VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

FRAMEWORK PRO AUTOMATIZOVANÉ TESTOVÁNÍ MCUXPRESSO CONFIG TOOLS

TESTING FRAMEWORK FOR AUTOMATIC TESTS OF MCUXPRESSO CONFIG TOOLS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **TOMÁŠ DUBOVSKÝ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ADAM CRHA**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Dubovský Tomáš**
Program: Informační technologie
Název: **Framework pro automatizované testování MCUXpresso Config Tools**
Testing Framework for Automatic Tests of MCUXpresso Config Tools
Kategorie: Vestavěné systémy

Zadání:

1. Seznamte se produktem MCUXpresso Config Tools
2. Nastudujte aktuálně používané principy objektově orientovaného návrhu v existujícím frameworku a analyzujte možnosti optimalizace stávajícího řešení na úrovni implementace i návrhu.
3. Navrhněte úpravu/rozšíření frameworku, které bude mít kladný vliv na efektivitu a výkonnost automatizovaného testování.
4. Implementujte navržené řešení.
5. Proveďte otestování funkčnosti naimplementovaného systému a zhodnoťte dopad na efektivitu testování.

Literatura:

- Dle pokynů vedoucího.

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

Vedoucí práce: **Crha Adam, Ing.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 26. října 2018

Abstrakt

Cílem této práce bylo analyzovat stávající řešení frameworku pro testování MCUXpresso Config Tools, dále navrhnout a implementovat různé možnosti optimalizace, které povedou ke zvýšení efektivity testování. Na základě profilování stávajícího řešení byly navrhnuty a implementovány tři způsoby optimalizace. Prvním je sjednocení spouštění externích aplikací pod nový modul runner. Druhým je implementace souběžnosti v jazyce Python za pomocí multiprocessingu, multithreadingu a asyncio, následné porovnání jednotlivých metod a výběru nejvhodnější z nich. Třetím je implementace podpory zřetězeného zadávání příkazů řádkového rozhraní v MCUXpresso Config Tools.

Abstract

The aim of this thesis was to analyze the existing solution of the MCUXpresso Config Tools testing framework, to design and implement various optimization options that will lead to increased testing efficiency. Three ways of optimization have been designed and implemented based on profiling of the existing solution. The first is to unify the launch of external applications under the new module runner. The second is implementing concurrency in Python using multiprocessing, multithreading, and asyncio, then comparing each method and selecting the most appropriate one. The third is the implementation of support for chained entry of commands in commands line interface in MCUXpresso Config Tools.

Klíčová slova

MCUXpresso Config Tools, testování, optimalizace, multiprocessing, multithreading, asyncio, python, OOP

Keywords

MCUXpresso Config Tools, testing, optimization, multiprocessing, multithreading, asyncio, python, OOP

Citace

DUBOVSKÝ, Tomáš. *Framework pro automatizované testování MCUXpresso Config Tools.* Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Adam Crha

Framework pro automatizované testování MCUXpresso Config Tools

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Adama Crhy. Další informace mi poskytli Ing. Michal Stareček a Ing. Petr Dohnal. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Tomáš Dubovský
15. května 2019

Poděkování

Děkuji Ing. Adamu Crhovi, za pomoc a odborný dohled při rešení bakalářské práce. Dále bych chtěl poděkovat za cenné rady Ing. Michalu Starečkovi a Ing. Petru Dohnalovi.

Obsah

Kapitola 1

Úvod

Pojem testování je při vývoji jakéhokoliv programu znám již od doby prvních počítačů. Ne vždy se jedná o zrovna příjemnou činnost, ale o to je to činnost mnohdy důležitější. Dnešním trendem ve firmách zabývajících se tvorbou softwaru je testování co nejvíce automatizovat a ušetřit tak potřebné lidské i finanční zdroje na samotný vývoj. Tento přístup však ne vždy dosáhne požadovaných cílů a často vidíme vývoj komplikovaných testovacích nástrojů, jejichž vývoj a údržba přesahují lidské i finanční náklady na manuální testování. Navíc výsledný nástroj netestuje aplikaci kompletně a délka testování je příliš dlouhá. Proto je nutné vždy nejprve provést kvalitní analýzu, zda se nám automatizované testování vyplatí. Následně dodržovat při vývoji nástroje stejné zásady, jako kdybychom vyvíjeli produkt samotný.

Vývoj různých programů pro mikroprocesorová zařízení již dávno není jen doménou profesionálů. Cílem firem vyvíjejících tato zařízení je snadná možnost jejich konfigurace pro různé účely i pro laika s minimální nutností znalosti kódu. K tomuto účelu jsou vyvíjeny grafické aplikace, kde si mnohdy uživatel může v intuitivním grafickém prostředí nastavit požadované parametry a samotný kód je mu vygenerován automaticky. Příkladem je sada programů `MCUXpresso Config Tools` [25] od společnosti `NXP`.

Tato sada nástrojů umožňuje práci s mnoha mikroprocesory, čímž vzniká požadavek na zaručení správnosti vygenerovaného kódu. Manuální testování v tomto případě by bylo příliš komplikované. Velký počet možností, nezáživné a zdlouhavé spouštění kompilace pro každý zdrojový kód, by znamenalo přílišné zatížení. Proto byl vytvořen framework pro automatizované testování `MCUXpresso Config Tools - MCUX Builder`. Tento framework v základu slouží k ověření správnosti vygenerovaných kódů z `MCUXpresso Config Tools`. Je to komplexní nástroj, který na vstupu přebírá konfigurační soubory s různým nastavením mikroprocesorů pro `MCUXpresso Config Tools` a umožňuje od automatického generovaní zdrojových souborů, kompilaci známými kompilátory (`Arm`[1], `GCC`[2], `Keil`[3]) a spuštění na hardwaru až po reportování vyhodnocených výsledků jednotlivých testů.

Tento framework však díky své obsáhlosti trpí návrhovými a optimalizačními problémy, které je potřeba řešit, aby došlo k efektivnímu využití dostupných zdrojů pro automatické testování a celý proces trval únosně dlouhou dobu i při velkém počtu testů. Celý framework je psán pomocí programovacího jazyka `Python` a proto se tato práce zaměřuje zejména na moderní možnosti optimalizace v rámci tohoto jazyka. Samotná optimalizace je rozdělena do dvou částí. První je objektový návrh daného frameworku a jeho částí a možnosti op-

[1] https://developer.arm.com/tools-and-software/embedded/arm-compiler
[2] https://gcc.gnu.org/
[3] http://www.keil.com/product/

timalizace na této úrovni. A druhá se týká možností optimalizace implementace, kde je primárně soustředěno na metody využití více procesů, vláken, či asynchronní zpracování.

V kapitole 2 lze naleznout popis jednotlivých nástrojů `MCUXpresso Config Tools`. Kapitola 3, pak uvede čtenáře do oblasti automatizovaného testování. Kapitola 4 provede čtenáře analýzou stávajícího řešení od objektového návrhu až po výkonnostní a zátěžové testy, s využitím profilovacích nástrojů v jazyce `Python`. Kapitola 5 se zaměřuje na návrh konkrétních optimalizačních řešení. Kapitola 6 následně provede čtenáře implementací jednotlivých optimalizací – implementací souběžnosti, podpory zřetězení příkazů a modulu `runner`. Kapitola 7 se pak zabývá otestováním funkčnosti implementovaných řešení a jejich dopadem na efektivitu testování. Vše je pak shrnuto v závěru 8.

Kapitola 2

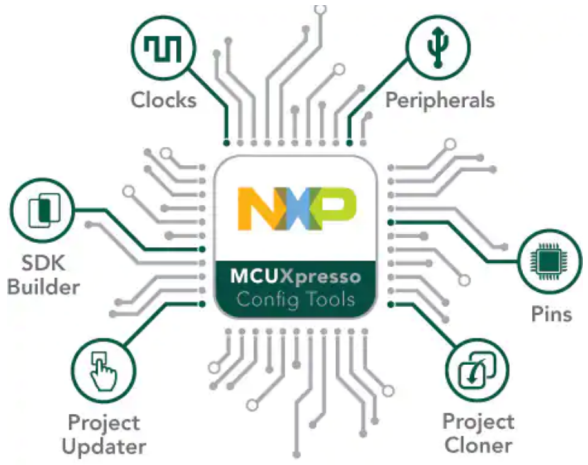
MCUXpresso Config Tools

V této kapitole si představíme sadu nástrojů MCUXpresso Config Tools od firmy NXP. Korektnost výstupů z těchto nástrojů má za cíl otestovat framework, jehož optimalizace je jedním z hlavních cílů této práce.

MCUXpresso Config Tools je integrovaná sada nástrojů, která pomáhá uživatelům při tvorbě softwaru pro mikrokontrolery od firmy NXP, založené na jádrech Arm® Cortex®-M[1], včetně i.MX RT[2], LPC[3] a Kinetis®. Tyto konfigurační nástroje umožňují vývojářům rychlé vygenerování SDK[4] balíčku a nastavení pinů, hodin a periferních zařízení až po vygenerování inicializačního C kódu [25].

Základními nástroji jsou: SDK Builder, Pins Tool, Clocks Tool, Peripherals Tool, Project Updater a Project Cloning viz. obrázek 2.1.

Všechny tyto nástroje jsou plně integrovány ve vývojovém prostředí MCUXpresso IDE[5], dále jsou dostupné ke stažení, či v online verzi. Nyní si jednotlivě představíme samotné nástroje, abychom získali ucelený přehled o možnostech této sady nástrojů.

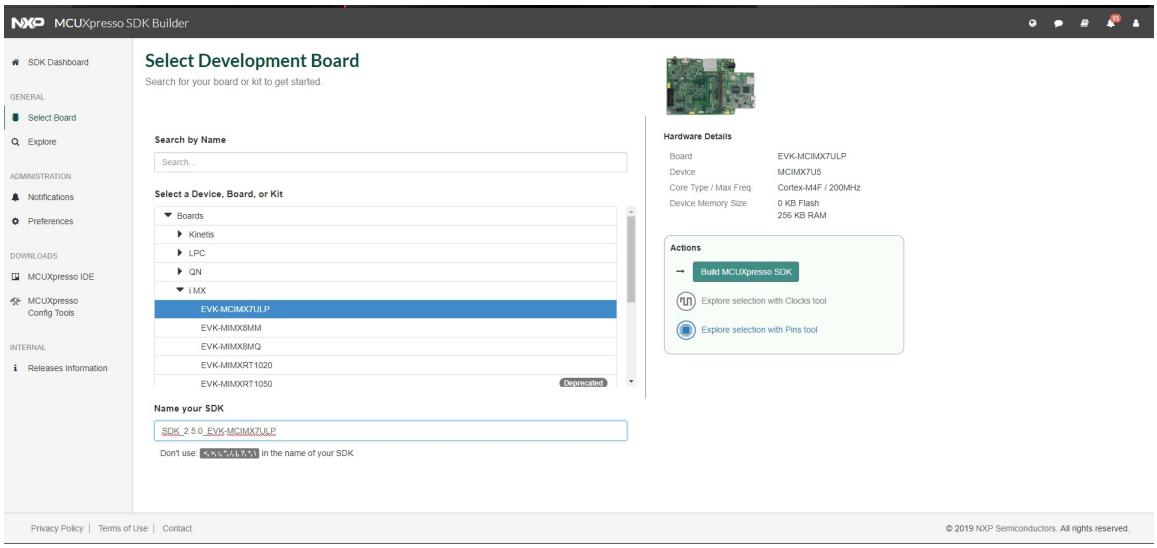


Obrázek 2.1: Sada nástrojů MCUXpresso Tools [23]

[1] https://en.wikipedia.org/wiki/ARM_Cortex-M
[2] https://www.nxp.com/docs/en/fact-sheet/IMXRTSERIESFS.pdf
[3] https://www.nxp.com/docs/en/brochure/BRKINLPCPWRMCU.pdf
[4] SDK – software development kit, více na https://en.wikipedia.org/wiki/Software_development_kit
[5] https://www.nxp.com/docs/en/fact-sheet/MCUXPRESSOIDEFS.pdf

2.1 SDK Builder

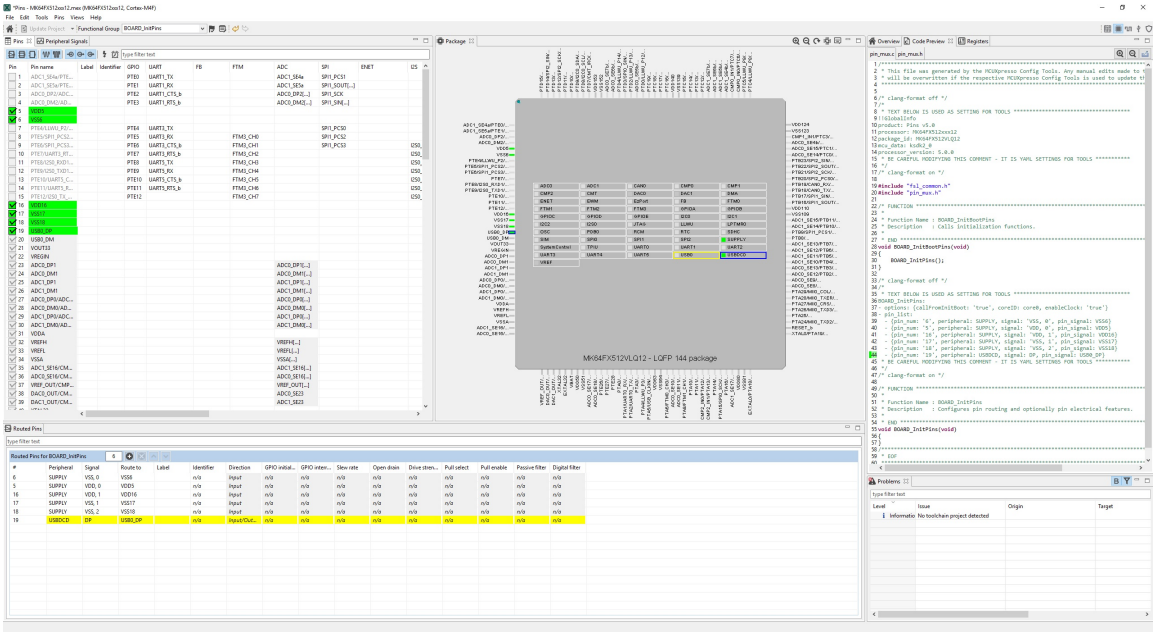
`MCUXpresso SDK Builder` slouží, jak již napovídá název, ke generování vlastních `SDK` balíčků. Primárním cílem tohoto nástroje je umožnit uživateli si vytvořit specifickou konfiguraci `SDK` balíčku pro svůj projekt, bez nutnosti si ho stahovat celý, protože se může jednat o značně velký soubor. Tento nástroj nám umožňuje si vybrat mikroprocesory, desky nebo kity, se kterými chceme pracovat viz. obrázek 2.2. Následně si vybrat operační systém a toolchain, který budeme používat. Další možností tohoto programu je přidání volitelné komponenty, jako například middleware, operační systém nebo softwarové knihovny. Poté si stačí vybraný balíček stáhnout. Tento nástroj je dostupný v online verzi [24].



Obrázek 2.2: Ukázka z nástroje MCUXpresso SDK Builder

2.2 Pins Tool

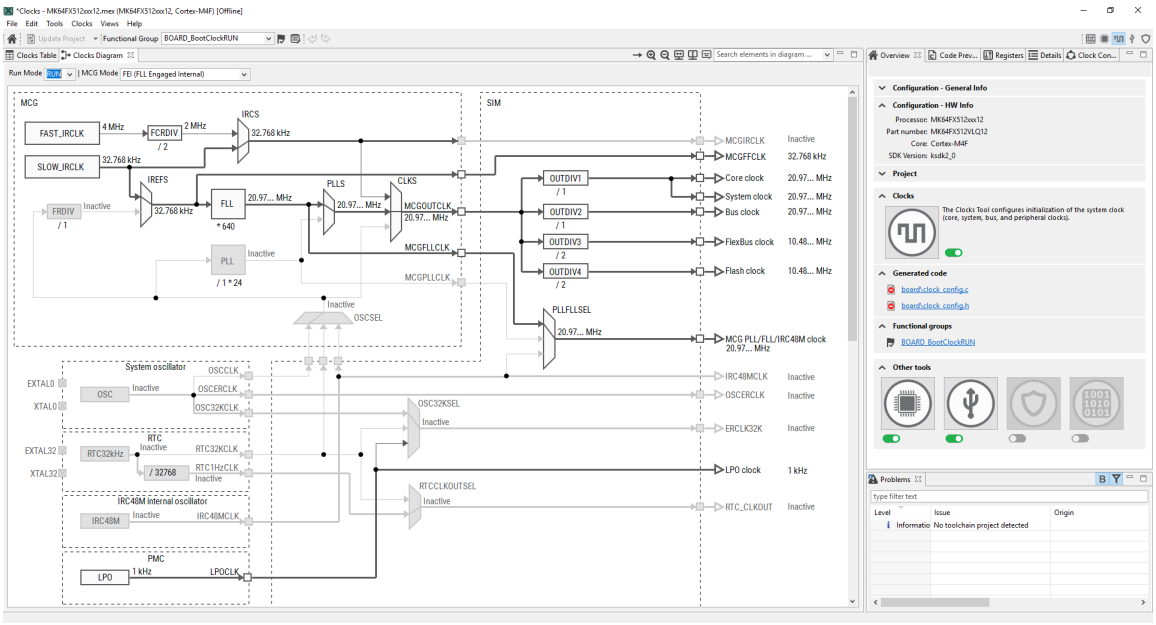
`Pins Tool` je nástroj sloužící pro konfiguraci jednotlivých pinů zařízení. Umožňuje vytvářet, prohlížet, měnit a modifikovat jakýkoliv element konfigurace pinů. Pomocí grafického rozhraní si jednoduše můžeme připojit jednotlivé signály na vybrané piny, či periferní zařízení, viz. obrázek 2.3. K přehlednosti nám pomáhá i grafické znázornění zařízení pomocí tzv. procesorového balíčku, kde vidíme jednotlivé `piny/periferie` [24] [27].



Obrázek 2.3: Ukázka z nástroje MCUXpresso Pins Tool

2.3 Clocks Tool

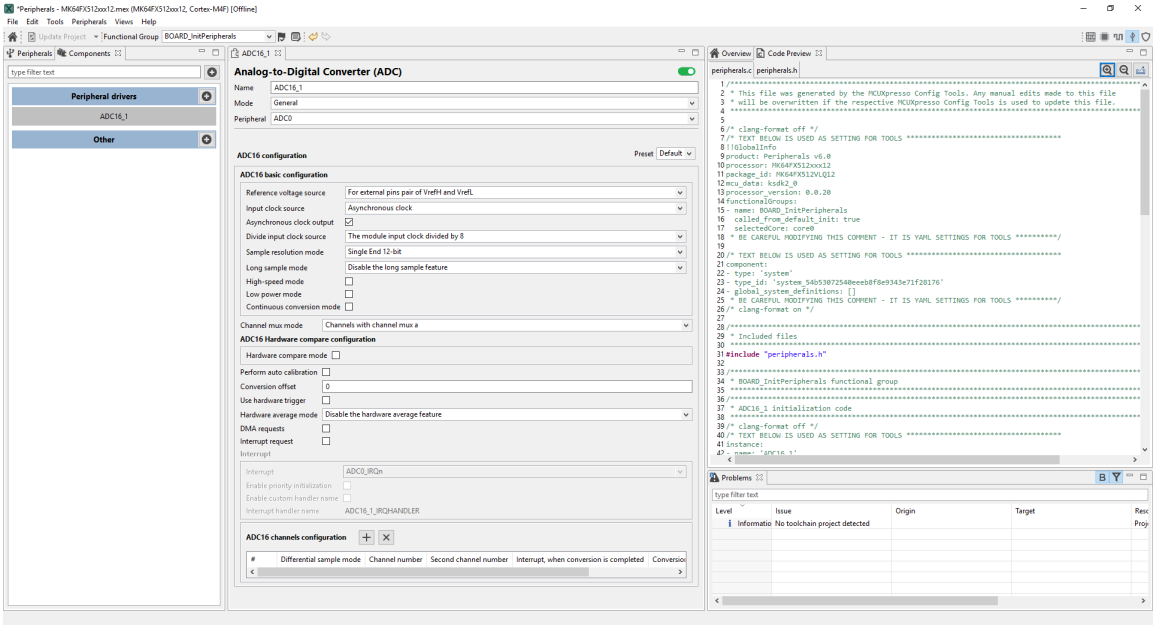
`Clocks Tool` je nástroj sloužící k zobrazení stromové struktury hodin v mikrokontrolerech. Taktéž slouží k nastavení a optimalizaci jednotlivých zdrojů a výstupů. Jednoduše si v něm lze nastavit hodnoty násobiček, sčítaček atd., k dosažení nejoptimálnějších výstupů pro konkrétní účel [24]. Pomocí grafického zobrazení lze mít snadno přehled, nad jinak komplikovanou strukturou cest jednotlivých hodin, viz. obrázek 2.4. Základní nastavení zdrojů hodin a výstupních frekvencí lze provádět v záložce `"Table view"`. Podrobnější nastavení pak najdeme v záložce `"Details view"`. Kontrolu a nastavení můžeme taktéž provádět v `"Diagram view"` [27].



Obrázek 2.4: Ukázka z nástroje MCUXpresso Clocks Tool

2.4 Peripherals Tool

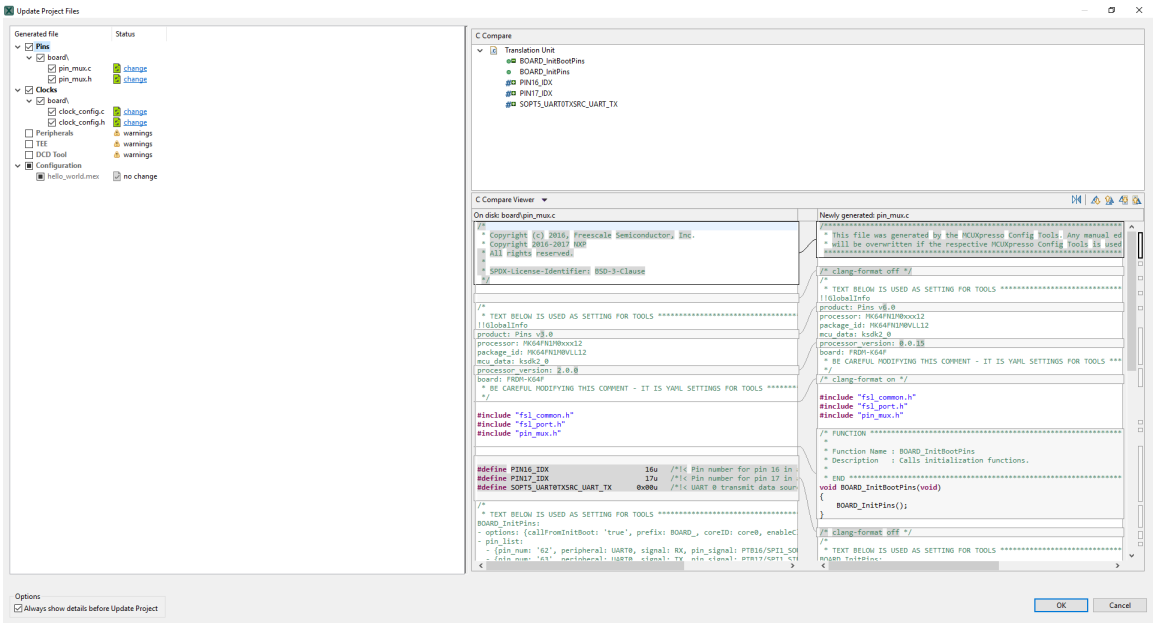
`Peripherals Tool` umožňuje uživatelům výběr chtěného periferního zařízení dle jejich návrhu, například `UART, ADC, SPI, I2C` ad. viz. obrázek 2.5. Nástroj vygeneruje inicializační struktury pro `MCUXpresso SDK` ovladače. Uživateli také nabízí konfiguraci vysokoúrovňového aplikačního kódu pro `USB` projekty. Dále také umožňuje rychlou validaci uživatelských nastavení, jejich bezkonfliktnost. A v případě objevení nějakého konfliktu nástroj uživatele vhodně upozorní [27][24].



Obrázek 2.5: Ukázka z nástroje MCUXpresso Peripherals Tool

2.5 Project Updater

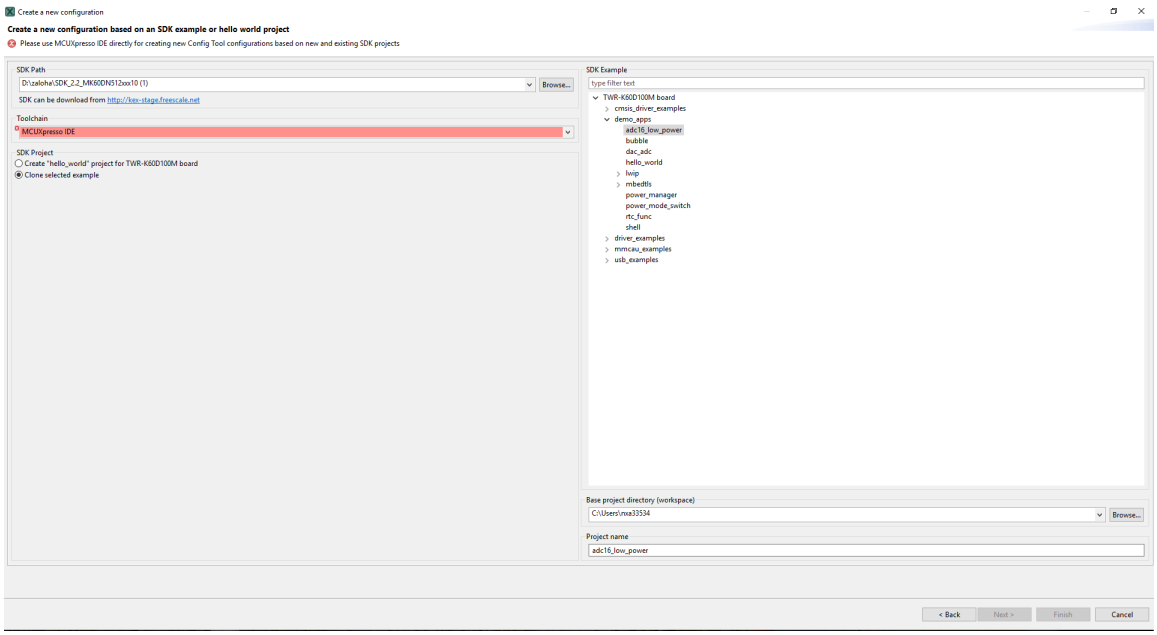
Tento nástroj pracuje přímo s existujícími `MCUXpresso IDE` projekty založenými na balíčku `SDK`. Aktualizuje zdrojové kódy projektu pro vygenerované zdrojové kódy z `Pins, Clocks a Peripherals` nástrojů viz. obrázek 2.6 [25].



Obrázek 2.6: Ukázka z nástroje Project Updater

2.6 Project Cloning

Díky tomuto nástroji je možné klonovat ukázkové SDK projekty pro IAR Embedded Workbench, Keil μVision a pro GCC ARM Embedded viz. obrázek 2.7. Výsledný projekt obsahuje všechny zdrojové soubory a knihovny pro sestavení projektu a může být jednoduše přizpůsoben, sdílen nebo vložen do systému správy verzí [25].



Obrázek 2.7: Ukázka z nástroje Project Clonig

2.7 Shrnutí

MCUXpresso Config Tools jsou celkově velmi užitečným pomocníkem při vývoji projektů pro vestavěné systémy. Díky intuitivnímu grafickému rozhraní umožňují snadnou konfiguraci potřebných nastavení bez hlubší znalosti jazyka C. Navíc jsou součástí souhrnné sady MCUXpresso softwaru a nástrojů a jsou kompatibilní se softwarovým vývojovým balíčkem MCUXpresso(SDK) a integrovaným vývojovým prostředím MCUXpresso (IDE). Díky tomu tvoří robustní a obsáhlý balík nástrojů, který může usnadnit a urychlit vývoj vestavěných systémů.

Kapitola 3

Automatizace testování

Testování softwaru je v dnešní době naprosto nezbytnou součástí životního cyklu vývoje. Samotné testování může probíhat na několika úrovních a přináší mnohá úskalí. V této kapitole si vysvětlíme základní způsoby testování a důvody pro a proti automatizaci.

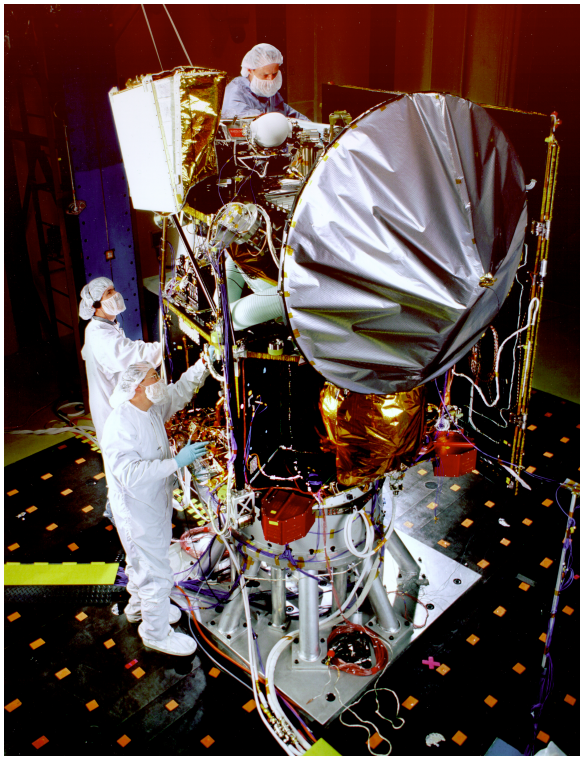
3.1 Testování a kvalita

Hlavní motivací pro testování je většinou ověření, že nám daný software bude fungovat správně. Chyby nebo selhání softwaru mohou mít drtivé následky. Pokud software nefunguje správně, může to vést ke ztrátě financí, času, reputace firmy nebo dokonce ohrožení zdraví, či životů [20].

Příklad velkého softwarového bugu:

Mars climate orbiter

Mars climate orbiter byla jedna ze dvou kosmických sond určených ke studování marťanského počasí a pátrání po vodě a oxidu uhličitém. Obrázek sondy 3.1. Při pokusu o přistání shořela. Při vyšetřování bylo zjištěno, že subdodavatel softwaru počítal v imperiálních jednotkách. Díky tomu došlo k vypočítání špatné trajektorie. Celková škoda byla více jak 327 milionů dolarů [30].



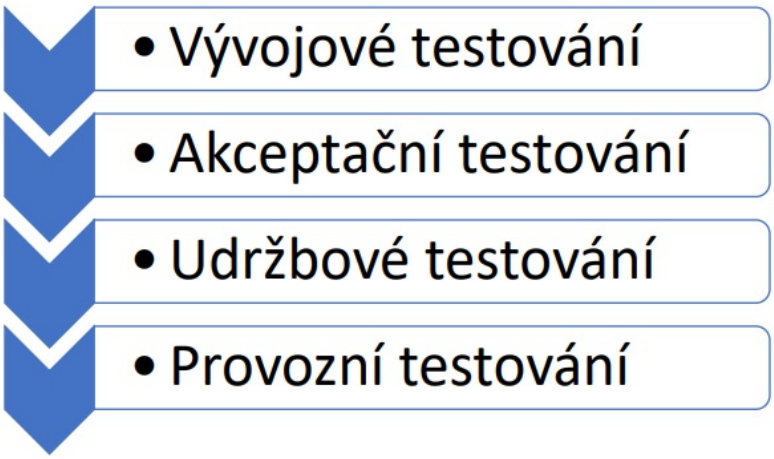
Obrázek 3.1: Ukázka kosmické sondy při testování [18].

Hlavní cíle testování:

- Nalezení defektů.
- Získání většího sebevědomí o úrovni kvality.
- Poskytnutí informací pro rozhodování.
- Předcházení defektům [20].

3.2 Fáze testování

Stejně jak vývoj softwaru prochází určitým cyklem, tak souběžně s ním máme i různé fáze testování. V průběhu vývoje se tudíž mění i motivace pro vytvoření testů. Na začátku a v průběhu vývoje bude jiná motivace, než před vydáním, jiná bude také po vydání a v průběhu údržby viz. obrázek 3.2. Proto si vysvětlíme jednotlivé fáze, ke kterým přiřadíme druhy testů v nich používaných.



Obrázek 3.2: Průběh fází testování.

Vývojové testování

Tato fáze testování většinou probíhá při vývoji. Hlavní motivací je zde oprava chyb ještě před dokončením softwaru. Do této fáze spadají například [20][6]:

- `unit testy` - Píší je většinou programátoři a jejich cílem je otestování určité jednotky (`unity`). Jedná se o testování základní funkčnosti kódu.
- `integrační testy` - Píší je většinou testeři a jejich cílem je ověření funkčnosti rozhraní mezi jednotlivými komponenty.
- `systémové testy` - Píší je většinou testeři a jejich cílem je ověření, zda systém vyhovuje požadavkům a specifikaci. Jedná se o nejvyšší úroveň testování v rámci vývoje.

Akceptační testování

Fáze, která probíhá před vydáním do produkce. Důležitou roli zde hraje zákazník (koncový uživatel). Cílem je ověření, že systém pracuje, tak jak se očekává a splňuje požadavky zákazníka [6].

Údržbové testování

Údržbové testování probíhá zejména při různých změnách po vydání. `Patche, upgrady`...atd. Hlavní zaměření je na novou funkcionalitu a `regresní testy`. Cíl je potvrzení, že dané změny nezpůsobí nové chyby [6].

Provozoní testování

Jedná se o jistý druh `akceptačního testování`, které se nezaměřuje na nalezení chyb, ale čistě na splnění požadavků. Zejména zajištění požadavků na provoz (dostupnost, spolehlivost atd.) Provádí se většinou těsně před uvedením do provozu/koupí od zákazníka [6].

3.3 Automatizované testování

Cílem automatizace testování je ušetření lidských a časových zdrojů na repetetivní, zejména `regresní testování`, abychom tyto zdroje mohli využít efektivněji. Důležité je zejména zvýšení `efektivnosti práce`. Automatizace je však často nákladný proces, který sám o sobě spotřebuje nemalé množství lidských, časových a často i finančních zdrojů. Proto je velmi důležitým bodem před zavedením automatizace důkladná `analýza aktuálního stavu` a zhodnocení, zda se automatizace vyplatí.

Dobrým příkladem kdy je vhodné použít automatizaci, je například pokud firma vydává v pravidelných intervalech nové verze softwaru. Na otestování základní funkčnosti máme určitou sadu `manuálních testů`, které testeři vykonávají před každou verzí a nových testů přibude vždy jen pár. V tomto okamžiku nám sice zavedení automatizace zabere na určitou dobu většinu testerů, ale vynaložené počáteční prostředky se nám brzy vrátí. `Automatizované regresní testy`, pak lze spustit kdykoliv a na údržbu a vývoj nových testů nám stačí menší zdroje.

Automatizaci testování lze dnes provádět mnoha způsoby. Od jednoduchých skriptů po složité nástroje. Záleží, o jaký druh testování se bude jednat. Níže jsou k jednotlivým fázím testování přiřazeny jednotlivé možnosti automatizace.

Vývojové testování

Sestavení regresní sady `unit` testů a její automatické spouštění. Využití nástrojů pro automatické generování `unit` testů. Propojení s nástroji pro kontinuální integraci a spouštění v pravidelných intervalech [6].

Akceptační testování

Testování požadavků pomocí `testování grafického rozhraní`. Na testování grafického rozhraní lze využít různé nástroje. Jedním z nejpoužívanějších nástrojů pro testování webových aplikací je například `Selenium`[1]. Pro aplikace napsané v `javě`, pak například `SWTBot`[2]. Jejich hlavní činností je simulace uživatele, kdy pomocí předprogramovaných scénářů provádějí úkony, které by prováděl i uživatel. Zejména při nasazování těchto nástrojů je potřeba provést podrobnou analýzu jejich užitečnosti. Jejich nasazení se vyplatí spíše u projektů, kde se v budoucnosti počítá s více verzemi produktu, nebo se jedná o velký projekt, kde je potřeba často spouštět regresní testy. Pokud například budu vyvíjet jednoúčelovou aplikaci, která bude mít omezený počet dostupných prvků grafického rozhraní, tak bude určitě efektivnější testování manuální [20].

Údržbové a provozní testování využívá podobné nástroje, jelikož se často v těchto fázích pouze spouští regresní sada automatických testů napsaných, již v předchozích fázích.

[1] https://www.seleniumhq.org/
[2] https://www.eclipse.org/swtbot/

Kapitola 4

Analýza stávajícího řešení

Tato kapitola se zabývá analýzou stávajícího řešení. Analýza je rozdělena do dvou částí. V první je analýza návrhu a výkonu testovacího frameworku `MCUX Builder`. V druhé je následně popsán způsob aktuálního nastavení a fungování. Jelikož je testovací framework `MCUX Builder` spouštěn na vzdálených virtuálních systémech, přes automatizační server `Jenkins` je znalost nastavení prostředí klíčová. Analýza návrhu je zaměřena zejména na efektivní využití jednotlivých tříd a udržitelnost kódu. Analýza výkonu je pak zaměřena na hledání výkonnostních problémů.

4.1 Analýza objektově orientovaného návrhu

Tato sekce se zabývá analýzou stávajícího frameworku `MCUX Builderu` po stránce objektově orientovaného návrhu. Hlavním cílem této analýzy je zhodnotit aktuální stav a nalézt případné problémy, jejichž vyřešení by mohlo dopomoci k vytvoření přehlednějšího a robustnějšího kódu.

4.1.1 Objektově orientovaný přístup

„Objektově orientovaný přístup k programování je založen na intuitivní korespondenci mezi softwarovou simulací reálného systému a reálným systémem samotným. Analogie je především mezi vytvářením algoritmického modelu skutečného systému ze softwarových komponent a výstavbou mechanického modelu pomocí skutečných objektů. Podle této analogie i ony softwarové komponenty nazýváme objekty. Objektově orientované programování pak zahrnuje analýzu, návrh a implementaci aspektů, kde jsou reálné objekty nahrazeny těmi softwarovými (virtuálními).“[31, str. 13]

Principy objektově orientovaného přístupu v pythonu

Koncept `OOP`[1] se v pythonu zaměřuje na vytváření znovupoužitelného kódu. Tento koncept je také známý jako `DRY`(Don’t Repeat Yourself) [22].

V pythonu koncept OOP dodržuje několik základních principů.

[1] OOP - objektově orientované programování, více viz. https://cs.wikipedia.org/wiki/Objektově_orientované_programování

Dědičnost

Proces používání detailů z nové třídy bez vytváření existují třídy. O mateřské třídě se někdy hovoří jako o předkovi a o třídě, která z ní dědí, jako o potomkovi. Potomek může přidávat nové metody nebo si uzpůsobovat metody z mateřské třídy. Je možné se setkat i s pojmy nadtřída a podtřída. Hlavní přínos je možnost rozšiřovaní existujících komponent o nové metody a tím je znovu využívat. Lze se tak vyhnout psaní redudantního kódu. Pokud dojde ke změně atributu v mateřské třídě, tak se tato změna automaticky podědí, což zajistí, že tato změna nemusí být prováděna u dalších několika tříd ručně [22] [11].

Zapouzdření

Schování privátních detailů třídy před dalšími objekty. Díky zapouzdření není třeba znát detaily implementace, ale stačí znát a používat rozhraní objektu. Díky zapouzdření je také možné změnit implementaci, pokud je použito stejné rozhraní objektu. V pythonu se označuje privátní atribut pomocí prefixu v podobě podtržítka, tj. „_“ nebo „___“ [22] [11].

Polymorfismus

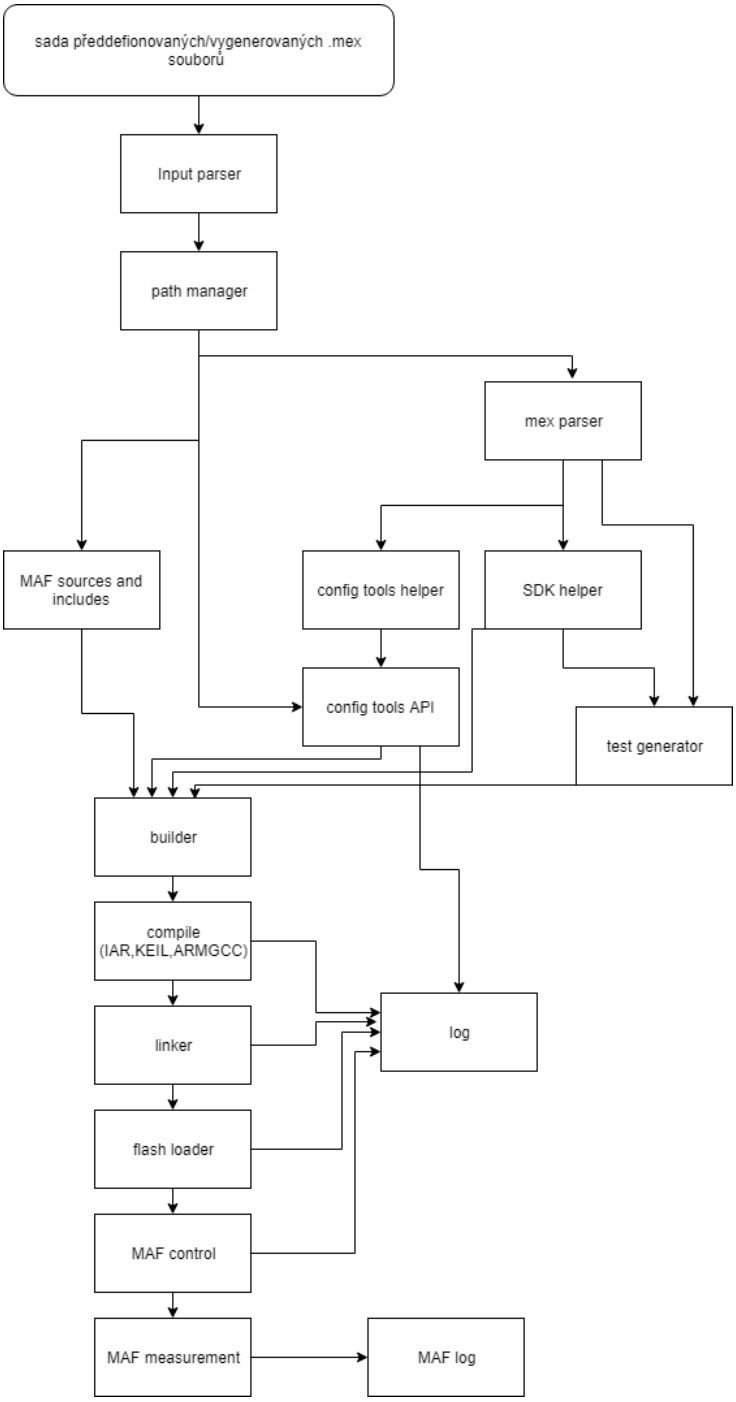
Polymorfismus umožňuje používat jednotné rozhraní pro práci s různými typy objektů. Ukázkou může být situace, kdy je potřeba nakreslit geometrický tvar. Je na výběr několik možností (čtverec, trojůhelník, kruh). Nicméně pro nakreslení tvaru lze použít stejnou metodu jako pro nakreslení jakéhokoliv tvaru. Polymorfismus v praxi znamená, že je možné mít různé třídy s metodami se stejnými parametry. Lze tak odlišit chování potomků [22].

4.1.2 MCUX Builder

`MCUX Builder` je nástroj sloužící ke zpracování zdrojových souborů (výstup z konfiguračních nástrojů `MCUXpresso Config Tools`). Jeho cílem je automatizované testování dat generovaných `MCUXpresso Config Tools`. Představa celého procesu je následující – automatické stažení dat (část v `MCUXpresso Config Tools`), nahrání předpřipravených konfigurací, vygenerování a uložení zdrojových souborů, kompilace se všemi nezbytnými definicemi a příznaky, vygenerování testovací aplikace, nalinkování a případné spuštění na reálném hardwaru. Vzhledem ke komplikovanosti tohoto konceptu jako celku, bylo rozhodnuto o jeho rozdělení do jednotlivých modulů, z nichž každý bude obstarávat určitou část.

Rozhraní pro programování aplikací MCUX Builderu

Vzhledem ke složitosti celého návrhu je `API(rozhraní pro programování aplikací)` vyjádřeno diagramem jednotlivých modulů `MCUX Builder` frameworku viz. obrázek 4.1.



Obrázek 4.1: API MCUX Builder frameworku

4.1.3 Moduly frameworku

V této části jsou popsány všechny důležité moduly `MCUX Builder` frameworku, které jsou klíčové pro pochopení jeho funkcionality.

input parser:

Tento modul se stará o zpracování jednotlivých argumentů zadaných z příkazového řádku.

Všechny vstupní parametry jsou zpracovány pomocí `input_parseru` (za pomocí `Argument-Parser` modulu). Jelikož jsou cesty ze vstupních parametrů registrovány v `PathManageru`, tak se zde vytváří instance třídy `PathManager`. Díky tomu jsou cesty přístupné ze všech dalších modulů.

path manager:

V tomto modulu je ošetřeno ukládání a zpracovávání cest pro spustitelné soubory, soubory obsahující konfigurace ze všech nástrojů `MCXUpresso Config Tools`, `.MEX soubory`[2], připojené soubory a `SDK` balíčky.

Pro zajištění, že všechny cesty jsou uložené na jednom místě je potřeba vytvořit speciální třídu `path_manager`. Kvůli zajištění pouze jedné instance, této třídy je použit objektový návrh jedináček. V průběhu vytvoření instance třídy jsou hledány a přiřazeny cesty ze systémových proměnných. Jakmile jsou zpracovány všechny cesty ze vstupních parametrů, proběhne jejich registrace právě v `path_manageru`.

MEX parser:

Tento modul se stará o získávání informací z `.MEX` souboru. Třída, která zajišťuje funkčnost tohoto modulu se nazývá `mex_parser`. V této třídě jsou všechny metody pro manipulaci s `.MEX` soubory. V průběhu procesu sestavování jsou požadovaná určitá data z `.MEX` souboru. Například je potřeba znát název procesoru, desky a nastavení periferií. Ke zpracování `.xml`[3] formátu(používány v `.MEX` souboru) a získání potřebných informací je použit `ElementTree`[4].

Config Tools Helper:

Tato část se stará o sestavení a spuštění příkazů rozhraní příkazového řádku pro `MCUXpresso Config Tools`.

SDK helper:

Tento modul slouží k získávání informací nezbytných k procesu sestavení programu s `SDK` repositáře. `MCUXpresso Config Tools` jsou úzce spojeny z `MCUXpresso SDK`[5]. Nastavení pinů, periferií a hodin v `MCUXpresso Config Tools` generuje kód s konfiguračními strukturami používanými v inicializačních funkcích z `MCUXpresso SDK`.

Přehled metod v této třídě:

- `get includes` - Tato metoda vrací pole souborů se závislostmi obsahující: společné ovladače (`common, gpio, port..`), ovladače nakonfigurované v `peripherals toolu`, defaultní závislosti (`CMSIS, debug_console`) a výstup, kam budou generovány zdrojové soubory `MCUXpresso Config Tools.`

[2] MEX(Microcontrollers Export Configuration)-soubor pro konfigurace mikrokontrolerů
[3] XML – Extensible Markup Language, více viz https://www.w3.org/XML/
[4] https://docs.python.org/2/library/xml.etree.elementtree.html
[5] https://www.nxp.com/docs/en/fact-sheet/MCUXPRESSOSDKFS.pdf

- `get_flags` - Tato metoda získá kompilační příznaky ze souboru `.json`[6] pomocí `flags_storage modulu` (popis výše).
- `get_defines` - Vrátí definice pro specifický procesor/desku. Skládá se z defaultních definic (`„DEBUG"..`) a `CPU` definice, která je složena z prefixu `„CPU_"` a jména balíčku. Pokud je zařízení vícejádrové je přidán suffix `„_core"`.
- `get_linker_file` - V této metodě se získá soubor pro linkování, který je použit pouze při sestavování aplikace (obsahuje linkovací fázi). Správný linkovací soubor je hledán v `SDK` repozitáři.
- `get_sdk_files_to_build` - Tato metoda vrací soubory s `SDK` k sestavení aplikace. Záleží na specifické aplikaci, se kterou je potřeba linkovat. Obsahuje: ovladače perifierií, adresář zařízení, `startup`, `system`, `fsl_clock...`

Config Tools API

Základní rozhraní pro programování aplikace `MCUX Builder` sestává ze třídy `config_tools_builder`, která obsahuje třídu `main`.

V tomto modulu jsou z `.MEX` souborů vygenerované zdrojové soubory, skrze `MCUXpresso Config Tool`. Následně jsou pro nakonfigurované periferní zařízení generovány testovací aplikace. Zdrojové soubory jsou poté přeloženy všemi dostupnými kompilátory. Pokud jsou vygenerované testy, tak jsou testovací aplikace zkompilovány a slinkovány s objektovými soubory. Výstupem jsou testovací aplikace v binární podobě, které budou spuštěny na `hardwaru`.

parametry:

```
usage:
config_tools_builder [-h] [-c CT_DIR] [-x CT_IMX_DIR] [-i IAR_DIR]
                     [-k KEIL_DIR] [-m MCUX_DIR] [-a ARM_GCC_DIR]
                     [-s SDK_DIR]
                     [-t {iar,keil,mcux,armgcc} [{iar,keil,mcux,armgcc} ...]]
                     [-v {jenkins,maf,saleae,verify}] [-o OUTPUT_DIR]
                     [-l {0,1,2,3}] [-rd] [-mp MAF_DIR]
                     [-ac ALL_CONFIG] [-cl CPU_LIST_FILE]
                     [-f MEX_FILE | -d MEX_DIR | -sf SOURCES_DIR]
```

`-c, -config_tool_dir`

Adresář s nainstalovaným `MCUXpresso Config Tools`, pokud zůstane prázdný, tak se použije proměnná env[7] `MCUX_CONFIG_DIR`.

`-x, -config_tool_imx`

Adresář s nainstalovanou `MCUXpresso Pins Tool for i.MX`[8] aplikací, pokud zůstane prázdný, tak se použije proměnná env `MCUX_IMX_DIR`.

`-i, -iar_dir`

Adresář s instalací kompilátoru `IAR`, pokud zůstane parametr prázdný, tak se použije pro-

[6] JSON – JavaScript Object Notation, více viz https://www.json.org/json-cz.html
[7] env – environment variable, více viz https://en.wikipedia.org/wiki/Environment_variable
[8] https://www.nxp.com/docs/en/user-guide/IMXPINSQSUG2.pdf

měnná env IAR_ARM_DIR.

-k, -keil_dir
Adresář s instalací kompilátoru Keil, pokud zůstane prázdný, tak se použije proměnná env KEIL_DIR.

-m, -mcux_dir
Adresář s instalací MCUXpresso IDE, pokud zůstane parametr prázdný použije se proměnná env MCUX_DIR.

-a, -arm_gcc_dir
Adresář s instalací ARM gcc, pokud zůstane parametr prázdný použije se proměnná env ARM_GCC_DIR.

-s, -sdk_dir
Cesta k SDK repozitáři, pokud zůstane parametr prázdný použije se proměnná env SDK_DIR.

-t, -toolchains
Výběr toolchainu, pokud parametr zůstane prázdný vyberou se všechny. Možnosti: iar, keil, mcux, armgcc.

-v, -variant
Výběr způsobu běhu programu, pokud parametr zůstane prázdný nevybere se žádný. Možnosti: jenkins, maf, saleae, verify.

-o, -output_dir
Cesta k výstupu, následná operace vytvoří podsložky <board>_<tool>.

-l, -logging_level
Úroveň logování: 0=DEBUG, 1=INFO, 2=WARNING, 3=ERROR. Výchozí je INFO. Možnosti: 0, 1, 2, 3.

-rd, -remove_data
Před startem smaže lokální data v lokaci: C:\ProgramData\NXP\mcu_data_v4 Výchozí: False.

-mp, -maf_dir
Závislosti pro MAF a zdrojový adresář.

-ac, -all_config
Vygeneruje pro všechny zdrojové soubory a všechny závislosti.

-cl, -cpu
list_file
Cesta k souboru se seznamem procesorů pro vygenerování zdrojových a .MEX souborů.

`-f, -mex_file`
Cesta k `.MEX` souboru.

`-d, -mex_dir`
Cesta k adresáři s `.MEX` soubory.

`-sf, -sources_dir`
Cesta k adresáři se zdrojovými soubory a `.MEX` soubory.

Příklad použití:

```
config_tools_builder.py -m
C:\nxp\MCUXpressoIDE_10.2.0_759\ide\
-s C:\mcu-sdk-2.0 -t armgcc -l 1
-mp C:\MCUX_Builder\MAF -ac no -v maf
-f C:\MCUX_Builder\test\mex\MAF\TWR-K64F120M_LLWU.mex
```

config tool helper

`Config tools helper` modul z diagramu tvoří hlavně třída `configToolsHelper`. Používá rozhraní příkazové řádky z `MCUXpresso Config Tools` k vygenerování zdrojových souborů z `.MEX` souborů. Sestavuje příkaz a volá `MCUXpresso Config Tools` z předdefinovanými paramatery.

test generator:

Tento modul je použit ke generování testovacích aplikací pro nakonfigurované periferie. Skládá se z rodičovské třídy `TestGenerator` a potomků (ty jsou rozděleny podle periferie). Modul si bere nezbytné informace z `.MEX` souborů (nastavení periferie, `init clock` funkce,..atd.) a vyplňuje korespondující šablonu. Pro tento účel je použit `Tenjin`[9] šablonový engine. Vygenerovaná testovací aplikace je uložena.

builder

Tento modul reprezentuje třída `Builder`. Celkem je zde jedna rodičovská třída `Builder` se společným nastavením. Následně jsou zde synovské třídy `IARBuilder, KeilBuilder, MCUXBuilder a ARMGCCBuilder` závisející na toolchainu. V tomto modulu je využit princip dědičnosti a taktéž princip polymorfismu, když jednotlivé synovské třídy obsahují stejnou metodu `link`. Nejdříve musí být nastaveny všechny parametry jako jsou závistlosti, definice, zdroje, `linker` soubory atd... Teprve poté můžeme spustit metodu `compile`. Po zavolání `compile` je sestaven a spuštěn příkaz pro kompilátor. V tomto modulu je využit i pricip zapouzdření, kdy jsou metody pro sestavení příkazů nastaveny jako privátní.

Příkazy builderu:

```
clean_build clean + compile + link
clean_compile clean + compile
```

[9] http://www.kuwata-lab.com/tenjin/

`build compile + link`

- `clean` - Smaže všechno ve výstupním adresáři.
- `compile` - Přidá prefixy do definic a závislostí, vezme všechny zdrojové soubory, sestaví pro ně příkazy a spustí je.
- `link` - Přidá prefixy do souborů pro linker, sestaví `ld` příkazy a spustí je.
- `add_source_files` - Přidá soubory k sestavení. Je možné je přidat jako soubor, pole, slovník nebo pole slovníků.
- `reset_source_files` - Vymaže všechny zdrojové soubory, používá se při sestavování více testovacích aplikací.
- `set_params` - Nastaví všechny nezbytné parametry pro sestavení (závislosti, definice, příznaky, linker soubory, zdrojové soubory, výstupní adresář, název aplikace).

Jelikož kompilační příznaky nejsou uloženy v `.yaml`[10] souboru z `SDK` generátoru ve správném formátu, tak musí správa kompilačních příznaků přejít na `MCUX Builder`. Bylo rozhodnuto, že budou uloženy v `.json` formátu. Každý toolchain má své vlastní příznaky pro `assembler(AS), C(CC), CPP(CX) a linker(LD)`. Při každém buildu generovaných zdrojových souborů jsou načítany správné příznaky z `.json` souboru. Společné příznaky a příznaky specifické pro jádro jsou spojeny a použity k vygenerování příkazu pro sestavení programu.

compile

Tento modul používá různé toolchainy k sestavení všech `.c` souborů.

linker

Nalinkuje všechny soubory `.o`[11] do jednoho binárního souboru `.elf`[12].

flash loader

Převede pro daný cíl binární `.elf` soubor do spustitelného `.bin` souboru. Binární soubor se `flashuje` pomocí `JLINK/Open SDA`[13].

MAF control

Tento modul poskytuje rozhraní pro řizení `MAFu` přes `TCP/IP` sokety.

[10] YAML – Ain't Markup Language, více viz https://cs.wikipedia.org/wiki/YAML
[11] .o – object file, více viz https://en.wikipedia.org/wiki/Object_file
[12] .elf – Executable and Linkable Format, více viz https://cs.wikipedia.org/wiki/Executable_and_Linkable_Format
[13] https://www.segger.com/products/debug-probes/j-link/

MAF measurement

Tento modul obsahuje rozhraní pro zachytávání `I2C, UART a SPI` pomocí `Saleae` logiky. Což znamená automatické měření elektrických charakteristik periferních zařízení `(UART, I2SC, PI, PIT, LPTMR, GPIO, ADC)`

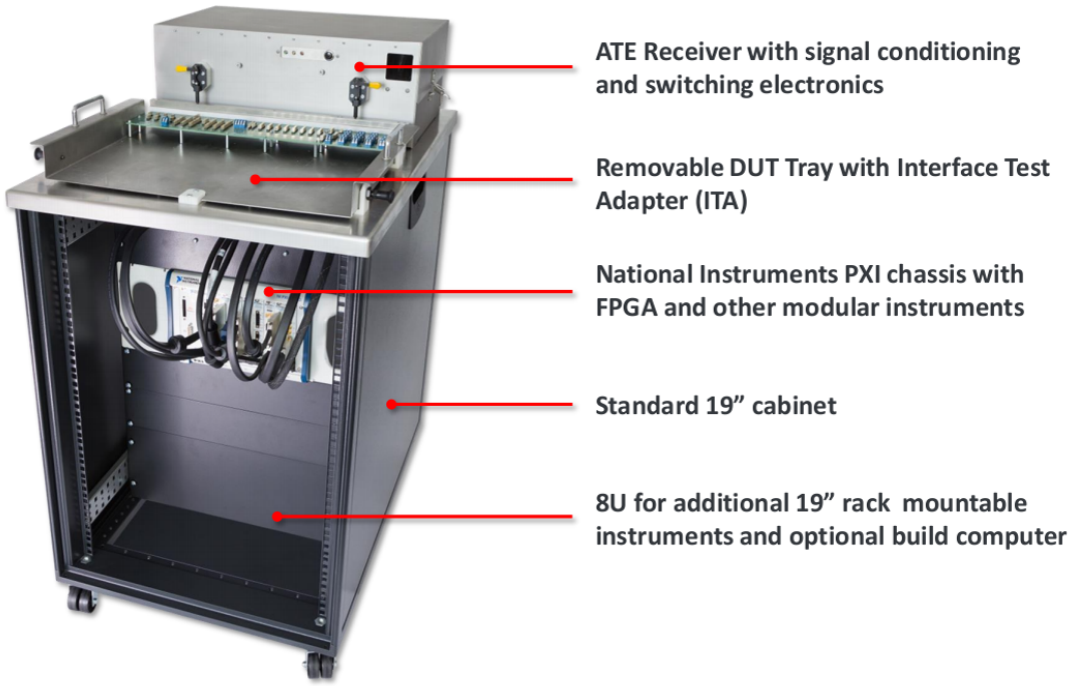
`Saleae` modul je použit pro kontrolu Saleae logického analyzátoru, který může být použit pro `hardwarové` testování mikrokontrolerů.

log

Modul `log` zajišťuje třída `Logger`, která je nakonfigurována, tak aby logovala zprávy typu `debug/info/warning/error` do konzole, ale stejně tak do předem definovaného souboru. Jednotlivé loggery jsou rozlišeny pomocí jména. Základní logovací level je nastaven jako `INFO`.

MAF

MAF je zkratka pro framework na měření a automatizaci `(Measurement and Automation Framework)`. `MAF` byl vyvinut v rožnovské pobočce firmy `NXP` jako univerzální tester s ohledem na automobilový průmysl.



Obrázek 4.2: MAF

Funkce:

- Postaven na National Instruments PXI[14],

[14] http://www.ni.com/cs-cz/shop/pxi.html

- konstrukce na bázi FPGA[15] pro maximální flexibilitu,
- programovatelné zdroje napájení,
- integrace s HSDIO[16],
- k dispozici jsou adaptéry pro rozhraní médií CAN, LIN a FlexRay,
- odolný konektor ITA(Iterface Test Adapter) > 10k cyklů,
- montář ve standartní 19"skříni,
- rozšiřitelný (PXI nebo 19"rack mountable instruments).

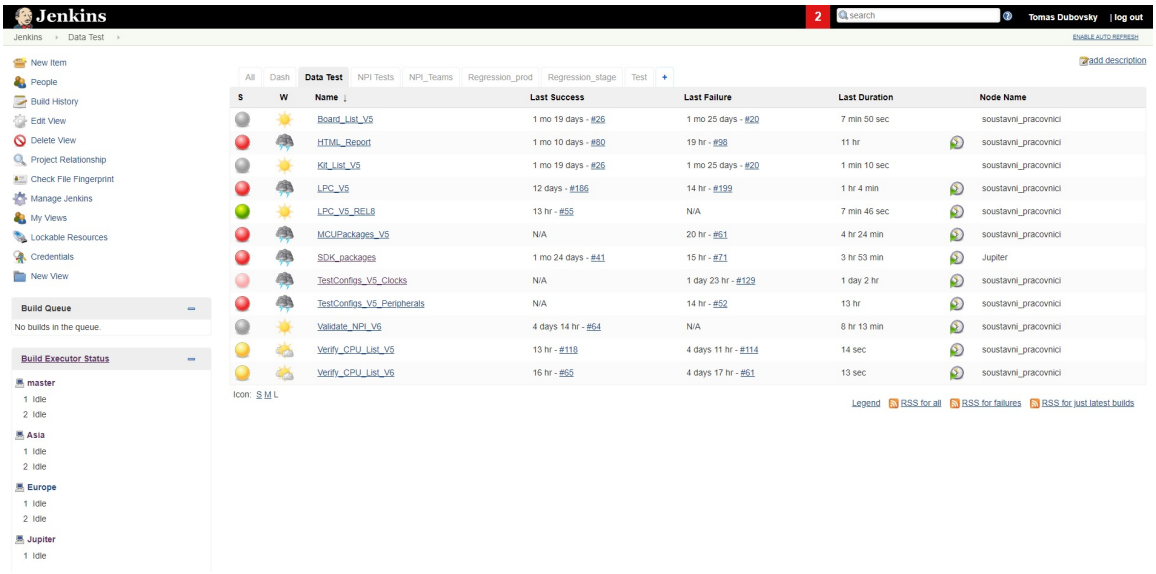
4.2 Jenkins

`Jenkins` je samostatný, open source automatizační server, který může být použit k automatizaci všech druhů úkolů spojených s vývojem, testováním a poskytováním nebo nasazováním `softwaru`. Jeho využití je zejména při `kontinuální integraci` [14].

V aktuálním nastavení `Jenkins` serveru pro `MCUX Builder` je několik jobů, které automaticky spouští `MCUX builder` podle jednotlivých testů viz. obrázek 4.3.

Toto řešení je výhodné, jelikož lze nastavit interval spouštění jednotlivých jobů dle jejich důležitosti. Taktéž lze efektivně rozložit zátěž na různé virtuální počítače.

Momentálně jsou používány celkem čtyři druhy jobů, které používají `MCUX Builder`. Ostatní joby, zde nejsou blíže popsány, jelikož nevyužívají testovací framework `MCUX Builder` a pro tuto práci jsou tudíž bezpředmětné.



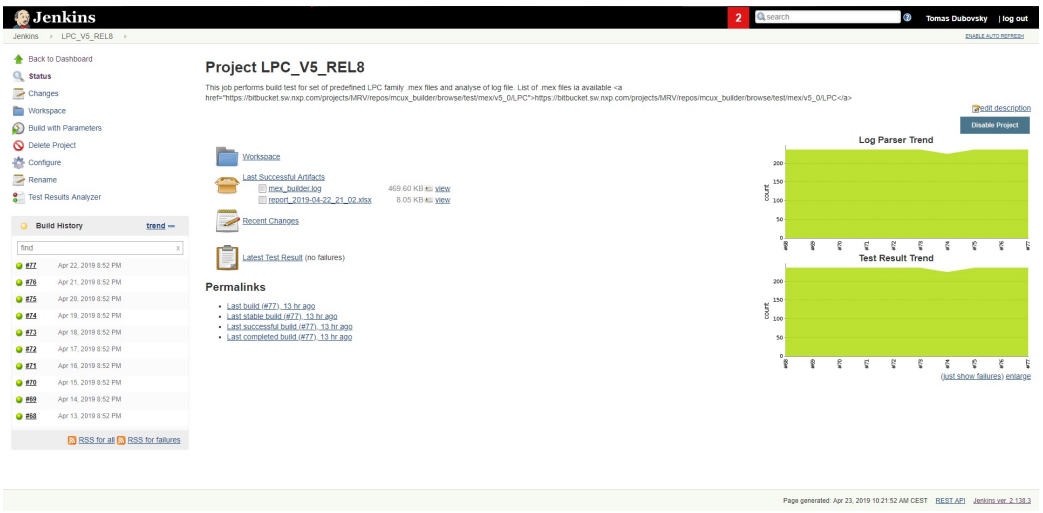
Obrázek 4.3: Ukázka prostředí Jenkins

[15] FPGA – Field Programmable Gate Array, více viz https://cs.wikipedia.org/wiki/Programovatelné_hradlové_pole

[16] http://sine.ni.com/nips/cds/view/p/lang/cs/nid/13556

4.2.1 MEX job

Tento druh jobů se stará o ověření `.MEX` souborů, generování zdrojových souborů a kompilace s vybranými toolchainy. Příkladem je například job `LPC_V5_REL8`, který má za cíl spustit test sestavení aplikace pro předdefinované `.MEX` soubory z rodiny mikrokontrolerů `LPC` viz. obrázek 4.4.



Obrázek 4.4: Ukázka .mex jobu

4.2.2 Test all config job

V tomto jobu se s využitím interního pluginu `MCUXpresso Config Tools` vygenerují všechny možné kombinace konfigurací pro periferní zařízení a hodiny. Následně se vygenerují všechny zdrojové soubory a spustí se kompilace s vybranými toolchainy.

4.2.3 All Pins job

Při běhu tohoto jobu se vygenerují všechny možné konfigurace pro piny za použití interního pluginu pro `MCUXpresso Config Tools`. Následně se vygenerují všechny zdrojové soubory a spustí se kompilace s vybranými toolchainy.

4.2.4 SDK Packages job

Tento job má za cíl ověřit funkčnost ukázkových aplikací z `SDK` balíčků po aktualizaci jejich kódu pomocí `MCUXpresso Config Tools`. Nejdříve se stáhne `SDK` balíček a následně se aktualizují zdrojové soubory pomocí `MCUXpresso Config Tools`. S takto aktualizovanými kódy se pak spustí build nad všemi podporovanými toolchainy – `IAR, KEIL, ARMGCC`.

4.2.5 Prostředí

K samotnému spouštění se používá osm virtuálních počítačů. Na každém počítači je stejné prostředí v podobě systému `windows server`. Všechny počítače mají čtyř-jádrové procesory.

4.3 Výkonnostní a zátěžové testy

Vzhledem k počtu zařízení, které má framework za cíl otestovat je velmi zásadní doba běhu samotného programu. Jakousi pomyslnou hranicí je doba běhu osm hodin. To je doba, kterou by framework neměl překročit ani při rapidním nárůstu dat k otestování. Právě osm hodin je stanoveno z několika hledisek [5]:

- Framework běží přes noc a tester potřebuje výsledky nejpozději ráno následujícího dne.
- Testování musí být kontinuální a změny provedené vývojáři se testují každý den.
- V případě úprav musí být možnost otestovat funkčnost celku za rozumnou dobu.

Pro účely testování byl vybrán v `MCUX Builderu` typ jobu `Kinetis_V5`. Tento job provádí build test pro sadu předdefinovaných `.MEX` souborů pro mikroprocesory rodiny `Kinetis`.

Tento zástupce byl vybrán, jelikož pokrývá hlavní funkcionalitu celého `MCUX Builderu`.

4.3.1 Profilování v pythonu

Jelikož je testovací framework `MCUX Builder` psán celý v programovacím jazyce `Python`, tak se při analýze výkonu zaměřuje v této práci na profilování v `pythonu`. Tato kapitola se zabývá možnostmi profilování. Dle dokumentace jsou v `pythonu` dostupné tři moduly pro profilování: `cProfile`, `profile` a `hotshot`.

Při profilování je cílem získat jistý profil o programu. Tento profil bude obsahovat sadu statistik, které budou popisovat jak často a jak dlouhou byly různé části programu spouštěny [29, str. 416-419].

Při návrhu profilování se musí vždy brát v potaz [15]:

- Výkonnost je častokrát velmi závislá na množství dat se kterými se pracuje, proto je potřeba profilovat s dostatečným množstvím dat, které bude odpovídat reálnému použití.
- Samotné profilování zabere nějaký čas, takže bez profilování bude aplikace o něco rychlejší (toto je potřeba řešit, pokud se pracuje v řádech milisekund).
- Výkonnost aplikace může být ovlivněna dalšími operacemi, které běží na stejném systému, proto pokud se porovnávají výsledky, je potřeba profilovat za stejných podmínek.
- Problém může být i mimo aplikaci, např. omezení na paměť, což může způsobit pomalý běh programu. Proto je potřeba brát v potaz i tyto možnosti.

Hotshot

Hotshot byl experimentální modul v jazyce `C`, který se zaměřoval na minimalizaci režie při profilování za cenu delších časů na zpracování dat. Momentálně již není podporovaný [10].

Profile

Profile je modul psaný čistě v `pythonu`, který ovšem přidává znatelnou režii do profilovaných programů. Případné úpravy a rozšíření tohoto modulu jsou však značně jednodušší [10].

cProfile

Nejpoužívanější modul v `pythonu` pro profilování. V podstatě jde o rozšíření v jazyce `C`. Díky tomu přidává tento modul pouze minimální režii profilovanému skriptu [10].

Pro potřeby této práce jsem se rozhodl využít modul `cProfile`, zejména kvůli nízké režii a doporučení jeho použití při programech s delší dobou běhu.

Použití cProfile:

V této práci se zabývám zejména využitím modulu `cProfile` pro profilování celých skriptů. Pro toto použití stačí zavolat `cProfile` následovně: `python -m cProfile [-o output_file] [-s sort_order] myscript.py`

- `o` - Zapíše výsledky profilování do souboru místo na `stdout`.
- `s` - Specifikuje způsob podle kterého budou statististiky tříděny (pouze pokud není použito -o).

Výsledky jsou vytisknuty v přehledné tabulce, kde jednotlivé sloupce mají následující význam [10]:

- `ncalls` - Počet volání.
- `tottime` - Celkový čas strávený v dané funkci.
- `percall` - Kvocient tottime dělený ncalls.
- `cumtime` - Představuje celkovou dobu strávenou v této a ve všech dílčích funkcích, je přesná i pro rekurzivní funkce.
- `percall` - Kvocient cumtime dělený primitivními voláními.
- `filename:lineno(funkce)` - Poskytuje příslušná data pro každou funkci

4.4 Profilování MCUX Builderu

V tomto konkrétním případu jsem se rozhodl nastavit na profilování stejné podmínky jako při reálném běhu programu. Framework `MCUX Builder` je spouštěn na virtuálních systémech umístěných na vzdálených serverech. Samotné spouštění a sběr výsledků je pak pomocí programu na kontinuální integraci `Jenkins`.

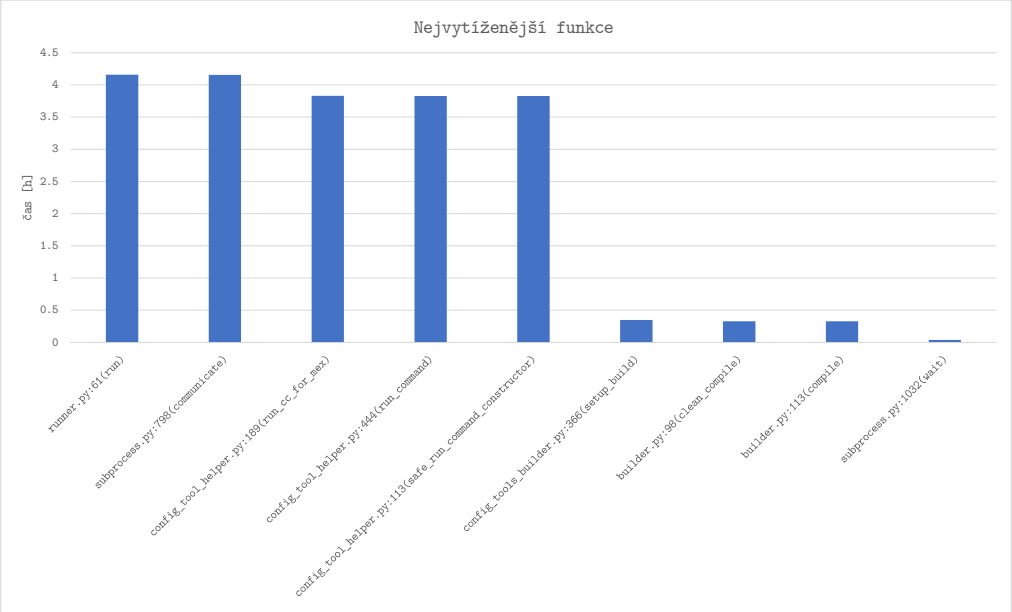
4.4.1 Profilování build testu

Konfigurace pro profilování build testu sestává z testovacího jobu `builder_optimalization`, který je kopií reálného jobu `Kinetis_V5`. Při této konfiguraci jsou z přednastavených `.MEX` souborů vygenerovány zdrojové soubory pomocí `MCUXpresso Config Tools` a následně jsou tyto zdrojové kódy zkompilovány pro jednotlivé toolchainy.

Nastavení pro profilování:

Samotné profilování probíhalo na stejném virtuálním počítači, kde běží `MCUX Builder` i standartně. Tento virtální systém má k dispozici čtyř-jádrový procesor `Intel`, operační systém je `Windows Server 2012 R2`[17]. Statistiky pro profilování jsou pak následující: nejvytíženější funkce 4.5 , celkový běh programu, spuštění s `cProfile` modulem a doba běhu jednotlivých funkcích.

V prvním grafu lze vidět přehled nejvytíženějších funkcí. V grafu vidíme celkový čas strávený v dané funkci, včetně všech dílčích funkcích. Jasně zde lze vidět, že největší časovou zátěž způsobuje volání externích programů – funkce *run, communicate, run_cc_for_mex, run_command, safe_run_command_constructor*.



Obrázek 4.5: Nejvytíženější funkce

Celková doba běhu se může drobně lišit v závislosti na aktuálním vytížení procesoru daného počítače, rychlosti připojení ke `GITu` a rychlosti běhu podprogramů. Z toho důvodu byl tento job spuštěn celkem `15x`, aby byla zajištěna maximální přesnost údajů. Přehled o době běhu najdete v tabulce 4.1.

Tabulka 4.1: Doba běhu jobu `Kinetis_V5`

	Nejkratší doba běhu	Nejdelší doba běhu	Průměrná doba běhu
Kinetis_V5	4h19min.	4h22min.	4h21min.

[17] https://cs.wikipedia.org/wiki/Windows_Server_2012

V tomto jobu jsou vykonávány dvě hlavní funkcionality. Generování zdrojových kódů přes `MCUXpresso Config Tools` a kompilace s jednotlivými toolchainy. Z předchozího grafu vyplývá, že největší časovou zátěží jsou funkce pro volání externích programů 4.5. Aby se ověřilo, která část zatěžuje běh programu více, tak byl spuštěn job `Kinetis_V5` ještě jednou. Tentokrát s měřením času ve funkci `compile`, která se stará o kompilaci vygenerovaných kódů. Z výsledů viz. tabulka 4.2 vychází, že samotná kompilace trvá zhruba `19,5` minuty. Což znamená, že z celkového průměrného běhu `4h21min.` programu, tvoří kompilace `7,5%`. A z toho je jasné, že největší časovou zátěží je tedy generování zdrojových kódů z `MCUXpresso Config Tools`.

Tabulka 4.2: Doba běhu funkce `compile`

	`compile`
Nejkratší doba běhu	0.29s
Nejdelší doba běhu	1.11s
Průměrná doba běhu	0.53s
Celková doba běhu	19.48min.
Celkový počet měřených kompilací	2212

4.4.2 Výsledek:

Z profilování jasně vychází, že největší časovou zátěž způsobuje externí volání aplikací. U jobu `Kinetis_V5` to je konkrétně generování zdrojových souborů pomocí `MCUXpresso Config Tools`.

Kapitola 5

Návrh optimalizace

Tato kapitola se zabývá možnostmi optimalizace testovacího frameworku `MCUX Builder`. Vzhledem k předchozím výsledkům profilování a analýzy stávajícího řešení se budeme zabývat zejména dvěmi možnostmi optimalizace. Zrychlením celkového běhu a zlepšením návrhu frameworku. Cílem první části optimalizace je snížit čas běhu programu při zachování stejných výsledků. Cílem druhé části je pak zlepšit přehlednost objektového návrhu, aby pozdější úpravy byly snazší a celkově se zvýšila udržitelnost kódu.

5.1 Souběžnost v pythonu

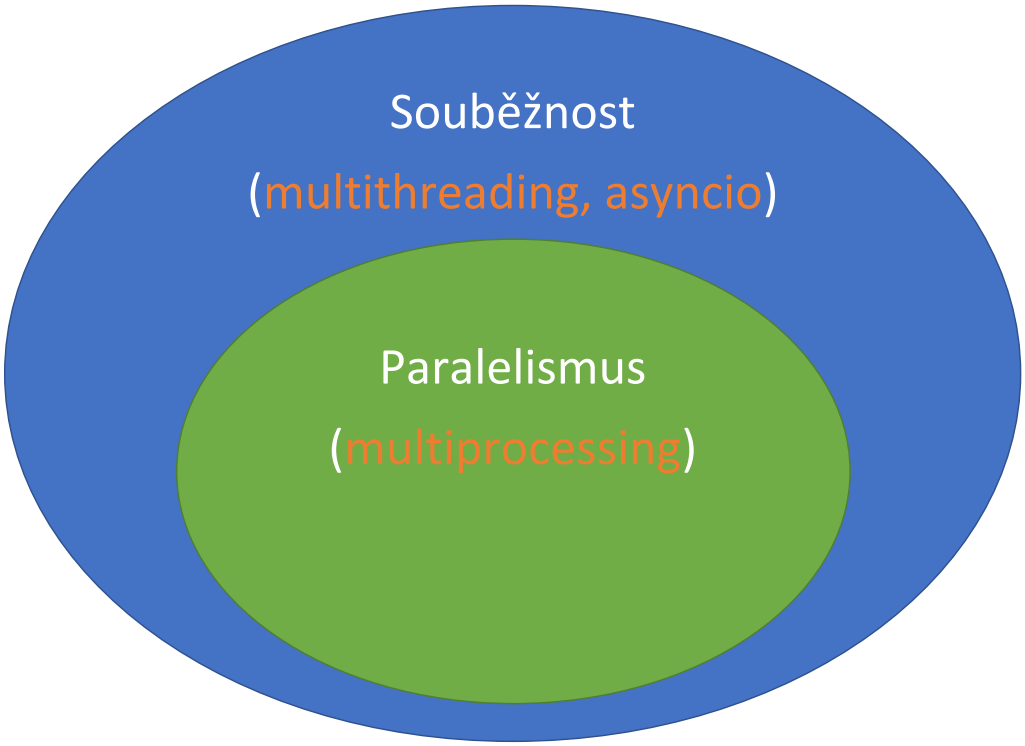
Vzhledem k velkému množství volání externích programů, jako `MCUXpresso Config Tools, IAR, KEIL, ARMGCC`, kde builder čeká na dokončení dané úlohy. Jsem se rozhodl tento problém řešit zavedením souběžného spouštění. Zejména na build pro jednotlivé toolchainy.

5.1.1 Souběžnost a parelelismus

Souběžnost znamená, že dva úkony mají současný průběh. Jako příklad si lze představit jednoduchý systém objednávky letenek. Zákazník si chce objednat letenku, daný systém musím zavolat do dané aerolinie a následně čeká na potvrzení letu. Po obdržení potvrzení píše a následně odešle email s informacemi o letu zákazníkovi. Pokud v okamžiku čekání na potvrzení letu, píše i mail, jedná se o `souběžné zpracování`. Pokud se rozhodne, že systém bude mít dvě linky, které zvládnou vyřizovat dva lety zároveň, tak poběží tzv. `paralelně`.

`Paralelismus` je v zásadě forma souběžnosti. Ale `paralelismus` je závislý na `hardwaru`. Například, jestli je v procesoru pouze jedno jádro, tak nemohou dvě operace běžet paralelně. Mohou pouze sdílet společný procesorový čas daného jádra. To je `souběžnost`, nikoliv však `paralelismus`. Pokud, ale je k dispozici více jader, tak je možné tyto dvě operace, nebo i více (záleží na počtu jader), provádět paralelně[16][21].

Parelismus tedy znamená, že se jedná o souběžnost. Souběžnost však neznamená, že se jedná o parelismus viz. obrázek 5.1.



Obrázek 5.1: Souběžnost

5.1.2 Synchronně vs asynchronně

V `synchronních` operacích, jsou úkoly vykonávány jeden za druhým. V `asynchronních` můžou být úkoly vykonávány v nezávislém pořadí. Jeden `asynchronní` úkol může začít a pokračovat v práci, zatímco se spouštění přesune na další nový úkol. `Asynchronní` operace neblokují (nedrží si řízení až do svého ukončení) jednotlivé operace a většinou běží na pozadí. Situace z předchozího příkladu, kdy v průběhu čekání na potvrzení začne systém psát už mail je právě `asynchronní` zpracovávání. Dané operace se neblokují [28].

5.1.3 Shrnuntí:

- `Synchronně:` blokující operace.
- `Asynchronně:` neblokující operace.
- `Souběžně:` mají současný průběh.
- `Paralelně:` probíhají paralelně [16].

5.1.4 Vlákna a procesy

`Python` podporuje vlákna již velmi dlouho. Vlákna nám umožňují souběžný běh operací. Od začátku však zde byl a je problém v podobě globálního zámku interpretu – `GIL (global`

interpreter lock), díky kterému použití vláken v pythonu nikdy nepovede k pravému paralelismu. Ovšem s multiprocessingem je, již možné i využití více jader.

5.1.5 Globální zámek interpretu - GIL

GIL je zamykací mechanismus, který umožní interpretu pythonu běžet ve stejném čase vždy pouze na jednom vláknu. Což znamená, že byte kód z pythonu může být vykonáván vždy pouze na jednom vláknu. GIL byl zaveden kvůli zjednodušení kontroly paměti cPythonu a lepší integraci s jazykem C. GIL zajistí, že více vláken nikdy neběží paralelně [4][16].

Fakta ohledně GIL[4]:

- Pouze jedno vlákno v daném čase.
- Interpret pythonu přepíná mezi vlákny, aby zajistil souběžnost.
- GIL je aplikovatelný pouze v CPythonu. Další implementace jako Jython nebo IronPython GIL nemají.
- GIL zajišťuje rychlejší běh jednovláknových programů.
- U vstupně/výstupně závislých operací, většinou GIL příliš nevadí.
- GIL umožňuje snažší integraci knihoven z jazyka C.

5.2 Zřetězení příkazů

Při analýze jsem došel k závěru, že jedním z největších optimalizačních problémů v MCUX builderu je generování zdrojových souborů z MCUXpresso Config Tools. Pro každý .MEX soubor se pro generování zdrojových souborů musela spustit nová instance MCUXpresso Config Tools. To bylo způsobeno hlavně tím, že MCUXpresso Config Tools nepodporovaly zřetězení příkazů v rozhraní příkazového řádku. Oním optimalizačním problémem, pak byla nutnost spouštět znovu celou aplikaci. Tento problém se týkal pouze rozhraní příkazového řádku. V grafickém rozhraní bylo možné opakovaně nahrávat různé .MEX soubory a generovat z nich zdrojové kódy. Na základě vzájemné domluvy s vývojáři MCUXpresso Config tools jsem pak implementoval možnost zřetězení příkazů. Implementace zřetězení je velmi důležitá pro dávkové spouštění, kde může značně ulehčit práci.

5.2.1 Návrh řešení

Práce s argumenty v MCUXpresso Config Tools je velmi komplexní. Některé argumenty jsou povinné, některé volitelné a jiné jsou povinné až při použití některého jiného argumentu. Všechny argumenty jsou tudíž nejdříve uloženy do pole, ze kterého jsou následně rozděleny do jednotlivých sekcí a ty jsou pak samostatně zpracovávány.

Hlavní částí je funkce pro zpracování argumentů. Pomocí algoritmu, který je implementován v této funkci jsou postupně zpracovávány argumenty a je rozhodováno, zda se vytvoří nová sekce nebo ne. Samotná sekce je potom tvořena mapou, respektive typem linkedHashMap.

5.2.2 Kolekce v javě

V `javě` je několik možností kolekcí, které implementují rozhraní `List`. Prvním typem je `ArrayList`(dynamické pole). Jedná se o seznam postavený nad polem. Největší výhodou `ArrayListu` je přístup k libovolnému prvku v konstantním čas $O(1)$, nevýhodou pak občasná nutnost kopírování všech prvků do nového pole (při rozpínání a zmenšování kolekce), kvůli které operace vkládání zabere až $O(n)$ operací. Tato struktura není pro ukládání parametrů vhodná, jelikož nelze dopředu vědět kolik argumentů bude daný parametr mít a rozpínání kolekce pak může být velké [2].

LinkedList

Další strukturou je `LinkedList` (spojový seznam). Jedná se o posloupnost žřetězených objektů, z nichž každý obsahuje ukládanou hodnotu. `LinkedList` v javě je obousměrně zřetězený (tj. uzel obsahuje ukazatel i na předchozí prvek). Největší nevýhodou `LinkedListu` je absence libovolného přístupu - pokud je potřeba přistoupit k i-tému prvku, tak je potřeba $O(i)$ operací na přeiterování všech předchozích prvků. Výhodou oproti dynamickému poli je absence realokace, díky které je spojový seznam vhodný pro použití v systémech, v nichž je nutnost garantovat odpověď v určitém časovém horizontu. Tuto strukturu jsem se však rozhodl nevyužít, právě kvůli absenci libovolného přístupu, kdy je občas potřeba zkontrolovat parametry v různém pořadí [2].

Vector

Mezi další možnosti patří struktura `Vector`, která je obdobou dynamického pole. Jediným rozdílem je, že je synchronizovaná. Synchronizovanost znamená, že lze tuto kolekci využívat najednou z více vláken. Nevýhodou synchronizace pak je `overhead`, který s sebou automaticky přináší - z tohoto důvodu se používá pouze tehdy, je-li k tomu skutečně pádný důvod.

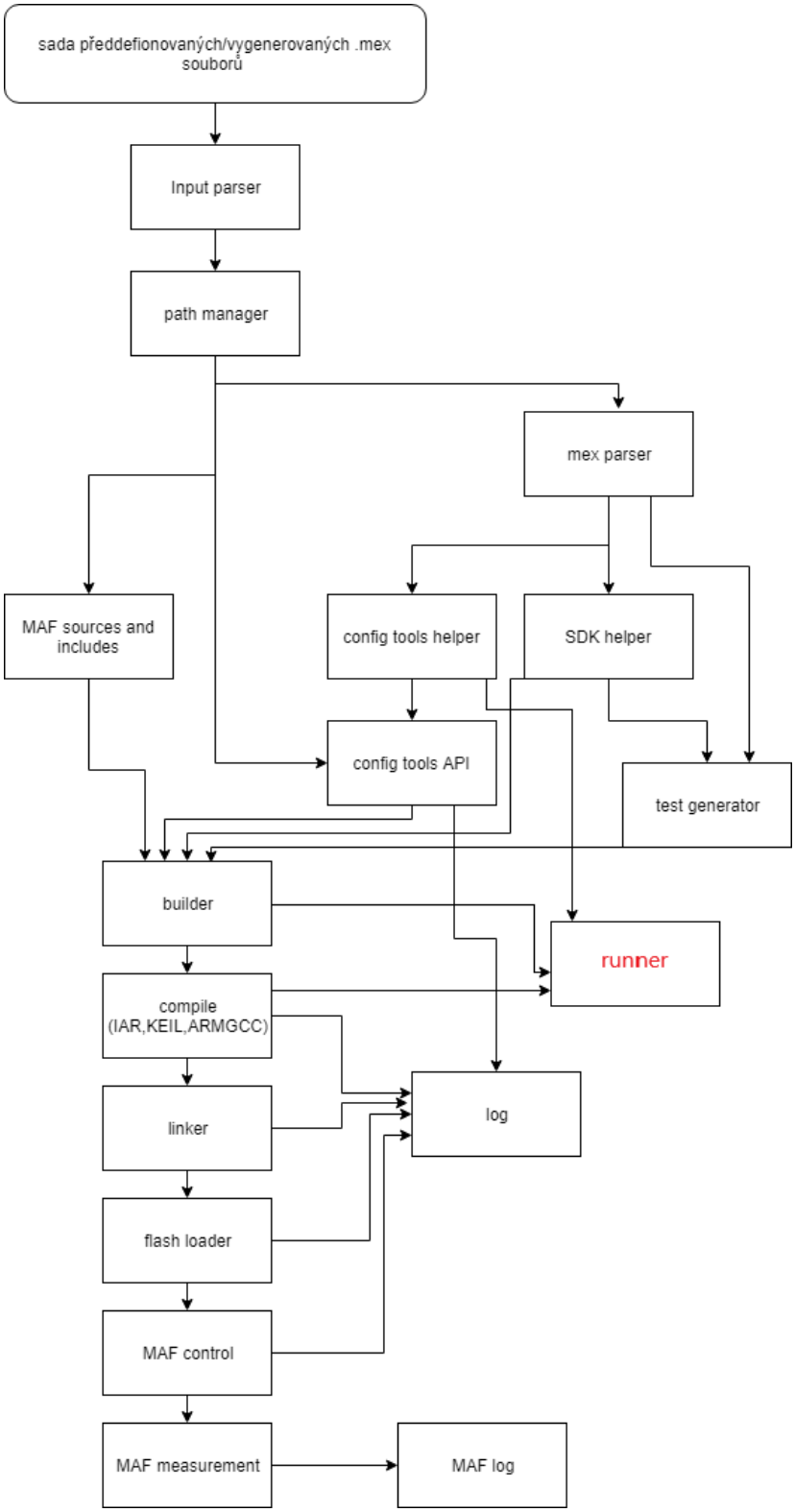
`Vector` je zastaralá kolekce, proto by neměla být v nových programech vůbec používána. Snychronizace kolekce lze dosáhnout i jinými prostředky. V této práci je uvedena pouze kvůli komplexnímu přehledu [2].

HashMap

`HashMap` je struktura, která byla vybrána jako vhodný kandidát. Konkrétně `LinkedHashMap`. `HashMap` odpovídá hashovací tabulce a její hlavní výhodou je vyhledávání podle prvku dle klíče v průměrně konstatním čase. Nepříjemností pak je negarantování jakéhokoliv pořadí prvků. Z toho důvodu byla použita `LinkedHashMap`, která vnitřně obsahuje spojovaný seznam a tím pádem je schopna garantovat predikovatelné pořadí prvků [3].

5.3 Sjednocení externího spouštění aplikací

V průběhu analýzy frameworku `MCUX Builder` jsem došel k závěr, že spouštění externích aplikací je řešeno poměrně nesystematicky. Vzhledem k potřebě implementace souběžnosti na jednom místě je potřeba tuto funkcionalitu sjednotit do jednoho bodu. Návrh na řešení tohoto problému spočívá ve vytvoření nového modulu `runner`, který bude implementovat jednotné rozhraní pro spouštění externích aplikací a sběr výsledků. Tato implementace taktéž umožní sjednocení souběžnosti do jednoho bodu viz. obrázek 5.2.



Obrázek 5.2: API MCUX Builder frameworku rozšířený o modul runner.

Kapitola 6

Implementace optimalizace

Tato kapitola popisuje implementaci jednotlivých návrhů optimalizace. Nejdříve se jedná o implementaci souběžnosti, dále pak implementaci zřetězení příkazů a poslední je implementace modulu `runner`. V implementaci souběžnosti a zřetězení příkazů jsou taktéž popsány experimenty, jejichž cílem bylo odhalit dopad optimalizace na běh aplikace.

6.1 Implementace souběžnosti

Tato sekce popisuje implementaci souběžnosti v testovacím frameworku `MCUX Builder`. Pro zjištění nejvíce vyhovující metody pro konkrétní případy užití `MCUX Builderu` jsem se rozhodl naimplementovat a otestovat všechny tři možnosti souběžnosti v pythonu – `multithreading, multiprocessing a asyncio`. V této kapitole tudíž naleznete popis implementace jednotlivých možností a výsledky výkonnostních testů.

6.1.1 Implementace multithreadingu

Při návrhu implementace multithredingu jsem potřeboval řešení, které bude vyhovovat následujícím požadavkům:

- Počet vláken není dopředu známý.
- Jednotlivé funkce spouštěné na více vláknech se mohou měnit.
- Optimalizace by měla být nezávislá na funkcích `MCUX Builderu`.
- Počet spouštění optimalizované funkce může být vysoký.

Dle těchto požadavků jsem jako nejvhodnější možnost zvolil implementaci přes třídu `ThreadPool`. `ThreadPool` je skupina předpřipravených vláken, čekajících na úkol. V případech, kdy je k dispozici velké množství úkolů je preferovanější způsob vytváření `thread poolu`, než inicializace vláken pro každý úkol [12].

`ThreadPool` zvládá souběžné vykonávání velkého množství úkolů následujícím způsobem:

- Pokud vlákno v `thread pool` dokončí svůj úkol, tak vlákno může být znovu použito.
- Pokud je vlákno ukončeno, jiné vlákno bude vytvořeno, aby ho nahradilo.

Pomocí třídy `ThreadPool` se tedy vytvoří `pool` s počtem vláken podle zařízení, na kterém je skript spuštěn. Pole z jednotlivými příkazy pro spuštění externích aplikací spolu s funkcí, která se stará o samotné spouštění jsou vloženy do mapy. `Mapa` je vestavěná funkce vyššího řádu, která aplikuje danou funkci na každý prvek seznamu a vrací seznam výsledků. Následně se zavolá dvojice funkcí `pool.close()` a `pool.join()`.

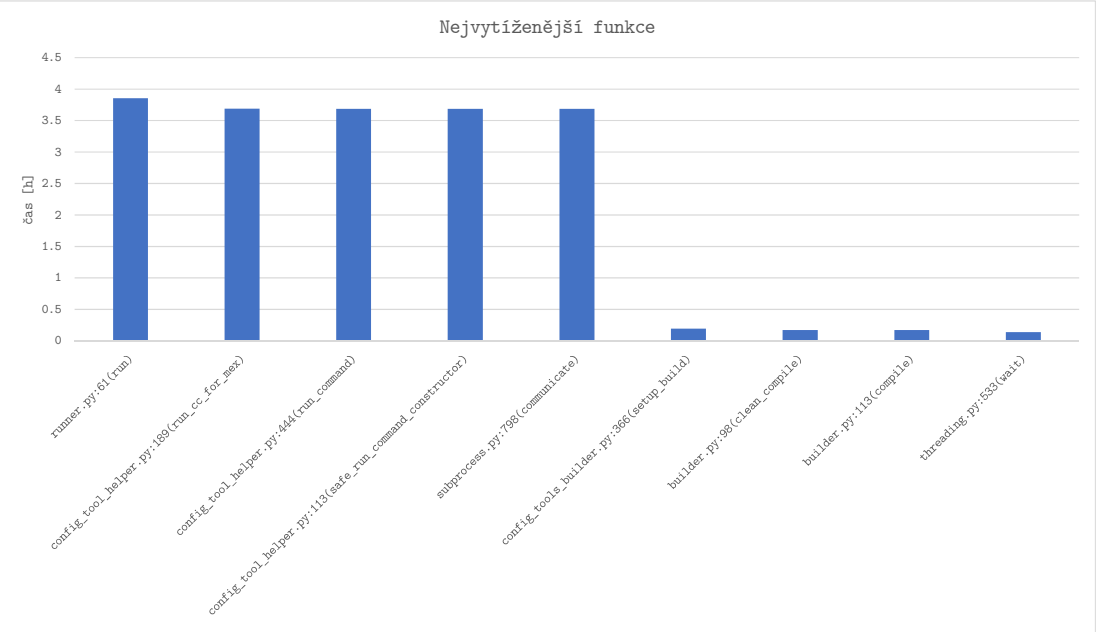
- `Pool.close()` se volá vždy, když už se nebude do instance poolu přidávat žádná další práce, což znamená že je typicky volána, když je dokončená paralelizovaná část hlavního programu, Jakmile budou všechny přiřazené práce dokončené, tak se ukončí všechny worker procesy.
- `Pool.join()` se volá vždy jako druhá funkce v pořadí a čeká na ukončení worker procesů, tato funkce zajišťuje jistý synchronizační bod, kde můžou být reportovány vyjímky, které se objeví v některém z worker procesů [9].

ukázka implementace:

```
pool = ThreadPool(multiprocessing.cpu_count())
pool.map(self.thread_run, commands)
pool.close()
pool.join()
```

6.1.2 `Kinetis_V5` multithreading experiment:

Při tomto experimentu byl spuštěn job `Kinetis_V5` a cílem bylo zjistit dopad použití `multithreadingu` na generování zdrojových souborů přes `MCUXpresso Config Tools`. V prvním grafu lze vidět přehled nejvytíženějších funkcí. V grafu vidíme celkový čas strávený v dané funkci, včetně všech dílčích funkcích. Stejně jako při spuštění bez optimalizace zůstává největší časová zátěž na volání externích programů. Největší pokles času je u volání funkce `communicate`, což vypovídá o lepší optimalizaci na úrovni čekání na výsledky externích podprogramů viz. 6.1.



Obrázek 6.1: Nejvytíženější funkce multithreading

Celková doba běhu se může drobně lišit v závislosti na aktuálním vytížení procesoru daného počítače, rychlosti připojení ke `GITu` a rychlosti běhu podprogramů. Z toho důvodu byl tento job spuštěn celkem `15x`, aby byla zajištěna maximální přesnost údajů. Přehled o době běhu najdete v tabulce 6.1.

Tabulka 6.1: Doba běhu jobu `Kinetis_V5`

	Nejkratší doba běhu	Nejdelší doba běhu	Průměrná doba běhu
Kinetis_V5	4h2min.	4h5min.	4h3min.

Pro zjištění doby běhu funkce `compile` při optimalizaci pomocí `multithreadingu`, byl job `Kinetis_V5` spuštěn ještě jednou, tentokrát s měřením času. Celkově kompilace trvaly `9.1` minuty, což je z celkového běhu `4h3min.` zhruba `3.7%`. Optimalizace se tedy projevila velmi pozitivně. Zejména průměrná doba kompilace se oproti běhu bez optimalizace zlepšila z `0.53s` na `0.25s`, což je zlepšení o `53%`.

Tabulka 6.2: Doba běhu funkce `compile` při multithreadingu

	`compile`
Nejkratší doba běhu	0.2s
Nejdelší doba běhu	0.71s
Průměrná doba běhu	0.25s
Celková doba běhu	9.1min.
Celkový počet měřených kompilací	2212

6.1.3 Implementace multiprocessingu

Podmínky pro implementaci `multiprocessingu` zůstávají velmi podobné těm u `multithreadingu`. Mezi ty nejdůležitější patří tyto:

- Počet jader procesoru není dopředu známý.
- Počet spouštění optimalizované funkce může být vysoký.

Pro implementaci multiprocessingu `Python` poskytuje knihovnu `multiprocessing`. `API` je potom velmi podobné jako při `multithreadingu`. V tomto případě je v podstatě vyměněna třída `ThreadPool` za třídu `pool`. Třída `pool` vytvoří `pool` s počtem jader podle procesoru zařízení na kterém je spuštěna. Mnohem důležitější je však pochopení změn, které se dějí na pozadí. Oproti `multithreadingu` umožňuje `multiprocessing` vytvářet programy, které běží souběžně a využijí opravdu celou kapacitu `CPU`. Knihovna `multiprocessing` dává každému procesu svůj vlastní interpretr se svým vlastním zámkem `GIL`. Což znamená, že i přes zdánlivou podobu je fungování knihovny `multiprocessing` oproti `multithreadingu` diametrálně odlišné. Využití `multiprocessingu` je zejména u výpočetně náročných operací [19].

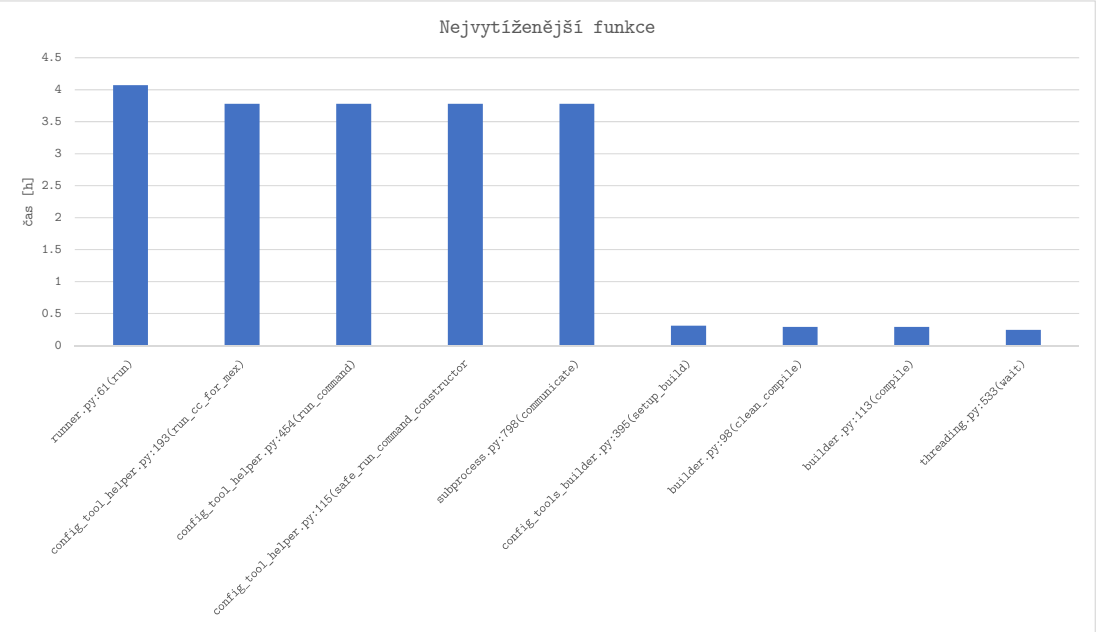
ukázka implementace:

```
pool = Pool(multiprocessing.cpu_count())
log = pool.map(proc_run, commands)
pool.close()
pool.join()
```

6.1.4 `Kinetis_V5` multiprocessing experiment:

Při tomto experimentu byl spuštěn job `Kinetis_V5` a cílem bylo zjistit dopad použití `multiprocessingu` na generování zdrojových souborů přes `MCUXpresso Config Tools`.

V prvním grafu lze vidět přehled nejvytíženějších funkcí. V grafu vidíme celkový čas strávený v dané funkci, včetně všech dílčích funkcích. Stejně jako při spuštění bez optimalizace zůstává největší časová zátěž na volání externích programů viz. 6.2.



Obrázek 6.2: Nejvytíženější funkce multiprocessing

Celková doba běhu se může drobně lišit v závislosti na aktuálním vytížení procesoru daného počítače, rychlosti připojení ke `GITu` a rychlosti běhu podprogramů. Z toho důvodu byl tento job spuštěn celkem `15x`, aby byla zajištěna maximální přesnost údajů. Přehled o době behu najdete v tabulce 6.3.

Tabulka 6.3: Doba běhu jobu `Kinetis_V5`

	Nejkratší doba běhu	Nejdelší doba běhu	Průměrná doba běhu
Kinetis_V5	4h12min.	4h15min.	4h13min.

Pro zjištění doby běhu funkce `compile` při optimalizaci pomocí `multiprocessingu`, byl job `Kinetis_V5` spuštěn ještě jednou, tentokrát s měřením času. Celkově kompilace trvaly `17.4` minuty, což je z celkového běhu `4h13min.` zhruba `6.9%`. Optimalizace se tedy téměř neprojevila. Zejména průměrná doba kompilace se oproti běhu bez optimalizace téměř nezměnila. Což vypovídá, že kompilace není příliš procesorově závislá.

Tabulka 6.4: Doba běhu funkce `compile` při multiprocessingu

	`compile`
Nejkratší doba běhu	0.33s
Nejdelší doba běhu	0.86s
Průměrná doba běhu	0.47s
Celková doba běhu	17.39min.
Celkový počet měřených kompilací	2212

6.1.5 Implementace asyncio

`Asyncio` je třetí a nejnovější možnost, jak v `pythonu` implementovat souběžnost. Plná podpora `asyncio` je až v pythonu verze `3.5`, ovšem vývoj pokračuje i v dalších verzích [21].

Již z názvu lze předpokládat, že se bude jednat o asynchronní zpracovávání úkolů. To ovšem zvládal i `multithreading`. Zásadní rozdíl je v tom, že `multithreading` funguje jako `preemtetivní multitasking` (systém sám rozhoduje, kdy přepne z jednoho vlákna do druhého a tuto změnu si prakticky vynutí). Vytváření procesů pomocí `multiprocessingu` má zase velkou režii.

Pro snazší představu problému s multithreadingem je uveden následující příklad: K dispozici jsou tři vlákna `V1, V2` a `V3`. Na všech třech vláknech začaly běžet `vstupně/výstupní` operace. `V3` už svou práci dokončilo. `V2` a `V1`, ale ještě stále čekají na `vstup/výstup`. Interpret pythonu přepne na `V1`, které ale stále čeká, tak se posune na `V2` to, ale taky čeká, tak se posune na `V3`, které je připraveno a interpret může spustit kód. Zde je jasně viditelný problém, kdy přepnutí na `V1` a následně na `V2` stálo režii, kterou lze ušetřit, pokud by interpret skočil rovnou na `V3`. Tento problém řeší právě `asyncio`. `Asyncio` ve své podstatě využívá pouze jedno jádro a jedno vlákno procesoru [16].

`Asyncio` nám poskytuje smyčku událostí a další zajímavé funkcionality. Smyčka událostí sleduje různé `vstupně/výstupní` události a přepíná na úkoly, které už jsou připraveny a pozastavuje ty, které čekají na `vstup/výstup`. Díky tomu není ztrácen čas na úkolech, které ještě nejsou připraveny ke spuštění.

Knihovna `asyncio` je vnitřně založena na `futures`. `Future` je objekt, který reprezentuje budoucí výsledek nějaké operace. Poté, co tato operace skončí, se dá výsledek zjistit pomocí metody `result()`. `Future` je takový slib, který nám říká, že v budoucnu něco dostaneme. Čekání na splnění tohoto slibu se provádí pomocí `await()(nebo loop.run_until_complete)`.

Shrnutí funkcionality je zhruba následující. Máme smyčku událostí, funkce a `vstupně/-výstupní` operace. Dáme naše funkce do smyčky událostí a požádáme ji, aby je za nás spustila. Smyčka událostí nám vrátí objekt `Future`. Čekáme na tomto objektu a čas od času ho zkontrolujeme, zda už má nějakou hodnotu. Jakmile ji má, tak ji použijeme v dalších operacích.

`Asyncio` pro spouštění a pozastavení úkolů používá `generátory` a `corountimes` [17].

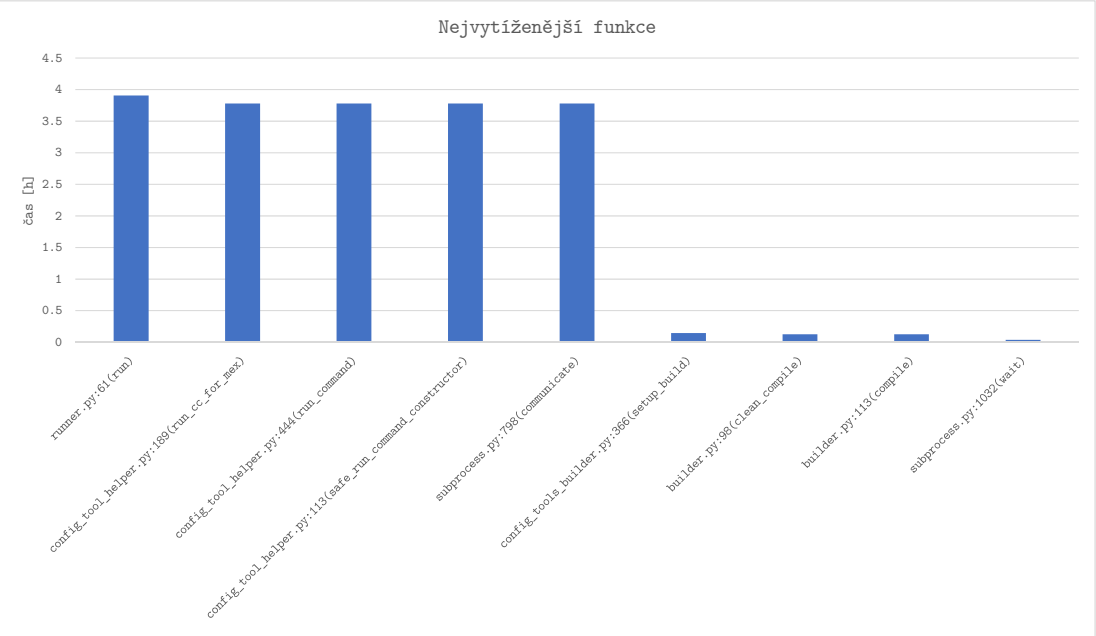
ukázka implementace:

```
futures = [self.run_async(command) for command in commands]
loop = asyncio.ProactorEventLoop()
asyncio.set_event_loop(loop)
loop.run_until_complete(asyncio.wait(futures))
```

6.1.6 `Kinetis_V5` asyncio experiment:

Při tomto experimentu byl spuštěn job `Kinetis_V5` a cílem bylo zjistit dopad použití `asyncio` na generování zdrojových souborů přes `MCUXpresso Config Tools`.

V prvním grafu lze vidět přehled nejvytíženějších funkcí. V grafu vidíme celkový čas strávený v dané funkci, včetně všech dílčích funkcích. Stejně jako při spuštění bez optimalizace zůstává největší časová zátěž na volání externích programů. Největší pokles času je u volání funkce `communicate`, což vypovídá o lepší optimalizaci na úrovni čekání na výsledky externích podprogramů viz. 6.3.



Obrázek 6.3: Nejvytíženější funkce asyncio

Celková doba běhu se může drobně lišit v závislosti na aktuálním vytížení procesoru daného počítače, rychlosti připojení ke `GITu` a rychlosti běhu podprogramů. Z toho důvodu byl tento job spuštěn celkem `15x`, aby byla zajištěna maximální přesnost údajů. Přehled o době běhu najdete v tabulce 6.5.

Tabulka 6.5: Doba běhu jobu `Kinetis_V5`

	Nejkratší doba běhu	Nejdelší doba běhu	Průměrná doba běhu
Kinetis_V5	4h1min.	4h3min.	4h2min.

Pro zjištění doby běhu funkce `compile` při optimalizaci pomocí `asyncio`, byl job `Kinetis_V5` ještě jednou, tentokrát s měřením času. Celkově kompilace trvaly `7.5` minuty, což je z

celkového běhu 4h2min. zhruba 3%. Optimalizace se tedy projevila velmi pozitivně. Zejména průměrná doba kompilace se oproti běhu bez optimalizace zlepšila z 0.53s na 0.20s, což je zlepšení o 62%.

Tabulka 6.6: Doba běhu funkce compile při asyncio

	compile
Nejkratší doba běhu	0.11s
Nejdelší doba běhu	0.63s
Průměrná doba běhu	0.20s
Celková doba běhu	7.46min.
Celkový počet měřených kompilací	2212

6.2 Implementace zřetězení příkazů

Tato sekce se zabývá implementací zřetězení příkazů. Nachází se v ní popis implementovaného rozhraní, příkazové řádky pro MCUXpresso Configuration Tools a popis samotného řešení. V závěru jsou pak uvedeny experimenty s cílem dokázat přínos daných řešení.

6.2.1 Rozhraní příkazové řádky pro MCUXpresso config tools

Tato část se zabývá pouze popisem částí, které jsou důležité pro optimalizaci frameworku MCUX Builderu. Kompletní popis rozhraní lze nalézt v referenčním manuálu viz. [26].

Od verze 5 MCUXpresso Config Tools je možné řetězit jednotlivé příkazy. Oddělovací prvek jednotlivých příkazů je -HeadlessTool. Každá část je pak vykonávána samostatně a nezávisle na ostatních. Příkazy, které nepožadují -HeadlessTool jako povinný parametr mohou být umístěny před první -HeadlessTool příkaz, nebo bez něj, pokud není použito zřetězení.

Příklad:

- -HeadlessTool Clocks -MCU MKL43Z256xxx4 -SDKVersion ksdk2_0 -ExportSrc C:/exports/src
- -HeadlessTool Pins -MCU MK65FN2M0xxx18 -SDKVersion ksdk2_0 -ExportSrc C:/exports/src
- -HeadlessTool Peripherals -MCU MK64FX512xxx12 -SDKVersion ksdk2_0 -ExportSrc C:/exports/src

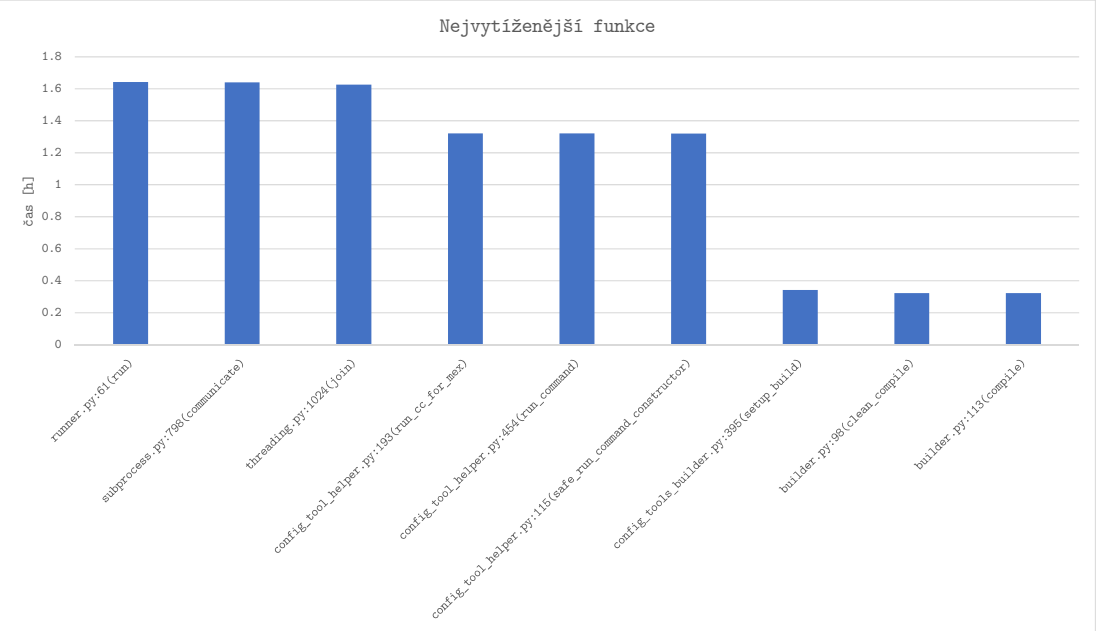
6.2.2 Základní struktura řešení

Základní struktura je pak následující. Zpracování začíná ve třídě Application, která se stará o všechny aspekty spouštění aplikace. V této třídě se uloží parametry do pole a zavolá se jejich zpracování. To je ve třídě CmdLine, kde je implementován algoritmus na roztřízení do jednotlivých sekcí podle daných pravidel. Samotná práce s jednotlivými sekcemi je pak ve třídě CmdSection, kde nalezneme funkce pro práci v LinkedHashMap.

6.2.3 `Kinetis_V5` zřetězení příkazů experiment:

Při tomto experimentu byl spuštěn job Kinetis_V5 a cílem bylo zjistit dopad použití zřetězení příkazů na generování zdrojových souborů přes `MCUXpresso Config Tools`.

V prvním grafu lze vidět přehled nejvytíženějších funkcí. V grafu vidíme celkový čas ztrávený v dané funkci, včetně všech dílčích funkcí. Stejně jako při spuštění bez optimalizace zůstává největší časová zátěž na volání externích programů. Je u nich, ale velmi značný pokles, v průměru o více než `50%` viz. 6.4.



Obrázek 6.4: Nejvytíženější funkce zřetězení příkazů

Celková doba běhu se může drobně lišit v závislosti na aktuálním vytížení procesoru daného počítače, rychlosti připojení ke `GITu` a rychlosti běhu podprogramů. Z toho důvodu byl tento job spuštěn celkem `15x`, aby byla zajištěna maximální přesnost údajů. Přehled o době běhu najdete v tabulce 6.7.

Tabulka 6.7: Doba běhu jobu `Kinetis_V5`

	Nejkratší doba běhu	Nejdelší doba běhu	Průměrná doba běhu
Kinetis_V5	1h49min.	1h53min.	1h51min.

6.3 Implementace modulu runner

Modul `runner` jsem navrhl, abych sjednotil způsob spouštění externích aplikací a programů z `MCUX Builderu`. Základem tohoto modulu je třída `CommandRunner`. Tato třída poskytuje rozhraní pro exekuci příkazů na spouštění externích aplikací. Toto rozhraní je postaveno na volání podprocesů v `pythonu` pomocí modulu `subprocess`.

Modul `subprocess` poskytuje konzistentní rozhraní pro vytváření a práci s přidanými procesy. Na rozdíl od jiných dostupných modulů nabízí vysoko úrovňové rozhraní, jehož cílem je nahradit funkce jako `os.system(), os.spawn*(), os.popen*(), popen2.*()` a `commands.*()` [13].

Tento modul definuje jednu třídu `Popen` a několik dalších `wrapper` funkcí. Konstruktor pro `Popen` si bere argumenty nutné pro nastavení nového procesu potomka, tak aby s ním rodičovský proces mohl pomocí `pipes` komunikovat. Poskytuje veškerou funkcionalitu modulů a funkcí, které nahrazuje. `API` je velmi konzistentní při různém použití.

Hlavní výhodou modulu `runner` je, že sjednocuje rozhraní pro volání externích aplikací pro celý builder. Krom zvýšení přehlednosti a udržitelnosti kódu, přináší tato implementace ještě jednu zásadní možnost, a to sjednocení implementace souběžnosti a možnost vyhodnocení použití nejlepší varianty optimalizace pro daný úkol. A to všechno bez zvýšení složitosti kódu a nezávislosti na úpravách v jednotlivých modulech.

6.3.1 Logování informací

Další důležitou částí nejen tohoto modulu je implementace logování různých informací. V průběhu exekuce je potřeba sbírat důležité informace o běhu jednotlivých programů. K tomuto účelu je potřeba číst `stdout` i `stderr`. K těmto informacím je navíc potřeba i různé `debugovací` informace, které jsou použity při hledání problémů. Aby byly dodrženy praktiky objektového návrhu je potřeba použít vhodné řešení, které bude tyto informace předávat a uchovávat v rámci jednotlivých tříd.

Pro toto řešení jsem zvolil `python` modul `logging`. `Logging` modul je používán většinou `python` knihoven třetích stran. Takže je velmi snadné integrovat logovací zprávy se zprávami z těchto knihoven a vytvořit, tak jednotný log pro danou aplikaci [1].

Po naimportování tohoto modulu, je možné použít tzv. `"logger"` k logování zpráv, které je žádoucí vidět. V základním nastavení je pět standartních úrovní pro určení důležitosti událostí. Jsou to tyto:

- `DEBUG`
- `INFO`
- `WARNING`
- `ERROR`
- `CRITICAL`

V nastavení loggeru, pak lze nastavit jaké úrovně je žádoucí logovat. Např. při standartním spuštění je nastaveno:

```
logging.basicConfig(level=logging.INFO)
```

,což říká, že budou logovány všechny zprávy s touto úrovní a vyšší [8].

V modulu runner je využití těchto úrovní následující:

```
self.logger.info("Return Code: " + str(self.returncode))
        if self.stdout:
            self.logger.debug("STDOUT: " + self.stdout.decode("utf-8")...
        if self.stderr:
            self.logger.error("STDERR: " + self.stderr.decode("utf-8")...
```

Zde lze vidět, že návratový kód je vracen vždy jako zprávu typu `INFO`. Výpis na `stdout` však za normálních okolností není potřeba vidět, proto je v úrovni `debug`. A `stderr` je potom logován jako `error`.

Přínos implementace modulu `runner` je velký, zejména kvůli jednotnému přístupu k volání podporgramů a konroly nad optimalizací.

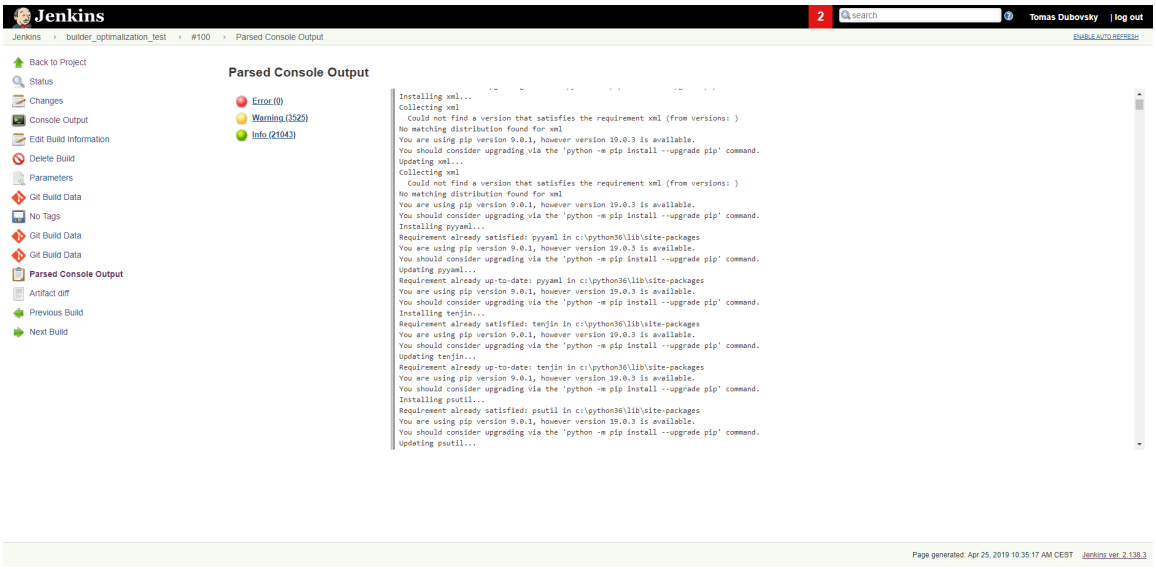
Kapitola 7

Ověření funkčnosti systému a dopad na efektivitu testování

Na ověření funkčnosti frameworku `MCUX Builder` byl sestaven testovací job `builder_optimalization`, který simuloval práci základního jobu `Kinetis_V5` . Na tomto testovacím jobu je ověřena základní funkčnost a případné nedostatky a možnosti zlepšení. Ve druhé fázi je ověřena funkčnost frameworku na reálném systému, kdy jsou spuštěny přímo joby vykonávající automatizované testování v pravidelných intervalech. Toto ověření testuje zejména případné zavlečení regresních chyb.

7.1 Testování `Kinetis_V5 job`

Při testu je spuštěna simulace reálného jobu `Kinetis_V5` na jobu `builder_optimalization`. Pomocí pluginu `Log Parser` v nástroji `Jenkins` jsou porovnány výstupní logy z obou jobů. Tento plugin umí zvýraznit body zájmu(info, warning, error) viz. obrázek 7.1 a lze tak přehledněji porovnat informace ze simulace s realnými výsledku jobu `Kinetis_V5` [7].



Obrázek 7.1: Log Parser

Výstupní log na testovacím jobu `builder_optimalization` se shoduje s výstupním logem na reálném jobu `Kinetis_V5`, což značí, že provedené změny nezměnily výsledky testování.

7.1.1 Výsledky a dopad na efektivitu testování

Na základě porovnání výsledků implemetace jednotlivých metod souběžnosti byla vybrána jako nejvhodnější možnost `asyncio`, jelikož dosahuje nejlepších časů pro metodu `compile` viz. tabulka 7.1.

Tabulka 7.1: Doba běhu funkce `compile`

	`multiprocessing`	`multithreading`	`asyncio`	`bez optimalizace`
Nejkratší doba běhu	0.33s	0.2s	0.11s	0.29s
Nejdelší doba běhu	0.86s	0.71s	0.63s	1.11s
Průměrná doba běhu	0.47s	0.25s	0.2s	0.53s
Celková doba běhu	17.39min.	9.1min.	7.46min.	19.48min.

U `multithreadingu` bylo dosaženo podobných výsledků jako u `asyncio`, u `multiprocessingu` jsou však výsledky jen o málo lepší než při běhu bez optimalizace, což se shoduje s obecnými doporučeními pro implementaci souběžnosti v `pythonu` [21]:

- Operace závislé na procesoru - `multiprocessing`.
- Operace závislé na vstupu/výstupu (rychlý I/O, malé množství vstupů) - `multithreading`.
- Operace závislé na vstupu/výstupu (pomalý I/O, velké množství vstupů) - `asyncio`.

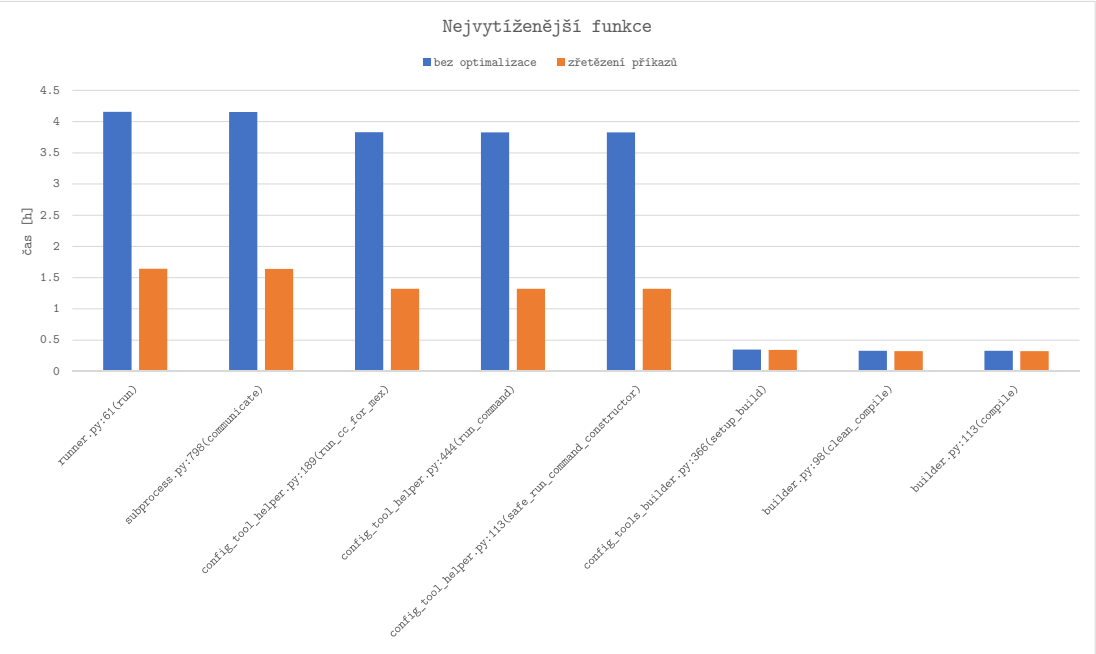
V tomto konkrétním případě se jedná o externí volání aplikací, které mají větší množství `I/O` operací.

Implementace zřetězení příkazů se pak projevila nejvíce. Oproti běhu bez optimalizace je časová náročnost u jobu `Kinetis_V5` snížena v průměru o `58%` viz.7.2.

Tabulka 7.2: Doba běhu jobu `Kinetis_V5`

	Nejkratší doba běhu	Nejdelší doba běhu	Průměrná doba běhu
bez optimalizace	4h19min.	4h22min.	4h21min.
zřetězení příkazů	1h49min.	1h53min.	1h51min.

Důležitým výsledkem při implementaci zřetězení příkazů je, že funkce `compile` zůstala časově stejně náročná jako před optimalizací pomocí zřetězení příkazů, což říká, že zavedení souběžnosti na tuto funkci je správný krok. Celkový přehled časové náročnosti funkcí před optimalizací a po zřetězení příkazů, pak ukazuje graf 7.2.



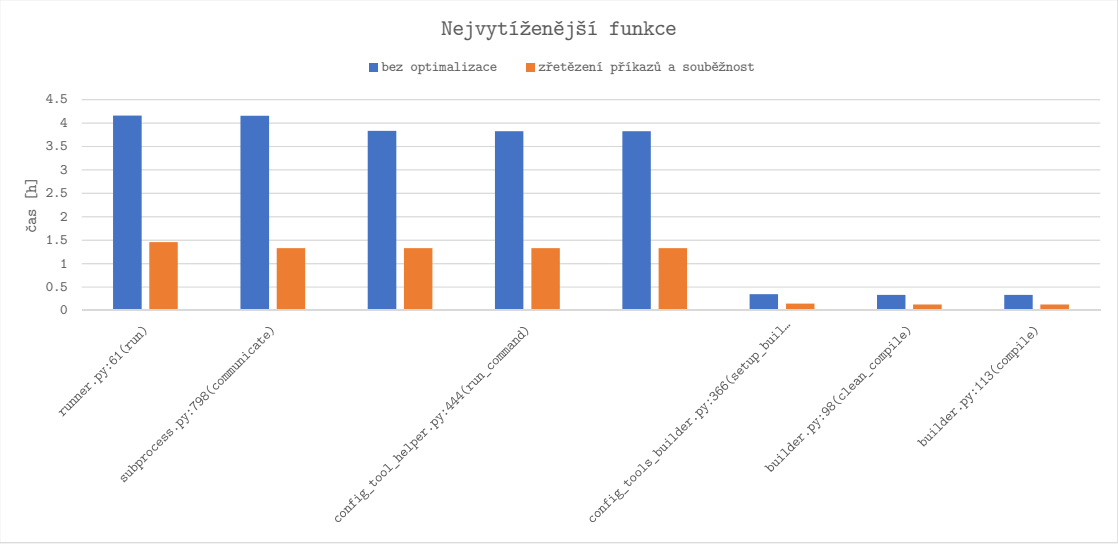
Obrázek 7.2: Nejvytíženější funkce srovnání

Při spuštění optimalizace pomocí zřetězení příkazů a souběžnosti současně došlo ke zlepšení v časové náročnosti o 62% viz.7.3, což je velký přínos pro efektivitu testování.

Tabulka 7.3: Doba běhu jobu `Kinetis_V5`

	Nejkratší doba běhu	Nejdelší doba běhu	Průměrná doba běhu
bez optimalizace	4h19min.	4h22min.	4h21min.
zřetězení příkazů a souběžnost	1h37min.	1h42min.	1h39min.

Celkový graf porovnávající nejvytíženější funkce při běhu jobu `Kinetis_V5` bez optimalizace a při běhu stejného jobu s optimalizacemi v podobě zřetězení příkazů a souběžnosti v podobě `asyncio`, pak ukazuje snížení časové náročnosti u všech funkcí o více jak polovinu viz. 7.3.



Obrázek 7.3: Nejvytíženější funkce srovnání

Díky jednotnému přístupu k volání externích aplikací přes modul `runner` se taktéž zjednodušila režie při hledání případných problémů v rozhraní pro spouštění externích aplikací a celkově se zlepšila udržitelnost a přehlednost kódu v této části.

7.2 Celkový dopad provedených změn na efektivitu testování

Největší přínos co do časové náročnosti má podpora zřetězení příkazů. Díky této optimalizaci je možné spouštět generování zdrojových kódů pro více mikrokontrolerů zaráz na jedné instanci `MCUXpresso Configuration Tools`. Tato změna zrychlila běh testovacího frameworku na ukázkovém jobu `Kinetis_V5` o `58%` . Spouštění kompilací s jednotlivými toolchainy, patřilo mezi nejvytíženější funkce, ale zřetězení příkazů tuto funkcionalitu neovlivnilo, proto byla implementována souběžnost. Jako nejvýhodnější řešení pro framework `MCUX Builder` byla vybrána souběžnost pomocí `Asyncio`. Vzhledem k různé časové náročnosti u různých toolchainů se díky této optimalizaci efektivněji využívá výkon a celkově tato změna zrychlila běh kompilace na ukázkovém jobu `Kinetis_V5` o `62%` . Implementace modulu `runner` zajistila sjednocení rozhraní pro spouštění externích aplikací. Zjednodušila přehlednost a udržitelnost frameworku `MCUX Builder`. Navíc umožnila přehlednou kontrolu implementace souběžnosti, kdy je díky sjednocení na jednom místě možné podle potřeby vybírat globálně mezi `multiprocessingem, multithreadingem` a `asyncio`. Celkově optimalizace zlepšila výkonost frameworku `MCUX Builder` co se týče časové náročnosti o `62%`. Toto urychlení umožnilo spouštění více běhů testů za jeden den a tím urychlilo celkové testování.

Kapitola 8

Závěr

Tato práce se zabývala možnostmi optimalizace frameworku `MCUX Builder`, který má za cíl automatizované testování `MCUXpresso Config Tools`. V práci bylo analyzováno stávající řešení na úrovní objektového návrhu i výkonu. V rámci analýzy byly objeveny výkonnostní problémy při volání externích aplikací. Byly navrhnuty a implementovány celkem tři možnosti optimalizace – optimalizace pomocí zavedení souběžnosti, implementace zřetězení příkazů a sjednocení volání externích aplikací.

V rámci zavedení souběžnosti byly analyzovány a testovány všechny tři možnosti souběžnosti v jazyce `Python` – `multiprocessing, multithreading` a `asyncio`. Cílem bylo zjistit nejoptimálnější možnost pro framework `MCUX Builder`. Na základě experimentů byla vybrána jako nejvhodnější možnost metoda `Asyncio`. Další optimalizací byla implementace podpory zřetězení příkazů v `MCUXpresso Config Tools`. Tato optimalizace byla navrhnuta v jazyce `Java` a pro implementaci byla provedena analýza nejvhodnější struktury pro ukládání a práci s argumenty. Jako nejvhodnější kandidát byla vybrána struktura `LinkedHashMap`, která byla i použita ve finálním řešení. Poslední optimalizací je rozšíření objektového návrhu o modul `runner`, který sjednocuje volání všech externích aplikací. Tento modul umožnil přehledné a udržitelné řešení, ve kterém je implementována i souběžnost.

Implementace souběžnosti a zřetězení příkazů značně urychlily běh frameworku `MCUX Builder`, čímž umožnily jeho další růst a možnost testování většího množství dat. Taktéž ušetřily `hardwarové` prostředky, které mohou být využity na spouštění dalších testů. Implementace třídy `runner` pak zpřehlednila a zjednodušila volání externích aplikací jako `MCUXpresso Config Tools`. Což znamená větší udržitelnost kódu v jedné z nejvytíženějších částí frameworku. V této třídě byly taktéž implementovány všechny možnosti souběžnosti a v budoucnu, tak lze snadno přepnout na tu nejvýhodnější možnost. Do budoucna je v plánu upravit a zpřehlednit logování jednotlivých problémů a taktéž optimalizace dalších částí frameworku.

Literatura

[1] Ajitsaria, A.: *Logging in Python*. [Online; navštíveno 21.04.2019].
URL https://realpython.com/python-logging/

[2] algoritmy.net: *Java (21) - Kolekce*. [Online; navštíveno 21.04.2019].
URL https://www.algoritmy.net/article/34009/Kolekce-21

[3] baeldung: *A Guide to LinkedHashMap in Java*. [Online; navštíveno 21.04.2019].
URL https://www.baeldung.com/java-linked-hashmap

[4] Beazley, D.: *Understanding the Python GIL*. [Online; navštíveno 21.04.2019].
URL http://www.dabeaz.com/python/UnderstandingGIL.pdf

[5] Board, I. S. T. Q.: *Certified Tester Foundation Level Extension Syllabus Agile Tester* . [Online; navštíveno 21.04.2019].
URL https://www.istqb.org/downloads/send/5-agile-tester-extension-documents/41-agile-tester-extension-syllabus.html

[6] Board, I. S. T. Q.: *Certifikovaný tester-Učební osnovy pro základní stupeň* . [Online; navštíveno 21.04.2019].
URL http://castb.org/wp-content/uploads/2017/01/ISTQB_CTFL_Syllabus_v2011-CZ_1_0_0.pdf

[7] Borghi, J.: *Log Parser*. [Online; navštíveno 21.04.2019].
URL https://plugins.jenkins.io/log-parser

[8] Foundation, P. S.: *logging — Logging facility for Python*. [Online; navštíveno 21.04.2019].
URL https://docs.python.org/3/library/logging.html

[9] Foundation, P. S.: *multiprocessing — Process-based "threading" interface*. [Online; navštíveno 21.04.2019].
URL https://docs.python.org/2/library/multiprocessing.html#module-multiprocessing.pool

[10] Foundation, P. S.: *The Python Profilers — Python 2.7.16 documentation*. Září 2011, [Online; navštíveno 21.04.2019].
URL http://www.citace.com/download/CSN-ISO-690.pdf

[11] gcx11: *Dědičnost a polymorfismus v Pythonu*. [Online; navštíveno 21.04.2019].
URL https://www.itnetwork.cz/python/oop/python-tutorial-dedicnost-a-polymorfismus

[12] Hager, C.: *Python Thread Pool*. [Online; navštíveno 21.04.2019].
URL https://www.metachris.com/2016/04/python-threadpool/

[13] Hellmann, D.: *subprocess – Work with additional processes*. [Online; navštíveno 21.04.2019].
URL https://pymotw.com/2/subprocess/

[14] Jenkins.io: *Jenkins User Documentation*. [Online; navštíveno 21.04.2019].
URL https://jenkins.io/doc/

[15] martinzofka: *Optimizing Python code performance with cProfile*. alookanalytics blog, Březen 2017, [Online; navštíveno 27.03.2019].
URL https://blog.alookanalytics.com/2017/03/21/python-profiling-basics/

[16] Masnun, A. A.: *Async Python: The Different Forms of Concurrency*. [Online; navštíveno 21.04.2019].
URL http://masnun.rocks/2016/10/06/async-python-the-different-forms-of-concurrency/

[17] Masnun, A. A.: *PYTHON ASYNCIO: FUTURE, TASK AND THE EVENT LOOP*. [Online; navštíveno 21.04.2019].
URL http://masnun.com/2015/11/20/python-asyncio-future-task-and-the-event-loop.html

[18] NASA: *Mars Climate Orbiter during tests*. [Online; navštíveno 21.04.2019].
URL https://cs.wikipedia.org/wiki/Mars_Climate_Orbiter#/media/File:Mars_Climate_Orbiter_during_tests.jpg

[19] Panjwani, S.: *MULTIPROCESSING VS. THREADING IN PYTHON: WHAT YOU NEED TO KNOW*. [Online; navštíveno 21.04.2019].
URL https://timber.io/blog/multiprocessing-vs-multithreading-in-python-what-you-need-to-know/

[20] PATTON, R.: *Testování softwaru*. Computer Press, 2002, ISBN ISBN 80-7226-636-5.

[21] Petr Viktorin, M. H. a. d.: *AsyncIO*. [Online; navštíveno 21.04.2019].
URL https://naucse.python.cz/lessons/intro/async/

[22] Programiz: *Python Object Oriented Programming*. [Online; navštíveno 21.04.2019].
URL https://www.programiz.com/python-programming/object-oriented-programming

[23] Semiconductors, N.: *MCUXConfigTools_BD*. [Online; navštíveno 21.04.2019].
URL https://www.nxp.com/assets/images/en/block-diagrams/MCUXConfigTools_BD.jpg

[24] Semiconductors, N.: *MCUXpresso Config Tools Fact Shet* . [Online; navštíveno 21.04.2019].
URL https://www.nxp.com/docs/en/fact-sheet/MCUXPRESSOCFTFS.pdf

[25] Semiconductors, N.: *MCUXpresso Config Tools - Pins, Clocks, Peripherals*. [Online; navštíveno 21.04.2019].
URL http://www.nxp.com/mcuxpresso/config

[26] Semiconductors, N.: *MCUXpresso Config Tools User's Guide (Desktop)*. [Online; navštíveno 21.04.2019].
URL https://www.nxp.com/docs/en/user-guide/GSMCUXCTUG.pdf

[27] Semiconductors, N.: *Quick Start Guide for MCUXpresso Config Tools* . [Online; navštíveno 21.04.2019].
URL https://www.nxp.com/docs/en/quick-reference-guide/MCUXDWQS.pdf

[28] Solomon, B.: *Async IO in Python: A Complete Walkthrough*. [Online; navštíveno 21.04.2019].
URL https://realpython.com/async-io-python/

[29] SUMMERFIELD, M.: *Python 3: výukový kurz*. Computer Press, 2010, ISBN 978-80-251-2737-7.

[30] Wikipedia: *Mars Climate Orbiter*. [Online; navštíveno 21.04.2019].
URL https://cs.wikipedia.org/wiki/Mars_Climate_Orbiter

[31] Zbyněk Křivka, D. K.: *Principy programovacích jazyků a objektově orientovaného programování IPP – II Studijní opora*. [Online; navštíveno 21.04.2019].
URL https://docplayer.cz/7075657-Principy-programovacich-jazyku-ipp-ii-studijni-opora.html