VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

SAMOČINNÉ TESTOVÁNÍ MIKROKONTROLERŮ

SELF-TESTING OF MICROCONTROLLERS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. FILIP DENK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JOSEF STRNADEL, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Denk Filip, Bc.**
Program: Informační technologie Obor: Počítačové a vestavěné systémy
Název: **Samočinné testování mikrokontrolerů**
Self-Testing of Microcontrollers
Kategorie: Vestavěné systémy
Zadání:

1. Nastudujte a zdokumentujte principy související s testováním elektronických systémů, obzvláště se zaměřte na principy vhodné pro samočinné testování mikrokontrolerů.
2. Vymezte i) množinu metod a prostředků samočinného testování mikrokontrolerů a ii) třídu a části mikrokontrolerů, na které budete tyto prostředky a metody aplikovat.
3. Diskutujte a shrňte požadavky kladené na technické řešení (např. programové vybavení) schopné samočinně otestovat mikrokontroler dle vymezení bodem 2 a navrhněte princip činnosti a způsob realizace tohoto řešení.
4. Řešení navržené v bodě 3 realizujte.
5. Funkčnost realizovaného řešení vhodně ověřte, demonstrujte a shrňte - zejména z hlediska jeho schopnosti odhalit poruchy a chyby (v mikrokontroleru i v programu, který vykonává).
6. Zhodnoťte vlastnosti realizovaného řešení, diskutujte možná rozšíření a modifikace realizovaného řešení a rozveďte ty, které považujete za nejperspektivnější z hlediska uplatnění v praxi.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Strnadel Josef, Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 9. května 2019

Abstrakt

Práce se zabývá funkční bezpečností elektronických systémů. Konkrétně se zaměřuje na samočinné testování mikroprocesoru a jeho periferií na softwarové úrovni. Cílem práce je navrhnout a implementovat soubor funkcí v jazyce C nebo v jazyce symbolických adres, které samočinně testují zvolené oblasti mikrokontroléru. Prostředky a metody použité v implementovaném řešení si také kladou za cíl splnit požadavky popsané v normě IEC 60730-1, příloha H, softwarová třída B. Zvolenou hardwarovou platformou je mikrokontrolér NXP LPC55S69, jež obsahuje dvě jádra ARM Cortex-M33. Výsledkem je demonstrační aplikace, která v průběhu vykonávání využívá implementované testovací funkce. Součástí je také uživatelské prostředí s možností injekce chyb.

Abstract

This Master's thesis deals with functional safety of electronic systems. Specifically, it focuses on self-testing of the microprocessor and its peripherals at the software level. The main aim of the thesis is to design and implement a set of functions written in programming language C or assembly language, which automatically test the selected areas of the microcontroller. Resources and methods used in the implemented solution also aim to meet the requirements according to the safety standard IEC 60730-1, Annex H, Software Class B. The microcontroller NXP LPC55S69 was chosen as a hardware platform. It consists of two ARM Cortex-M33 cores. As a result, the example application is provided, which uses implemented test functions at the run-time. Example application also contains a graphical user interface with fault injection ability.

Klíčová slova

funkční bezpečnost, samočinné testování, mikrokontrolér, mikroprocesor, vestavěný systém, testování, ARM Cortex, VLSI, IEC 60730, IEC 61508, MCUXpresso

Keywords

functional safety, self-test, microcontroller, microprocessor, embedded system, testing, ARM Cortex, VLSI, IEC 60730, IEC 61508, MCUXpresso

Citace

DENK, Filip. *Samočinné testování mikrokontrolerů*. Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Josef Strnadel, Ph.D.

Samočinné testování mikrokontrolerů

Prohlášení

Prohlašuji, že jsem tuto semestrální práci vypracoval samostatně pod vedením pana Ing. Josefa Strnadela, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Filip Denk
15. května 2019

Poděkování

Děkuji vedoucímu práce Ing. Josefu Strnadelovi, Ph.D. za odborné vedení a užitečné rady do teoretické a praktické části. Také děkuji manželce a rodičům za podporu při studiu.

Obsah

Kapitola 1

Úvod

Elektronická zařízení nás obklopují stále častěji, ať už v soukromí či na veřejnosti. Mnoho takových elektronických zařízení by při selhání dokázalo ublížit člověku, zničit věci okolo či znečistit životní prostředí. Proto je potřeba mít tato zařízení pod neustálou kontrolou od úplného počátku, tzn. od výroby přes testování za provozu až po likvidaci zařízení.

Cílem této práce je poskytnout přehled technik, pomocí kterých jsou elektronická zařízení samočinně testována, a následně vybrané techniky použít pro implementaci samočinných testů na konkrétním mikrokontroléru.

Práce se (mimo úvod a závěr) dělí na 4 hlavní kapitoly. První kapitola poskytuje všeobecný přehled v oblasti testování mikrokontrolérů. V úvodu je vysvětlen pojmem *Funkční bezpečnost* a dále jsou popsány bezpečnostní normy IEC 61508 a IEC 60730. Zejména bezpečnostní norma IEC 60730 je ve vztahu k této práci důležitá, protože určuje požadavky, dle kterých jsou v rámci řešení implementovány softwarové testy. Následně jsou rozebrány principy hardwarového a softwarového testování mikrokontrolérů, jednotlivé modely poruch a také možnosti detekce takových poruch.

V další kapitole, která se věnuje návrhu řešení, jsou na úvod vymezeny oblasti mikrokontroléru, na které jsou aplikovány implementované testy. Poté jsou návrhy takových testů a jejich požadavky jednotlivě popsány. Tato kapitola také obsahuje popis vybrané hardwarové platformy a zdůvodnění jejího výběru. Na samotném závěru této kapitoly se nachází návrh demonstrační aplikace, jejíž cílem je ukázat, jak je možné implementované bezpečnostní testy integrovat do reálné aplikace.

Hlavní část této práce představuje kapitola *Implementace* (4). Popisuje způsoby, jakými jsou jednotlivé testy implementovány. U každého testu je okomentován úsek kódu, který zachycuje hlavní podstatu daného testu. Součástí implementace je také schopnost injekce poruch (za běhu aplikace), které musí bezpečnostní testy rozpoznat. Tato kapitola rozebírá jednak způsob, jakým jsou poruchy injektovány, ale také obsahuje naměřené časy trvání jednotlivých testů a také maximální možnou dobu, po jejíž uběhnutí aplikace zaručeně rozpozná danou poruchu.

Poslední kapitola dosažené výsledky vyhodnocuje. Obsahuje 4 názorné případy injekce poruchy, které jsou implementovanými testy rozpoznány. Na úplném závěru kapitoly se nachází stručný výčet zajímavostí, které objevily ve fázi vývoje.

Kapitola 2

Testování mikrokontrolérů

Tato kapitola si klade za cíl vysvětlit a přiblížit čtenáři problematiku testování mikrokontrolérů. Na začátku kapitoly jsou vysvětleny základní pojmy, které se v průběhu celé práce vyskytují. Dále kapitola popisuje základní bezpečnostní normu IEC 61508 a také normu IEC 60730, jež je pro tuto práci stěžejní. Dále je popsáno testování na úrovni úrovni hardware a software. Zejména poslední část tvoří základ pro návrh a testování.

2.1 Základní pojmy

Na úvod kapitoly je vhodné definovat základní pojmy, jejichž význam je sice na první pohled zřejmý, avšak bývají často zaměňovány či špatně pochopeny.

- **Funkční bezpečnost** – Funkční bezpečnost je součást celkové bezpečnosti systému. Sleduje bezpečnostní aspekty, které se vztahují k funkci zařízení nebo systému a zajišťuje, aby systém fungoval správně v reakci na vstupy [10]. Dnešní elektronické systémy jsou obvykle tak složité, že je prakticky nemožné, aby byly při vývoji otestovány všechny možné stavy systému. Proto musí funkční bezpečnost umět za běhu zabránit nebezpečným chybám nebo pokud už k takové chybě dojde, tak musí zajistit nebo udržet bezpečný stav z hlediska konkrétní nebezpečné události.

  Jako názorný příklad funkční bezpečnosti je možné uvést protipožární systém, který při detekci kouře v místnosti spustí hasicí proces a zároveň informuje integrovaný záchranný systém. Naopak za instanci funkční bezpečnosti nelze považovat pasivní systémy, např. ohnivzdorné dveře apod.

- **Defekt** v elektronickém systému – (anglicky *defect*) neúmyslný rozdíl mezi implementovaným hardwarem a jeho designem. Defekty se dělí na procesní, materiálové, zapouzdřovací a defekty způsobené stářím [5]. Může jít například o zkrat mezi vodiči nebo o rozpojený vodič.

- **Porucha** – (anglicky *fault*) model defektu na logické úrovni [4]. Defekty, které nejsou pokryty poruchovým modelem, nemusí být odhaleny, v důsledku čehož pak může být vadný čip akceptován. Jako příklad lze uvést poruchu **stuck-at**.

- **Chyba** – (anglicky *error*) rozdíl mezi očekávanými a aktuálními výsledky, které systém poskytuje. Nemusí vždy platit, že porucha vyvolá chybu. Zde může slovo chyba představovat také chybu v softwaru či návrhu hardware [4].

- **Selhání** – (anglicky *failure*) odchylka v činnosti obvodu nebo systému od jeho specifikovaného chování. Původní (bezchybové) funkcionality systému lze dosáhnout pouze opravou [4].
- **Výnosnost výroby** – poměr výrobků, které prošly akceptačním testem, vůči celkovému počtu výrobků [35].

$$Výnosnost\ výroby = \frac{akceptované\ výrobky}{celkový\ počet\ výrobků} \tag{2.1}$$

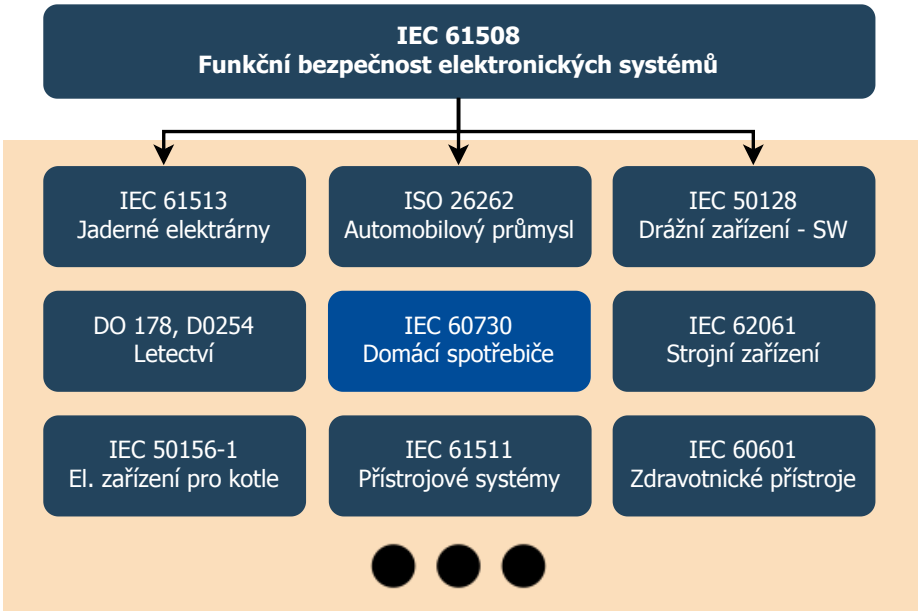
- **Míra zamítnutí** – poměr vadných výrobků, které prošly akceptačním testem, vůči celkovému počtu výrobků, které prošly akceptačním testem [35].

$$Míra\ zamítnutí = \frac{falešně\ akceptované\ výrobky}{celkový\ počet\ akceptovaných\ výrobků} \tag{2.2}$$

- **Kombinační logický obvod** – obvod, jehož výstup závisí pouze na stavech vstupních signálů.
- **Sekvenční logický obvod** – obvod, jehož výstup závisí na stavech vstupních signálů a zároveň na posloupnosti vstupů minulých (na vnitřním stavu). Sekvenční logický obvod se skládá ze dvou částí – kombinační a paměťové.
- **MTTR** – (anglicky *Mean time to repair*) množství času, které je vynaloženo na opravu systému a jeho následné uvedení do původního (bezporuchového) stavu [6].
- **MTBF** – (anglicky *Mean time between failure*) představuje střední dobu mezi dvěma poruchami. Tato veličina je určena pro zařízení, která se opravují. Za poruchu se však nedá počítat pravidelná údržba zařízení (mazání, výměna opotřebovaných dílu apod.) [6].
- **MTTF** – (anglicky *Mean time to failure*) metrika, která říká, jak dlouhá je předpokládaná životnost zařízení (střední doba do poruchy). Je určená pro zařízení, která se neopravují [6].

2.2 Bezpečnostní normy

Pro zajištění požadované úrovně bezpečnosti elektronických systémů je nutné dodržovat odpovídající normy, které se na danou skupinu zařízení vztahují. V případě samočinného testování mikrokontrolérů se jedná zejména o dvě normy, které jsou v následujících podkapitolách podrobněji popsány. V úvodu této sekce lze vidět přehled vybraných bezpečnostních norem, na kterém je zvýrazněna norma IEC 60730.



Obrázek 2.1: Přehled vybraných bezpečnostních norem. Na diagramu lze vidět obecnou normu IEC 61508 a normy od ní odvozené. Barevně zvýrazněná je norma IEC 60730, na jejíž aplikaci se zaměřuje tato práce.

2.2.1 IEC 61508

Český název normy: *Funkční bezpečnost elektrických/elektronických/programovatelných elektronických systémů souvisejících s bezpečností.*

Tato norma je základním bezpečnostním standardem platným pro všechny druhy průmyslu. Základní koncepce spočívá v tom, že jakýkoli systém související s bezpečností musí fungovat správně a pokud selže, tak musí být stále bezpečný (předvídatelný). Struktura této normy se skládá ze 7 dílů. První 3 díly jsou základní, následující 4 díly jsou podpůrné pro první 3 [25]. Obsah jednotlivých dílů je stručně shrnut v následujícím seznamu:

1. **Všeobecné požadavky** – zahrnuje všechny aspekty, které musí být zváženy, když jsou E/E/PE systémy použity k vykonávání bezpečnostních funkcí. Dále stanovuje úrovně integrity bezpečnosti SIL a požadavky na dokumentaci [12].
2. **Požadavky pro E/E/PE systémy související s bezpečností** – specifikuje požadavky na činnosti, které mají být vykonávány při navrhování a výrobě systémů souvisejících s bezpečností E/E/PE s výjimkou softwaru, který je řešen v IEC 61508-3 [13].
3. **Požadavky na software** – tento díl definuje specifické požadavky vztahující se na podpůrné nástroje používané k vývoji a konfiguraci systému souvisejícího s bezpečností v rámci IEC 61508-1 a IEC 61508-2. Vyžaduje, aby byly specifikovány bezpečnostní funkce softwaru a systematické schopnosti softwaru. Stanovuje požadavky na fáze a činnosti životního cyklu bezpečnosti, které se použijí při navrhování a vývoji softwaru souvisejících s bezpečností [14].

4. **Definice a zkratky** – obsahuje definice a vysvětlení pojmů, které se používají v částech 1 až 7 řady norem IEC 61508. Definice jsou seskupeny do kapitol tak, aby byly příbuzné pojmy správně pochopeny v kontextu [15].

5. **Příklady metod určování úrovní integrity bezpečnosti** – poskytuje metody, které pomáhají určit úrovně integrity bezpečnosti pro systémy související s bezpečností E/E/PE systémů. Aplikace principů ALARP[1] [16].

6. **Metodické pokyny pro aplikaci dílu 2 a dílu 3** – obsahuje postupy pro výpočet pravděpodobnosti hardwarových poruch; popisuje obvyklé příčiny poruch a aplikování požadavků na software [17].

7. **Přehled technik a opatření** – obsahuje přehled různých bezpečnostních technik a opatření týkajících se IEC 61508-2 a 61508-3 [18].

Úrovně integrity bezpečnosti

K vykonávání důležité práce jsou nutné spolehlivé pracovní prostředky. A čím důležitější ta práce je, tím spolehlivější by měly být i prostředky. V případě bezpečnostních systémů je prací myšleno zajištění bezpečnosti. Čím větší význam má bezpečnost systému, tím menší musí být četnost výskytu nebezpečných poruch. Míru četnosti výskytu nebezpečných poruch vyjadřuje integrita bezpečnosti systému, dále SIL[2], definovaná v části 4 normy IEC 61508 jako pravděpodobnost, že bezpečnostní systém bude uspokojivě plnit požadované bezpečnostní funkce za daných podmínek během stanovené doby.

V normě je SIL definována jako diskrétní hodnota mezi SIL1–SIL4. SIL4 znamená nejvyšší úroveň integrity, naopak SIL1 nejnižší. Kategorie SIL tedy reprezentuje výslednou pravděpodobnost výskytu nebezpečné poruchy definované bezpečnostní funkce.

Norma přiřazuje k SIL pravděpodobnost výskytu nebezpečných chyb ve dvou případech. Jeden platí pro systémy činné nepřetržitě (systém s vysokým vyžádáním a druhý pro systémy pracující na základě jednotlivého požadavku (systém s nízkým vyžádáním). Podle normy je od systémů s nízkým vyžádáním požadována činnost méně často než jednou za rok.

Následující tabulka zobrazuje jednotlivé SIL a k nim přiřazené pravděpodobnosti výskytu nebezpečných chyb u obou výše zmíněných případů.

Tabulka 2.1: Úrovně integrity bezpečnosti systému dle IEC 61508 [15].

Úroveň integrity bezpečnosti	**Režim s vysokým vyžádáním (nebezpečné poruchy/hod)**	**Režim s nízkým vyžádáním (pravděpodobnost poruchy na vyžádání)**
SIL1	$\geq 10^{-6}$ až $< 10^{-5}$	$\geq 10^{-2}$ až $< 10^{-1}$
SIL2	$\geq 10^{-7}$ až $< 10^{-6}$	$\geq 10^{-3}$ až $< 10^{-2}$
SIL3	$\geq 10^{-8}$ až $< 10^{-7}$	$\geq 10^{-4}$ až $< 10^{-3}$
SIL4	$\geq 10^{-9}$ až $< 10^{-8}$	$\geq 10^{-5}$ až $< 10^{-4}$

Jak je uvedeno v definici, je účelem kategorií SIL zadat úkoly návrhářům. V případě jednoduchého elektromechanického hardwaru lze prokázat dosažení určité kategorie SIL

[1] ALARP – as low as reasonably practicable (dokud jsou náklady na snížení rizika rozumné).
[2] SIL – safety integrity level (integrita bezpečnosti systému).

použitím údajů o četnosti poruch. Pro složité systémy a v případě softwaru, kde poruchy nejsou náhodné, ale systematické, nelze SIL tímto způsobem prokázat. Potom SIL vyjadřuje preciznost uplatněnou ve fázi vývoje produktu. Jinými slovy, protože nelze spolehlivě prokázat četnost nebezpečných poruch softwarového produktu, je nutné obrátit pozornost do procesu jeho vývoje. V tomto případě vyžaduje SIL 1 základní znalosti projektování a dodržování zásad řízení jakosti. Vyšší SIL následně vyžaduje, aby byl tento základní požadavek zpřísněn [34].

2.2.2 IEC 60730-1

Český název normy: *Automatická elektrická řídicí zařízení pro domácnost a podobné účely.*

Tato norma pokrývá mechanickou, elektrickou, elektronickou, ekologickou odolnost, elektromagnetickou kompatibilitu[3] a abnormální stavy vztahující se na domácí spotřebiče. Zejména *Příloha H: Požadavky na elektronické řízení*, která podrobně popisuje testovací a diagnostické metody pro zajištění bezpečného provozu vestavěného řídícího hardwaru a software pro domácí spotřebiče.

Pro hodnocení ochranných opatření a pro odolnost proti poruchám a předcházení rizikům je potřeba klasifikovat řídící funkce s ohledem na jejich poruchové chování. Pro účely posouzení konstrukce řídící funkce je třeba rozdělit požadavky do tří různých tříd:

- **Třída A** – Řídicí funkce, které nemohou způsobit vážné škody či ublížení na zdraví.
- **Třída B** – Řídicí funkce, jejichž účelem je zabránit nebezpečnému stavu spotřebiče.
- **Třída C** – Řídicí funkce, jejichž účelem je zabránit speciálnímu nebezpečí, jako např. výbuchu apod.

Velké spotřebiče, jako jsou pračky, myčky nádobí, sušičky, ledničky, mrazničky a sporáky, spadají do třídy B. Výjimkou je samozřejmě zařízení, které spadá do třídy C, protože by mohlo způsobit výbuch (například plynový vařič)).

V následující tabulce je uveden přehled testů, které jsou vyžadovány normou IEC 60730-1, softwarová třída B. Obecně platí, že každou testovanou komponentu lze otestovat různými způsoby, což poskytuje flexibilitu pro výběr vhodného opatření pro dané zařízení.

V-model

Norma IEC 60730-1 také předepisuje doporučený průběh návrhu a vyvíjení produktu. Koncepce na obrázku 2.2 je převzatá z normy IEC 61508-3 a dále upravená pro potřeby standardu IEC 60730-1. Tento návrhový a vývojový cyklus je znám jako V-model. Popis vlastností tohoto modelu je částečně převzat z [19]. Vyznačuje se především tím, že hierarchicky rozděluje jednotlivé úrovně požadavků a specifikací systému. Každé takové úrovni odpovídá příslušné testování. Obecně lze V-model rozdělit na 3 hlavní části:

1. **Specifikace** – levá část modelu, za kterou zodpovídá vývojář. Představuje vůbec první kroky v celém průběhu vývoje produktu. Začíná definicí požadavků na software, pokračuje návrhem systému (architektury) a končí detailní specifikací jednotlivých modulů. Každá z těchto fází má svůj výstup – návrhový/specifikační dokument.

[3] EMC – Electromagnetic compatibility (elektromagnetická kompatibilita). Vlastnost elektrického nebo magnetického přístroje nebo nástroje spočívající v tom, že neovlivňuje jiný objekt včetně sebe samotného a že odolává působení ostatních přístrojů.

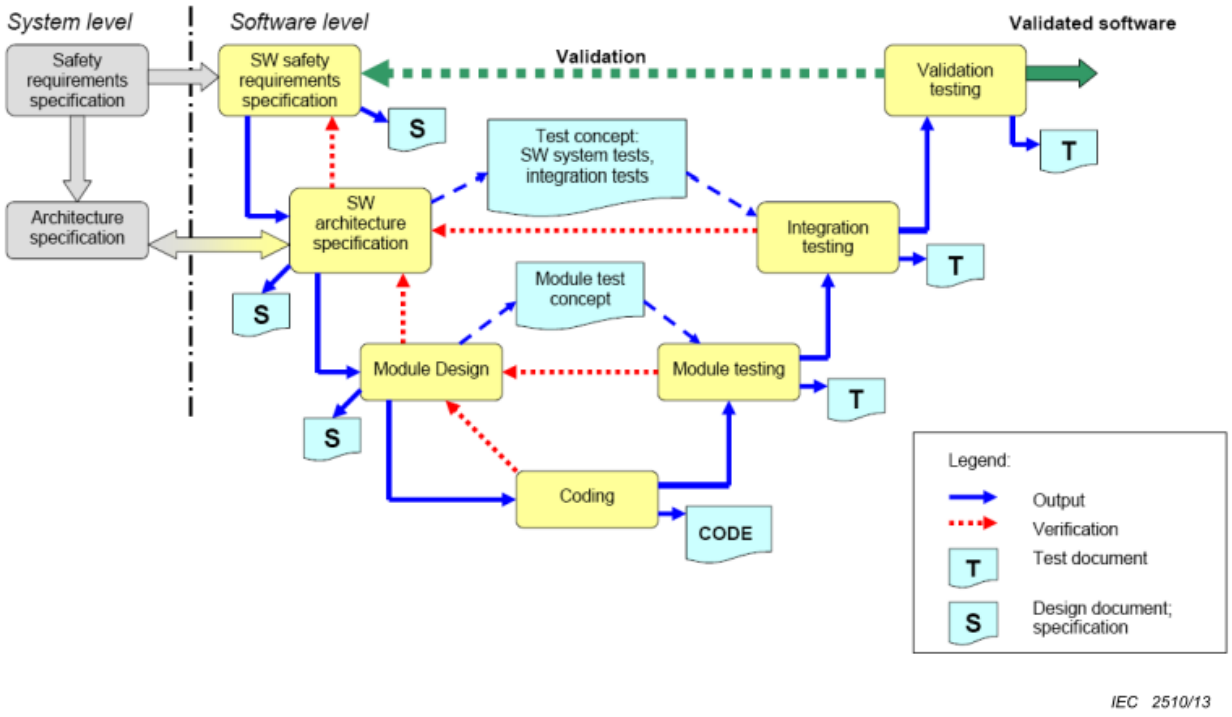
Tabulka 2.2: Přehled softwarových testů, které jsou vyžadované normou IEC 60730-1, softwarová třída B [11].

Testovaná komponenta	**Testovaná chyba**	**Přijatelná opatření**
CPU registry	Stuck-at	test statické paměti
Programový čítač	Stuck-at	test statické paměti
Frekvence přerušení a tok programu	špatná frekvence	watchdog; časovač počítající přerušení
Hodinová frekvence	špatná frekvence	porovnání frekvence vůči referenčnímu zdroji
Variabilní paměť (RAM)	všechny bitové poruchy	test statické paměti; kontrolní součet
Nevariabilní paměť (ROM)	DC porucha	kontrolní součet
Interní komunikační kanály	Stuck-at	bitová redundance
Externí komunikace	Hammingova vzdálenost	kontrolní součet
Digitální vstup/výstup	Specifické chybové stavy	kontrola hodnoty na vstupu/výstupu
Analogový vstup/výstup	Specifické chybové stavy	kontrola hodnoty na vstupu/výstupu

2. **Implementace** – hlavní část celého vývojového cyklu, jejímž výstupem je kód aplikace. Vývojář implementuje vůči dané specifikaci (z předchozích kroků).

3. **Testování** – pravá část modelu, za kterou zodpovídá tester. Po implementaci jsou nejdříve testovány a ověřovány menší jednotky, komponenty a celky a teprve následně jejich větší seskupení až po kompletní systém. Výhodou této strategie je, že testování dříve odhalí chyby v kritických modulech, a že může být zahájeno dříve, než jsou všechny komponenty hotovy. Výstupem jednotlivých fází je tzv. test-report, což je dokument, který vyhodnocuje testování dané úrovně.

Pro dodržení normy IEC 60730 je samozřejmě možné vyvíjet software také pomocí jiných metodik (např. agilní metodiky), ale pouze za předpokladu dodržení předem stanovených procesů včetně fáze návrhu a testování [11].



Obrázek 2.2: V-model – vývojová metodika, která je doporučena normou IEC 60730-1 [11].

Norma IEC 60730 neobsahuje pouze část 1, ale také část 2 (IEC 60730-2), která se nazývá *Automatická elektrická řídicí zařízení pro domácnost a podobné účely.* Tato část se dále dělí na mnoho dalších podčástí, které se nazývají např. *Zvláštní požadavky na časové relé a časové spínače*, *Zvláštní požadavky na elektrické automatiky hořáků* či *Zvláštní požadavky na řídicí zařízení pro snímání teploty* apod.

2.2.3 Další bezpečnostní normy

V následujícím seznamu jsou pro příklad uvedeny další bezpečnostní normy, které jsou v některých odvětvích průmyslu velice významné.

- **IEC 61511** – *Bezpečnostní přístrojové systémy pro sektor průmyslových procesů.*
- **IEC 61513** – *Jaderné elektrárny – Systémy kontroly a řízení důležité pro bezpečnost.*
- **IEC 50128** – *Drážní zařízení – Sdělovací a zabezpečovací systémy a systémy zpracování dat – Software pro drážní řídicí a ochranné systémy.*
- **IEC 60601** – *Zdravotnické elektrické přístroje.*

2.3 Hardwarové testování

Tato sekce poskytuje přehled různých aspektů testování, kterým je nutné v oblasti hardware porozumět. V této sekci je popsáno jak a proč se mikrokontroléry testují.

Zavedení integrovaných obvodů s sebou doprovázelo potřebu testování těchto zařízení. Obvody z počátku šedesátých let, které obsahovaly pouhé jednotky tranzistorů, se označují jako *malý stupeň integrace* (SSI[4]). Jako *střední stupeň integrace* (MSI[5]) se označuje obvod složený z desítek tranzistorů. Testování těchto obvodů nebyla do začátku 80. let 20. století tak veliká výzva, ale postupně se integrace zvyšovala a tak vznikl *vysoký stupeň integrace* (LSI[6]) se stovkami až tisícovkami tranzistorů. Následoval *velmi vysoký stupeň integrace* (VLSI[7]) obsahující desítky až sta tisíce tranzistorů. V 90. letech tento pokrok vyústil v zařízení se stovkami miliony tranzistorů (ULSI[8]) v jediném křemíkovém mikročipu, se kterými přišlo mnoho nových testovacích problémů. Termínem VLSI se často označuje také stupeň ULSI.

Postupným snižováním velikosti čipů se zvyšuje pravděpodobnost, že se ve výrobním procesu objeví integrovaný obvod, který bude obsahovat výrobní defekt. Stačí pouze jeden vadný tranzistor či vodič a celý čip nemusí fungovat správně či na požadované frekvenci. V důsledku toho se očekává, že malá část výrobků (čipů) bude vadných. Proto je neustále nutné provádět testování, aby byla zaručena bezporuchovost výrobku bez ohledu na to, zda je výrobkem samostatný čip, nebo elektronický systémem složený z mnoha takových čipů. Také je nutné testovat komponenty v různých fázích během výrobního procesu. Například pro výrobu elektronického systému je potřeba nejprve integrovaný obvod vyrobit, použít tento obvod k sestavení desky plošných spojů (PCB[9]) a pak použít tuto PCB pro sestavení systému.

Existuje obecné tvrzení – *pravidlo deseti* – které říká, že při procesu zajištění kvality stojí detekce defektu integrovaného obvodu 10krát více času a peněz v každé další fázi [35]. Elektronické testování zahrnuje testování integrovaného obvodu v různých fázích výroby a také během provozu systému. Testování se používá nejenom pro zjištění bezporuchových zařízení, desek plošných spojů a systémů, ale také pro zlepšení efektivity produkce v různých fázích výroby, a to analýzou příčiny závad v případě výskytu poruch. V některých systémech se provádí periodické testování, které zajišťuje bezporuchový provoz systému a iniciuje opravy při zjištění poruch. Ve výsledku je tedy testování důležité nejenom pro designéry a testovací inženýry, ale také pro management a koncové uživatele [20].

Kvůli vadám v materiálu a v maskách používaných k výrobě integrovaných obvodů, je statisticky nemožné vyrábět 100% obvodů daného druhu bez defektů. Defekty se objevují jak v průběhu výroby, tak během používání zařízení/součástky. Defekty stejného typu, které se objevují pravidelně, by měly být považovány za indikátor potřeby zkvalitnit výrobní nebo návrhový proces. Analytická metoda, jejímž cílem je identifikovat místa možného vzniku defektů při výrobě se nazývá *analýza možného výskytu a vlivu vad* (FMEA[10]).

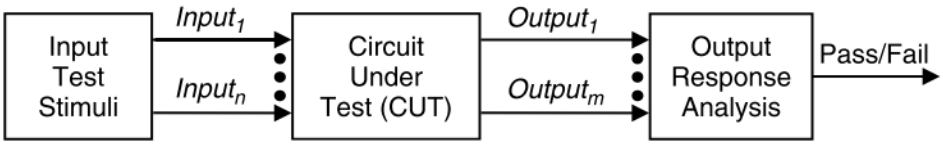
Na obrázku 2.3 lze vidět typický postup při testování. Na začátku takového procesu jsou vygenerovány vstupní testovací stimuly (vektory), které se aplikují na vstupy testovaného

[4] SSI – small-scale integration (malý stupeň integrace).
[5] MSI – medium-scale integration (střední stupeň integrace).
[6] LSI – large-scale integration (vysoký stupeň integrace).
[7] VLSI – very large-scale integration (velmi vysoký stupeň integrace).
[8] ULSI – ultra large-scale integration (ultra vysoký stupeň integrace).
[9] PCB – printed circuit board (deska plošných spojů).
[10] FMEA – failure mode and effects analysis (analýza možného výskytu a vlivu vad).

Tabulka 2.3: Typické zdroje defektu PCB (převzato z [5]).

Typ defektu	Četnost výskytu [%]
Zkrat	51
Jiná komponenta	13
Ohnuté vodiče	8
Otočená komponenta	6
Chybějící komponenta	6
Výkonnostní defekt	5
Chybné analogová specifikace	5
Chybná logika	5
Otevřený spoj	1

obvodu (CUT[11]). Jakmile provede CUT výpočet nad vstupními vektory, přichází na řadu analýza výstupních hodnot. Obvody, jejichž výstupy odpovídají správným hodnotám, jsou považovány za bezchybné (prošly testem). Naopak obvody, jejichž výstupy neodpovídají správným hodnotám, jsou považovány za chybné.



Obrázek 2.3: Postup při testování integrovaného obvodu (převzato z [35]).

2.3.1 Generování testů

Mějme testovaný obvod (CUT) s n vstupy. Jak již bylo zmíněno – každý vstupní vzorek se nazývá testovací vektor. Pak generování testů představuje proces, při kterém se vytváří testovací vektory a k nim odpovídající výstupy. Cílem tohoto procesu je najít co nejmenší množinu testovacích vektorů, která je schopná detekovat co nejvyšší počet chyb. Při vysokém n je pro úplné otestování obvodu zapotřebí vytvořit/vygenerovat mnoho těchto vstupních vektorů, přesněji 2^n. Je však obtížné zjistit, kolik testovacích vektorů je zapotřebí k zajištění uspokojivé míry zamítnutí (2.1).

Pokud je CUT kombinační logický obvod, je možné na něj aplikovat všech 2^n testovacích vektorů, testujících Stuck-at (2.3.2) chyby. Tento přístup se nazývá *vyčerpávající testování*. Pokud obvod projde tímto testem, můžeme předpokládat, že obvod neobsahuje funkční poruchy, bez ohledu na jeho vnitřní strukturu. Vyčerpávající testování bohužel není praktické, pokud je n velké a navíc není tak účinné při testování sekvenčních logických obvodů, protože i když jsou aplikovány všechny vstupní vektory, tak není zaručeno, že byly navštíveny všechny možné stavy.

Předchozí příklad vystihuje hlavní myšlenku *funkčního testování*, kde je testován každý záznam v pravdivostní tabulce pro daný kombinační obvod, aby se zjistilo, zda obvod produkuje správné výsledky. Problémem obou případů je nedostatek kvantitativního měření defektů, které jsou detekovány sadou testovacích vektorů.

[11] CUT – circuit under test (testovaný obvod).

Praktičtějším přístupem je výběr konkrétních testovacích vektorů založených na znalosti struktury obvodu a množině chybových modelů (2.3.2). Tento přístup se nazývá *strukturální testování*. Šetří čas a zlepšuje efektivitu testovacího procesu, protože celkový počet vstupních vektorů, je díky zaměření na konkrétní chyby, které by vyplynuly z výrobních defektů, snížen.

2.3.2 Chybové modely

Struktura této sekce, která popisuje nejčastější chybové modely, je převzatá z knihy *VLSI Test Principles and Architectures: Design for Testability* od L.T. Wang [35], kapitola *Fault Models*.

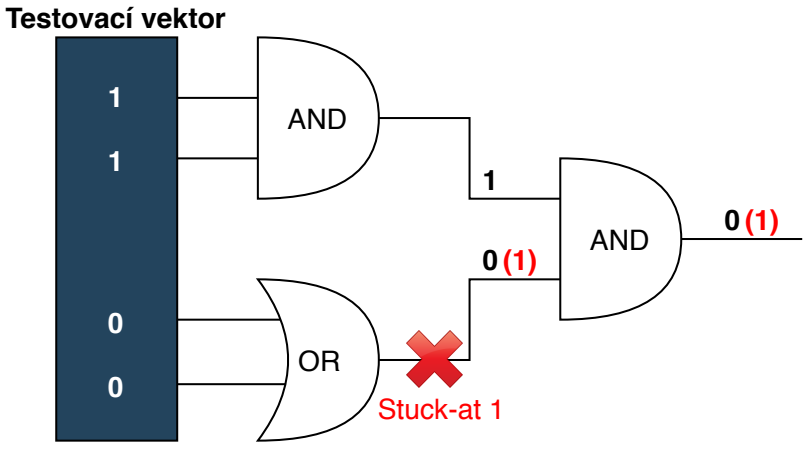
Vzhledem k rozmanitosti defektů VLSI (ULSI) je obtížné generovat vhodnou sadu testů. Proto jsou pro generování a vyhodnocování testů nezbytné právě *modely poruch*. Obecně by měl dobrý chybový model splňovat dvě kritéria:

1. Měl by přesně reflektovat chování konkrétního defektu.
2. Měl by být výpočetně efektivní z hlediska simulace chyb a generování testovacích vektorů.

V současnosti již bylo navrženo mnoho chybových modelů, ale bohužel žádný přesně neodráží chování všech možných defektů, ke kterým může dojít. Výsledkem je, že se při generování a vyhodnocování testovacích vektorů běžně používá kombinace různých chybových modelů. V následujících podsekcích jsou specifikovány ty nejčastější.

Stuck-at

Jedná se zřejmě o nejznámější model. Stuck-at chyba ovlivňuje stav logického signálu na vodiči v logickém obvodu (na hlavním vstupu/výstupu, na vstupu/výstupu interních tranzistorů apod.). Ovlivňuje jej tak, že se signál na vadném vodiči jeví jako konstantní hodnota – logická 1 nebo logická 0. Typicky se takový stav značí jako *SA0* (stuck-at 0) nebo *SA1* (stuck-at 1).



Obrázek 2.4: Jednoduchá ukázka stuck-at chyby. Na vstupech hradel AND a OR je přiveden testovací vektor. Očekávaný výstup obvodu je logická 0. Avšak mezi hradlem OR a AND se objevila stuck-at 1 chyba, což ovlivnilo výslednou hodnotu celého obvodu – logická 1. Červeně vyznačené hodnoty signálů ukazují vliv stuck-at chyby.

Ačkoli je fyzicky možné, aby vodič nabýval pouze hodnoty SA0 nebo SA1, mohou být pomocí testů (vyvinutích na detekci stuck-at chyby) detekovány také mnohé další defekty v obvodu. Byla navržena např. metoda N-detekce, určená pro detekci dalších defektů, které stuck-at chybový model nepokrývá [22]. V takové N-detekující množině testovacích vektorů, je každá jednotlivá stuck-at chyba detekována nejméně N různými testovanými vektory. Nicméně vstupní vektory generované pomocí modelu stuck-at nemusí nutně zaručit zjištění všech možných závad. Proto jsou zapotřebí další modely poruch.

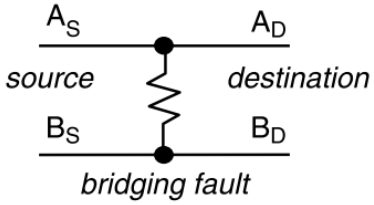
Tranzistorová chyba

Tranzistor může být trvale spojen či rozpojen. Stuck-at chybový model v tomto případě nedokáže přesně reflektovat chování tranzistorové poruchy (trvale spojen či rozpojen) v CMOS[12] logických obvodech, protože pro konstrukci CMOS logických hradel je použito více tranzistorů.

Zkrat/otevřený spoj

Vady v zařízeních VLSI mohou obsahovat také otevřený spoj nebo zkraty ve vodičích. Otevřený spoj ve vodiči má tendenci chovat se jako trvale rozpojený tranzistor. Na druhou stranu, otevřený spoj má tendenci se chovat jako stuck-at chyba.

Zkrat mezi dvěma prvky je obvykle označován jako *bridging fault* (přemostění). Zasažené mohou být tranzistorové terminály nebo spojení mezi tranzistory a hradly. Zkrat k napájení nebo zemi je ekvivalentní ke stuck-at chybovému modelu. Nicméně když jsou dva vodiče signálu zkratovány navzájem, je nutné aplikovat *bridging fault* modely.



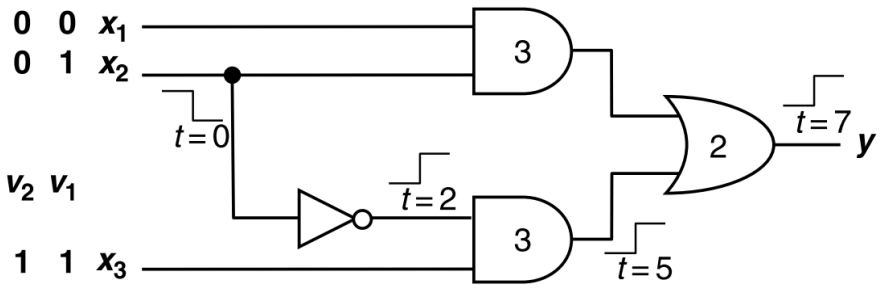
Obrázek 2.5: Bridging fault (převzato z [35]).

Zpoždění a přeslech

Bezporuchová činnost logického obvodu vyžaduje nejen správnou funkci logiky, ale také distribuci logických signálů v určeném časovém limitu. Zpoždění se objevuje stále častěji se snižující se velikostí obvodu.

Níže, na obrázku 2.6, je znázorněn jeden z několika možných druhů testů na zpoždění. Testovaná cesta začíná v $\mathbf{x_2}$ a vede přes spodní hradlo AND, následně přes hradlo OR a nakonec na výstup $\mathbf{y}$. Mějme testovací vektory $\mathbf{v_1}$ a $\mathbf{v_2}$, mezi kterýma nastane přechod v čase *t=0*. Dle uvedených zpoždění (čísla na hradlech) se dá očekávat, že pokud je obvod bez defektů, tak bude na výstupu $\mathbf{y}$ v čase *t=7* detekován přechod z logické **0** na **1**.

[12] CMOS – complementary metal–oxide–semiconductor.



Obrázek 2.6: Testování zpoždění – cesta z **x2**, přes spodní hradlo AND a následně hradlo OR, až na výstup **y** by měla mít v bezchybovém stavu zpoždění 7 (převzato z [35]).

2.3.3 Úrovně abstrakce chybových modelů

Navrhování obvodu je proces transformace z vysoko-úrovňového popisu na nízko-úrovňový popis obvodu. Jednotlivé úrovně takových modelů určují stupeň abstrakce, jakým se návrhář/tester dívá na použité komponenty.

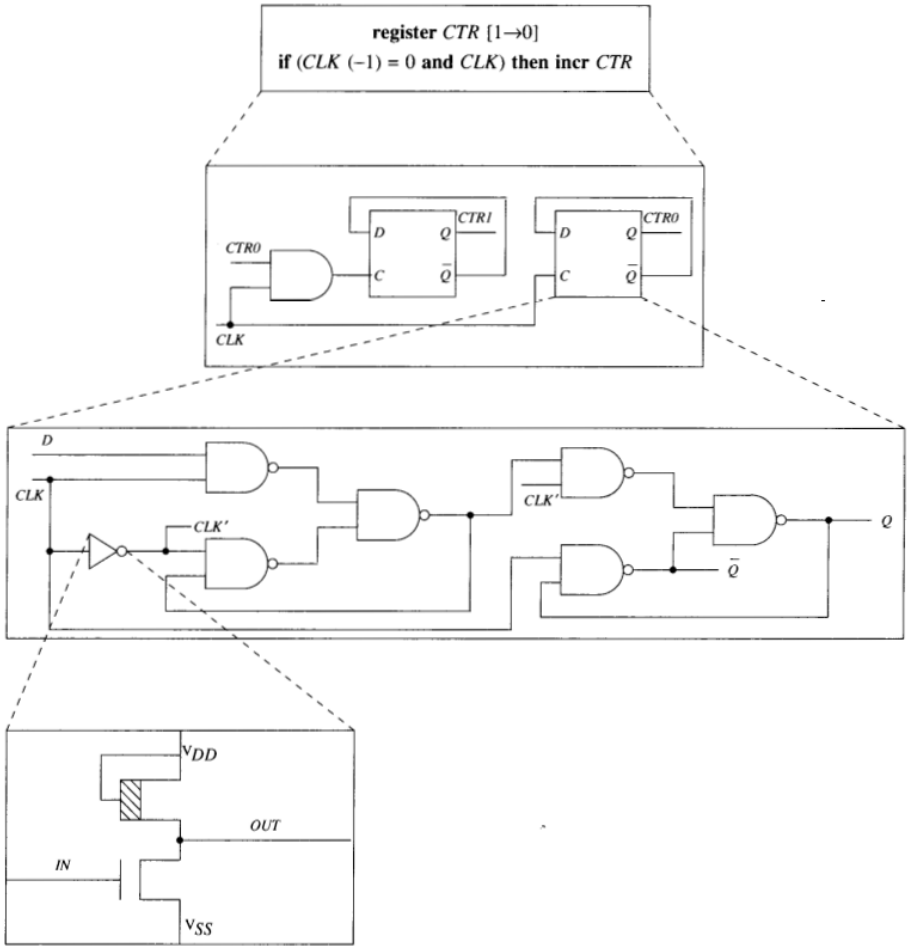
Počínaje specifikací, kdy jsou detailně popsány požadavky na výsledný systém, je popsán návrh na nejvyšší úrovni abstrakce – *behaviorální úroveň*. Vývoj na této úrovni probíhá v daném programovacím jazyce, typicky VHDL, Verilog nebo SystemC a je simulován tak, aby byla ověřena funkční ekvivalence vůči specifikaci.

Další úrovní je *úroveň přenosů mezi registry* (anglicky RTL[13]), která obsahuje více strukturálních informací, konkrétně sekvenční a kombinační logické obvody. Sekvenční obvody (registry – obvykle implementované jako D klopný obvod) synchronizují činnost obvodu s hranou hodinového signálu a jsou jedinými prvky v obvodu, které se chovají jako paměť. Kombinační logika provádí veškeré logické funkce v obvodu a skládá se z logických hradel. Předtím, než začne syntéza na logickou úroveň, musí být RTL popis také verifikován, aby splnil požadavky na funkčnost dané specifikací.

Logická úroveň (někdy také nazývaná *hradlová*) je výsledkem syntézy popisu RTL. Popisuje celý obvod pouze pomocí logických hradel, tzn. hradla AND, NAND, OR, NOR apod. Implementace logické úrovně by měla být ověřena co nejdetailněji, aby byla zaručena správná funkčnost výsledného návrhu. Nejpoužívanějším chybovým modelem na této úrovni je stuck-at. Dále také model zpoždění či bridging fault. V závěrečném kroku musí být popis logické úrovně transformován na popis fyzické úrovně.

Nejnižší úroveň abstrakce se nazývá *fyzická úroveň* (někdy také nazývaná *tranzistorová*). Tato úroveň popisuje obvod pomocí jednotlivých tranzistorů, což je nejmenší možná jednotka, na kterou je možné navrhovaný obvod rozložit. Při verifikaci se používá tento popis pro ověření časových a frekvenčních omezení. Chybový model používaný na této úrovni abstrakce je tranzistorová chyba – trvale spojen/rozpojen.

[13] RTL – register-transfer level (úroveň přenosů mezi registry).



Obrázek 2.7: Ukázka jednotlivých úrovní abstrakce na 2-bitovém čítači. Směrem od vrchu dolů klesá úroveň abstrakce – od behaviorálního modelu přes RTL, logickou úroveň a nakonec úroveň tranzistorovou (převzato z [1]).

2.4 Softwarové testování

Tento druh testování je znám pod zkratkou SBST[14] Jde o případ, kdy se testování provádí za běhu a neprovádí jej dedikovaná HW jednotka, ale procesor. Ten zpracovává jednotlivé instrukce testu, který je implementovaný v daném programovacím jazyce, například v jazyce C nebo jazyce symbolických adres. Z tohoto principu plynou dvě veliké nevýhody a to:

- **Zátěž procesoru** – procesor by měl být minimálně vytěžován vykonáváním bezpečnostních mechanismů. Měl by naopak vykonávat aplikaci samotnou. Mezi těmito dvěma přístupy je potřeba najít kompromis.
- **Delší exekuční doba testu** – je zřejmé, že hardwarová implementace testu je rychlejší, než vykonání testu na CPU. Za zmínku stojí například porovnání softwarové implementace výpočtu kontrolního součtu CRC[15] při testování paměti, oproti hardware modulu, který vypočítá kontrolní součet nad stejným množstvím dat podstatně rychleji [32].

Softwarová implementace má ale také výhody a to např. nezávislost na hardware při úpravě/rozšíření testu. Znamená to, že kdyby proběhla aktualizace hardwarového testu, tak se musí upravit návrh hardware, musí proběhnout nová verifikace (popř. výroba) a tím pádem se zvýší cena daného produktu. U softwarového testování cena při aktualizaci testu nepředstavuje takové zvýšení nákladů.

Současné mikrokontroléry již obsahují vestavěné softwarové mechanismy on-line detekce chyb. Jedná se například o testování platnosti op-kódu, přetečení zásobníku, chyby paměti apod. Jsou prováděny příslušným firmwarem, který neovlivňuje běžící aplikaci.

V řadě aplikací se objevuje závislost na správném načasování. Tyto podmínky by neměly být samočinným testováním porušeny. To samé platí pro neporušení dat, se kterými pracuje aplikace. Typicky je aplikační logika prováděna v pevně daných periodách. To znamená, že pro bezpečnostní testování je vyhrazen čas mezi jednotlivými periodami [30]. Při integraci bezpečnostních testů do již implementované aplikace je velice důležité správné časování zohlednit. Vůbec nejlepší je, pokud vývojář již při vývoji zná požadavky jak bezpečnostních testů, tak samotné aplikace. Může tím předejít významným komplikacím.

V praxi je běžné, že se některé softwarově implementované testy nestihnou vykonat za jednu periodu. Velmi často k tomu dochází například při testování paměti ROM a RAM. V takovém případě se test vykonává *parciálně*. Co to znamená, je znázorněno na následujícím příkladu.

Mějme aplikaci, ve které se každých 5 ms vykonává v přerušení aplikační kód a bezpečnostní funkce (pro zjednodušení pouze testování paměti RAM). Vykonání aplikačního kódu trvá 3 ms, tím pádem na testování paměti zbývají necelé 2 ms (pokud se počítá s menší rezervou). Velikost paměti RAM je 8192 B, přičemž průměrně je blok o velikosti 32 B otestován za 50 μs. Pokud tedy zbývají necelé 2 ms – celá paměť RAM se nestihne otestovat celá najednou (trvalo by to cca 12.8 ms). To je tedy důvod, proč se test paměti vykonává parciálně – každou periodu se otestuje následující jiný blok paměti.

V následujících sekcích jsou popsány principy softwarového testování jednotlivých komponent mikrokontrolérů a prevence proti poruchám. Většina z nich je aplikovatelná a dostatečná pro splnění normy IEC 60730, což je cílem této práce. Metody navržené v této

[14] SBST – software based self testing (softwarové samočinné testování).

[15] CRC – Cyclic redundancy check (Cyklický redundantní součet). Matematická funkce používaná pro detekci chyb během přenosu či ukládání dat.

sekci jsou inspirovány dokumenty [26, 30, 9, 33]. Autory těchto dokumentů jsou vývojáři/-designéři velkých společností jako je NXP Semiconductors či Cypress Semiconductors. Dá se tedy očekávat, že metody jimi popsané, jsou standardně používané v praxi.

2.4.1 Testování CPU registrů

Cílem testu procesorových registrů je odhalit **stuck-at** poruchy v registrech jádra. Prakticky to znamená, že pokud má alespoň jeden bit registru jádra konstantně hodnotu logické 0/1, tak test musí takový bit odhalit.

Pro detekci stuck-at poruchy je typicky implementován testovací algoritmus **Checkerboard** (česky *šachovnice*). Jeho princip spočívá v nahrávání a verifikaci dvou testovacích vzorů. Jedná se o sekvenci 4 kroků:

1. Nahrání vzoru **0xAAAAAAAA** do registru.
2. Verifikace hodnoty v registru.
3. Nahrání vzoru **0x55555555**.
4. Verifikace hodnoty v registru.

Zmíněné vzory nejsou zvoleny náhodně – jsou si navzájem negací. Záměrem je otestovat nejen stuck-at poruchu, ale také možnost, zda hodnota nějaké paměťové buňky neovlivňuje hodnotu sousední buňky (což by např. vzor 0xFFFF0000 a jeho negace detekovat nemusela).

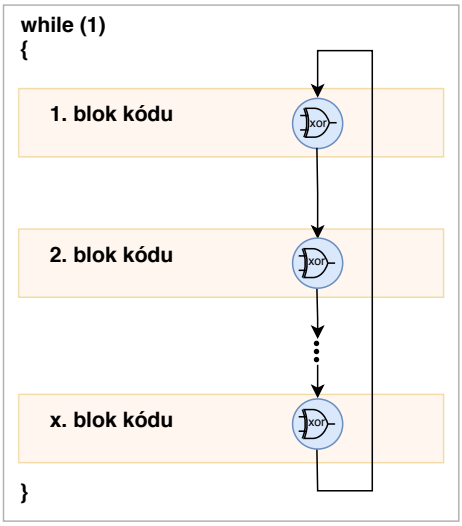
2.4.2 Testování frekvence přerušení a toku programu

Posloupnost vykonávání programu a frekvence přerušení jsou dvě odlišné věci, avšak úzce spolu souvisí. To je taky důvod, proč se nachází ve stejné sekci. Mechanismus přerušení je jedna ze stěžejních součástí aplikací ve vestavěných systémech. Z hlediska normy IEC 60730-B se musí testovat frekvence takových přerušení. Kontroluje se, zda se testované přerušení provádí tak často, jak se očekává – tzn. ne moc často, ani ne moc zřídka.

Pro každý případ se využívá jiný mechanismus. Pro detekci moc častého vykonávání přerušení slouží **počítadlo**, které se inkrementuje přímo v testovaném přerušení. Hodnota tohoto počítadla se jednou za určitou dobu kontroluje (v jiném přerušení nebo v nekonečné smyčce na pozadí), a pakliže překročí hranice tolerance, test detekuje chybu. Naopak pro detekci málo častého vykonávání přerušení lze elegantně vyřešit pomocí **watchdog** modulu, který v případě poruchy není obnoven.

Účelem testu správného toku programu je zjistit, zda aplikace prochází jednotlivými části kódu tak, jak se očekává. Metoda, která se pro tento test používá se nazývá **CFCSS**[16]. Na obrázku 2.8 je znázorněn její princip.

[16] CFCSS – Control Flow Checking by Software Signatures (Kontrola toku programu pomocí softwarových podpisů).



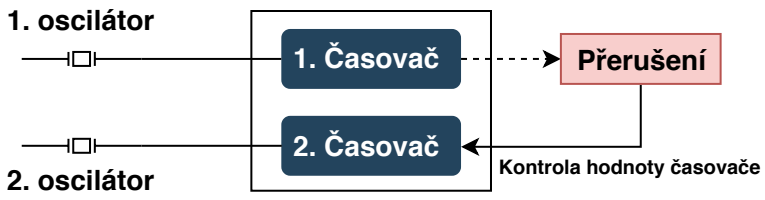
Obrázek 2.8: Test správného toku programu.

Test kontroly toku programu je založen na rozmístění předem definovaných uzlů v rámci vlákna (přerušení, smyčka atd.) a následné kontroly, zda je aktuální uzel přímý následovník uzlu předchozího. Kontrola se provádí jednoduchou operací XOR (více v implementační části 4.2.2).

2.4.3 Testování hodinové frekvence

Test správné frekvence hodinového signálu ověřuje, zda-li je hodinový signál, který řídí procesor, dostatečně přesný. K tomu je potřeba minimálně dvou nezávislých zdrojů hodinového signálu. Typicky se pro testování používá časovač s funkcí **capture** (česky *zachycení*). V principu jde o to, že se do hardwarového modulu (časovače) přivedou 2 zdroje hodinového signálu – primární a sekundární. Primární představuje hodinový signál procesoru, sekundární představuje jiný nezávislý zdroj hodinového signálu. S každou náběžnou hranou sekundárního (capture) signálu je zachycen počet náběžných hran signálu primárního. Uživatel pouze zkontroluje, zda je počet náběžných hran primárního signálu (za jednu periodu sekundárního signálu) v mezích tolerance.

Pokud není k dispozici funkce zachycení, je možné použít dva časovače, přičemž každý s jiným zdrojem hodinového signálu. Pak lze v obslužné rutině přerušení pomalejšího časovače vyčíst hodnotu časovače druhého a následně porovnat jejich hodnoty a test vyhodnotit. Nevýhoda této metody je, že zbytečně spotřebuje 2 časovače namísto jednoho. Princip je znázorněn na následujícím obrázku:



Obrázek 2.9: Princip testu hodinové frekvence s použitím dvou časovačů a dvou nezávislých zdrojů hodinového signálu.

Pro testování správné frekvence hodinového signálu se také dá použít **watchdog**. Musí však umět pracovat v režimu **window**. To znamená, že není požadováno pouze obnovovat watchdog před vypršením časového limitu, ale zároveň platí, že nesmí být obnoven příliš brzo. Tím se vymezí horní a dolní hranice tolerance.

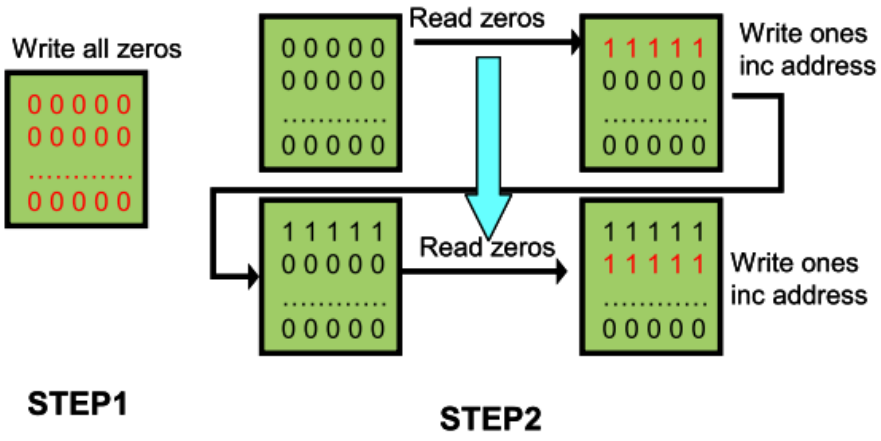
2.4.4 Testování variabilní paměti (RAM)

Ve variabilní paměti se nachází data, která v průběhu vykonávání programu mění hodnotu. Nejčastější a zároveň asi nejjednodušší způsob testování variabilní paměti — je vzhledem ke standardu IEC 60730-B — periodické testování statické paměti.

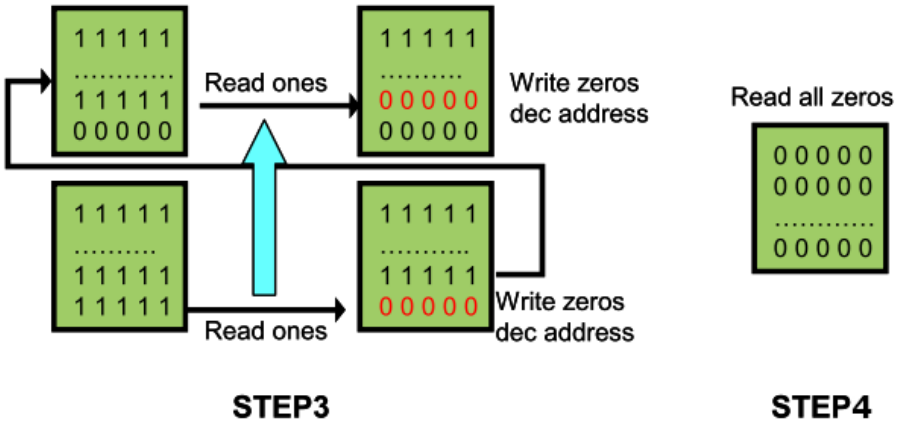
Test RAM paměti se provádí za účelem detekce **stuck-at** poruch nebo detekce ovlivňování sousedních paměťových buněk navzájem. Dle standardu se test paměti RAM považuje za dostatečně otestovaný, pokud je implementován alespoň jeden ze dvou přípustných způsobů: **test statické paměti** nebo **bitová redundance**.

Test statické paměti je nejčastěji implementován jednou z variant **March** algoritmů. Na obrázku 2.10, 2.11 a v následujícím seznamu jsou popsány jednotlivé kroky **MarchX** algoritmu:

1. Záloha testované oblasti.
2. Vynulování testované oblasti.
3. Od nejnižší po nejvyšší adresu (vzestupně) probíhá načtení a kontrola hodnoty 0. Zároveň se zapisuje hodnota 1.
4. Sestupně probíhá načtení a kontrola hodnoty 1. Zároveň se zapisuje hodnota 0.
5. Kontrola, zda je celá testovaná oblast rovna hodnotě 0.
6. Navrácení zálohy do původního stavu.



Obrázek 2.10: První dva kroky MarchX algoritmu (převzato z [9]).



Obrázek 2.11: MarchX algoritmus – krok číslo 3 a 4 (převzato z [9]).

Existují také další varianty March algoritmů – např. **MarchC**.

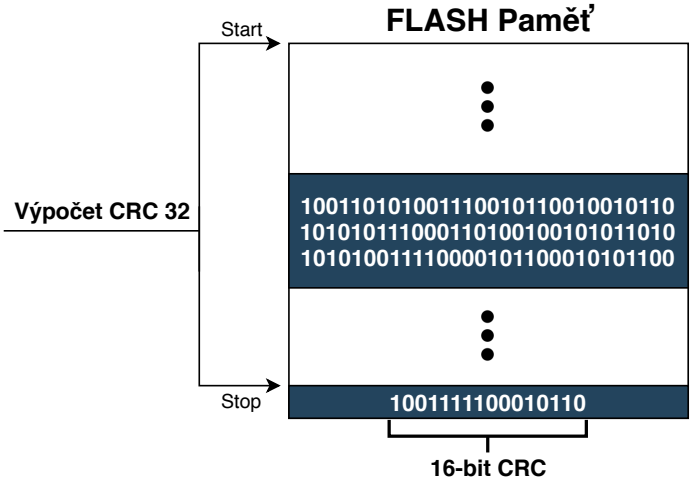
Testovat paměť RAM lze také pomocí **bitové redundance**. Nicméně tento způsob je více paměťově náročný, protože každému slovu v paměti (32 bitů) náleží paritní bit, který určuje buď sudou nebo lichou paritu.

2.4.5 Testování nevariabilní paměti (ROM)

V paměti ROM se nachází kód a konstantní data celé aplikace. Chyba v těchto datech se může objevit například v průběhu přenosu z desktopového počítače do FLASH paměti mikrokontroléru. Cílem testu je odhalit jakoukoliv bitovou chybu ve využité oblasti paměti ROM.

Standard IEC 60730-B definuje pro testování paměti ROM několik variant: **bitová redundance** (stejně jako u RAM) a **cyklický redundantní součet (CRC)**. Poslední jmenovaná varianta je nejčastěji implementována. Navíc existuje mnoho variant výpočtu CRC [21].

Počítání cyklického redundantního součtu probíhá nejprve na platformě, kde se aplikace kompiluje. Po kompilaci se vypočítané hodnota CRC uloží na předem definované místo v paměti ROM. Po nahrání aplikace do ROM paměti mikrokontroléru má aplikace za úkol spočítat CRC hodnotu znovu (ze stejné paměťové oblasti) a vyhodnotit, zda došlo k chybě či nikoliv.



Obrázek 2.12: Testování nevariabilní paměti (ROM).

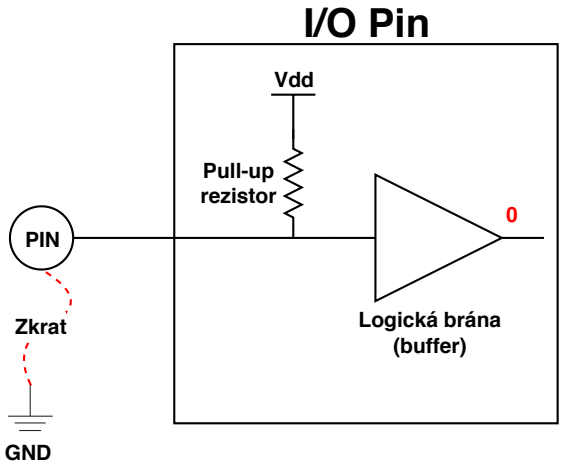
2.4.6 Testování digitálního vstupu/výstupu

Mikrokontroléry dovolují přiřadit vstupním/výstupním pinům různé funkce (GPIO, USB, I2C atd.). Cílem je otestovat, zda na vstupu nebo výstupu není porucha **stuck-at**.

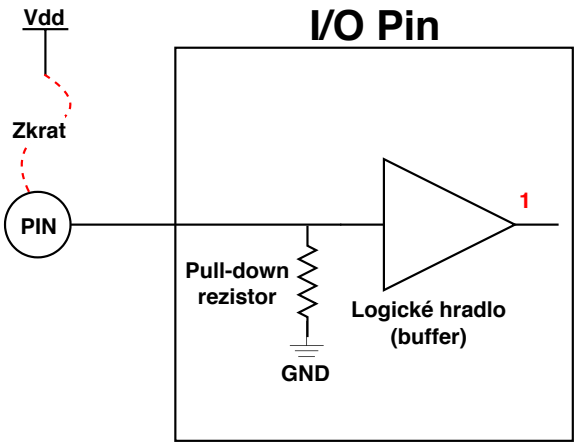
Standard definuje způsob testování jako tzv. **kontrolu důvěryhodnosti**, kdy jsou za běhu programu kontrolovány nepřípustné stavy vstupů a výstupů. Prakticky to například znamená periodickou kontrolu vstupního pinu na jeho očekávanou hodnotu.

Kontrolovat hodnotu na příslušném pinu je typicky možné provádět na dvou různých místech: **na testovaném pinu** nebo **na jiném pinu**. Pokud je prováděna kontrola hodnoty přímo na testovaném pinu, tak se využívá zpětné vazby, která je součástí pinu. Naopak, pokud se kontrola hodnoty provádí na jiném pinu, tak to znamená, že výstupní pin (jehož hodnota je testována) je připojen jako vstup na jiný pin, kde se kontroluje vstupní hodnota.

Existuje také další způsob testování hodnoty na pinu a to pomocí **pull-up/pull-down** rezistorů. Při výrobě totiž může dojít ke zkratu k zemi nebo naopak ke zkratu k napájení. Pokud je tedy testovaný pin nastavený jako **pull-down**, jeho očekávaná hodnota je **logická 0**. Pro pin nastavený jako **pull-up** je naopak očekávaná hodnota **logická 1**. Na následujících obrázcích 2.13 a 2.14 je zobrazeno testování pinu na zkrat k zemi a zkrat k napájení.



Obrázek 2.13: Princip testu digitálního vstupu – zkrat k zemi. Červená hodnota **0** ukazuje, že pokud je pin zkratován k zemi, vstupní hodnota pinu je logická 0 namísto očekávané hodnoty 1.



Obrázek 2.14: Princip testu digitálního vstupu – zkrat k napájení. Červená hodnota **1** ukazuje, že pokud je pin zkratován k napájení, vstupní hodnota pinu je logická 1 namísto očekávané hodnoty 0.

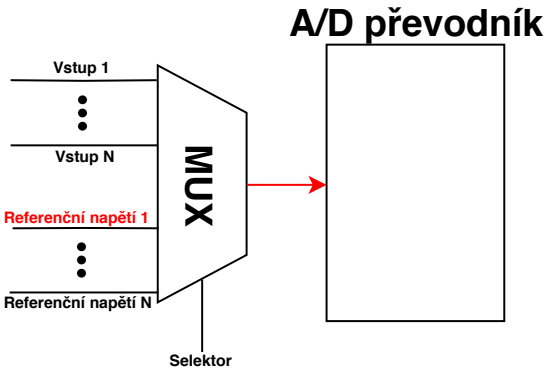
Jelikož je změna hodnoty daného pinu (za účelem testování) v určitých aplikacích nežádoucí, tak je dle standardu IEC 60730-B dostačující pouze kontrolovat hodnotu, která je na pinu očekávaná.

2.4.7 Testování analogového vstupu/výstupu

Analogově digitální převodník (dále A/D) je součástka, která převádí vstupní analogový signál na digitální signál. Používá se např. pro zpracování dat ze senzoru. Podle standardu IEC 60730-B se pro otestování A/D převodníku využívá (stejně jako digitální vstup/výstup) **kontrola důvěryhodnosti**, což znamená kontrolu nepřípustných stavů na vstupním/výstupním pinu. Dále se testuje **multiplexor**, který přepíná vstupní kanály převodníku.

Testování převodníků probíhá následovně:

1. Nastavení vstupního kanálu pomocí multiplexoru. Ideálně využít interní reference (např. 3.3 V, 1 V, 5 V apod.).
2. Spuštění převodu.
3. Po skončení převodu se testuje, zda výstupní hodnota převodníku odpovídá hodnotě očekávané (s určitou tolerancí).



Obrázek 2.15: Test A/D převodníku – přivedení referenčního napětí na vstup.

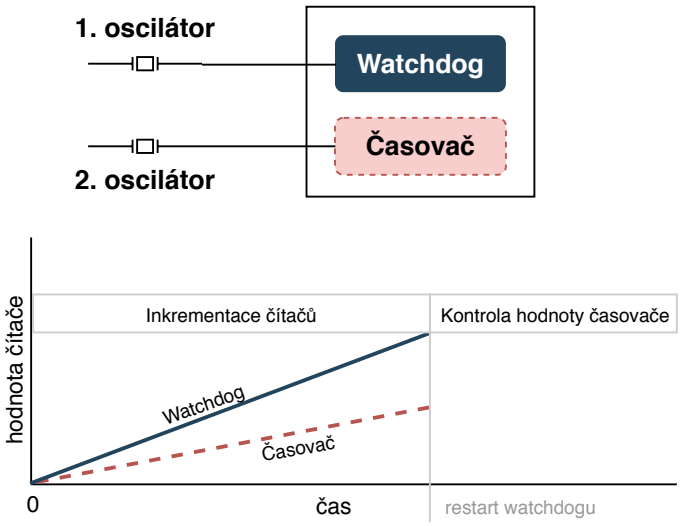
Čím více kanálů se takovým způsobem otestuje, tím větší je pravděpodobnost, že A/D převodník pracuje správně.

Pokud aplikace využívá také digitálně analogový (D/A) převodník, tak je vhodné tyto moduly (A/D a D/A převodník) zakomponovat do jednoho testu. Výstup D/A převodníku se v tomto případě zapojí na vstup A/D převodníku.

2.4.8 Testování watchdogu

Časovač nazývaný watchdog se nejčastěji používá pro detekci uváznutí systému a následné zotavení. Funguje jako časovač, který po překročení určité hranice restartuje zařízení. Proto je jeho funkčnost z pohledu bezpečnosti kritická. Použití watchdog modulu už bylo několikrát zmíněno v předchozích sekcích, a proto je vhodné testovat také samotnou funkčnost watchdogu. Konkrétně to znamená, že watchdog musí restartovat systém pouze tehdy, kdy je to od něj vyžadováno (ne nikdy jindy).

Princip testu je v podstatě stejný jako test hodinového signálu 2.4.3 s tím rozdílem, že se provede pouze jednou a to ihned po restartu zařízení. Na obrázku 2.16 je zobrazena časová posloupnost testu. Ihned po restartu je inicializován watchdog a k tomu navíc časovač, který je řízen **nezávislým zdrojem hodinového signálu**. Poté se simuluje uváznutí pomocí nekonečné smyčky. Jakmile watchdog restartuje zařízení, tak se pouze zkontroluje hodnota referenčního čítače. Pokud se pohybuje v předem stanovených mezích, může se považovat test watchdogu za úspěšný.

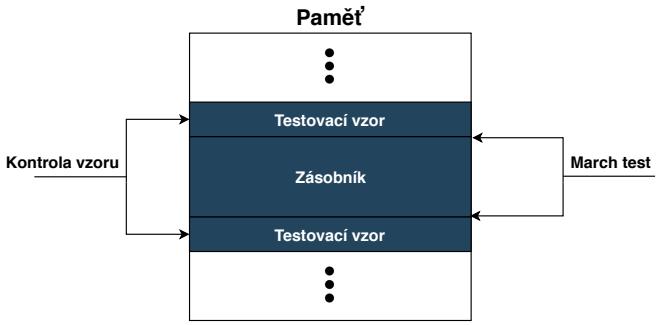


Obrázek 2.16: Princip testu funkčnosti watchdogu.

2.4.9 Testování zásobníku

Test zásobníku není přímo vyžadován normou, ale je doporučené jej testovat také, protože se jedná o jednu z klíčových komponent procesoru. Za běhu programu je neustále používán jak pro lokální proměnné, tak pro ukládání argumentů funkcí při zanořování apod. Cílem testování zásobníku je detekovat dva druhy chybových stavů – **stuck-at** a přetečení/podtečení zásobníku. Stuck-at porucha se může vyskytnout v pracovní oblasti zásobníku. Naopak k přetečení/podtečení dochází mimo tuto oblast. Druhá zmíněná varianta může být zapříčiněna například špatným návrhem aplikace, kdy je vyhrazena pro zásobník malá paměťová oblast, která je nedostatečná, protože aplikace se zanořuje do mnoha funkcí.

Pro detekci stuck-at poruchy se typicky implementuje jedna z variant *March* algoritmů, jehož princip je již popsán v návrhu testu pro Testování variabilní paměti (RAM). Pro vysvětlení principu detekce přetečení/podtečení slouží obrázek 2.17, na kterém je zobrazena pracovní oblast zásobníku. Navíc je v oblasti **nad** a **pod** zásobníkem rezervován prostor (typicky jednotky bajtů), který se při inicializaci naplní vhodně zvolenými vzory. Tyto vzory se v průběhu programu neustále načítají z paměti a pokud dojde k jakékoliv změně hodnoty v této oblasti paměti, došlo s největší pravděpodobností k přetečení nebo podtečení zásobníku.



Obrázek 2.17: Princip testování přetečení/podtečení zásobníku.

2.4.10 Redundantní proměnné

Kromě testování RAM paměti – které zaručí, že hodnota proměnných nebude změněna vlivem poruchy RAM paměti – existují také metody, které také testují, zda nebyla hodnota proměnné změněna, ale na rozdíl od RAM testů, tato metoda testuje vnější vlivy jako např. zápis do proměnné v obsluze přerušení s vyšší prioritou.

Pro detekci takových neočekávaných a nežádoucích jevů jsou využívány 2 následující metody, které se liší pouze ve 3. odrážce. Ty jsou založené na tom, že každá proměnná má svoji bitově inverzní kopii v jiné oblasti paměti.

Kontrola před zápisem

- Redundantní proměnná je definována v paměťovém regionu, který se nachází daleko od umístění originální proměnné.
- Když se do originální proměnné zapisuje, tak je redundantní proměnná aktualizována bitově inverzní hodnotou.
- Když se z originální proměnné čte, tak je porovnána s redundantní proměnnou pomocí operace XOR. Pokud je výsledek 0, vše je v pořádku.
- Poskytuje plnou ochranu proti přepsání zevnějšku, ale za cenu větší výpočetní náročnosti.

Periodická kontrola

- Redundantní proměnná je definována v paměťovém regionu, který se nachází daleko od umístění originální proměnné.
- Když se do originální proměnné zapisuje, tak je redundantní proměnná aktualizována bitově inverzní hodnotou.
- Když se z originální proměnné čte, tak je porovnána s redundantní proměnnou pomocí operace XOR. Pokud je výsledek 0, vše je v pořádku.
- Poskytuje pouze částečnou ochranu proti přepsání zevnějšku, ale s menším dopadem na paměťovou a výpočetní náročnosti.

2.4.11 Redundantní programování

Obecně se jedná o metodu, jejíž cílem je vytvořit určitou úroveň tolerance poruch. Hlavním znakem této metody je, že program, jehož požadavky jsou na začátku vývoje přesně specifikovány, je implementován n-krát. Velice podstatným požadavkem také je, aby implementaci prováděly různé týmy vývojářů, v různých jazycích pomocí různých metod.

Program pak za běhu vykonává všechny verze najednou – buď sériově nebo paralelně. Problém trochu nastává v případě porovnávání výsledků. V systému musí být jasně daný postup vyhodnocování. Tento proces provádí modul nazýván **hlasovač**.

Nevýhodou této metody je jednoznačně vysoká cena takového systému, která rapidně stoupá s počtem redundancí n. Také pochopitelně rostou nároky na výpočetní výkon (sériová i paralelní verze). Problém také nastává při porovnávání výsledků.

Čím více odlišných metod se při vývoji duplikátního programu použije, tím více bude výsledný systém odolnější vůči systematickým chybám v softwaru.

Kapitola 3

Prostředky a návrh řešení

V úvodu této kapitoly jsou vymezeny oblasti mikrokontroléru, které budou testovány implementovanými bezpečnostními testy. Poté následuje popis vybrané hardwarové platformy a vývojového prostředí, které vybranou platformu dostatečně podporuje.

Samotný návrh bezpečnostních testů je rozdělen do jednotlivých podkapitol. Na začátku procesu návrhu je nutné znát požadavky kladené na výsledné řešení. Tyto požadavky jsou jednotlivě popsány u každého testu v sekci Testy zvolených oblastí mikrokontroléru.

Poslední část této kapitoly popisuje návrh demonstrační aplikace.

3.1 Vymezení testovaných oblastí

Účelem bezpečnostního testování je periodicky kontrolovat, zda zdroje a prostředky, které aplikace využívá, pracují tak, jak se očekává — bezporuchově.

Výsledkem této práce je software. Ve výsledném řešení se tedy pro účely testování využívá pouze softwarových prostředků. Neznamená to však, že by nebyly různé hardwarové moduly jako např. CRC, analogově digitální převodník apod., ale znamená to, že veškeré testovací rutiny jsou řízeny softwarově, bez dodatečného hardwaru.

Následující tabulka definuje množinu komponent, které je požadováno dle standardu IEC 60730-B 2.2 testovat. U každé komponenty v seznamu je specifikováno, zda bude testována ve výsledném řešení této práce.

Tabulka 3.1:

Testovaná komponenta	**Součást řešení**
CPU registry	ANO
Programový čítač	ANO
Frekvence přerušení a tok programu	ANO
Hodinová frekvence	ANO
Variabilní paměť (RAM)	ANO
Nevariabilní paměť (ROM)	ANO
Interní komunikační kanály	**NE**
Externí komunikace	**NE**
Digitální vstup/výstup	ANO
Analogový vstup/výstup	ANO, pouze VSTUP

Test **interních komunikačních kanálů** není navržen ani implementován, protože interní komunikační kanály nejsou v řešení této práce využity (komunikace mezi jádry). To samé platí pro test **externí komunikace** – nevyužitá komponenta.

I přes to, že ve standardu IEC 60730, softwarové třídy B, není přímo vyžadováno testování **zásobníku** a **watchdogu**, zahrnul jsem je do návrhu a implementace také, protože jsou za běhu využívány. Zásobník je využíván jádrem např. pro předání argumentů, lokální proměnné apod. Watchdog se používá např. pro testování hodinové frekvence procesoru. Z těchto důvodů je nezbytně nutné kontrolovat tyto moduly také.

3.2 Zvolená hardwarová platforma

Při výběru hardwarové platformy, na které bude řešení implementováno, je nezbytně nutné provést studii proveditelnosti (anglicky *feasibility study*). Jednoduše řečeno – je potřeba zjistit, zda má vybraná platforma dostatečné prostředky (ať už se jedná o software či hardware) na to, aby splnila požadavky dané normou IEC 60730.

Konkrétně se jedná například o následující požadavky:

- **Frekvence procesoru** – frekvence by měla být dostatečná na to, aby procesor stihl za běhu obsloužit jak aplikační část, tak bezpečnostní testy.
- **Minimálně dva nezávislé zdroje hodinového signálu** – více popsáno v návrhu modulu Testování hodinové frekvence.
- **Dostatek paměti ROM a RAM**
- **Hardwarová podpora pro výpočet CRC** – používá se často pro odhalení bitových poruch při testování paměti ROM.
- **Existence modulu watchdogWatchdog – periferie, která restartuje systém při uváznutí.** – z pohledu bezpečnostní normy IEC 60730 jde o nezbytně nutnou součást, která detekuje uváznutí při vykonávání kódu.
- **Podpora ladění**

Jako vhodnou hardwarovou platformu jsem zvolil vývojový kit LCP55S69-EVK od společnosti NXP, který je osazený mikrokontrolérem **LPC55S69**. Jedná se o zařízení se dvěma jádry ARM Cortex-M33. Tento mikrokontrolér je určen pro trh, kde je vyžadována zejména bezpečnost a nízká spotřeba. Frekvence jádra dosahuje až 100 MHz. Více informací o LPC55S69 se nachází v tabulce 3.2.

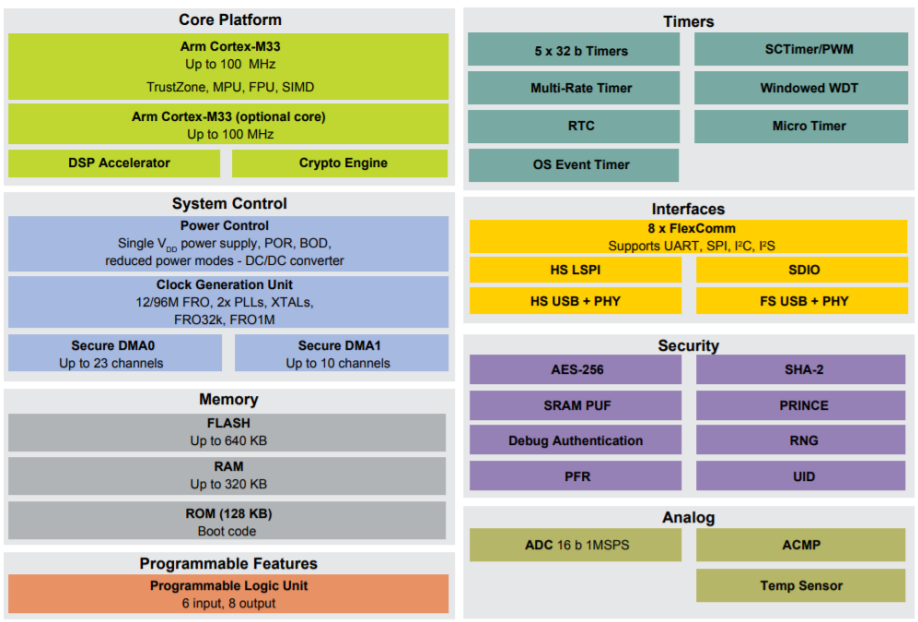
LPC55S69 jsem zvolil hlavně proto, že disponuje prostředky, které umožní a usnadní splnit požadavky kladené normou IEC 60730-B. Jedná se zejména o tyto vlastnosti:

- **Více nezávislých zdrojů hodinového signálu** – frekvence 32 kHz, 1 MHz a 96 MHz.
- **Dostatek paměti ROM a RAM**
- **Hardwarová podpora pro výpočet CRC**
- **Watchdog modul**
- **7 různých časovačů** – SCTIMER, CTIMER, SysTick, RTC, Micro-Tick, Multi-Rate, Watchdog.

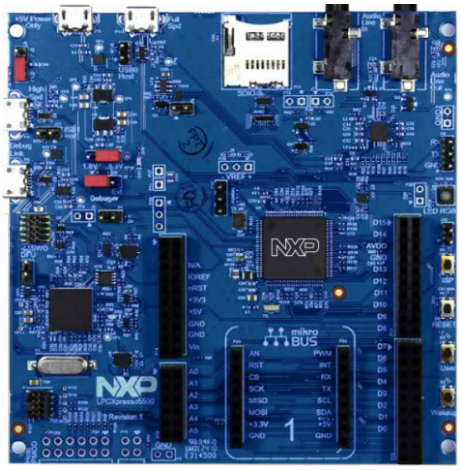
Tabulka 3.2: Přehled vlastností mikrokontroléru LPC55S69 (převzato z [7]).

Jádro	ARM Cortex-M33
Počet jader	2
Architektura	Armv8-M (32-bit)
Maximální frekvence procesoru	100 MHz
Paměť ROM	640 KB
Paměť RAM	320 KB
Počet I/O pinů	64
Minimální napájecí napětí	1.8 V
Maximální napájecí napětí	3.6 V
Sériové rozhraní	USB, UART, I2C, I2S, SPI

Následující diagram 3.1 zobrazuje komponenty zvoleného mikrokontroléru LCP55S69.



Obrázek 3.1: Blokový diagram zachycující komponenty mikrokontroléru LCP55S69 (převzato z [24]).



Obrázek 3.2: Vývojový kit LCP55S69-EVK (převzato z [24]).

3.3 Zvolené vývojové prostředí

Výběr vhodného vývojového prostředí (dále *IDE*[1]) je dle propagačních materiálů ([24]) omezený na 3 možnosti:

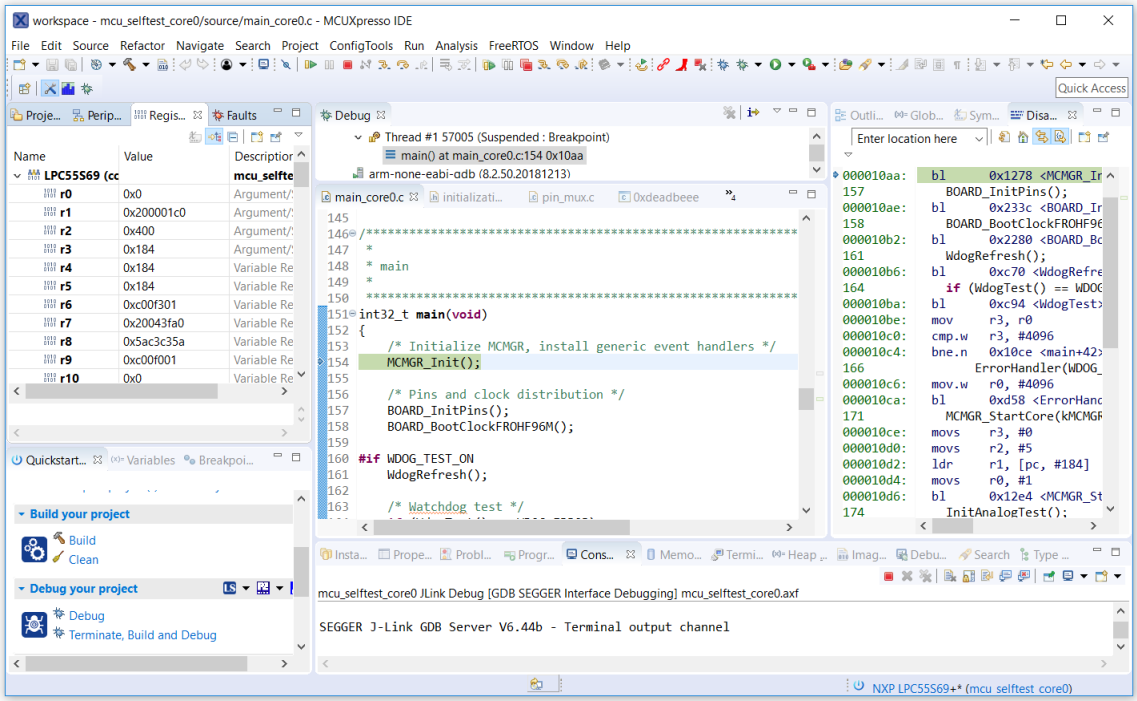
- IAR Embedded Workbench
- μVision IDE
- MCUXpresso IDE

Všechny výše jmenované IDE mají v podstatě identickou funkčnost. Velký rozdíl je však v možnosti bezplatného používání. IAR Embedded Workbench nabízí buď bezplatnou plně funkční 30-denní verzi nebo bezplatnou verzi, která není časově omezená, avšak nepovolí kompilaci aplikace s velikostí vyšší než 32KB. Další možností je tedy μVision IDE, které nabízí bezplatnou edici, avšak se stejným omezením při kompilaci jako u předchozího IDE.

Zvoleným IDE je tedy MCUXpresso [28] od společnosti NXP, které je zdarma ke stažení v plně funkční verzi. Nativně podporuje programování i ladění v jazyce C a v jazyce symbolických adres. Neméně důležitá vlastnost IDE je podpora při fázi linkování výsledné aplikace, popř. *post-build*[2] akce. MCUXpresso IDE používá pro linkování nástroj GNU Linker [8], který v této práci používám zejména pro umístění kódu a dat na specifické místo v paměti mikrokontroléru. Jak je již zmíněno v předchozí sekci, LPC55S69 se skládá ze dvou jader, a proto musí vybrané IDE umět pracovat s více-jádrovými aplikacemi. V MCUXpresso IDE existuje pro každé z jader samostatný projekt, přičemž propojení obou projektů obstarává právě GNU Linker.

Na následujícím obrázku 3.3 lze vidět typické rozložení prvků v MCUXpresso IDE při ladění. V levé části se nachází přehled stavu registrů jádra (lze přepnout např. na stavy registrů periferií). V prostřední části se nachází zdrojový soubor. Zeleně vyznačený řádek ukazuje na příkaz (v jazyce C), na kterém je právě aplikace zastavená. V pravé části je vyobrazeno v podstatě to samé jako v části prostřední, ale s tím rozdílem, že instrukce odpovídají posloupnosti kódu v jazyce symbolických adres.

[1] IDE – Integrated Development Environment (vývojové prostředí).
[2] Post-build akce – akce provedené po sestavení celé aplikace, např. výpočet CRC.



Obrázek 3.3: Vývojové prostředí MCUXpresso IDE v módu ladění.

Většina procesorů/vývojových kitů od společnosti NXP (včetně LPC55S69) je podporována konfiguračními nástroji. Jedná se zejména o nástroje:

- **Clock tool** – konfigurace zdroje hodinových signálů a jejich distribuce v rámci mikrokontroléru. Nastavuje se zde například hodnota děliček, výstup multiplexorů a podobně.
- **Pin tool** – konfigurace vlastností a funkce jednotlivých pinů. Konkrétně se nastavuje např. funkce pinu (GPIO, USB, ADC a další), analogový/digitální mód, směr pinu a podobně.
- **Peripheral tool** – konfigurace jednotlivých periferií na mikrokontroléru. Uživatel by měl být schopen nastavit tímto nástrojem např. časovač podle svých preferencí. Jelikož je tento nástroj složitý na vývoj, podporuje výrazně méně zařízení. Je to dáno složitostí a unikátností některých periferií (k nahlédnutí uživatelský manuál pro LPC55S69 [31]).

Výhoda MCUXpresso IDE je, že tyto výše jmenované konfigurační nástroje jsou do IDE integrovány, což usnadňuje vývoj (uživatel nemusí nástroje instalovat a navíc nemusí exportovat vygenerovaný kód, ale jednoduše aktualizuje kód v otevřeném projektu).

3.4 Návrh řešení

Vzhledem k bezpečnostní normě IEC 60730-B jsem se rozhodl zvolit pro návrh a vývoj metodiku nazývanou V-model, která je zmíněnou normou doporučená. Jelikož jsem jediná osoba, která řeší tuto práci, tak zastávám pochopitelně pozici architekta, vývojáře i testera. Jak je již zmíněno v sekci popisující V-model 2.2.2, zvolená metodika se skládá ze tří

hlavních částí. Každá úroveň specifikace či testování by měla ideálně produkovat samostatné výstupy — dokumenty. Tyto výstupy jsou pro zjednodušení zakomponovány do technické zprávy (vyjma kódu).

V následujícím seznamu jsou popsány fáze V-modelu tak, aby bylo zřejmé, kde přesně se výstupy jednotlivých fází v technické zprávě vyskytují.

1. **Specifikace**
   (a) **Definice požadavků** – tabulka 2.2 převzatá z normy IEC 60730 definuje přijatelná opatření (požadavky) pro otestování konkrétních komponent mikrokontroléru.
   (b) **Návrh systému (architektury)** – sekce Softwarová architektura popisuje návrh architektury implementovaného systému.
   (c) **Detailní specifikace jednotlivých modulů** – za modul se této práci považuje softwarový test. V sekci Testy zvolených oblastí mikrokontroléru lze najít specifikaci každého takového testu v jednotlivých podkapitolách. Součástí této specifikace jsou také požadavky proveditelnosti.
2. **Implementace**
   (a) Implementační část, respektive kód, není součástí technické zprávy. Lze jej však nalézt v příloze A. Nicméně v sekci 4.2 se nacházejí v jednotlivých podkapitolách úryvky kódu, které mají za cíl přiblížit čtenáři stěžejní část z pohledu implementace.
3. **Testování**
   (a) **Testování samostatných modulů** – v sekci Softwarové testy je u každého implementovaného testu popsáno, jakým způsobem je do systému injektována porucha a jak je rozpoznána.
   (b) **Integrační testování a testování kompletního systému** – vzhledem k relativně nízké složitosti řešení, jsou integrační a systémové testování sloučeny do jedné úrovně. Výsledek této závěrečné fáze je shrnut v kapitole Vyhodnocení.

3.4.1 Softwarová architektura

Návrh architektury je z hlediska použitelnosti, bezpečnosti a výkonu důležitá fáze. Rád bych zde na úvod zmínil tři poučky z knihy Chrise Hobbse [10]:

“Použitelnost je o vytváření systému, který je pro uživatele jednoduše použitelný.”
“Zabezpečení je o vytváření systému, který je pro určené lidi jednoduše použitelný, naopak pro všechny ostatní je použitelný velmi těžko (ideálně vůbec).”
“Bezpečnost je o zamezení vykonávání nevhodné činnosti těch pravých lidí. I zkušení někdy udělají chybu; v bezpečném systému tyto chyby nevystupňují v nežádoucí události, poškození či jiné bezpečnostní komplikace.”

Na celé softwarové řešení lze nahlížet z nejvyšší abstraktní úrovně jako na dvě oddělené části – program vykonávaný na mikrokontroléru a uživatelské rozhraní, které je spuštěné na straně PC. Vzhledem k relativně jednoduché struktuře implementovaného řešení, je návrh architektury rozdělený pouze na 3 části, jež jsou specifikovány v podkapitolách níže.

Adresářová hierarchie

Výsledné implementované řešení je rozdělené do následující struktury (platí pro oba dva projekty):

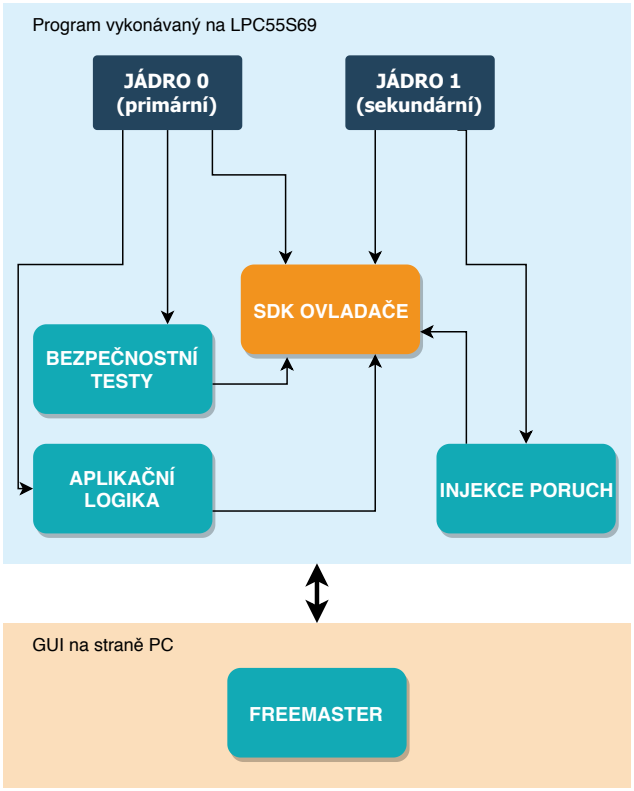
- **CMSIS** – hlavičkové soubory, které obsahují makra a definice usnadňující přístup k řídicím registrům čipu.
- **device** – soubory sloužící k nastavení distribuce hodinových signálů a nastavení funkce jednotlivým pinům.
- **drivers** – SDK[3] ovladače k jednotlivým periferiím.
- **freemaster** – grafické uživatelské rozhraní.
- **linker** – soubory pro GNU Linker. Obsahují definici paměťového prostoru a umístění kódu a dat v tomto prostoru.
- **mcmgr** – MCMGR (Multicore Manager) je knihovna, která obstarává komunikaci mezi jádry. V této práci je konkrétně použitá pro spuštění sekundárního jádra.
- **source** – obsahuje soubory vztahující se k aplikační logice.
  - **source/safety** – implementace bezpečnostních testů. Každému testu odpovídá jeden zdrojový a jeden hlavičkový soubor. Navíc se tam také nachází hlavičkový soubor *safety.h*, který propojuje všechny implementované testy a navíc definuje kódy specifických poruch.
- **startup** – obsahuje jediný zdrojový soubor, který definuje tabulku vektorů a inicializační funkci, která se vykoná ihned po restartu zařízení.
- **tools** – utilita Srecord[4], která je použita pro výpočet CRC.

Závislost komponent

Jak je již zmíněno v úvodu kapitoly Adresářová hierarchie, z nejvyšší abstraktní úrovně se lze dívat na implementovaný systém jako na dvě části. Právě tyto dvě části, jejich logicky oddělené podčásti a vzájemné závislosti mezi nimi jsou vykresleny na diagramu 3.4.

[3] SDK – Software Development Kit. Vývojové nástroje umožňující vytváření aplikací pro určité platformy.

[4] Srecord [23] – utilita, která poskytuje pokročilou manipulaci se soubory, které se přímo nahrávají do paměti ROM. Jde např. o funkce jako spojení/rozdělení souborů, výpočet CRC apod.



Obrázek 3.4: Závislosti jednotlivých komponent.

Jak je na první pohled zřejmé, SDK ovladače jsou komponentou, na které je závislá každá další komponenta. Je to dáno tím, že poskytuje obslužné rutiny ke všem periferiím, či zajišťuje podporu více-jádrovým aplikacím.

Organizace dat

V této sekci je popsána organizace dat pouze v rámci implementovaných testů. Každý test si udržuje svoje proměnné, které jsou za běhu programu aktualizovány. Aby bylo zaručeno, že těmto proměnným může změnit hodnotu pouze daný modul, je k tomuto účelu využito koncepce klíčového slova *static* v programovacím jazyce C. Příklad definice statické proměnné:

```
1    static uint32_t variable = 0;
```

Toto klíčové slovo má více vlastností, ale bezpečnostní testy jej využívají hlavně z důvodu rozsahu viditelnosti proměnné, který je omezen pouze na daný soubor. Proměnné v jednotlivých modulech jsou uloženy v paměti RAM. Naopak veškerá konstantní data jsou uložena v paměti ROM.

3.4.2 Testy zvolených oblastí mikrokontroléru

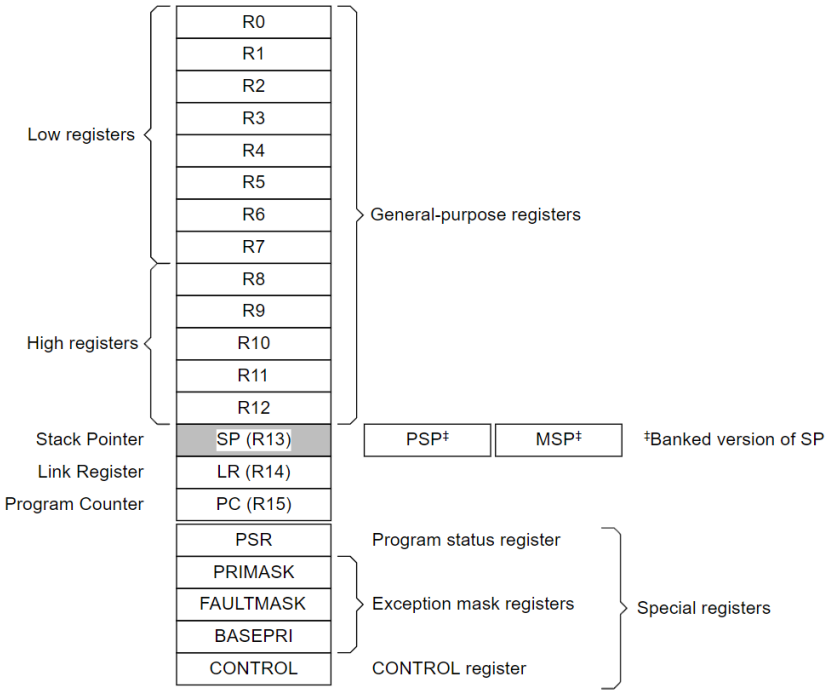
Tato sekce se dělí na podsekce, jež představují jednotlivé testy, které jsou vymezené v sekci 3.1. Každá podkapitola popisuje výběr metody, která bude pro testování dané oblasti implementována. Dále definuje podmínky proveditelnosti.

Pro každý test existuje více způsobů, jak jej implementovat, aby byly splněny požadavky normy IEC 60730-B. Principy se v zásadě neliší, spíše záleží na prostředcích, kterými disponuje hardware a software.

CPU registry

Pro testování všech CPU registrů bude implementována metoda **Checkerboard**. Funkce testovaných registrů jádra ARM Cortex-M33 jsou v odrážkách níže stručně popsána:

- **R0–R12** – univerzální I/O registry.
- **SP (stack pointer)** – ukazatel na vrchol zásobníku. Tento registr ukazuje na *Main Stack Pointer* nebo na *Process Stack Pointer*. Volba závisí na bitu číslo 1 v registru CONTROL.
- **LR (link register)** – uchovává návratovou adresu.
- **PC (program counter)** – uchovává adresu právě vykonávané instrukce navýšenou o 4.
- **Speciální registry** – PSR (program status register), PRIMASK (exception mask), FAULTMASK (fault mask), BASEPRI (base priority mask), CONTROL (control register).



Obrázek 3.5: Registry jádra ARM Cortex-M33 (převzato z [2]).

Test registrů CPU musí být vykonáván v přerušení s nejvyšší prioritou nebo nesmí být za běhu přerušen. Kdyby byl test přerušen například v průběhu testování registru SP, mohl by uživatel neúmyslně používat jiný zásobník, než zamýšlel.

Frekvence přerušení a tok programu

Pro testování frekvence přerušení bude použita globální proměnná, která se v nekonečné smyčce inkrementuje. Kontrola hodnoty bude probíhat jak v přerušení (horní a dolní mez) tak v samotné nekonečné smyčce.

Správný tok programu bude implementován popsanou metodou **CFCSS**. V hlavní smyčce budou rozmístěný 2 uzly. V přerušení od A/D převodníku je uzlů celkem 6.

Testy frekvence přerušení a toku programu nekladou žádné speciální požadavky. Pouze je na zodpovědnosti uživatele správně definovat hranice tolerance (frekvence přerušení) a vhodně rozmístit uzly (kontrola toku).

Hodinová frekvence

V implementovaném řešení bude použita metoda se dvěma časovači. Přičemž referenční časovač je použitý výhradně pro tento test a druhý časovač, který bude test vyhodnocovat, řídí celou aplikaci (generuje trigger signál do A/D převodníku).

Hlavní požadavek proveditelnosti je nutnost mít alespoň 2 nezávislé zdroje hodinového signálu – což mikrokontrolér LPC55S69 má (32 kHz a 12 MHz). Samozřejmě existuje možnost tyto dva různé signály porovnat.

Variabilní paměť (RAM)

Pro tento test bude implementován algoritmus **MarchX**. Součástí implementace bude navíc záloha dat testované oblasti.

Tento test se provádí periodicky po předem určených blocích variabilní paměti. Test vyžaduje, aby běh každého takového cyklu testu nemohl být přerušen. Dále můžou být kladeny požadavky na latenci testu. Ta se dá snížit např. zvětšením bloku testovaného za jednu periodu, zvýšením frekvence vykonávání testu nebo optimalizací (implementace v jazyce symbolických adres).

Nevariabilní paměť (ROM)

Mikrokontrolér LPC55S69 má integrován hardwarový modul pro výpočet CRC součtu. V implementovaném řešení bude tedy využito SDK ovladačů pro práci s tímto modulem.

Test paměti ROM bude přerušitelný a bude se provádět parciálně. Požadavky na latenci testu se dají snížit stejným způsobem jako u paměti RAM.

Digitální vstup/výstup

Implementovány budou oba dva způsoby testování digitálního vstupu/výstupu. Jak detekce proti zkratu, tak redundantní kontrola (1 výstupní pin, 2 kontrolní vstupní piny).

Požadavky testu jsou závislé na způsobu testování, resp. na aplikaci. Minimálním požadavkem je, aby byl uživatel schopný číst hodnotu na vstupním pinu.

Analogový vstup

V implementovaném řešení budou testovány 3 kanály. Referenční napětí 3.3 V, 1V a uživatelský vstup. Tím bude pokryt požadavek na testování multiplexoru A/D převodníku a také mechanismus převodu samotný.

Požadavkem testu je možnost připojit na vstup A/D převodníku více různých referenčních napětí.

Watchdog

Test funkčnosti watchdog modulu bude probíhat pouze jednou a to ihned po restartu. Implementovaný test nejprve simuluje uváznutí systému v nekonečné smyčce. Následně po restartu (vyvolaným watchdogem) zkontroluje, zda watchdog restartoval zařízení po očekávané době.

Požadavkem tohoto testu je, aby byl watchdog modul řízen nezávislým zdrojem hodinového signálu. Pokud se používá watchdog na testování správné frekvence hodinového signálu procesoru, tak musí umět pracovat v režimu **window**. 2.4.3.

Zásobník

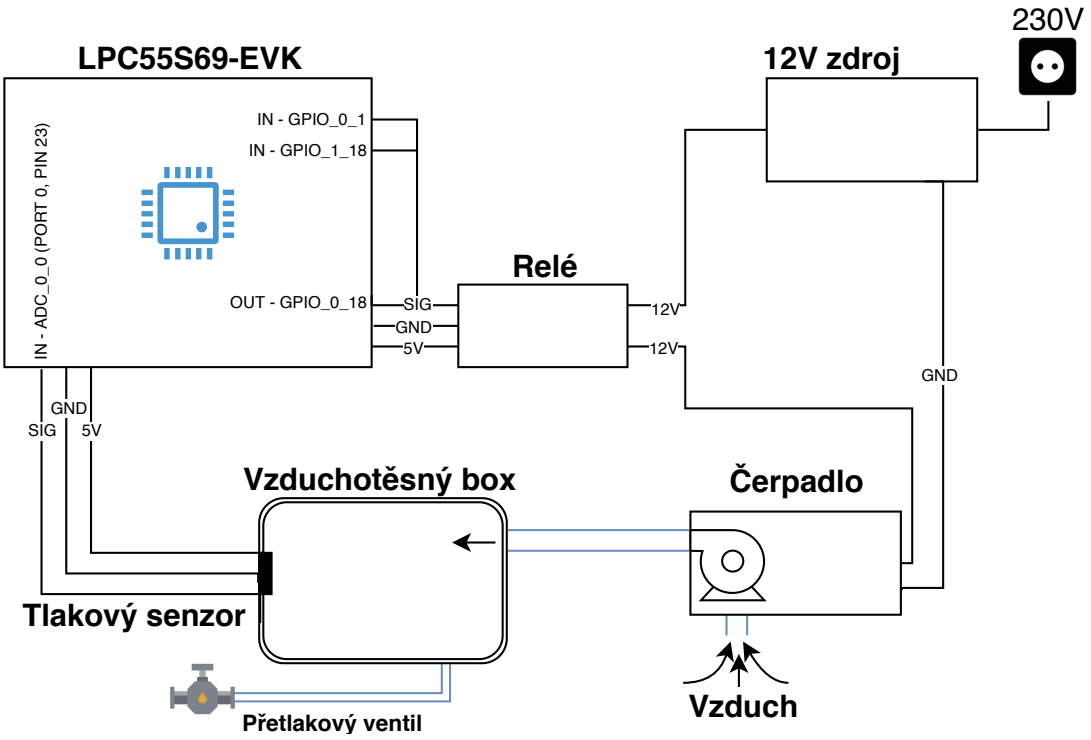
Paměťová oblast vyhrazená pro zásobník bude testována **MarchX** algoritmem. Přetečení a podtečení bude kontrolováno pomocí vyplnění oblasti okolo zásobníku předem známými vzory.

Hlavní požadavek tohoto testu je, že nesmí být přerušen (platí pouze pro detekci stuck-at poruchy). Dalším požadavkem je možnost vyhradit si vlastní paměťový prostor nad a pod zásobníkem (vyžaduje určitou znalost práce s linkerem).

3.4.3 Demonstrační aplikace

Účelem demonstrační aplikace je ukázat, jak lze integrovat bezpečnostní testy do reálné aplikace, kdy mikrokontrolér vykonává na pozadí bezpečnostní testy a zároveň musí obsloužit samotnou aplikační logiku s reakcí na okolní podněty.

Návrh demonstrační aplikace z pohledu propojení jednotlivých hardwarových komponent je popsán na následujícím diagramu 3.6.



Obrázek 3.6: Diagram zobrazující komponenty navrženého systému a jejich vzájemné propojení. Modré spoje reprezentují trubičky, které vedou vzduch.

Jedná se o jednoduchý systém regulace tlaku ve vzduchotěsné nádobě. Mikrokontrolér snímá (pomocí A/D převodníku) data z tlakového senzoru, který je umístěný v nádobě. Na základě toho buď zapíná/vypíná vzduchové čerpadlo (pomocí elektromagnetického relé). Pokud by došlo k poruše, v jejíž důsledku by bylo čerpadlo neustále zapnuté, tak by se otevřel přetlakový ventil a tím pádem by nedošlo ke zničení nádoby/čerpadla.

Součástí demonstrační aplikace je také (nedestruktivní) injekce poruch za běhu. Při situaci, kdy mikrokontrolér nějakou poruchu detekuje, musí provést odpovídající opatření. Nejnebezpečnější element (z pohledu ublížení na zdraví) v celém systému je vzduchotěsná nádoba, která je natlakovaná – z toho vyplývá, že první úkon, který je nutné vykonat po detekci poruchy, je vypnout čerpadlo.

Konkrétní implementace bezpečnostní funkce a celé aplikační logiky je popsána v implementační sekci 4.3.

Kapitola 4

Implementace

Implementace je stěžejní částí celé práce. Na jejím úvodu je popsáno, jak probíhá inicializační fáze programu a co se vykonává za běhu. Hlavní náplní této kapitoly je detailní popis jednotlivých bezpečnostních testů, které vychází ze svého návrhu 3.

V podkapitolách se vyskytují úryvky kódu, které zachycují hlavní podstatu daného testu. Součástí implementace je také injekce poruch (za běhu aplikace), které musí bezpečnostní testy rozpoznat. Tato kapitola rozebírá jednak způsob, jakým jsou poruchy injektovány, ale také obsahuje naměřené časy trvání jednotlivých testů. Navíc maximální možnou dobu, po jejíž uběhnutí aplikace zaručeně rozpozná danou poruchu.

Veškerý kód, na který se v této práci odkazuji, je dostupný na paměťovém médiu A a sám jsem jej implementoval. Výjimkou jsou pouze hlavičkové soubory CMSIS [3] a SDK ovladače [29], které poskytuje výrobce mikrokontroléru LPC55S69.

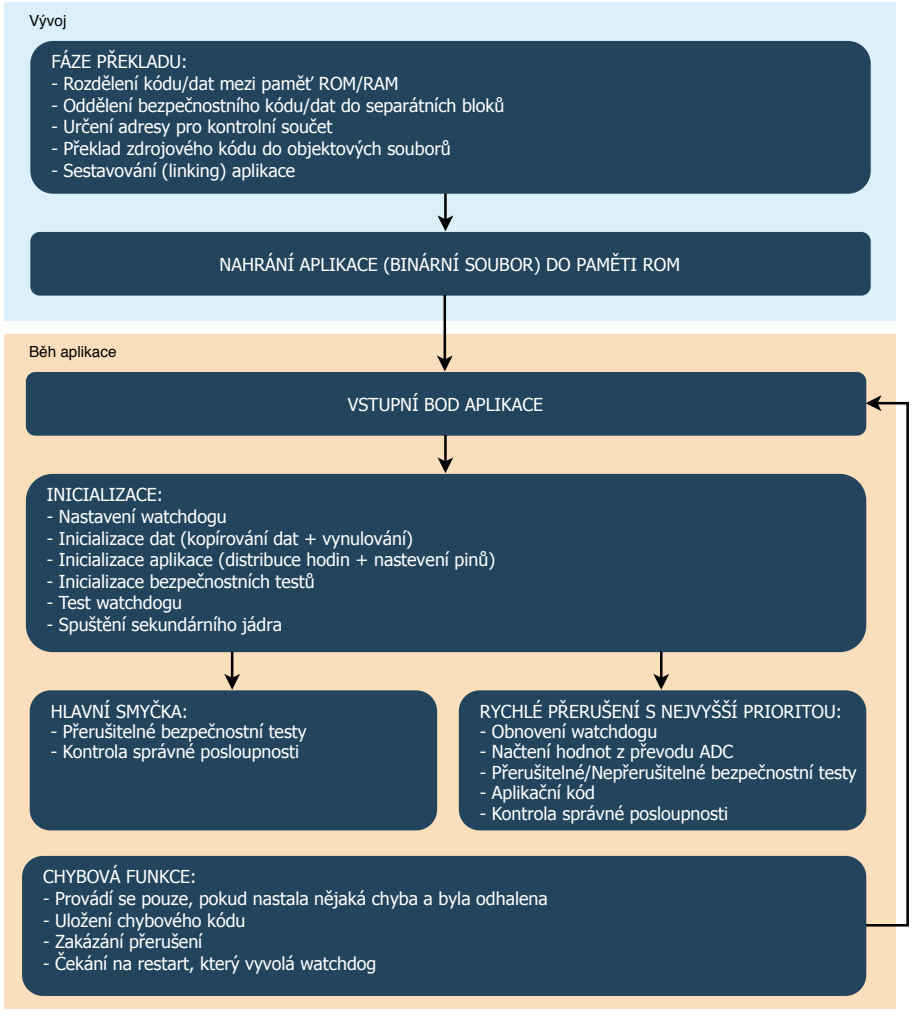
API všech implementovaných bezpečnostních funkcí je specifikováno v příloze B.

4.1 Inicializační fáze a běh programu

Typická architektura software, který vykonává vestavěné zařízení, se skládá ze tří stěžejních bloků kódu:

- **Inicializační fáze** – vykonává se pouze jednou (po restartu zařízení) a má za úkol správně nastavit hardware a inicializovat data.
- **Nekonečná smyčka** – úsek kódu, který se vykonává neustále dokola. Cílem je vykonávat činnost, pro kterou je aplikace a zařízení určené.
- **Přerušení** – úseky kódu, které se vykonávají v reakci na určité události nebo se vykonávají s přesně určenou periodou. Přerušení má vyšší prioritu, než nekonečná smyčka, tudíž je vykonáváno přednostně.

Diagram 4.1 popisuje činnosti, které se v jednotlivých fázích (popsaných výše) provádějí. Součástí diagramu je také zachycení přechodu z vývojové části. Je nutné podotknout, že se jedná o posloupnosti vykonávané na primárním jádru. Sekundární jádro má tuto posloupnost velice primitivní, jelikož pouze injektuje chyby.



Obrázek 4.1: Diagram zobrazující posloupnost kroků, které se provádí za běhu aplikace na primárním jádru.

V pravé části diagramu se nachází **rychlé přerušení s nejvyšší prioritou**. V implementovaném řešení se toto přerušení vykonává pokaždé, kdykoliv A/D převodník dokončí převod signálu z tlakového senzoru. Každý takový převod analogového signálu na digitální musí být iniciován triggerem[1]. A právě takový signál generuje periodicky každých 10 ms časovač **CTIMER**.

Veškerý kód, který implementuje inicializační fázi a tvoří hlavní kostru aplikace je dostupný v příloze A. Konkrétně se jedná o soubory *main_core0.c, main_core1.c*, které tvoří hlavní kostru aplikace. Dále *initialization.c*, ve kterém je implementována inicializace časovače generující trigger. V souborech *startup_lpc55s69_cm33_core0.c a startup_lpc55s69_cm33_core1.c* se nachází kód vykonávaný bezprostředně po restartu zařízení.

Kód inicializace časovače, který generuje trigger pro A/D převodník, je v následujícím bloku. Nastavuje se v něm vstupní vstupní hodinový signál, hodnota děličky hodinového signálu a také porovnávací hodnota čítače (při které se vygeneruje trigger).

[1]Trigger – signál, který funguje jako spouštěč určitého procesu. Zdrojem triggeru může být jak hardware, tak i software.

```
void InitCTIMER1(void)
{
  /* Use 96 MHz clock for CTIMER1 */
  CLOCK_AttachClk(kFRO_HF_to_CTIMER1);

  ctimer_config_t config;
  config.mode = kCTIMER_TimerMode;
  config.prescale = 95U;
  CTIMER_Init(CTIMER1, &config);

  ctimer_match_config_t matchConfig;
  matchConfig.enableCounterReset = true;
  matchConfig.enableCounterStop = false;
  matchConfig.matchValue = 5000U;
  matchConfig.outControl = kCTIMER_Output_Toggle;
  matchConfig.outPinInitState = true;
  matchConfig.enableInterrupt = false;

  CTIMER_SetupMatch(CTIMER1, kCTIMER_Match_3, &matchConfig);
  CTIMER_StartTimer(CTIMER1);
}
```

4.2 Softwarové testy

Vůbec nejdůležitější část celé práce je popsána právě v této sekci. Každá podkapitola odpovídá testované oblasti, které jsou vymezené v tabulce 3.1. U těchto jednotlivých oblastí je přiložen vývojový diagram ukazující posloupnost vykonávání testu. U popisu implementace samozřejmě nechybí úryvek kódu, zachycující klíčové části a také časové vlastnosti kódu (jak dlouho se daná funkce vykonává či za jaký nejhorší čas rozpozná poruchu). Vzhledem k rozsahu některých funkcí jsou však úryvky kódu buď zjednodušené (zkrácené) nebo se jedná o pseudokód.

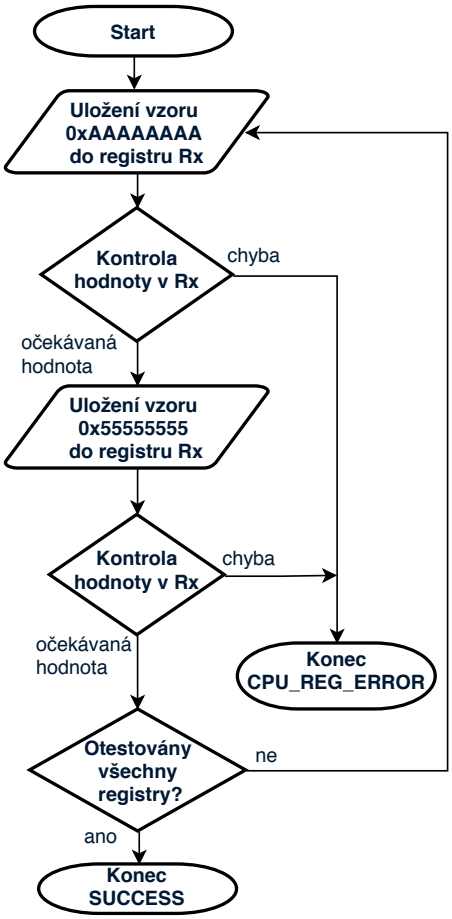
Jak již bylo zmíněno dříve – součástí implementovaného řešení je také injekce poruch, které by měly být rozpoznány implementovanými testy. Popis způsobu injekce do každého z implementovaných testů je v následujících sekcích popsán také.

4.2.1 CPU registry

Test CPU registrů je implementován algoritmem **Checkerboard**, popsaným v návrhu 2.4.1. Kód je dostupný v souboru *cpu_reg.S*.

Samotný test vykonává funkce **RegisterTest()**, která nesmí být za běhu přerušena. Funkce testuje všechny registry jádra kromě registrů FAULTMASK, PRIMASK, BASEPRI a CONTROL.

Následující vývojový diagram 4.2 zobrazuje posloupnost testování registrů CPU:



Obrázek 4.2: Test CPU registrů – vývojový diagram. Slovo *Kontrola* v diagramu znamená načtení hodnoty z registru Rx a následné porovnání s očekávaným vzorem. Registr **Rx** představuje registry R0–R12, SP, LR a PSR.

Délka vykonání celé funkce **RegisterTest()** zabere cca 4.6 μs. Pro představu je zde přiložen krátký úryvek kódu, který testuje registr R4. Posloupnost odpovídá vývojovému diagramu.

```
1  /* R4 */
2  LDR R4, =0xAAAAAAAA
3  CMP R4, R0
4  BNE ERROR_LABEL
5  LDR R4, =0x55555555
6  CMP R4, R1
7  BNE ERROR_LABEL
```

Součástí implementované funkce je také testování programového čítače (dále jen PC). Testování této komponenty je hodně specifické, protože registr PC s každou vykonanou instrukcí mění svoji hodnotu. Každopádně nelze aplikovat stejný princip testu jako na předchozí registry, protože hodnota, která je do PC registru nahrána, se v dalším taktu procesoru použije jako adresa vykonávané instrukce. Princip testu je tedy následující:

1. Na uživatelem zvolenou adresu v RAM paměti se uloží dvě předem známé instrukce: první inkrementuje hodnotu v registru R0, druhá přesune hodnotu registru R5 do PC.

2. Registr R0 se vynuluje a do registru R5 se uloží adresa PC navýšená o 2.

3. Ihned po 2. kroku se nahraje do registru PC uživatelem zvolená adresa z 1. kroku.

4. Vykonávání se přesune na testovanou adresu, kde se vykonají dvě předem známé instrukce (vložené v 1. kroku). Inkrementuje se tedy registr R0 a potom se do PC registru nahraje hodnota R5.

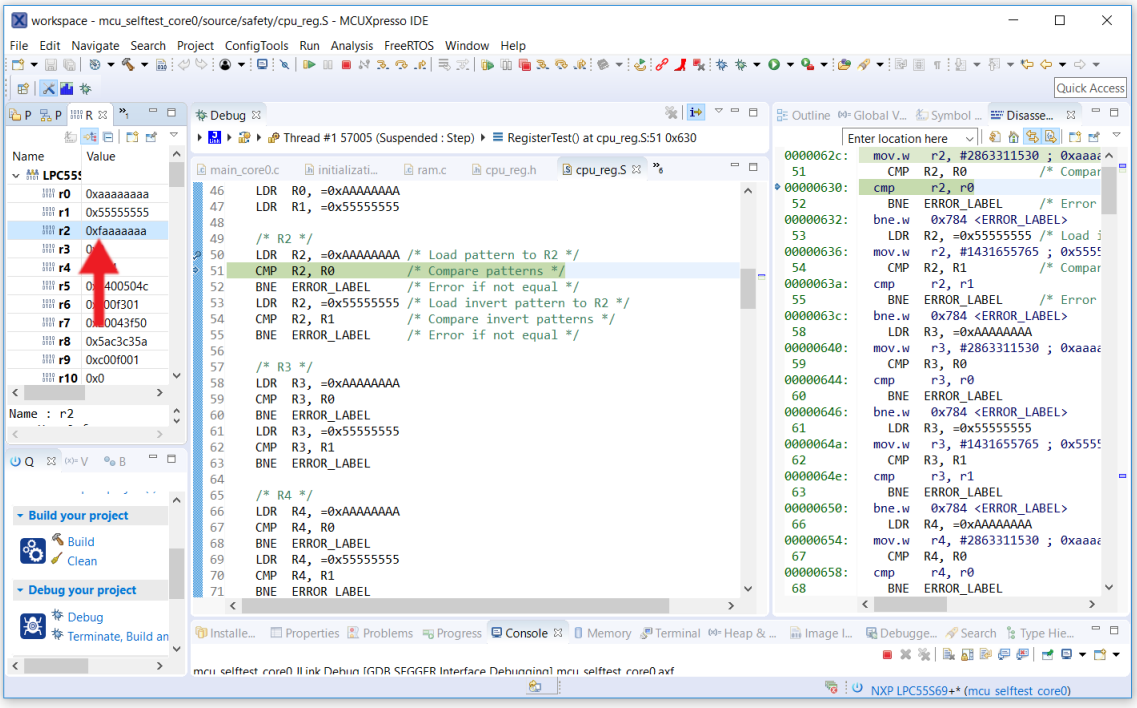
5. Vykonávání se vrací zpět do bezpečnostního testu, kde se pouze zkontroluje, zda se hodnota v R0 rovná hodnotě 1.

Tímto způsobem lze tedy PC registr otestovat. Čím více testovacích adres je použito, tím důkladněji je registr prověřen. V implementovaném řešení jsou použity 2 testovací adresy: **0x2003FFFC** a **0x20000000**.

Injekce poruchy

Injektovat poruchu (nedestruktivně) do registrů CPU za běhu, je velmi obtížné, protože by se injekce musela trefit přesně mezi dvě instrukce, které jsou vykonávány bezprostředně za sebou. Proto jsem v tomto případě zvolil injekci poruchy v módu ladění, což lze vidět na obrázku 4.3.

V nejhorším případě tento test odhalí poruchu v registrech CPU za 10 ms. Je to totiž délka periody, ve které se test vykonává. Pokud je požadavek mít latenci co nejnižší, stačí délku periody zkrátit.

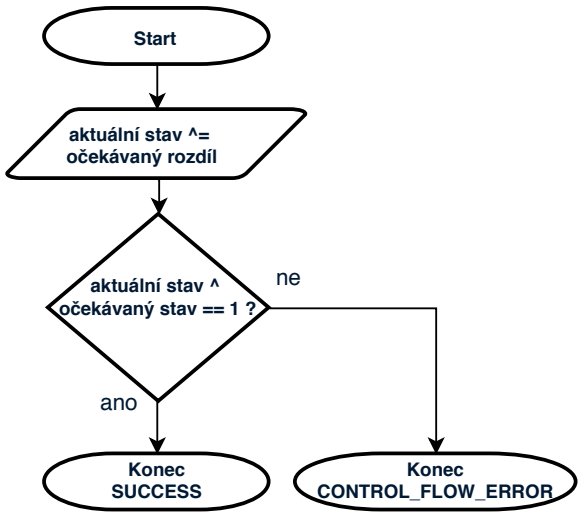


Obrázek 4.3: Injekce chyby – CPU registry. Kód je zastavený na testování registru R2, konkrétně na instrukci, která vykoná porovnání vzoru s hodnotou v R2. Červená šipka ukazuje na hodnotu registru R2, která je pozměněna na 0x**f**aaaaaaa, v důsledku čehož test detekuje chybu.

4.2.2 Frekvence přerušení a tok programu

Pro detekci poruchy v řízení toku programu je použita metoda **CFCSS** a pro kontrolu frekvence přerušení se používají dvě různá počítadla. Implementace testu frekvence přerušení a toku programu je dostupná v souboru *control_flow.c*.

Následující diagram 4.4 popisuje posloupnost testu kontroly toku programu.



Obrázek 4.4: Test toku programu – vývojový diagram. Znak ^ značí logickou operaci XOR.

Tento test vykonává funkce **ControlFlowTest()**, jejíž délka vykonání se pohybuje okolo 2 μs. Využívá se jednoduchého principu logické operace **XOR**. Funkce je volána na místech, která uživatel zvolí za vhodná. Tyto místa se nazývají **uzly**. Při vykonávání testu na daném uzlu se nejprve provede operace XOR nad aktuálním uzlem a očekávaným rozdílem. Po této operaci by se už hodnota aktuálního uzlu měla rovnat hodnotě uzlu následujícího. Pokud hodnoty nejsou stejné (což ověří opět operace XOR), pravděpodobně došlo k přeskočení některého z uzlů. Popsaný princip demonstruje pseudokód v této sekci.

```
#define SIGN_1 0x2 // 0b10
#define SIGN_2 0x1 // 0b01
#define DIFF_1_2 0x3 // 0b11

currentNode = SIGN_1;
currentNode ^= DIFF_1_2;
if (currentNode ^ SIGN_2)
{
  return CONTROL_FLOW_ERROR;
}
```

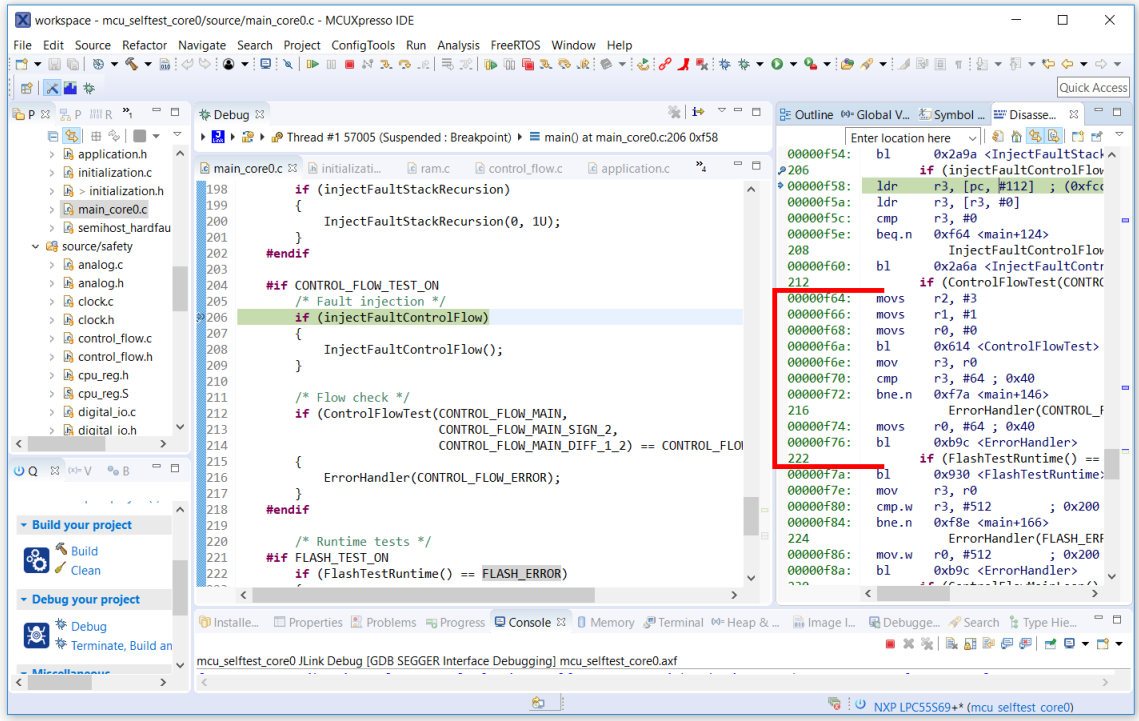
Testování frekvence přerušení je implementováno pomocí globální proměnné, která je v průběhu aplikace inkrementována v hlavní nekonečné smyčce. Čítač se inkrementuje při každém vykonávání nekonečné smyčky. Kontrola je jednoduchá – v obslužné rutině přerušení od A/D převodníku (perioda 10 ms) se kontroluje, zda hodnota čítače splňuje horní/dolní limit a následně se vynuluje. Navíc se v nekonečné smyčce kontroluje, zda čítač nepřekročil horní limit – čímž se detekuje nevykonávané přerušení. Popsanou funkcionalitu zajišťují funkce **ControlFlowMainLoop()** a **ControlFlowADC()**, které trvají přibližně 1 μs.

Injekce poruchy

Způsobů, jak injektovat poruchu vykonávání programu či frekvence přerušení je několik. Za zmínění stojí např. tyto:

1. **Vložení zpoždění** na jakékoliv místo v programu. Pozor, tuto potenciální poruchu frekvence přerušení může dříve detekovat watchdog.
2. **Inkrementovat PC**. Tato metoda přičte k registru PC konstantní číslo, což má za následek přeskočení jednoho kontrolního uzlu. Injekce poruchy touto cestou je implementována a popsána dále v této sekci.
3. Jelikož je kontrola toku vložena do všech bezpečnostních test stačí jakýkoliv test vypnout/zakomentovat a test tuto poruchu detekuje (protože vynechá jednu XOR operaci).

Následující obrázek 4.5 popisuje simulaci poruchy toku programu.



Obrázek 4.5: Control flow test – Červeně vyznačená oblast představuje skok do funkce, která kontroluje správný tok programu (provádí XOR uzlů). Tento skok do funkce včetně předání argumentů se při injekci chyby přeskakuje.

Kód funkce **InjectFaultControlFlow()**, která implementuje injekci výše popsané poruchy, lze vidět v krátkém úryvku kódu. Funkce je napsána v jazyku C, ale využívá instrukce jazyka symbolických adres.

```
1   void InjectFaultControlFlow(void)
2   {
3     /* Disable interrupts */
4     __asm("CPSID i");
5
6     /* Increment PC (skip following control flow check) */
7     __asm("MOV R8, LR");
8     __asm("ADD R8, #0x16");
9     __asm("MOV PC, R8");
10  }
```

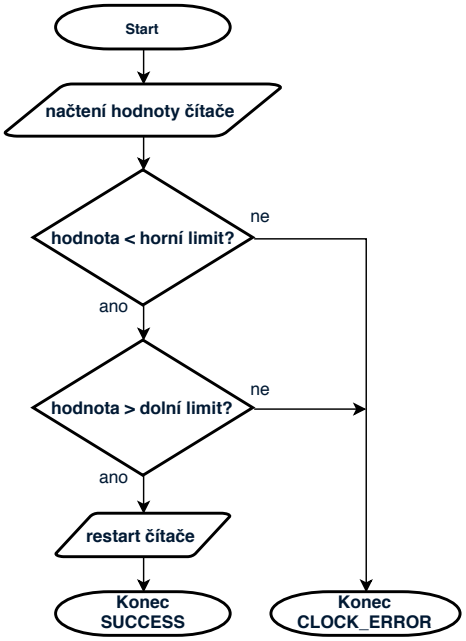
Latenci těchto testů nelze přesně určit, protože záleží na uživateli, kde přesně v kódu jednotlivé uzly rozmístí, a jak dlouhá je perioda jednotlivých přerušení. Prakticky se jedná o jednotky milisekund.

4.2.3 Hodinová frekvence

Test hodinové frekvence je dostupný v souboru *clock.c*. Vykonává jej funkce **ClockTest()**.

Test ke svému chodu vyžaduje jeden systémový prostředek – referenční časovač s nezávislým zdrojem hodinového signálu. V implementovaném řešení je k tomuto účelu inicializován **CTIMER** modul, jež je nastaven na zdroj hodinového signálu o frekvenci 32 kHz. Doba vykonání funkce **ClockTest()** se pohybuje okolo 350 μs.

Vývojový diagram testu:



Obrázek 4.6: Test hodinové frekvence – vývojový diagram.

Očekávaná hodnota referenčního časovače se vypočítá následovně:

Obrázek 4.7: Výpočet očekávané hodnoty referenčního časovače.

$$f_{in} * t = EXPECTED$$

$$32768 * 0.01 = 327.68$$

f_{in}	frekvence hodinového signálu časovače [Hz]
t	perioda testování hodnoty referenčního časovače [s]
EXPECTED	očekávaná hodnota

Úryvek kódu z funkce **ClockTest()** se nachází v následujícím bloku. Posloupnost přesně odpovídá předchozímu vývojovému diagramu.

Na mikrokontroléru LPC55S69 se test hodinové frekvence vyvíjí poměrně obtížně, protože časovače na tomto zařízení se při ladění nezastaví. Naštěstí existuje nástroj FreeMaster [27], který umožňuje sledovat stavy proměnných a registrů za běhu.

Injekce poruchy

Implementovány jsou dvě metody injekce poruchy:

```
    /* Read counter value */
    counterValue = CTIMER0->TC;

    /* Check limits */
    if ((counterValue > HIGH_LIMIT) || (counterValue < LOW_LIMIT))
    {
      return CLOCK_ERROR;
    }

    /* Reset CTIMER */
    CTIMER0->TCR |= CTIMER_TCR_CRST_MASK;
    CTIMER0->TCR &= ~CTIMER_TCR_CRST_MASK;
```

1. Změna zdroje hodinového signálu referenčního časovače za běhu. Původní frekvence 32 kHz je nahrazena frekvencí 1 MHz. K injekci je použita funkce SDK ovladače:

```
    CLOCK_AttachClk(kFRO1M_to_CTIMER0);
```

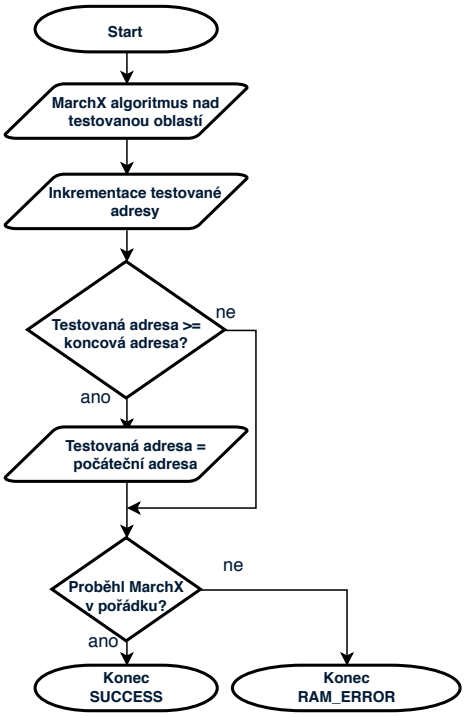
2. Změna komparační hodnoty časovače, který generuje trigger pro A/D převodník. V důsledku zpožděného přerušení od A/D převodníku se pochopitelně opozdí také kontrola hodnoty referenčního časovače.

V případě implementované aplikace je latence testu cca 10 ms, což opět znamená, že latence je závislá na periodě vykonávání testu. Tato latence však platí pro případ, že hodinový signál bude vychýlený maximálně do 20 %. Pokud bude frekvence výrazně vyšší, tak tuto poruchu detekuje test frekvence přerušení (latence $<$ 10 ms), nebo naopak – pokud bude frekvence testovaného hodinového signálu výrazně nižší, tak tuto poruchu detekuje watchdog nebo znovu test frekvence přerušení (latence $<$ 10 ms).

4.2.4 Variabilní paměť (RAM)

Test paměti RAM je implementován v souboru *ram.c* a *ram_asm.S*. Oproti předchozím testům je tento test rozdílný v tom, že je tzv. **parciální**. Je to dáno tím, že RAM paměť má určitý rozsah, který není možné (softwarově) otestovat za kratší časový úsek. Proto se paměť RAM testuje po blocích, jejichž velikost je předem definována.

V souboru *ram.c* je funkce **RAMTest()**, jejíž doba vykonání zabere cca 3.8 μs. Tato funkce zajišťuje správné (parciální) volání funkce **RAMMarchX()** ze souboru *ram_asm.S*, jež provádí samotné testování paměti. Tato funkce je implementována v jazyce symbolických instrukcí. Velikost testovaného bloku je nastavena na 16 B. Vývojový diagram 4.8 zobrazuje posloupnost příkazů ve funkci **RAMTest()**.

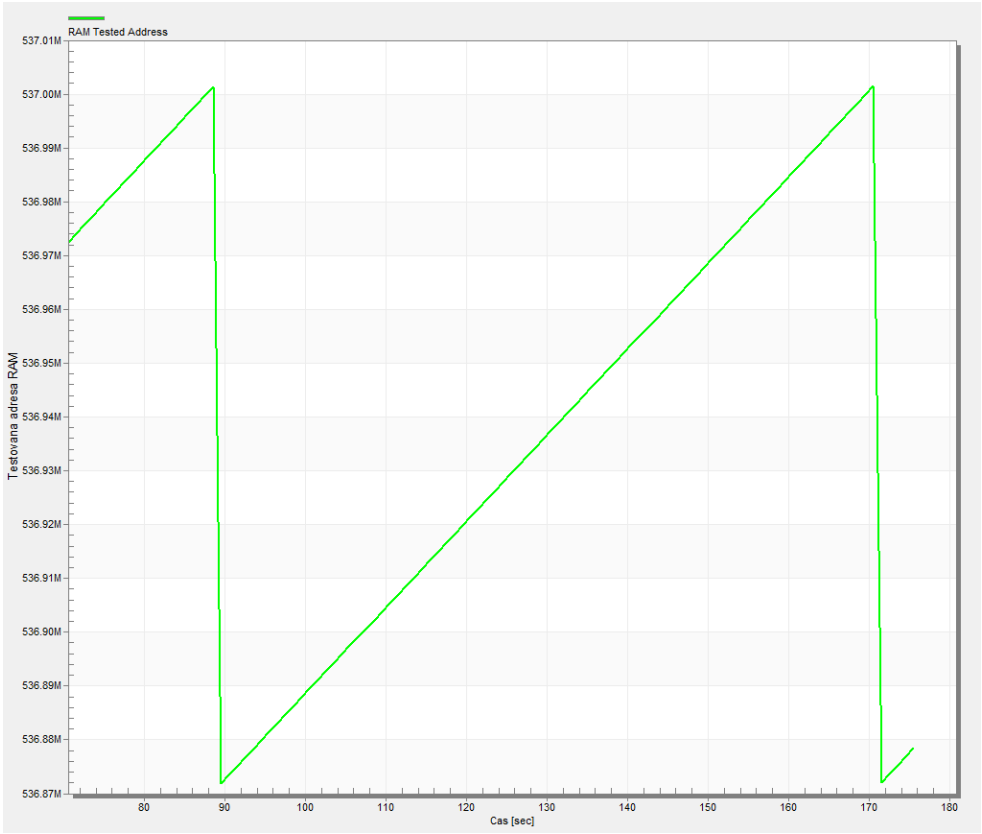


Obrázek 4.8: Test variabilní paměti RAM – vývojový diagram.

Úryvek kódu z funkce **RAMMarchX()**, konkrétně se jedná o 3. krok z popisovaného testu na obrázku 2.11.

```
1   /* STEP 3: Read 1st pattern, write 2nd pattern - ascending */
2   LDR R0, =#0x20000000 /* start address */
3   LDR R3, =#0 /* counter */
4   LDR R4, =#0x55555555U /* 1st pattern */
5
6   RAM_READ_1P_WRITE_2P_ASC:
7   LDR R6, [R0, R3] /* read 1st pattern */
8   CMP R6, R4 /* check if 1st pattern */
9   BNE ERROR_LABEL
10  STR R5, [R0, R3] /* write 2nd pattern */
11  ADDS R3, #4
12  CMP R2, R3
13  BHI RAM_READ_1P_WRITE_2P_ASC
```

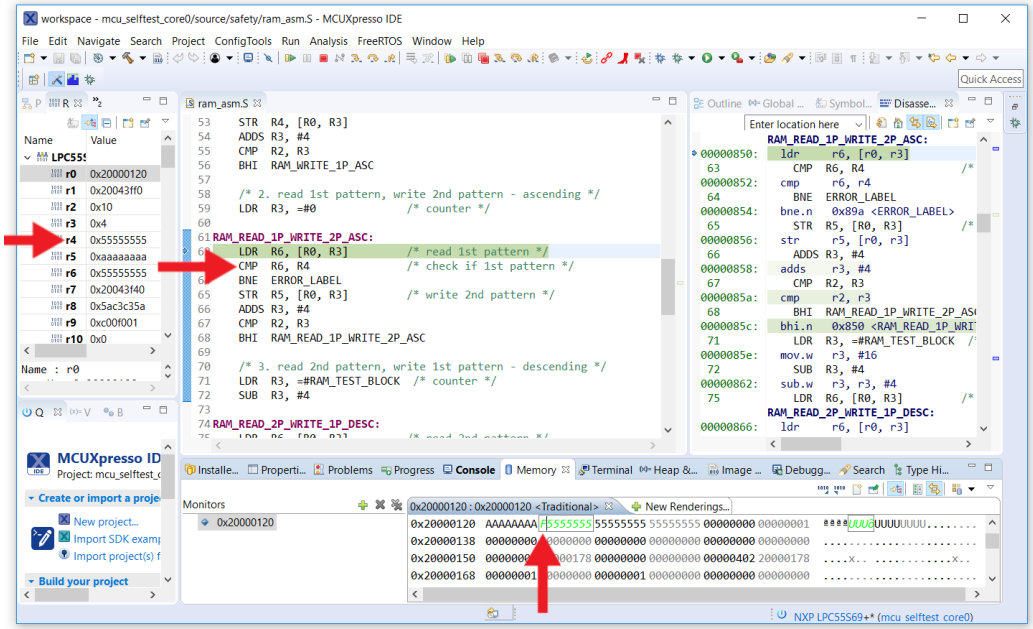
Snímek 4.9 z měření doby vykonání testu paměti RAM přes celou vymezenou paměťovou oblast:



Obrázek 4.9: Testování paměti RAM. Celá testovaná oblast paměti RAM (rozsah 0x20000000-0x20001FFFF) je otestována za cca 80 s.

Injekce poruchy

Podobně jako u testování CPU registrů – jsem pro injekci poruchy zvolil ladící mód vývojového prostředí. Princip je stejný, v průběhu testu je program pozastaven a hodnota na testované adrese je uživatelem pozměněna. Test po spuštění ihned poruchu paměti detekuje. Více na obrázku 4.10.



Obrázek 4.10: Injekce chyby – testování paměti RAM. Obrázek zachycuje vývojové prostředí MCUXpresso IDE v módu ladění. Vykonávání kódu je zastaveno v průběhu MarchX testu. Spodní červená šipka ukazuje na místo v paměti, které je aktuálně testováno, ale hodnota na této adrese byla uživatelem záměrně změněna. MarchX algoritmus je ve fázi čtení 1. vzoru. Ve dvou následujících instrukcích detekuje test (simulovanou) poruchu paměti.

Dle snímku z měření 4.9 je v nejhorším případě latence testu **80** s. Je to opravdu dlouhá doba, ale je to dáno rozsahem testované paměti (0x20000000–0x20020000) a také tím, že velikost testovaného bloku paměti je pouhých 16 B. Svoji roli hraje také délka trvání periody – 10 ms. Latenci testu lze vypočítat podle následující rovnice.

Obrázek 4.11: Výpočet latence testu paměti RAM.

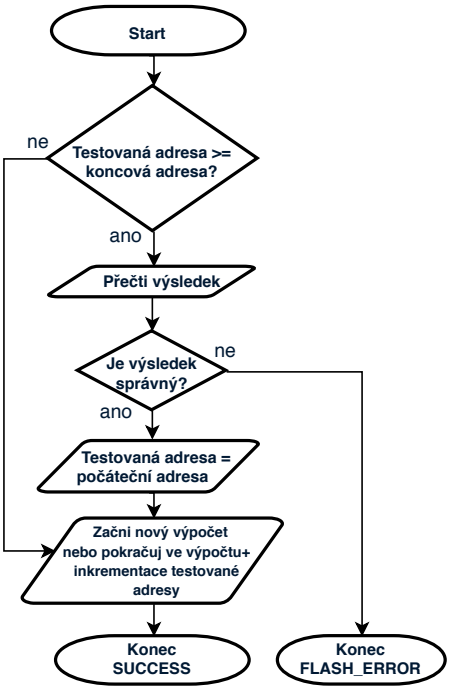
$$\frac{r}{s} * t = LATENCY$$

$$\frac{131'072}{16} * 0.01 = 81.92$$

r	rozsah testované oblasti [B]
s	velikost testovaného bloku paměti [B]
t	perioda testování paměti RAM [s]
LATENCY	latence testu

4.2.5 Nevariabilní paměť (ROM)

Implementace testu se nachází v souboru *flash.c*. Test vykonává funkce **FlashTestRuntime()**. Tento test je **parciální** (stejně jako u paměti RAM). Velikost testovaného bloku je nastavena na 32 B. Následující vývojový diagram 4.12 zobrazuje posloupnost instrukcí ve funkci **FlashTestRuntime()**. Jeden cyklus funkce trvá 940 ns.



Obrázek 4.12: Test nevariabilní paměti ROM – vývojový diagram.

Pro výpočet CRC na LPC55S69 je využitá podpora ze strany hardware – **CRC modul**. Ten je možné inicializovat pro různé typy výpočtu CRC. Jelikož se kód testovací funkce shoduje s výše uvedeným vývojovým diagramem, je zde pro představu uveden alespoň úryvek kódu z inicializace hardwarového CRC modulu (jedná se o známou variantu CRC-32/MPEG-2).

```
1   void InitFlashTest(uint32_t start, uint32_t end)
2   {
3     startAddress = start;
4     endAddress = end;
5     currentAddress = start;
6
7     config.polynomial = kCRC_Polynomial_CRC_32;
8     config.reverseIn = false;
9     config.complementIn = false;
10    config.reverseOut = false;
11    config.complementOut = false;
12    config.seed = 0xFFFFFFFF;
13
14    /* First reset */
15    CRC_Init(CRC_ENGINE, &config);
16  }
```

Injekce poruchy

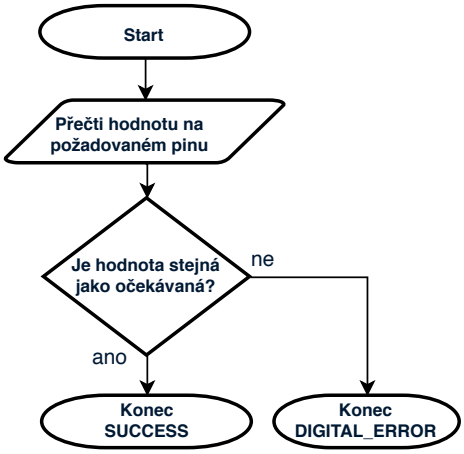
Vzhledem k tomu, že za běhu aplikace se mi **nepovedlo změnit obsah paměti ROM** (i přesto, že k tomuto účelu existují SDK ovladače), tak jsem přistoupil k jiné metodě. Po fázi kompilace celé aplikace není výsledný binární soubor ihned stažen do paměti mikro-

kontroléru, ale je trochu upraven. Konkrétně – v paměti ROM je definován region, který je nevyužitý. V oblasti tohoto regionu stačí změnit jediný bit, a poté soubor nahrát do paměti mikrokontroléru. Po spuštění aplikace test detekuje (simulovanou) poruchu paměti ROM. Je to způsobeno tím, že po kompilaci se do výsledného binárního souboru uloží CRC součet. Ten se znovu počítá za běhu aplikace. Jakmile se však výsledky neshodují, tak je detekována chyba.

Latence testu paměti ROM je v nejhorším případě 3.6 ms – je to doba, za kterou test zkontroluje celý rozsah paměti ROM. Pokud by byl test umístěn v přerušení s periodou 10 ms, byl by celý vykonán za 8.03 s.

4.2.6 Digitální vstup/výstup

Test digitálního vstupu/výstupu se nachází v souboru *digital_io.c*. Obsahuje jedinou funkci a to **DigitalPinValueTest()**, jejíž doba vykonání činí 1.2 μs. Tato primitivní funkce pouze kontroluje, zda vstupní nebo výstupní hodnota pinu odpovídá hodnotě očekávané. Průběh funkce je zobrazen na diagramu 4.13.



Obrázek 4.13: Test digitálního vstupu/výstupu – vývojový diagram.

V aplikaci lze funkci **DigitalPinValueTest()** použít např. pro kontrolu redundantních vstupů (což je případ této práce). Výstup pinu, který řídí relé, je připojen na dva nezávislé vstupní piny. Na všech třech pinech se tedy musí objevit v jeden časový okamžik stejná hodnota. Kontrola těchto pinů vypadá pak následovně:

Mimo redundantní kontroly popsané výše, je ve výsledném řešení implementována také kontrola pinu proti zkratu s pinem sousedním. Testovaný pin je interně nastaven jako **pull-down** – výchozí vstupní hodnota je logická 0. Sousední pin je zvolen dle schéma zapojení a je nastaven jako výstupní s logickou hodnotou 1. Pokud by tyto dva piny byly zkratované, tak test detekuje poruchu na testovaném pinu (zjistí, že hodnota na vstupním pinu je logická 1).

Injekce poruchy

V této sekci jsou popsány 3 varianty. Poruch jsou zavedeny fyzicky – například pouhým rozpojením vodiče, zkratováním dvou sousedních pinů nebo připojení výstupního pinu k zemi.

```
void CheckPins(void)
{
  uint8_t errorCount = 0;

  /* Delay */
  for(uint32_t i = 0; i < 50U; i++) __NOP();

  /* Check if all pins have appropriate value */
  if (DigitalPinValueTest(0, 18U, pumpState) == DIGITAL_ERROR) errorCount++;
  if (DigitalPinValueTest(0U, 1U, pumpState) == DIGITAL_ERROR) errorCount++;
  if (DigitalPinValueTest(1U, 14U, pumpState) == DIGITAL_ERROR) errorCount++;

  if (errorCount) ErrorHandler(DIGITAL_ERROR);
}
```

1. Zkrat dvou sousedních pinů, které jsou kontrolovány na očekávanou hodnotu. Dle schéma zapojení se jedná o pin **P18, 1** a **P18, 3**.

2. Fyzicky odpojit alespoň jeden ze dvou vodičů, fungujících jako redundantní kontrola výstupního pinu, který řídí relé (popsáno výše v této sekci).

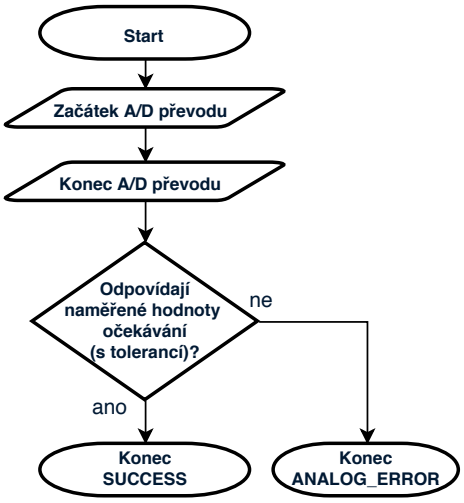
3. Uzemnit výstup, který řídí relé, právě v okamžiku, kdy je očekávaná hodnota logická 1 (sepnuté čerpadlo).

Latence testu závisí na délce periody přerušení, ve kterém je test volán. V případě tohoto řešení se (v nejhorším případě) opět jedná o dobu 10 ms.

4.2.7 Analogový vstup

Kód tohoto testu je dostupný v souboru *analog.c*. Funkce, která provádí kontrolu naměřených hodnot, se nazývá **AnalogTest()** a její exekuční čas se pohybuje kolem 1.8 μs.

Diagram 4.14 zobrazuje průběh testu analogového vstupu. Začátek A/D převodu je hardwarově odstartován každých 10 ms trigger signálem z modulu **CTIMER**. Jakmile je převod dokončen, dojde k přerušení, v jehož obslužné rutině se provádí většina bezpečnostních testů (včetně testu analogového vstupu).



Obrázek 4.14: Test analogového vstupu – vývojový diagram.

Před prvním převodem je nezbytně nutné správně inicializovat **ADC** modul na mikrokontroléru LPC55S69. A právě tuto činnost provádí funkce **InitAnalogTest()**. Mimo základního nastavení ADC modulu, jsou v této funkci nastaveny jednotlivé vstupní kanály, které jsou vždy po spuštění převodu měřeny. Konkrétně se jedná o interní referenční napětí **1 V, 3.3 V** a výstupní signál tlakového senzoru.

Následující (upravený) úsek kódu ukazuje, jakým způsobem se v aplikaci čtou výsledky převodu z FIFO fronty, které jsou dále použity jako vstup do kontrolní funkce.

```
    value3V3 = (ADC0->RESFIFO[0] & ADC_RESFIFO_D_MASK);
    value1V = (ADC0->RESFIFO[0] & ADC_RESFIFO_D_MASK);
    valueUser = (ADC0->RESFIFO[0] & ADC_RESFIFO_D_MASK);

    AnalogTest(value3V3, value1V, valueUser);

    /*****************************************************/

    uint32_t AnalogTest(uint32_t value3V3, uint32_t value1V, uint32_t valueUser)
    {
    if (value3V3 > ANALOG_3V3_HIGH || value3V3 < ANALOG_3V3_LOW)
    {
      return ANALOG_ERROR;
    }

    if (value1V > ANALOG_1V_HIGH || value1V < ANALOG_1V_LOW)
    {
      return ANALOG_ERROR;
    }

    if (valueUser > ANALOG_USER_HIGH || valueUser < ANALOG_USER_LOW)
    {
      return ANALOG_ERROR;
    }

    return SUCCESS;
    }
```

Limitní hodnoty jako jsou např. *ANALOG_1V_HIGH* a *ANALOG_1V_LOW* jsou předem vypočítány v souboru *analog.h*. Pro představu je v následujícím bloku kódu znázorněn výpočet limitních hodnot, pro referenční napětí 1 V.

```
#define ANALOG_REFERENCE_VOLTAGE 3.3f
#define ANALOG_MIN 0U
#define ANALOG_MAX 65535.0f
#define ANALOG_TOLERANCE_VOLTAGE 0.3f
#define ANALOG_TOLERANCE ((ANALOG_MAX * ANALOG_TOLERANCE_VOLTAGE) / ANALOG_REFERENCE_VOLTAGE)

/* 1V */
#define ANALOG_1V_EXPECTED (ANALOG_MAX/ANALOG_REFERENCE_VOLTAGE)
#define ANALOG_1V_LOW (ANALOG_1V_EXPECTED - ANALOG_TOLERANCE)
#define ANALOG_1V_HIGH (ANALOG_1V_EXPECTED + ANALOG_TOLERANCE)
```

Injekce poruchy

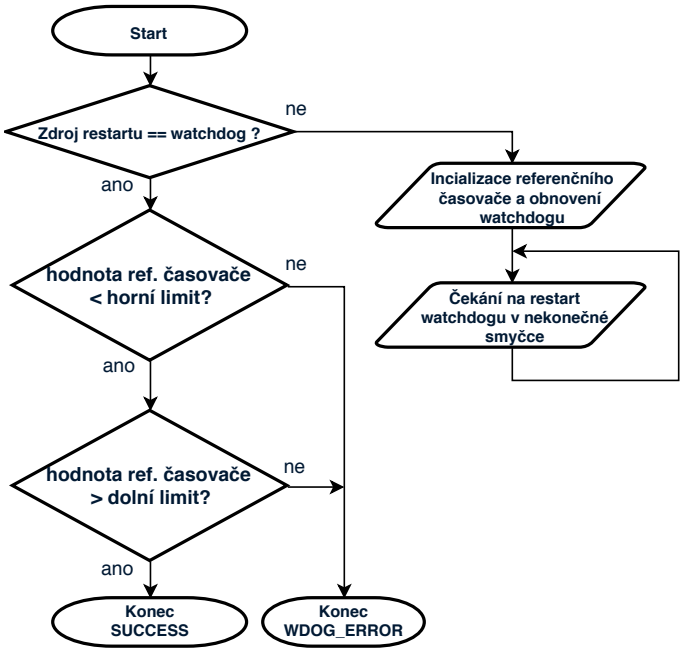
Softwarově je implementována jedna varianta. Existuje také možnost poruchu způsobit fyzicky.

1. **Softwarově** – dle standardu IEC 60730-B musí být testován také multiplexor vstupů do převodníku. Pokud je tedy změněno číslo kanálu referenčního napětí 1 V, na číslo, které odpovídá kanálu 3.3 V – test by měl detekovat poruchu.
2. **Fyzicky** – odpojení vodiče, který vede signál z tlakového senzoru. Jedná se o pin **P19, 4**.

Latence testu je závislá na dvou faktorech: jak často je spouštěn **převod A/D převodníku** a jak často je kontrolován **výsledek** daného převodu. V implementovaném řešení je nejhorší možný čas detekce poruchy 10 ms.

4.2.8 Watchdog

Implementace testu je dostupná v souboru *wdog.c*. Tvoří jej funkce **WdogTest()** a **WdogRefresh()**. Test je prováděn pouze jedenkrát, a to po restartu, protože kdyby se měl provádět za běhu, tak by se aplikace musel neustále restartovat (což je nežádoucí).



Obrázek 4.15: Test watchdogu – vývojový diagram.

Inicializace modulu je provedena bezprostředně po restartu zařízení (pokud je použití watchdogu povoleno). Jakmile je watchdog povolen a nastaven, nejde již jeho běh zastavit či zakázat, dokud nenastane další restart – jedná se o bezpečnostní mechanismus daný výrobcem.

```
1   /* Enable Watchdog */
2   SYSCON->CLOCK_CTRL |= SYSCON_CLOCK_CTRL_FRO1MHZ_CLK_ENA_MASK;  /* Enable 1MHz */
3   SYSCON->WDTCLKDIV = 1U;           /* Divide by 2 (500kHz) */
4   SYSCON->AHBCLKCTRL.AHBCLKCTRL0 |= SYSCON_AHBCLKCTRL0_WWDT_MASK; /* Enable WDT clock */
5
6   wwdt_config_t config;
7   WWDT_GetDefaultConfig(&config);
8   /*
9   * Set period 12 ms. Input to WD is 500kHz and wdog has fixed 4 prescaler.
10  * Thus wdog counter is clocked by 125kHz.
11  * */
12  config.timeoutValue = 1500U;
13  config.warningValue = 0;
14  config.windowValue = 0xFFFFFF; /* Maximum possible TV value = disable windowing */
15  config.enableWatchdogReset = true;
16  config.clockFreq_Hz = 1000000U;
17  WWDT_Init(WWDT, &config);
```

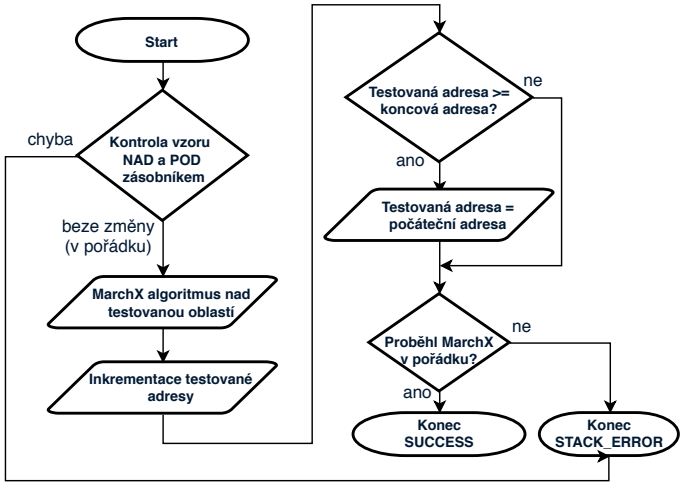
Injekce poruchy

Injekce poruchy je implementována pouze jedním způsobem a to **změnou děličky hodinového signálu**, který se vyskytuje na vstupu watchdog modulu. Tím se watchdogu sníží obnovovací perioda, v důsledku čehož se zařízení restartuje. Následující test modulu watchdog odhalí poruchu.

Latence testu odpovídá obnovovací periodě, která je v rámci inicializace nastavena ihned po restartu. V implementovaném řešení je délka této periody 12 ms.

4.2.9 Zásobník

Bezpečnostní test zásobníku je implementován v souboru *stack.c*. Funkce, která testování zajišťuje se jmenuje **StackTest()** a jeden její cyklus trvá přibližně 5 μs. Implementace této funkce se dělí na dvě části – **kontrola přetečení/podtečení zásobníku** a **MarchX** test paměťové oblasti, která je vyhrazená pouze pro zásobník.



Obrázek 4.16: Test zásobníku – vývojový diagram.

Implementace funkce **StackTest()**:

```
1   uint32_t StackTest(void)
2   {
3     /* Check if pattern below + above the stack is untouched */
4     if (stackUnderflow != pattern || stackOverflow != pattern)
5     {
6       return STACK_ERROR;
7     }
8
9     /* March test of Stack */
10    uint32_t status = RAMMarchX(currentAddress);
11
12    /* Increment tested address */
13    currentAddress += RAM_TEST_BLOCK;
14    if (currentAddress >= (uint32_t)&stackTop)
15    {
16      currentAddress = (uint32_t)&stackBase;
17    }
18    return status;
19  }
```

Injekce poruchy

V seznamu jsou uvedeny 3 různé způsoby, jakými se injektuje porucha

1. **Přepsání vzoru** nad/pod zásobníkem. Tímto způsobem se simuluje přetečení/podtečení zásobníku.

2. **Nekonečná rekurzivní funkce** – tato metoda zavolá rekurzivní funkci, která se donekonečna zanořuje. Jelikož se argumenty funkce ukládají na zásobník, tak je po určitém čase detekováno přetečení. Může se však stát, že poruchu detekuje dříve watchdog nebo test frekvence přerušení a toku programu. Tuto metodu implementuje funkce **InjectFaultStackRecursion()** v souboru *stack.c*.

3. **Změna hodnoty v paměti** v režimu ladění. Je velice obtížné injektovat poruchu paměti RAM — ve které se zásobník nachází — za běhu (nedestruktivně). Proto je tato porucha zanesena do paměti v režimu ladění. Více informací na demonstračním obrázku 4.10.

Latence testu je závislá na poruše, která má být detekována. Pokud se jedná o přetečení/podtečení, tak je v nejhorším případě doba detekce poruchy cca 10 ms. Pokud se však jedná o stuck-at poruchu v paměťovém regionu zásobníku, tak se v nejhorším případě latence pohybuje kolem 2.56 s. Tato reakční doba může být rapidně zkrácena např. **zmenšením** paměťové oblasti zásobníku, častějším provádění funkce **StackTest()** nebo také zvětšením testovaného bloku ve funkci **RAMMarchX()**.

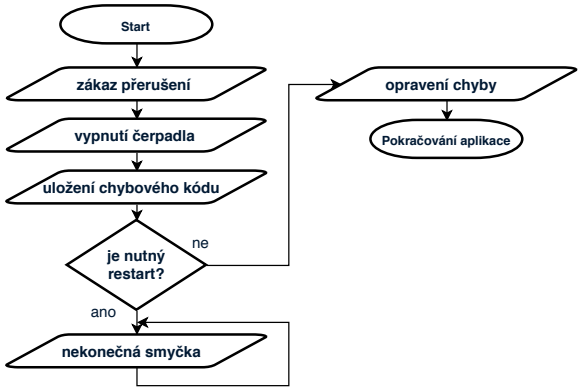
4.3 Demonstrační aplikace

Implementace demonstrační aplikace, která obsahuje integrované bezpečnostní testy, se nachází v souborech:

- **Primární jádro** –*main_core0.c*, *application.c/.h*, *initialization.c/.h*.
- **Sekundární jádro** – *main_core1.c*.

V souboru *main_core0.c* je implementována posloupnost příkazů, která je na obrázku 4.1 označená jako INICIALIZACE. Tento soubor také obsahuje implementaci hlavní nekonečné smyčky a obslužné rutiny přerušení, ve které je volána většina bezpečnostních testů.

Soubor *application.c/.h* implementuje aplikační logiku. Tvoří jej funkce **ErrorHandler(), Regulator()** a **CheckPins()**. Funkce **ErrorHandler()** je bezpečnostní funkce, která je volána při detekci jakékoliv poruchy4.17.



Obrázek 4.17: Průběh funkce **ErrorHandler()**.

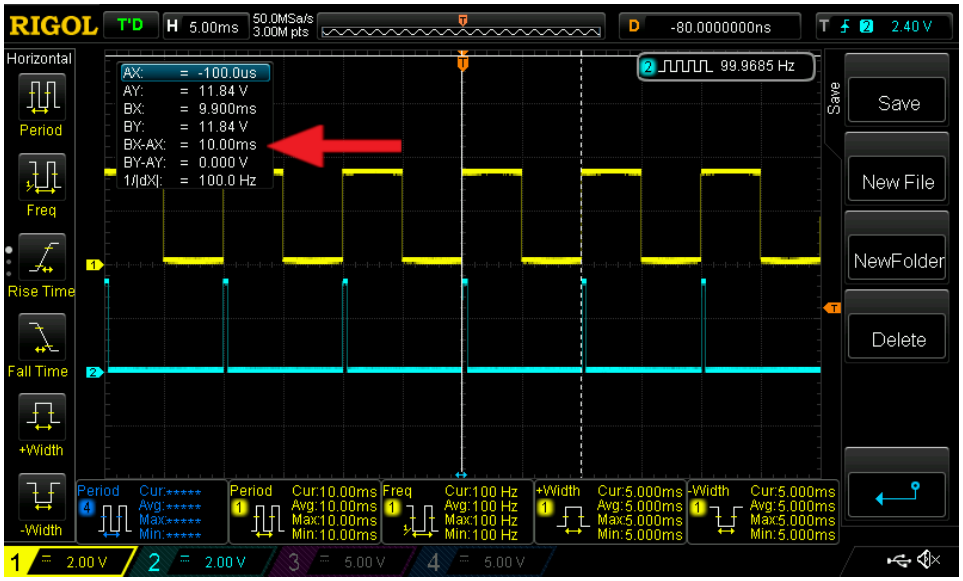
Chybový kód se v bezpečnostní funkci ukládá na předem vyhrazené místo v paměti RAM. Je zaručeno, že po restartu watchdogu zůstane tato paměťová oblast nezměněná. Opravení chyby není ve výsledném řešení implementováno, každá porucha čeká na restart watchdogu, který nastane nejpozději 12 ms po vstupu do nekonečné smyčky. Po restartu se vykonávání programu zastaví a čeká se, až uživatel provede PoR[2]. Funkce **ErrorHandler()** obsahuje sekce, které jsou předem připravené pro individuální obsluhu všech testovaných typů poruch.

Jak již plyne z názvu – funkce **Regulator()** vykonává primitivní regulaci spínání a vypínání vzduchového čerpadla. Kód funkce CheckPins() je již přiložen a popsán v sekci popisující implementaci digitálního vstupu/výstupu 4.2.6.

V souboru *initialization.c* je implementována pouze inicializace modulu **CTIMER**, která generuje trigger pro A/D převodník (kód v sekci 4.1). V souboru *initialization.h* je ke každému implementovanému testu jeden přepínač, pomocí kterého si lze daný test zapnout nebo vypnout. Jedná se o preprocesorové definice, takže při jakékoliv změně těchto přepínačů, je nutné aplikace celou znovu zkompilovat.

V sekci Inicializační fáze a běh programu je zmíněno, že modul **CTIMER** generuje každých 10 ms trigger signál pro spuštění převodu v A/D převodníku. Po ukončení převodu je generováno přerušení, v jehož obsluze se čte výsledek převodu a zároveň se v tomto přerušení vykonávají bezpečnostní testy. To znamená, že na vykonání převodu analogového signálu a na vykonání bezpečnostních testů je vymezená perioda 10 ms. Tyto časové údaje jsou také ověřené měřením na digitálním osciloskopu Rigol DS1054Z 4.18, 4.19. Obě dvě měření byly provedené za pomocí dvou pinů, které byly připojené na vstup osciloskopu.

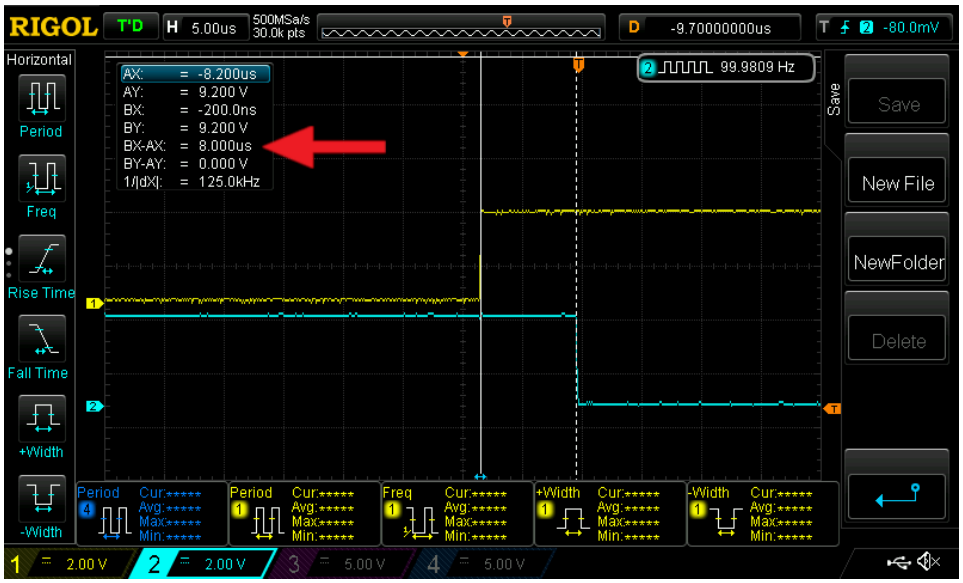
Na prvním obrázku je zachyceno měření periody signálu trigger. Žlutý signál představuje výstup z modulu **CTIMER**, jež s nástupnou hranou startuje převod modulu **ADC**. Jak lze vidět v levém horním rohu – šipka zobrazuje periodu 10 ms. Délka modrého signálu ve stavu logické 1 osciluje okolo hodnoty 400 μs. Tato doba představuje celkový výpočetní čas, potřebný pro vykonání bezpečnostních testů v přerušení.



Obrázek 4.18: Záznam z měření časových parametrů aplikace na osciloskopu Rigol DS1054Z. Červená šipka ukazuje na měřenou periodu – 10 ms.

[2]PoR – Power-on reset.

Další měřenou vlastností byla doba trvání A/D převodu. Ta byla měřená tak, že na prvním pinu byla opět (žlutě vyznačená) perioda signálu trigger. Druhý pin překlápěl svoji hodnotu při každém vstupu do přerušení, které signalizuje konec A/D převodu. Naměřená doba trvání jednoho převodu je 8 μs.



Obrázek 4.19: Záznam z měření časových parametrů aplikace na osciloskopu Rigol DS1054Z. Červená šipka ukazuje naměřenou dobu trvání A/D převodu.

4.3.1 Parametry použitého hardwarového vybavení

V této sekci jsou popsány parametry elektronických zařízení, která byla použita pro realizaci demonstrační aplikace.

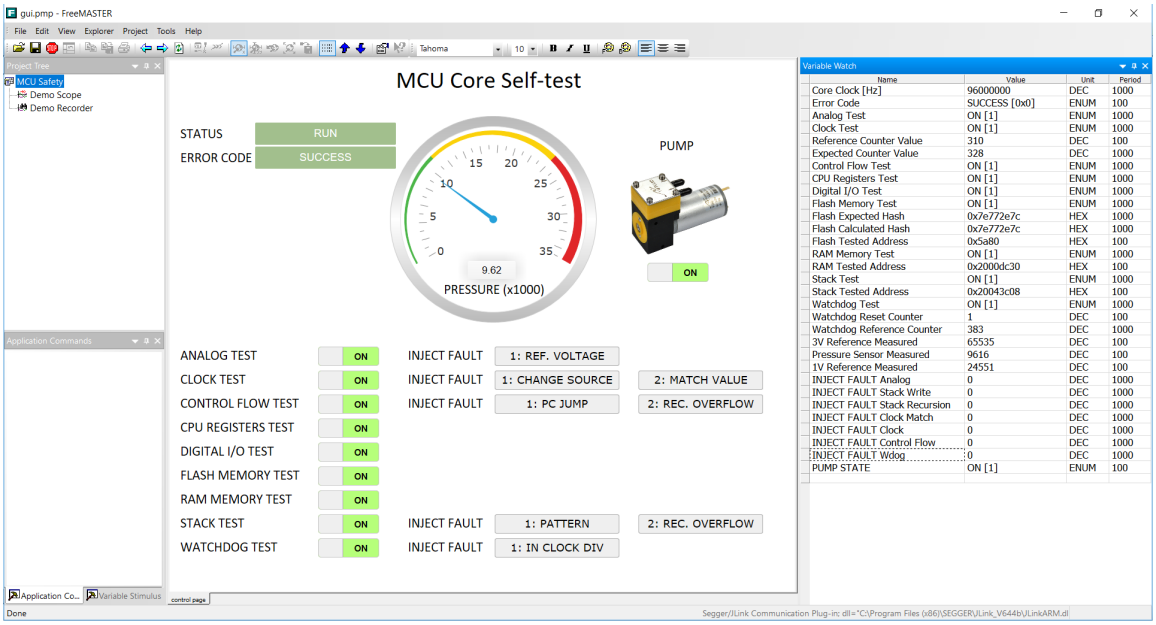
- **Micro DC vzduchové čerpadlo**
  - maximální tlak: 100 kPa (1 bar).
  - maximální průtok: 3.2 L za minutu.
  - vstupní napětí: DC 12 V.
- **Relé**
  - řídicí napětí: 5 V.
  - maximální řízené napětí: 30 V.
  - maximální řízený proud: 10 A.
- **12 V zdroj**
  - vstupní napětí: AC 220 V
  - výstupní napětí: DC 12 V.
  - maximální výstupní proud: 15 A.

4.4 Grafické uživatelské rozhraní

Grafické uživatelské rozhraní slouží pro sledování stavu aplikace, případně pak pro injekci různých poruch, které jsou softwarově implementované. K tomuto účelu jsem zvolil ladicí software **FreeMaster** [27]. Je to nástroj, který komunikuje s cílovou platformou (např. J-Link Debugger) a dokáže za běhu aplikace monitorovat a upravovat stav libovolných proměnných.

FreeMaster umožňuje svým uživatelům vytvořit si vlastní HTML stránku a vložit do ní libovolné grafické prvky, které mohou aplikaci ovládat, či jen zobrazovat její stav. V souboru *safety.htm* je taková HTML stránka implementována.

Výsledek lze vidět na obrázku 4.20. V pravé části se zobrazuje stav vybraných proměnných v čase. Hlavní část tvoří samotné uživatelské prostředí. Každý bezpečnostní test má svůj indikátor, zda je zapnutý či vypnutý. Hned vedle indikátoru jsou umístěna tlačítka, která zajistí injekci poruchy. FreeMaster disponuje také funkcí **Scope**, což je v podstatě softwarová simulace osciloskopu, která měří hodnotu zvolených proměnných. V řešení této práce je funkce **Scope** několikrát použita – např. 5.5 nebo 4.9.



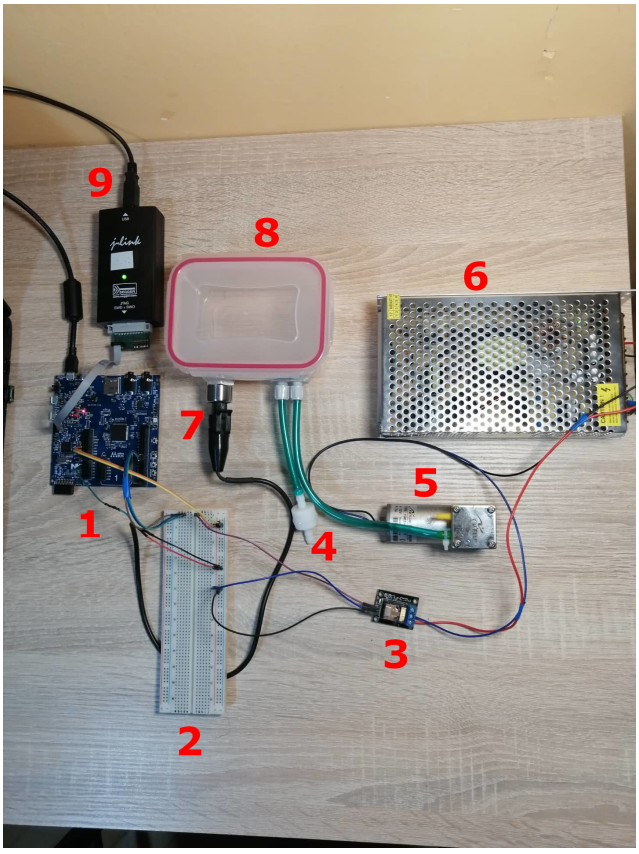
Obrázek 4.20: Grafické uživatelské prostředí FreeMaster.

Kapitola 5

Vyhodnocení

Tato kapitola analyzuje dosažené výsledky. U vybraných případů je popsán proces validace implementovaných bezpečnostních testů. Na konci této kapitoly jsou zmíněny zajímavosti/komplikace, které se v průběhu vývoje vyskytly.

Dle návrhu demonstrační aplikace z pohledu hardware (3.6), byl sestaven výsledný systém regulace tlaku ve vzduchotěsné nádobě.



Obrázek 5.1: Pohled na celou demonstrační aplikaci. 1–MCU, 2–propojovací deska, 3–relé, 4–přetlakový ventil, 5–čerpadlo, 6–zdroj, 7–tlakový senzor, 8–vzduchotěsný box a 9–Jlink Debugger.

Následující graf zobrazuje průběh regulace tlaku. Vrchní hodnota reprezentuje stav čerpadla – zapnuto nebo vypnuto. Ve spodní části grafu je vidět průběh tlaku v uzavřené nádobě (hodnota tlakového senzoru převedená A/D převodníkem).

Obrázek 5.2: Na grafu je vyobrazen průběh regulace tlaku.

V následujících sekcích jsou vybrány 4 případy injekce poruchy. Porucha hodinové frekvence a watchdogu je injektována softwarově, zatímco porucha digitálního a analogového vstupu je injektována fyzicky. Pro každý níže popsaný případ byla vybrána jedna varianta zanesení poruchy. Všechny implementované varianty lze nalézt v kapitole Implementace.

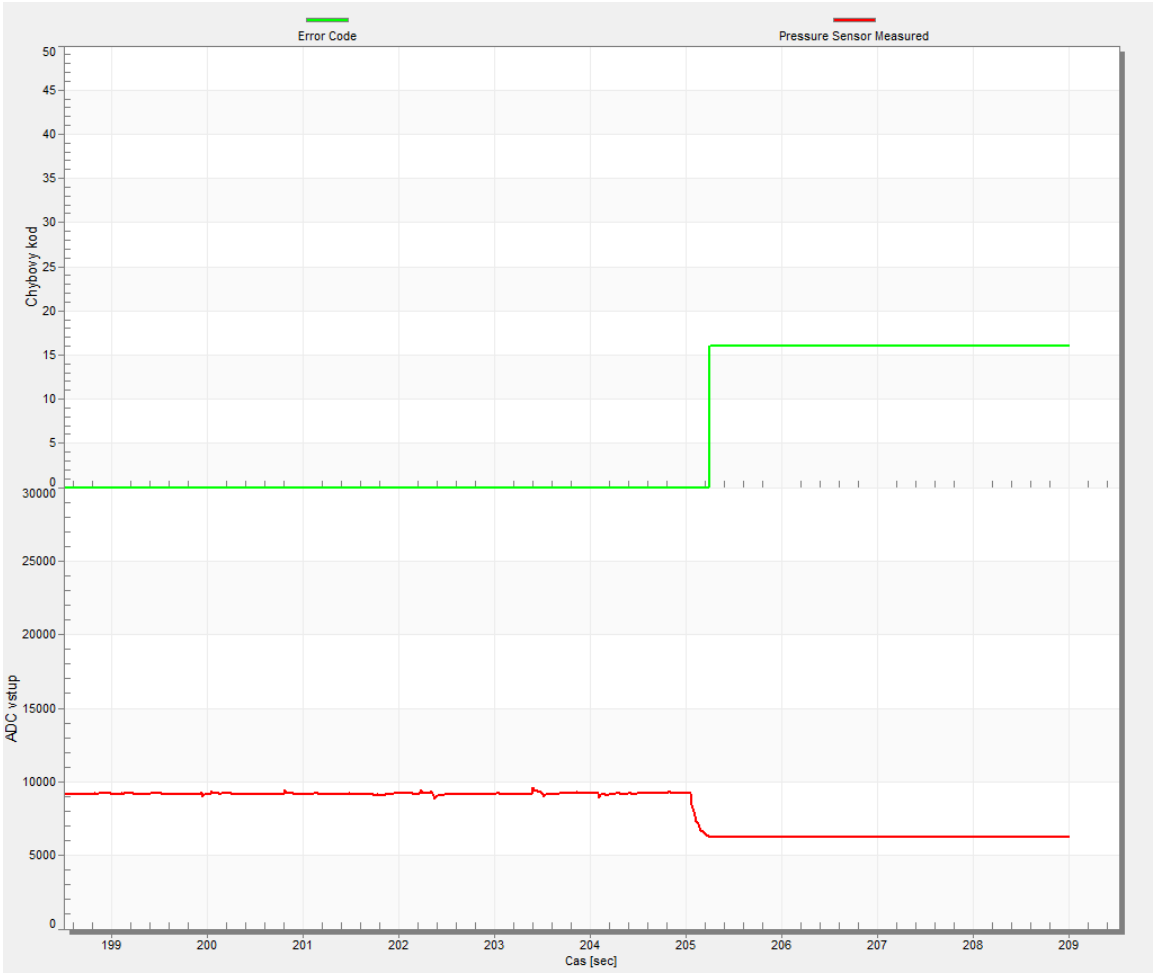
5.1 Test digitálního vstupu/výstupu

Injekce poruchy, kterou by měl rozpoznat test digitálního vstupu/výstupu, je realizována fyzickým rozpojením vodiče. Konkrétně se rozpojí jeden ze dvou vstupních kontrolních vodičů, který je v systému použit jako redundantní zpětná vazba signálu, který řídí relé. Na grafu (5.3) lze vidět průběh validace. Spodní 3 veličiny (červená, modrá a fialová) představují signál, který řídí relé. Červená čára znázorňuje průběh výstupního pinu v čase. Další dvě veličiny — modrá a fialová — znázorňují průběh dvou nezávislých kontrolních pinů. Pokud je jeden z nich odpojen, tak funkce **DigitalPinValueTest()** detekuje poruchu (zelená čára – chybový kód 256).

Obrázek 5.3: Injekce chyby – testování digitálního vstupu/výstupu.

5.2 Test analogového vstupu

Validace testu analogového vstupu je (stejně jako předchozí validace) prováděna fyzicky – odpojením vodiče signálu tlakového senzoru od vstupního pinu. Průběh je zobrazen na grafu (5.4), kde červená veličina představuje průběh naměřené hodnoty tlaku v nádobě. V okamžiku odpojení vodiče signálu je funkcí **AnalogTest()** detekována porucha analogového vstupu (zelená veličina – chybový kód 16).



Obrázek 5.4: Injekce chyby – testování analogového vstupu.

5.3 Test hodinové frekvence

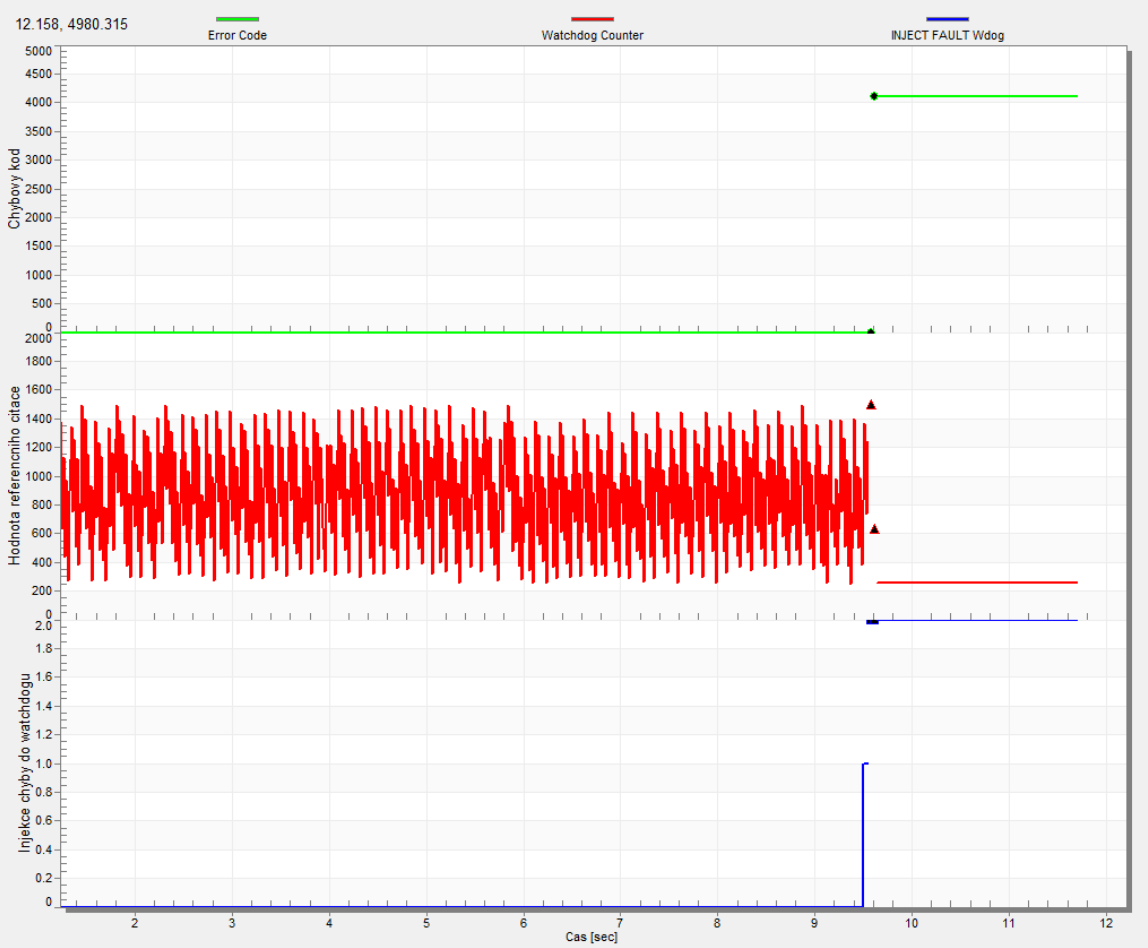
Pro injekci poruchy hodinového signálu procesoru je použita metoda, která změní zdroj hodinového signálu referenčního časovače.

Obrázek 5.5: Injekce chyby – změna zdroje hodinového signálu do referenčního časovače. Zelená veličina – chybový kód, červená – hodnota referenčního čítače, modrá – čas injekce.

Původní frekvence referenčního časovače je 32 kHz, ale za běhu je nahrazena frekvencí 1 MHz. Na obrázku (5.5) jde vidět, že stav referenčního čítače osciloval kolem hodnoty 310. Jakmile se změnil zdroj hodinového signálu (proběhla injekce), tak se hodnota čítače okamžitě zvýšila. Při testu hodinové frekvence (**ClockTest()**) přesáhla hodnota referenčního čítače vrchní hranici tolerance – porucha byla detekována.

5.4 Test watchdogu

Injekce poruchy je implementována pouze jedním způsobem a to **změnou děličky hodinového signálu**, který se vyskytuje na vstupu watchdog modulu. Konkrétně se za běhu změní dělící faktor z hodnoty 1 na hodnotu 0. To zapříčiní zkrácení obnovovací periody watchdogu, v důsledku čehož se nestihne watchdog modul obnovit a nastane restart zařízení. Poté proběhne test modulu watchdog (**WdogTest()**), který odhalí poruchu (watchdog restartuje zařízení po 6 ms, což není očekávané chování).



Obrázek 5.6: Injekce chyby – změna hodnoty dělícího faktoru na vstupu hodinového signálu *watchdog* modulu.

Modrý signál na obrázku (5.6) indikuje čas injekce poruchy. Červený signál zobrazuje stav čítače watchdogu. Zelená veličina představuje chybový kód, který se ihned po injekci poruchy změní na hodnotu 4096 (chybový kód watchdogu).

5.5 Zajímavosti a komplikace

Ve fázi vývoje se u implementace některých testů objevily zajímavosti či menší komplikace. V následujícím seznamu je pár z nich uvedeno.

- **Časovače v ladícím režimu** – časovače na mikrokontroléru LPC55S69 se nezastaví v ladícím režimu. To znamená, že když se v MCUXpresso IDE pozastavilo vykonávání kódu, časovače pořád inkrementovaly svoji hodnotu. Docela obtížně se kvůli tomu ladil vývoj testů. Pro zobrazení hodnoty čítačů za běhu byl tedy použit nástroj FreeMaster, který je použit ve výsledku jako grafické rozhraní.

- **Softwarový zápis do paměti ROM** – i přes veškeré úsilí se nepovedlo implementovat (simulovat) injekci poruchy do paměti ROM za běhu. Snahou bylo změnit jediný bit v testované paměti. V poskytovaném SDK existuje ovladač pro zápis do ROM paměti, nicméně při snaze použít jej, docházelo k nepříjemné situaci – přístup do upraveného bloku v ROM paměti pokaždé vygeneroval *HARD FAULT*.

- **Vícenásobná detekce poruchy** – ve fázi validace implementovaných testů se několikrát stalo, že injektovanou poruchu odhalil jiný test, než bylo očekáváno. Např. při injekci přetečení zásobníku, kdy je do sebe sama zanořována rekurzivní funkce, zahlásil jako poruchu jako první test frekvence přerušení.

  Poté jsem si také uvědomil, že je velice pravděpodobné, že by tuto injekci poruchy odhalil také watchdog. Ve výsledku je to velice pozitivní fakt.

- **Perioda přerušení** – v implementovaném řešení se používá pouze jedno přerušení (resp. jeho softwarová obsluha). Na začátku vývoje jsem odhadl, že pokud budou všechny bezpečnostní testy vykonávány v obsluze tohoto přerušení, nebude to trvat déle než 10 ms. To je sice pravda, ale celková doba vykonání je cca 400 μs (25krát kratší). Tento fakt lze vidět na obrázku z měření (4.18), ze kterého vyplývá, že perioda by mohla být mnohem kratší. Tím by se výrazně snížila latence většiny testů.

- **Speciální registry** – v implementovaném řešení nejsou testovány registry jádra FAULTMASK, PRIMASK, BASEPRI a CONTROL, protože práce s nimi v režimu ladění nefunguje tak, jak bych očekával. Nakonec jsem se rozhodl tyto registry netestovat.

Kapitola 6

Závěr

Dle zadání byly implementovány bezpečnostní testy, které za běhu aplikace ověřují funkčnost dvou-jádrového mikrokontroléru LPC55S69. Testy byly navrženy s ohledem na standard IEC 60730 (příloha H, softwarová třída B), který se zabývá bezpečným provozem vestavěného řídícího hardwaru a softwaru pro domácí spotřebiče.

Výsledkem práce je také demonstrační řešení, které má za úkol regulovat tlak ve vzduchotěsné nádobě. Cílem demonstračního řešení je názorná ukázka integrace bezpečnostních testů v reálné aplikaci. Součástí implementace je také injekce poruch prostřednictvím sekundárního jádra, ovládaného skrze grafické uživatelské rozhraní.

Práce pro mě představuje veliký přínos v oblasti vestavěných systémů a jejich bezpečnosti, protože pro správný návrh testů bylo nutné nastudovat příslušnou část standardu IEC 60730 a také referenční manuál mikrokontroléru. Neméně významným přínosem byla pro mě fáze vývoje, protože jsem 2 z celkových 9 testů implementoval v jazyce symbolických adres (často nesprávně označovaného jako *assembler*).

V budoucnu by bylo přínosné vylepšit tuto práci následujícími úpravami:

- Rozšíření testu CPU registrů – nyní je pokryto cca 80 % registrů.
- Implementovat test externí komunikace, která je v praxi často využívána.
- Použít sekundární jádro nejenom pro injekci poruch, ale také pro dodatečnou kontrolu bezpečnosti celého mikrokontroléru. Snížilo by se tím vytížení primárního jádra.

Pokud by se tahle vylepšení implementovala, pak by zbývalo už jenom automatizovat validaci testů. To znamená, že by se v určité sekvenci injektovaly do mikrokontroléru poruchy a zároveň by kontrolní mechanismus validoval, zda danou poruchu nějaký test detekoval.

Literatura

[1] Abramovici, M.; Breuer, M. A.; Friedman, A. D.: *Digital Systems Testing And Testable Design*. IEEE Press, 1990, ISBN 0-7803-1062-4.

[2] ARM: *Cortex-M3 Devices Generic User Guide, Core registers*. [Online; navštíveno 23.01.2019].
URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0552a/CHDBIBGJ.html

[3] ARM: *Cortex Microcontroller Software Interface Standard*. [Online; navštíveno 12.5.2019].
URL https://developer.arm.com/tools-and-software/embedded/cmsis

[4] Avienzis, A.; Laprie, J.-C.; Randell, B.; aj.: *Basic concepts and taxonomy of dependable and secure computing*. [Online; navštíveno 24.1.2019].
URL http://ieeexplore.ieee.org/document/1335465/

[5] Bushnell, M. L.: *Essentials Of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Kluwer Academic Publishers, 2000, ISBN 9780792379911.

[6] Christiansen, B.: *MTTR, MTBF, or MTTF? – A Simple Guide To Failure Metrics*. [Online; navštíveno 24.1.2019].
URL
https://limblecmms.com/blog/mttr-mtbf-mttf-guide-to-failure-metrics/

[7] Components, F.: *LPC55S6x Series Microcontrollers - Product information*. [Online; navštíveno 17.01.2019].
URL https://ie.farnell.com/nxp/lpc55s69jbd100k/mcu-32bit-100mhz-hlqfp/dp/2992430?scope=partnumberlookahead&ost=LPC55S69JBD100K

[8] Foundation, F. S.: *GNU linker documentation*. [Online; navštíveno 1.04.2019].
URL https://sourceware.org/binutils/docs/ld/

[9] Freescale Semiconductor, I.: *IEC 60730B Safety Routines for Kinetis MCUs*. [Online; navštíveno 9.03.2019].
URL https://www.nxp.com/docs/en/application-note/AN4873.pdf

[10] Hobbs, C.: *Embedded Software Development for Safety-Critical Systems*. CRC Press, 2016, ISBN 978-149-8726-702.

[11] IEC 60730-1: *Automatic electrical controls - Part 1: General requirements*. International Electrotechnical Commission, 2015, 1161 s.

[12] IEC 61508-1: *Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 1: General requirements*. International Electrotechnical Commission, 2010, 127 s.

[13] IEC 61508-2: *Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 2: Requirements for electrical/electronic/programmable electronic safety-related systems*. International Electrotechnical Commission, 2010, 187 s.

[14] IEC 61508-3: *Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 3: Software requirements*. International Electrotechnical Commission, 2010, 234 s.

[15] IEC 61508-4: *Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 4: Definitions and abbreviations*. International Electrotechnical Commission, 2010, 68 s.

[16] IEC 61508-5: *Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 5: Examples of methods for the determination of safety integrity levels*. International Electrotechnical Commission, 2010, 97 s.

[17] IEC 61508-6: *Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 6: Guidelines on the application of IEC 61508-2 and IEC 61508-3*. International Electrotechnical Commission, 2010, 237 s.

[18] IEC 61508-7: *Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 7: Overview of techniques and measures*. International Electrotechnical Commission, 2010, 296 s.

[19] Ing. Michal Pavlík, P.: *Vývojové modely*. [Online; navštíveno 10.03.2019]. URL http://www.umel.feec.vutbr.cz/bdts/index.php/embedded-systemy/vyvojove-modely

[20] Jha, N. K.; Gupta, S.: *Testing of Digital Systems*. Cambridge University Press, 2003, ISBN 0521773563.

[21] Koopman, P.: *Best CRC Polynomials*. [Online; navštíveno 10.3.2019]. URL https://users.ece.cmu.edu/~koopman/crc/

[22] Ma, S. C.; Franco, P.; McCluskey, E. J.: *An experimental chip to evaluate test techniques: Experimental results*. [Online; navštíveno 26.01.2019]. URL https://ieeexplore.ieee.org/document/529895

[23] Miller, P.: *Srecord*. [Online; navštíveno 5.05.2019]. URL http://srecord.sourceforge.net/

[24] NXP Semiconductors: *LPC55S6x MCU Family Fact Sheet*. [Online; navštíveno 17.1.2019]. URL https://www.nxp.com/docs/en/fact-sheet/LPC55S6XFS.PDF

[25] Prášek, R.: *Functional Safety*. [Online; navštíveno 15.01.2019]. URL https://www.qmprofi.cz/33/funkcni-bezpecnost-uniqueidmRRWSbk196FNf8-jVUh4EjjkeFupcvnhRmUsfGJb5uA/

[26] Semiconductor, C.: *PSoC 3 and PSoC 5LP - IEC 60730 Class B Safety Software Library*. [Online; navštíveno 9.11.2018].
URL https://www.cypress.com/file/134271/download

[27] Semiconductors, N.: *FREEMASTER: FreeMASTER Run-Time Debugging Tool*. [Online; navštíveno 3.4.2019].
URL https://www.nxp.com/freemaster

[28] Semiconductors, N.: *MCUXpresso IDE*. [Online; navštíveno 5.04.2019].
URL https://www.nxp.com/support/developer-resources/software-development-tools/mcuxpresso-software-and-tools/mcuxpresso-integrated-development-environment-ide:MCUXpresso-IDEs

[29] Semiconductors, N.: *MCUXpresso SDK Builder*. [Online; navštíveno 12.5.2019].
URL https://mcuxpresso.nxp.com/en/welcome

[30] Semiconductors, N.: *Safety Class B with PMSM Sensorless Drive*. [Online; navštíveno 4.11.2018].
URL https://www.nxp.com/docs/en/application-note/AN5321.pdf

[31] Semiconductors, N.: *UM11126 LPC55S6x User manual*. [Online; navštíveno 6.04.2019].
URL https://www.nxp.com/docs/en/user-guide/UM11126.pdf

[32] STMicroelectronics: *Using the CRC peripheral in the STM32 family*. [Online; navštíveno 9.11.2018].
URL https://www.st.com/content/ccc/resource/technical/document/application_note/39/89/da/89/9e/d7/49/b1/DM00068118.pdf/files/DM00068118.pdf/jcr:content/translations/en.DM00068118.pdf

[33] Sundaram, M.: *Understand safety guidelines for electronic controls*. [Online; navštíveno 23.10.2018].
URL https://www.edn.com/design/test-and-measurement/4411232/Understand-safety-guidelines-for-electronic-controls

[34] Uher, J.: *Úvod do funkční bezpečnosti I: norma ČSN EN 61508*. [Online; navštíveno 20.01.2019].
URL http://automa.cz/cz/casopis-clanky/uvod-do-funkcni-bezpecnosti-i-norma-csn-en-61508-2004_08_32520_3609/

[35] Wang, L.-T.; Wu, C.-W.; Wen, X.: *VLSI Test Principles and Architectures: Design for Testability*. Elsevier, 2006, ISBN 978-0-12-370597-6.

Přílohy

Seznam příloh

Příloha A

Obsah CD

Adresářová struktura přiloženého CD:

/src/	zdrojové soubory projektů a technické zprávy
/license.txt	licence
/analog_error.mp4	demonstrační video
/xdenkf00.pdf	technická zpráva
/schematic.pdf	schéma zapojení mikrokontroléru na vývojové desce
/user_manual.pdf	uživatelský manuál pro LPC55S69

Příloha B

API bezpečnostních testů

B.1 Soubor analog.h

```
/*
* Initialization of LPADC module (clock, control register, trigger, sequence).
*
* */
void InitAnalogTest(void);

/*
* Check of measured values.
*
* value3V3...measured value 3V3
* value1V....measured value 1V
* valueUser..measured user value
*
* return.....ANALOG_ERROR or SUCCESS
*
* */
uint32_t AnalogTest(uint32_t value3V3, uint32_t value1V, uint32_t valueUser);
```

B.2 Soubor clock.h

```
/*
* This function initialize the CTIMER module.
* */
void InitClockTest(void);

/*
* Function checks if CTIMER counter value meets the limits.
*
* return.....CLOCK_ERROR or SUCCESS
*
```

```
* */
uint32_t ClockTest(void);
```

B.3 Soubor control_flow.h

```
/*
* Initialization of frequency and flow check variables.
*
* */
void InitControlFlowTest(void);

/*
* Flow check. Performs XOR operation on ADC0_ISR or main loop flow.
*
* flow....main or adc_isr flow check
* node....expected node
* diff....expected diff
*
* return..CONTROL_FLOW_ERROR or SUCCESS
*
* */
uint32_t ControlFlowTest(uint32_t flow, uint32_t node, uint32_t diff);

/*
* Frequency check of main loop.
*
* return..CONTROL_FLOW_FREQ_ERROR or SUCCESS
*
* */
uint32_t ControlFlowMainLoop(void);

/*
* Frequency check of ADC0_ISR. Resets main loop counter.
*
* return..CONTROL_FLOW_FREQ_ERROR or SUCCESS
*
* */
uint32_t ControlFlowADC(void);
```

B.4 Soubor cpu_reg.h

```
/*
* Tests R0-R12, SP, LR, PC, APSR and FPSCR CPU registers.
*
* return...CPU_REG_ERROR or SUCCESS
*
```

* */
uint32_t RegisterTest(void);

B.5 Soubor digital_io.h

/*
* Check of value on chosen port and pin.
*
* port.....tested port
* pin.....tested pin
* expectedValue..expected value
*
* return.....DIGITAL_ERROR or SUCCESS
*
* */
uint32_t DigitalPinValueTest(uint8_t port, uint8_t pin, uint8_t expectedValue);

B.6 Soubor flash.h

/*
* Initialization of CRC module.
*
* start...start address
* end.....end address
*
* */
void InitFlashTest(uint32_t start, uint32_t end);

/*
* Partial CRC hash calculation.
*
* return...FLASH_ERROR or SUCCESS
* */
uint32_t FlashTestRuntime(void);

B.7 Soubor ram.h

/*
* Initialization of addresses.
*
* start....start address
* end.....end address
* */
void InitRAMTest(uint32_t start, uint32_t end);

/*

```
* Partial RAM memory checking. Calls RAMMarchX().
*
* return.....RAM_ERROR or SUCCESS
* */
uint32_t RAMTest(void);

/*
* March X algorithm.
*
* current....current tested address
*
* return.....RAM_ERROR or SUCCESS
* */
uint32_t RAMMarchX(uint32_t current);
```

B.8 Soubor stack.h

```
/*
* Initialization of addresses.
*
* testPattern...pattern above and below the stack
* */
void InitStackTest(uint32_t testPattern);

/*
* Check of overflow or underflow. Also performs RAMMarchX() test on stack area.
*
* return STACK_ERROR or SUCCESS
* */
uint32_t StackTest(void);
```

B.9 Soubor wdog.h

```
/*
* Watchdog refresh sequence.
*
* */
void WdogRefresh(void);

/*
* Check if watchdog restart correctly (at the right time).
*
* return...WDOG_ERROR or SUCCESS
*
* */
uint32_t WdogTest(void);
```

Příloha C

Uživatelský manuál

Tento manuál slouží ke zprovoznění demonstrační aplikace na vývojovém kitu LPC55S69-EVK. Kromě samotného vývojového kitu není potřebný žádný dodatečný hardware.

1. Nainstalujte si MCUXpresso IDE, které je zdarma ke stažení na oficiální webu – záložka **SOFTWARE AND TOOLS -> MCUXpresso IDE -> Learn More**).
2. Ze stejné webové stránky si také stáhněte SDK balíček – **Select Development Board -> LPCXpresso55S69**.
3. Pro podporu ze strany IDE, je nutné SDK balíček do MCUXpresso IDE importovat. Zde je jednoduchý návod, jak to lze provést metodou **Drag & Drop**.
4. Dalším krokem je importování projektů do vývojového prostředí. Jelikož je mikrokontrolér LPC55S69 dvoujádrový, je potřeba importovat projekty 2. Na přiloženém CD jsou projekty uložené ve složkách */src/mcu_selftest_core0* a */src/mcu_selftest_core1*. Pro správné importování je tedy nutné metodou **Drag & Drop** přetáhnout soubory *.cproject* (z těchto složek) do MCUXpresso IDE.
5. V projektu *mcu_selftest_core0* je soubor */source/initialization.h*. V něm si uživatel může vypnout/zapnout jednotlivé bezpečnostní testy.
6. Nyní se stisknutím tlačítka **F7** spustí kompilace celé aplikace. Jako první se kompiluje projekt pro sekundární jádro, poté projekt pro jádro primární.
7. Ve vývojovém prostředí se v levém dolním rohu nachází **Quickstart Panel**. V něm je tlačítko **Debug**, po jehož stisknutí se stáhne projekt do paměti mikrokontroléru. Po nahrání se stisknutím tlačítka **F8** aplikace spustí.
8. Na vývojovém kitu začne blikat LED dioda střídavě – červeně a modře. Je to indikace toho, že primární (modrá barva) i sekundární (červená barva) jádro vykonává kód a nikde neuvázlo.
9. Pro interakci s aplikací je nutné spustit grafické uživatelské prostředí otevřením souboru */src/mcu_selftest_core0/freemaster/gui.pmp*. Ještě před tím si nainstalujte FreeMaster.
10. V aplikaci FreeMaster již stačí stisknout kombinaci kláves **CTRL-K** pro zahájení komunikace.