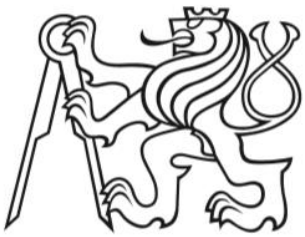
Bakalářská práce




ČESKÉ
VYSOKÉ
UČENÍ
TECHNICKÉ

F3

Fakulta elektrotechnická
Katedra řídící techniky


# Generování kódu pro distribuované řízení z modelů v Simulinku


**Petr Bláha**


**Vedoucí práce: doc. Ing. Zdeněk Hurák Ph.D.**

**Květen 2018**

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Bláha** Jméno: **Petr** Osobní číslo: **434679**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra řídicí techniky**

Studijní program: **Kybernetika a robotika**

Studijní obor: **Systémy a řízení**

## II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

**Generování kódu pro distribuované řízení z modelů v Simulinku**

Název bakalářské práce anglicky:

**Code Generation for Distributed Control from Simulink Models**

Pokyny pro vypracování:

V tomto projektu budou prozkoumány možnosti, které dostupné nástroje pro automatické generování kódu z modelů v Simulinku nabízí pro distribuované řízení. Cílem je (získat dovednost) vygenerování kódů pro více spolupracujících dílčích regulátorů modelovaných v jediném diagramu v Simulinku. Tyto kódy budou následně rozeslány na odpovídající dílčí řídicí jednotky a tam zkompilovány. Po spuštění kódů realizujících regulační úlohy budou změřené veličiny zasílány zpět do operátorského PC pro účely vizualizace i další (offline) analýzy. Ve vygenerovaných kódech bude implementována i funkčnost bezdrátové komunikace mezi jednotlivými dílčími řídicími jednotkami, které tak budou moci spolupracovat. Práce je motivována vývojem experimentální platformy pro distribuované řízení konvoje autonomních autodráhových autíček.
Dílčí pokyny:
1. Seznamte se s problematikou generování kódu pomocí nástroje Simulink Coder. Předveďte zvládnutí této základní dovednosti při uvažování jediné cílové (angl. target) platformy - v tomto projektu uvažujte platformu BeagleBone Blue. Pro předváděcí úlohu zvolte jednoduchou regulaci stejnosměrného motoru.
2. Prozkoumejte systematické možnosti generování kódu pro více cílových platforem. Předvěďte tuto funkčnost pro (alespoň) tři cílové jednotky BeagleBone Blue.
3. Prozkoumejte možnosti zakomponování bezdrátové komunikace mezi jednotlivými dílčími cílovými platformami do vygenerovaného kódu. Funkčnost opět předveďte s využitím několika jednotek BeagleBone Blue.

Seznam doporučené literatury:

[1] Dokumentace k produktu Simulink Coder firmy The Mathworks. Dostupné online na https://www.mathworks.com/help/pdf_doc/rtw/index.html.
[2] Dokumentace k produktu Embedded Coder firmy The Mathworks. Dostupné online na https://www.mathworks.com/help/ecoder/.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

**doc. Ing. Zdeněk Hurák, Ph.D., katedra řídicí techniky FEL**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **31.01.2018** Termín odevzdání bakalářské práce: **25.05.2018**

Platnost zadání bakalářské práce: **30.09.2019**

| doc. Ing. Zdeněk Hurák, Ph.D.<br>podpis vedoucí(ho) práce | prof. Ing. Michael Šebek, DrSc.<br>podpis vedoucí(ho) ústavu/katedry | prof. Ing. Pavel Ripka, CSc.<br>podpis děkana(ky) |
|---|---|---|

## III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

Prohlášení

Prohlašuji, že jsem psal tuto práci sám a uvedl jsem všechny použité zdroje. V Praze, 20. Května 2018

……

Petr Bláha

## Abstrakt

V této práci je prozkoumána možnost automatického generování kódu z modelů v Simulinku prostřednictvím nástroje Simulink Coder za účelem distribuovaného řízení. Generovaný kód bude následně nahrán na cílová zařízení, kde bude zkompilován a spuštěn. Jednotlivá zařízení by měla být schopna si navzájem předávat dat, což by mělo umožnit spolupráci těchto zařízení při řešení problému. Práce je motivovaná vývojem experimentální platformy pro distribuované řízení konvoje autonomních autodráhových autíček. Z toho důvodu je celá práce zaměřena na generování kódu pro platformu Beaglebone Blue, která obsahuje řadu vestavěných modulů, jako například drivery motorů, což usnadní vývoj a výrobu takového auta. Výsledky jsou prezentovány formou návodu, popisujícím možné dosažení daného cíle.



## Abstract

In this thesis it is discussed the possibility to automatically generate source code from Simulink models with the use of Simulink Coder. Goal is to be able to generate code for distributed systems. He code should be build afterwards and loaded on target device. Devices should be able to communicate with each other, which should lead to their cooperation. The work is motivated by experimental platform for convoy platooning of slot cars. Because of that, the work targets on developing code for platform BeagleBone Blue. Results of this thesis will be presented as acookbook for solving this problem.

# Obsah

# Úvod

Ovládání modelu popřípadě robota prostřednictvím Simulinku nebo generování řídících programů prostřednictvím Simulink Coderu do takovýchto zařízení je dnes již běžná praxe. V průmyslu je dokonce setkáváme s tím, že je generovaným kódům důvěřováno více, než těm, které napsali lidé. Na druhou stranu je velice obtížné najít informace o tom, že by se někdo pokoušel o generování kódu pro distribuované řízení i přesto, že je v dnešní době levné elektroniky distribuované řešení problémů stále populárnější.

Neexistence generován kódu pro distribuované řízení, o které jsem se dozvěděl během práce na zdokonalení experimentální platformy pro distribuované řízení konvoje autonomních autodráhových autíček, vedla k mému zájmu o toto téma a k vypracování této práce. Jelikož současné řešení je rozsáhlé a komplikované, což v podstatě znemožňovalo rozšíření mimo projekt samotný, vznikla potřeba najít jednodušší způsob a automatické generování kódu se zdálo jak cesta správným směrem.

## Hlavní výhody generovaného kódu

- Bezchybnost: Psaný kód obsahuje v průměru zhruba 25 chyb na 1000 řádků kódu. V generovaném řešení jsou však chyby jen minimálně. Simulink Coder navíc obsahuje nástroj pro kontrolu chyb, čímž se dají eliminovat nedostatky úplně.
- Usnadnění: Oproti psanému kódu je návrh v Simulinku jednoduchý. Člověk je oproštěn od potřeby myslet nad realizací svého návrhu a může se naplno věnovat pouze podstatným věcem.
- Rozšířenost: Simulinku je v komunitě lidí zabývajících se řízením velice používaný nástroj
- Zpracování dat: Je velice jednoduché z takto generovaného programu získat naměřená data pro další analýzu

# Generování kódu prostřednictvím Simulink Coderu

Sestavení modelu pro generovaný kód probíhá standardně, neboť Simulink Coder podporuje překlad nejen standartních funkcí Simulinku, Stateflow grafů, vložených subsystémů a Matlab funkcí ale i některých dalších toolboxů. Kód je generován pro programovací jazyky C nebo C++, podle preference uživatele. Je možné pouze vygenerovat zdrojové soubory, nicméně cílem je generování programů. K tomu je třeba nakonfigurovat správný cíl, na kterém bude kód spouštěn. Jelikož v rámci této práce je kód generován na platformu BeagleBone Blue, ke které je možno stáhnout podpůrný balík „Simulink Coder Support Package for BeagleBone Blue Hardware“, je toto nastavení velice jednoduché. V položce „Model Configuration Parameters“ stačí v záložce „Hardware Implementation“ vybrat u pole „Hardware board“ položku „BeagleBone Blue“, tím se nastaví veškeré potřebné informace o hardwaru a systémový cílový soubor s toolchainem pro správné generování kódu. Dále je třeba nastavit IP adresu zařízení, uživatelské jméno a heslo pro připojení k zařízení. To se nachází v již zmíněné záložce v okně „Hardware board settings“ „Target hardware resource“. Dáje je zde možné nastavit, do které složky bude zkompilovaný program nahrán. Zároveň se jedná o složku, do které se nahrávají logy vytvořené programem.

## Vkládání vlastního kódu

Jelikož cílem automatického generování kódu je mimo jiné minimalizace času stráveného nad vývojem aplikace, intuitivní způsob vkládání vlastního kódu, přímá úprava generovaných zdrojových souborů, nepřipadá v úvahu. Simulink Coder tak poskytuje několik nástrojů, kterými lze vlastní kód do generovaného programu vložit.

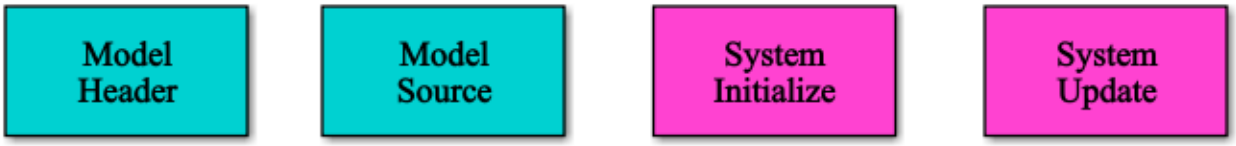
- Code Replacement Library
- Simulinkové bloky
- Voláním z matlabové funkce
- Kompilací do .mex souboru

### Code Replacement Library

Jedním z možných řešení je nastavení knihovny pro automatické nahrazení kódu. Ta je, pokud nakonfigurována, volána během samotného generování zdrojových souborů. Knihovna nahrazuje části textu podle určitých specifikovaných parametrů, jako je třeba jméno funkce, nebo označení vstupů a výstupů. Náhrada kódu nemusí vždy dopadnout podle očekávání a je proto vhodné generovaný kód zkontrolovat. Výběr použité knihovny je možné udělat v „Code Generation> Interface”.

## Bloky Simulink Coderu

Další možností je využití některého z bloků, které přidává Simulink Coder. Tyto bloky, nacházející se v knihovně Simulinku „rtwlib/Custom Code", která je dnes nazývána „Simulink Coder". Kód je vkládán do některé ze sekcí generovaného kódu podle dvou parametrů. Jméno bloku určuje sekci, do které bude text vkládán.




Obrázek 1: Ukázka bloků pro vlastní kód

Dále pak v rámci jednotlivých bloků nalezneme více polí, do kterých lze psát. Ty určují pozici v rámci jednotlivých částí sekcí. Bloky označené jako „Model" se dají použít například pro deklaraci funkcí, globálních proměnných nebo importování některé externí knihovny. Bloky označené jako „System" vkládají kód do již volaných funkcí. Tento rozdíl je také vyjádřen barevným odlišením bloků. Například „Systém initialize" vkládá kód do inicializační funkce generovaného programu a „Systém update" vkládá do funkce vypočítávající krok běhu programu.U systémových bloků je možné zvolit uložení kódu na začátek, do středu a na závěr. Takto vložený kód je však až za odpovídajícím automaticky generovaným kódem.

## Volání zfunkce Matlabu

Vložení bloku umožňujícího volání matlabového kódu otevírá cestu pro vlastní kód prostřednictvím Matlab Coderu. Do Simulinku stačí vložit blok „MATLAB function", který dovoluje definici vstupů i výstupů. Otevřením tohoto bloku se otevře příslušný soubor obsahující deklaraci volané matlabové funkce. V této funkci se musí zavolat Matlab Coder funkcí `coder.ceval('function_name', args)`. Ta zavolá funkci definovanou v některých ze zdrojových souborů. Jelikož Matlab předem neví, jaký datový typ má výstup volané funkce, je třeba proměnnou, do které bude tato hodnota uložena předem inicializovat. V případě, že se volaná funkce nachází v běžných knihovnách, které je možno importovat, třeba říct Simulinku, kde tyto soubory hledat. Simulink má dvě místa, s odpovídajícím uživatelským rozhraním, kde je možné provést tuto konfiguraci. První se nachází v „Model Configuration Parameters" v záložce „Simulation Target". Zde se určují soubory používané při simulacích volaných tlačítkem „Run". Druhá v záložce „Code Generation>Custom Code" určuje, které soubory budou použity při volání „Deploy to Hardware". V této sekci je také možno zatrhnout, že budou použity stejné soubory, které jsou nastaveny v „Simulation Target". Tato nastavení obsahují dvě textová pole. Jedno slouží k přímému vkládání kódu do určitých sekcí a druhé, do kterého se mezerou odděleně píší zdrojové

soubory, popřípadě složky, kde se tyto soubory nacházejí. Simulink, jak se zdá, přijímá jak cesty relativní, tak absolutní.

## Struktura generovaného kódu

Pro vládání vlastního kódu je vhodné znát strukturu, jakou mají generované soubory. Tím lze předejít nevhodnému umístění kódu, což by mohlo mít za následek chybu kompilace nebo hůře nevyzpytatelné chování aplikace.

### main.c

Hlavní soubor, který definuje spouštění a ukončování aplikace je ert_main.c. Jméno se může lišit podle zvolených parametrů generace kódu. Zde ve funkci main, která je vstupní funkcí programu dochází napřed k inicializaci externích zdrojů, například je zde inicializována Robotics API, která je použita v BeagleBone Blue pro komunikaci s LED diodami, drivery motorů, čítačem pulsů a senzory jako třeba barometr. Dále je inicializován vlastní model a real-time funkce Simulink Coderu. Je zde také inicializován semafor pro řízení kroků aplikace. Dále jsou zde funkce ošetřující správné ukončení aplikace a funkce baseRateTask zajišťující běh celé aplikace.Ta pomocí while cyklu volá funkci vypočítávající krok modelu. Zde také pomocí semaforu dochází ke správnému časování dané funkce tak, aby odpovídalo nastavené vzorkovací frekvenci.

### Soubory modelu

Dále jsou generovány zdrojové soubory pro každý použitý model. Vždy v páru zdrojový a hlavičkový soubor. Hlavičkový soubor .h zajišťuje import potřebných knihoven, deklarace proměnných a funkcí. Celý soubor je ošetřen podmínkou pro preprocesor`#ifndef`, čímž je zajištěno, že nedojde k dvojité deklaraci. Dále má takovouto podmínku část importů a většina deklarací proměnných. Vlastní kód vložený do tohoto souboru je také zahrnut v této podmínce.

Veškeré funkce jsou pak definovány ve zdrojovém souboru .c. Ten obsahuje funkce step, initialize a terminate. Funkce step zajišťuje výpočet jednoho kroku modelu.

Dále je generováno ještě několik souborů definující například použité datové typy.

### Code Report

Veškeré generované soubory je možné prohlédnout prostřednictvím Code Reportu. Tato funkce zajistí vygenerování přehledu o generovaném kódu ve formátu HTML. Mimo samotné zdrojové soubory jsou

zde uvedeny obecké informace o vygenerovaném kódu a také odkazy na kód, který byl vygenerován pro jednotlivé Simulinkové bloky. Tato funkce je ve výchozím nastavení Simulinku zapnutá a probíhá po sestavení aplikace. V případě, že jsou zdrojové soubory zachovány, je možné zprávu generovat zavoláním funkce `coder.report.generate`.

## Urychlení generace

Během testování generování kódu na jednoduché aplikaci, která pouze blikala diodou a odečítala teplotu, jsem naměřil, že Simulinku zabere celí proces generování a kompilace přibližně dvě minuty. To je sice relativně krátká doba, nicméně pro větší aplikace by tento čas mohl výrazně narůst. Nutnost urychlení celého procesu se také projeví při vývoji aplikace pro distribuované systémy, neboť je třeba ji generovat pro každý cíl zvlášť. Simulink Coder nabízí několik způsobů, jak urychlení dosáhnout.

- Incremental build
- Paralelní sestavení
- Generování pouze kódu
- Vypnutím Code Reportu

### Incremental build

Jedním ze způsobů, jak Simulink urychluje generování kódu je takzvaný Incremental Build. Jedná se o metodu, při které dochází k novému generování pouze v případě, kdy dojde ke změně modelu, nebo submodelu. Generován znova je pouze změněný model a zbytek je použit z předchozí generace. V Simulinku je dále možno nastavit, jakým způsobem bude určena změna. Tím způsobem je možné znovu generování i úplně vypnout.

### Paralelní sestavení

Zejména pro rozsáhlejší modely se ukáže být vhodné použití paralelního sestavení. K tomu je zapotřebí Parallel Computing Toolbox. Matlab v tomto případě dokáže urychlit sestavení modelu tím, že podsystémy a modely obsažené v hlavním modelu sestaví ve vlastním vlákně prostřednictvím workera zmíněného Parallel Computing Toolboxu. K určení, kolik workerů je třeba inicializovat a jak efektivní tento postup je se dá využít nástroje „Build Status“. Ten umožňuje pozorovat průběh sestavení kódu, využití workerů a dobu kompilace jednotlivých částí modelu. Paralelní sestavení se umožní inicializací workerů, například zavoláním funkce `parpool` dále je třeba v modelu nastavit „Enable parallel model reference builds“ v panelu „Model Referencing“.

### Generováním pouze kódu

Sestavení aplikace, které obvykle probíhá po generování kódu, nemusí být vždy nutné. Pro případy, kdy se generovaný kód stává součástí větší aplikace je proto možné zvolit pouze generování zdrojových souborů. Mimo ně je generován také makefile pro sestavení příslušné aplikace. Tato možnost se dá zvolit zatrhnutím „Generate code only“ checkboxu v „Model Configuration Parameters“ v záložce „Code Generation“.

### Vypnutím Code Reportu

Code report je velice užitečný nástroj při návrhu aplikace, zvláště pokud je do ní vkládán vlastní kód. Nicméně během ladění parametrů jako je třeba návrh regulátoru je tento nástroj zcela zbytečný a pouze vede k prodloužení doby potřené ke generování kódu. V nastavení modelu v „Code Generation> Report“ je tedy možné tuto funkci vypnout.

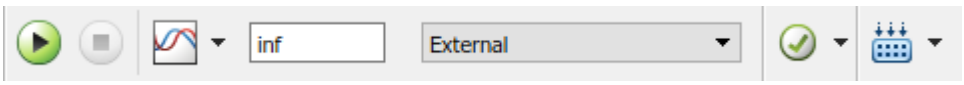
## Odečítání hodnot

Získávání hodnot z běžícího programu je velmi důležité jak pro samotný návrh programu, tak pro optimální nastavení, popřípadě je to nezbytné, pokud je cílem programu shromažďování dat. Simulink Coder nabízí několik způsobů jak data odečítat. Zde jsou uvedeny některé z nich.

- External Mode
- Logování do souboru
- API Interface

### External Mode

Nejjednodušším a velice efektivním způsobem pro získání dat z chodu programu je použití externího módu. Jedná se o způsob simulace, ve kterém je z modelu vygenerována aplikace Simulink Coderem a přitom zůstává spojení se Simulinkem, prostřednictvím kterého jsou nahrávána data do simulinkových scopů.Toto spojení je realizováno prostřednictvím TCI/IP protokolu, což zajišťuje spolehlivý přenos dat téměř v reálném čase.




Obrázek 2: Ukázka ovládacího panelu Simulinku

External mód se spouští oproti běžnému překladu modelu tlačítkem „Run“ namísto „Deploy to Hardware“. Díky tomu přebírá takto generovaný kód některé vlastnosti simulace. Jako externí zdrojové soubory jsou použity soubory definované pro normální simulaci namísto těch definovaných pro generování kódu. Během kompilace je také nastavena hodnota EXT_MODE, což umožňuje další flexibilitu externího kódu pro tento případ. Při generování kódu se také mohou lišit jména některých proměnných. Zjevnou nevýhodou tohoto způsobu je nutnost běhu Simulinku.

## Logování do souboru

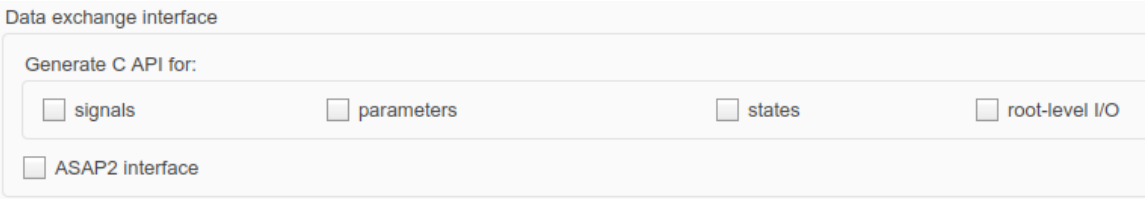
Nezávislost běhu vygenerovaného programu na Simulinku se dá zařídit například logováním do souboru. Simulink Coder umožňuje logování dat, jako normální simulinkový model, které je nastaveno prostřednictvím Data Import/Export. Oproti normálnímu modelu však nepodporuje logování signálů ve formátu dataset. Dále není možné logovat stavy signálů. Do souborů mohou být dále ukládány data prostřednictvím bloku toWorkspace a data scopů, u kterých je nastaveno, aby ukládaly data.Výchozí velikost bufferu dat je 1024b.Je-li aplikace spuštěna prostřednictvím Simulinku, mají logy omezenou velikost na 16KB, poté dojde k vtvoření nového.

Data logovaná těmito způsoby jsou ukládána do souboru, nesoucí jméno modelu, za podtržítkem následuje číslo určující, o kolikátý běh aplikace se jedná a za dalším podtržítkem o kolikátý soubor se jedná v rámci jednoho běhu aplikace. Dále je možno použít blok „to File“, u kterého je možno nastavit jméno souboru, do kterého budou data logována. K tomu aby vygenerovaná aplikace podporovala logování dat, je třeba nastavit „Enable MAT-file logging” v nastavení „Code Generation> Interface”.

Soubory s logy jsou vytvářeny ve stejném adresáři, ve kterém se nachází aplikace. Pro jejich snadné nahrání vývojový počítač je možné použít matlabového příkazu `getFile`. Ten požaduje dva vstupn9 argumenty. První je odkaz na strukturu vrácenou příkazem `beagleboneblue`, ten určí, ze kterého zařízení budou data přenášena. Druhým argumentem je název souboru, který má být zkopírován. Název souboru může také osahovat regulární výraz, takže třeba příkaz`beagleboneblue(target, 'example_1_*.mat')` přenese veškeré logyvygenerované při prvním spuštění aplikace.

## API Interface

Teoreticky nejmocnějším nástrojem pro získání dat je C API, které je možné vygenerovat spolu s aplikací. Takovéto rozhraní umožňuje přístup externího programu k datům vygenerované aplikace. Oproti předchozím metodám, které uměly pouze data získávat, může být toto rozhraní nakonfigurováno tak, že umožní například i změnu parametrů simulinkových bloků za chodu aplikace. Prostřednictvím „Code Generation> Interface” je možné nastavit, zda bude toto API generováno a co vše bude zahrnovat.




Obrázek 3: Rozhraní pro nastavení generovaného rozhraní

## Simulink Coder s BeagleBone Blue

Simulink Coder, od verze Matlabu 2017b umožňuje instalaci podpůrného balíku, který zajišťuje propojení s hardwarem BeagleBone Blue. Simulink Coder Support Package for BeagleBone Blue Hardware, jak zní tento balík celým názvem,obsahuje řadu nástrojů pro usnadnění vývoje kódu na tuto platformu. Existují však i další podpůrné balíky pro jiné platformy. Jejich použití je velice podobné tomu zde popsanému a liší se jen v detailech jako je například jméno funkce nebo simulinkového bloku. Dále mohou obsahovat některé další bloky specifické pro danou platformu.

### Propojení s BeagleBone Blue

První propojení s BeagleBone je možné provést dvěma způsoby. Prostřednictvím USB kabelu, nebo připojením na hotspot, který zařízení vytváří. Po připojení kabelu dojde na zařízeních s operačním systémem Windows k automatické instalaci ovladače pro BeagleBone. Po jeho instalaci se vytvoří virtuální spojení, které přiřadí zařízení IP adresu. Ta je liší pro různé typy připojení. Prostřednictvím Wi-Fi acces pointu je to http://192.168.8.1, a prostřednictvím USB je to buď http://192.168.6.2, nebo http://192.168.7.2. Poté, co je navázáno spojení, je třeba toto zařízení nakonfigurovat. O to se postará již zmíněný podpůrný balík, stačí tedy prostřednictvím Add-On manageru spustit nastavení tohoto balíku. To otevře interaktivní okno umožňující instalaci ovladače zařízení, následně prostřednictvím uvedené IP adresy provede prostřednictvím SSH připojení konfiguraci zařízení. V rámci této konfigurace je také možné nastavit připojení Wi-Fi. Tím je deska připravena.

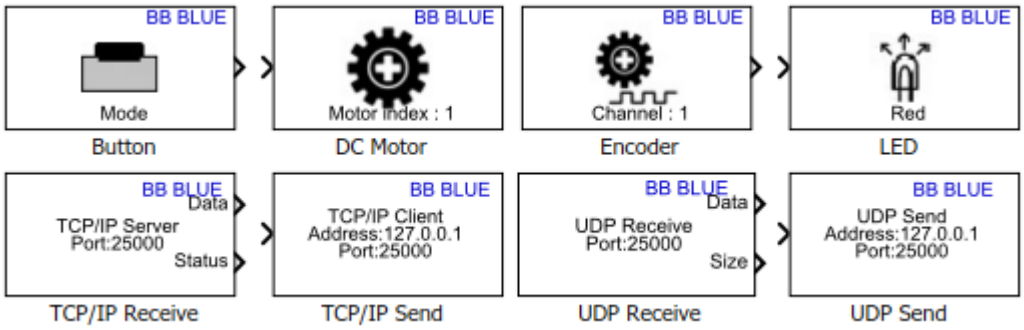
Jakmile je deska nakonfigurována, je již snadné provést se k ní připojit. To uděláme zavoláním funkce `beagleboneblue` v Matlabu. Ta otevře SSH připojení s nakonfigurovaným zařízením na portu 443.

### Knihovna pro BeagleBone Blue

Simulink v rámci podpůrného balíku pro BeagleBone Blue dostává několik nových bloků pro vývoj aplikací na tuto platformu. Jedná se o následující bloky

- Button
- DC Motor
- Encoder

- LED
- TCP/IP Receive
- TCI/IP Send
- UDP Receive
- UDP Send




Obrázek4: Ukázka bloků pro BeagleBone Blue

## Button

Tento blok čte stav tlačítka na zařízení. Tento blok dokáže sledovat tlačítka „Mode“ a „Pause“. Výstupem tlačítka je bezznaménkový osmibitový inteager nabývající pouze hodnot 0 a 1. Jedna pro stisknuté tlačítko, nula pro uvolněné. Dále je možné nastavit, s jakou frekvencí bude tlačítko kontrolováno.

## DC Motor

Prostřednictvím tohoto bloku lze ovládat driver pro jeden ze čtyř možných DC motorů. O který z motorů se jedná, se nastavuje v rámci bloku podle indexu motoru. Dále se také nastavuje, jakým způsobem bude motor reagovat na nulový výkon. Ten bude buď nechán ve stavu volného otáčení, nebo bude brzděn. Motor přijímá signál v rozsahu -100 až 100, kde znaménko určuje směr rotace motoru a absolutní hodnota dodávaný výkon.

## Encoder

BeagleBone obsahuje čtyři vstupy pro čítače pulsů. Tento blok vrací počet pulzů jednoho z nich od jeho posledního restartování jako klasický inteager. U tohoto bloku je také možné nastavit, s jakou frekvencí odčítá pulzy. Dále se dá nastavit, kdy dojde k restartování počtu pulzů. Restartování buď prováděno není, nebo je provedeno při každém odečtení hodnoty nebo je možné ho provést externím signálem.

## LED

Tento blok, jehož vstupem je osmibitový bezznaménkový inteager, ovládá jednu ze dvou diod, které jsou určeny pro uživatele. Jedná se o červenou a zelenou diodu v prostoru tlačítek. Dioda je vypnutá pro hodnotu nula, jinak je rozsvícená.

### Bloky UDP

Tyto bloky, jak již název napovídá, slouží ke komunikaci prostřednictvím protokolu UDP. Oproti blokům přistupujícím k hardwaru je možné jejich použití v rámci normální simulace v prostředí Simulink. Vstupem, popřípadě výstupem z takovéhoto bloku je jednorozměrné pole až 32 bitových inetegerů, popřípadě singlů nebo doublů. U těchto bloků se nastavuje port, na kterém je komunikace prováděna. Dále, u UDP Send se nastavuje, na jakou IP adresu budou data posílána. Tato adresa může být konkrétní adresa některého zařízení, popřípadě adresa 255.255.255.255, což je adresa pro vysílání do celé sítě. Oproti tomu blok Receive musí mít navíc nastavenou frekvenci, se kterou bude data odečítat, velikost přijímaných dat a jejich datový typ. Tento blok navíc obsahuje druhý výstupní port, který udává, zda jsou na portu připravena data k přijetí.

### Bloky TCP/IP

TCP/IP bloky se z hlediska signálů používají stejně, jako bloky UDP. Vzhledem k faktu, že TCP/IP je oproti UDP bezpečná komunikace, je však třeba specifikovat některé další údaje. Na blocích je třeba specifikovat, který z nich se chová jako server a který jako klient. U klienta je třeba specifikovat IP adresu serveru a port, kdežto u serveru stačí pouze přijímaný port.

## Přístup k ostatním senzorům

BeagleBone Blue má v sobě vestavěnou řadu dalších senzorů, ke kterým však Support package nedodává bloky pro odečítání jejich hodnot. Z tohoto důvodu je pro jejich čtení zapotřebí vlastní kód. Jednou z možností, jak přistoupit k těmto senzorům je sepsání vlastního driveru, který by byl následné volán prostřednictvím Simulink bloku Matlab Function jak je tomu popsáno v předešlé části této práce. Psaní vlastního driveru je však zbytečná práce, neboť tyto drivery již existují. Jsou zahrnuty v Robotics API od společnosti Strawson Design. Tato API je použita pro již existující bloky, které přistupují k hardwaru zařízení, čímž se nabízí ji použít i pro další senzory.

### Volání Robotics API

Na první pohled se zdá přístup k senzorům prostřednictvím tohoto API jednoduché. Stačí pouze napsat vlastní kód, který bude volán přes matalovou funkci volanou v Simulinku. Tento přístup však narazí na problém. Jelikož není Robotics API nainstalována na kompilujícím počítači, nelze ji importovat bez použití upraveného makefile. I přesto, že je tento takovíto makefile použit pro sestavení celého projektu, matlabová funkce je na kód přeložena vlastní kompilací, která Robotics API nezvládne přeložit. Namísto toho je však možné použít drobný trik s preprocesorem.

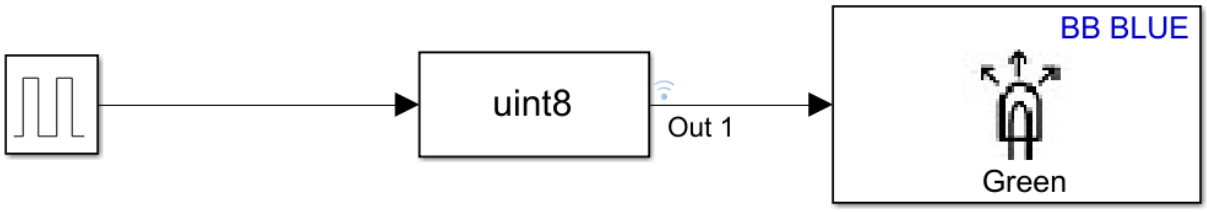
Při použití jakéhokoliv bloku, co využívá Robotics API jsou importovány všechny potřebné soubory pro použití celé API. Díky tomu je možné pomocí bloku „Model Source“ nadefinovat potřené funkce využívající tuto API a dojde k jejich správnému přeložení. Následně je potřeba zavolat požadovanou

funkci. To se udělá opět prostřednictvím volání funkce`coder.ceval` přes blok matlabové funkce. Jelikož této funkci nelze udat jako zdroj soubor, ve kterém je funkce reálně implementována, je potřeba vytvořit další zdrojový soubor obsahující definici funkce. Aby se zabránilo duplicitní dané funkce, je třeba ji zahrnout do preprocesorové podmínky. Například klíčové slovo MODEL není definováno během překladu bloku matlabové funkce, ale je již nadefinované při sestavování celého programu.

Jelikož je přístup k některým senzorům relativně pomalý, je v Robotics API jejich čtení rozděleno do dvou funkcí. První funkce získává hodnotu ze samotného senzoru. Takovéto funkce jsou pomalé, proto jsou definovány tak, že data ukládají do předem alokované datové struktury, na kterou se jim v rámci volání předává ukazatel. To umožňuje například jejich volání v samostatném vlákně, čímž nezpomalují simulaci. Požadovaná data se pak dají získat buď voláním další, již rychlé, funkce popřípadě přímo extrahovat z příslušné datové struktury.

## Hello world

Jak je již v programování zvyklostí, prvním programem, který programátor napíše, je program Hello World. Tento program, který pouze vypíše tento text. Samozřejmě, takový program nemá v rámci Simulinku žádný smysl a i samotná implementace by byla obtížná. Za určitou obdobu tohoto programu by se dal považovat program generovaný ze schématu na obrázku 1.




Obrázek 5: Blokové schéma jednoduchého programu

Z tohoto schématu je jasně patrné, že tento program bude pouze blikat LED diodou na frekvenci nastavené v pulzním generátoru. I přesto, že se jedná o jednoduché schéma, které nikterak nepřispívá v řešení případného problému, jedná se o velice užitečné schéma a jeho zařazení do vyvíjených aplikací bych doporučil, neboť umožňuje rychlou a nenáročnou kontrolu běhu programu. Blikající dioda nejen že zaručuje, že byl program spuštěn, ale také ukazuje, jestli se program ve svém běhu nezasekl. Navíc se takto dá velice jednoduše zkontrolovat, zda je zařízení nakonfigurováno správně.

# Generování kódu pro distribuované systémy prostřednictvím Simulink Coderu

V této části práce budou použity poznatky z předchozích kapitol ke generování kódu pro distribuované systémy. Bude zde představeno několik možných přístupů k řešení spolu s jejich výhodami a nevýhodami jak pro návrh, tak pro samotné použití. Dále zde budou popsány možnosti vzájemné komunikace mezi jednotlivými zařízeními. Vzhledem k důvodu sepsání této práce je realizována pouze jedna varianta, která zde bude podrobně rozebrána, u ostatních bude pouze nastíněn způsob jejich realizace. Rozebrány zde budou tyto přístupy k řešení problému.

- Externí přístup
- Matlabový skript
- Simulink Real-Time

## Externí přístup

Pod tímto pojmem jsou zahrnuty různé možnosti, které jsou realizovatelné mimo prostředí Matlabu. V Simulinku bude v takovémto případě vygenerován zdrojový univerzální kód pro jedno zařízení, popřípadě bude sestavena jedna aplikace.

V nejjednodušším případě se může jednat o jednoduchý skript, který vezme aplikaci vygenerovanou a sestavenou Simulinkem a nahraje ji do požadovaných zařízení distribuovaného systému. Tím odpadá nutnost sestavení aplikace pro každé zařízení a celé řešení je tak velice rychlé i pro systémy osahující velké množství zařízení.Tento přístup však nemá žádnou flexibilitu a dá se proto použít jen v případě spolupráce více rovnocenných zařízení.

O něco větší možnost nabízí vlastní sestavení aplikace. Tím je možné provést drobné zásahy do kódu například prostřednictvím preprocesorových podmínek a provést tak nastavení pro jednotlivá zařízení, ovšem to však povede ke zpomalení vývoje, neboť bude muset být každá aplikace sestavena samostatně.

Změnit některé parametry je také možné pomocí vygenerovaného rozhraní. Tento způsob je vhodný, pokud je generovaný kód použit v rámci větší aplikace, která v případě potřeby muže přes toto rozhraní provést například optimální nastavení regulátoru.

V případě větších rozdílů v rámci celého systému může však být problém navrhnout vhodné Simulinkové schéma. Popřípadě bude generovaný kód zbytečně složitý. V takovém případě je lepší použít některou z jiných možností. Z tohoto důvodu nebyla tato možnost vyvíjena. Nicméně zde uvedu příklad takovéhoto skriptu pro Bash.

### Možná realizace problému

Nejprve provedeme sestavení pro  jedno zařízení za použití deploy to hardware, následně zavoláme následující skript, který se postará o nahrání programu do ostatních zařízení a jeho následného spuštění. Jedná se o jednoduchý skript, který pomocí kopírování přes SSH nahraje aplikaci do všech cílových zařízení a zařídí její spuštění.

```
#!/bin/bash


devices = (“10”, “11”, 12”)
        for dev in ${devices[@]}; do
            scp ./example user@10.1.1.$dev:~/
            ssh user@10.1.1.$dev
            ./example &
            exit
        done
```

## Matlab skript

Z hlediska vývoje samostatné aplikace za použití Simulink Coderu by však bylo vhodnější udržet celý vývojový proces uvnitř prostředí matlabu. Matlab obsahuje celou řadu užitečných funkcí, které to umožní. Je možné tedy napsat skript, který se o vše postará. Jelikož je vývoj prováděn v prostředí matlabu, je zde veliká možnost specializace kódu pro jednotlivá zařízení. Je tedy možné velmi pohodlně změnit kteroukoliv hodnotu v modelu. Také je možné využívat varianty modelu popřípadě použít pro některá zařízení model zcela odlišný. Aplikace může být tímto způsobem vyvíjena i pro zcela odlišná zařízení.

Složitost tohoto skriptu se tedy bude odvíjet od požadavků na vyvíjený systém. Zásadní nevýhodou oproti předchozímu řešení je potřeba zkompilovat kód pro každé zařízení systému, čímž se může značně prodloužit délka kompilace. V předchozí části práce je proto uvedeno několik způsobů jak tento proces urychlit. S tím souvisí i další potencionální problém a to sice nutnost synchronizace zařízení v rámci vyvíjeného systému. Za další nevýhodu je možné považovat i fakt, že celý systém nemusí být zahrnutelný v rámci jednoho simulinkového schématu ale může být rozložený do více modelů. V rozsáhlejších systémech se to však může ukázat jako výhoda, neboť je možné vyvíjet nezávisle více částí systému najednou.

## Použitelné matlabové funkce

Zde budou uvedeny některé matlabové funkce umožňující sepsání skriptu pro správnou distribuci aplikací v rámci systému.

- beagleboneblue
- slbuild/rtwbuild
- isModelRunning
- runModel
- stopModel

### Funkce beagleboneblue

Distribuce kódu do cílových zařízení je prováděna prostřednictvím protokolu SSH. Funkce `beagleboneblue`zajišťujenavázání tohoto spojení. Funkce buď nevyžaduje žádné argumenty, v takovém případě naváže spojení s posledním propojeným zařízením, nebo vyžaduje argumenty tři. Prvním argumentem je IP adresa cílového zařízení. Další dva jsou pak v tomto pořadí uživatelské jméno a heslo uživatele cílového zařízení. Návratová hodnota je odkaz na strukturu obsahující informace o spojení. V případě, že funkce spojení nenaváže, vyhodí chybovou hlášku.

Tato funkce pochází z podpůrného balíku pro BeagleBone Blue, pro připojení na jiná zařízení však existují obdobné funkce.

### Funkce slbuild/ rtwbuild

Tyto funkce sestaví model, jehož jméno bylo argumentem funkce. Funkce `rtwbuild` navíc dokáže sestavit a nahrát do zařízení subsystém modelu, čímž může kód pro celý systém být obsažen v jednom simulinkovém modelu.

### Funkce isModelRunning

Tato funkce umožňuje zjistit, zda model běží na cílovém zařízení. Funkce vrací logickou jedničku, pokud model běží. Jako vstupní argument je třeba udat odkaz na spojení, vrácený předchozí funkcí a jméno modelu, jehož běh je kontrolován.

### Funkce runModel

Tato funkce zajistí spuštění požadovaného modelu. Vstupní argumenty jsou stejné jako pro funkci `isModelRunning`.

### Funkce stopModel

Tato funkce zajistí zastavení požadovaného modelu. Vstupní argumenty jsou stejné jako pro funkci `isModelRunning`. Tato funkce je realizována prostřednictvím bashového příkazu `killall`,

Který však není na zařízení BeagleBone nainstalován. Pro jeho použití je tedy potřeba nainstalovat knihovny psmisc.

## Ukázka realizace

Zde je uvedena možná realizace pomocí matlabového skriptu, který může nahrávat různé modely do cílových zařízení. Tento konkrétní kód byl úspěšně otestován na několika zařízeních BeagleBone.

```
models = ['Model1'; 'Model2'];
devices = ['192.168.2.177'; '192.168.2.128'];
dataCount = 2;


numberOfDevices = length(devices(:,1));
defaultData = zeros(dataCount, numberOfDevices);


tic()
for i =1:numberOfDevices
   ip = devices(i,:);
   b = beagleboneblue (ip, 'debian', 'temppwd');
   id = i;
   slbuild (models(i,:))
   runs = isModelRunning(b, models(i,:));
if runs
      fprintf ("Running at %s\n", ip);
end
   toc()
   fprintf("-----------------------------------------------------------\n");
end
```

Jak je patrné, celí skript je velice jednoduchý. Navíc obsahuje některé řádky, které jsou zdekvůli zajištění komunikace a odečítání dat. První dva řádky kódu specifikují, který model bude nahrán do kterého zařízení. Model je reprezentován svým jménem a zařízení svojí IP adresou. Skript pak iteruje přes každé zařízení, pro něhož nakonfiguruje Simulink, přenastaví model a následně vygeneruje kód, sestaví a spustí jej na daném zařízení. Jako poslední část smyčky je pak ověření, zda byl kód skutečně spuštěn. Kód by samozřejmě mohl být dále vylepšován, například by mohl potřené proměnné nahrávat ze souboru, popřípadě by mohl být lépe ošetřen případ, pro který se z nějakého důvodu sestavená aplikace nespustila.

Problémem však je, že žádný z dosud popsaných způsobu nerealizuje datové spojení dvou aplikací systému. Nejvíce se tomu blíží generování jednotlivých aplikací ze submodelů, u něhož je možné snadné vygenerování rozhraní, nicméně ani to samo o sobě nerealizuje spojení mezi více zařízeními.

## Komunikace mezi více zařízeními

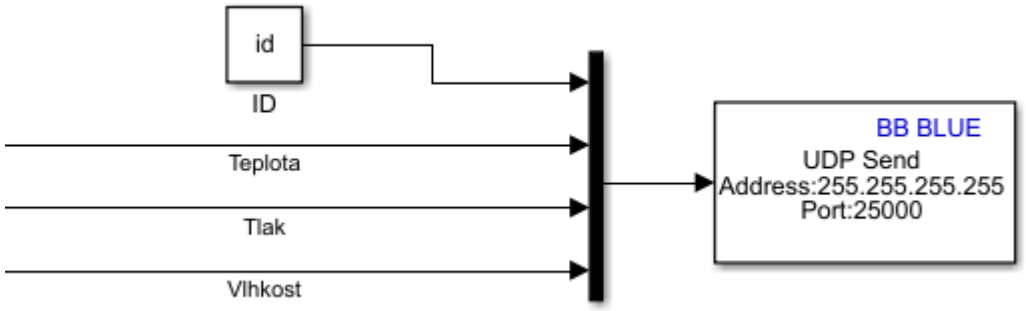
Tato část se tedy bude zabývat tím, jak problém komunikace vyřešit. Například externí mód používá pro datový přenos do simulinkových scopů protokol TCP/IP. Tím je zajištěn spolehlivý přenos dat spět do Simulinku. Jelikož podpůrné balíky obsahují bloky pro komunikaci prostřednictvím sítě, nabízí se možnost takovéto spojení realizovat vlastnoručně. Mnou provedená realizace používá namísto protokolu TCP/IP protokol UDP. Díky tomu je možné vysílat do celé sítě a sledovat komunikaci v rámci Simulinku.Není tedy třeba používat externí nástroj pro monitorování komunikace jako je třeba Wireshark. Použitý protokol se však dá velice jednoduše změnit, aby vyhovoval potřebě konkrétního systému.

Jelikož tyto boky neumožňují rozlišit, ze kterého zařízení byla data odeslána, pokud ovšem nebude použita na každé spojení vlastní sadu bloků se svým portem, je zde navrhnut a realizován jednoduchý komunikační protokol, který tento problém odstraní. Specifikace tohoto protokolu by se dala popsat následovně.

1. Veškerá komunikace v rámci jednoho portu se musí řídit těmito pravidly
2. Odesílaná data mají stejnou délku
3. Na prvním místě je vždy uvedeno id zařízení, ze kterého byla data odvysílána
4. Id zařízení musí být kladná
5. Za id zařízení následují ostatní hodnoty
6. Data by měla být vždy řazena tak, aby byly odpovídající hodnoty na stejných indexech
7. Pokud nemáme data pro splnění bodu 6, doplníme daty mimo očekávaný rozsah
8. Příjemce dat by měl být připraven na možnost přijetí výchozích hodnot z bloku Receive
9. Je třeba udržet data do doby, ve které přijdou nová validní data

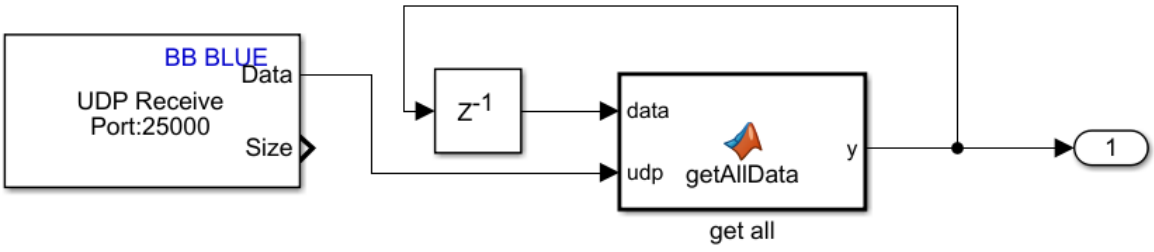
## Realizace komunikace

Na následujících obrázcích zachycující Simulinková schémata je ukázána jedna z možností jak tento protokol realizovat.




Obrázek 6: Ukázka odesílání dat v rámci protokolu

Jak ukazuje schéma na obrázku 6, je odesílání dat tímto protokolem velice jednoduché, jediné na co si stačí dát pozor je pořadí signálů. Na straně příjemce je však situace o něco složitější.




Obrázek 7: Ukázka příjmu dat v rámci protokolu

```
function y = getAllData(data, udp)
if length(data(1,:)) == (length(udp) -1)
if (udp (1) > 0)
        data (udp(1), :) = udp(2:length(udp));
end
end
y = data;
```

Jelikož můžou být přijímána data z více zdrojů, je třeba je nejprve utřídit. To je provedeno matlabovou funkcí getAllData. Ta nejprve zkontroluje validitu dat a následně provede jednoduché přiřazení do pole data. Jedná se o dvojrozměrné pole, v němž jsou po řádcích ukládánypřijatá data od jednotlivých zařízení. Výstupní data jsou použita zároveň jako vstupní hodnota pole data, tím je zaručeno, že data zůstanou zachována i pokud zrovna není dostáván validní signál. Pro eliminaci algebraické smyčky a také na určení výchozí hodnoty pole data je zde použit blok zpoždění. Výchozí hodnota je nastavena na základě znalosti přijímaných dat pomocí matlabu.Funkce může být dále velice jednoduše upravena, pokud je třeba odečítat hodnoty pouze z některých zařízení.

Díky tomu, že komunikační bloky pro BeagleBone fungují i v rámci normální simulace, je možné tímto protokolem sledovat všechny potřebné signály. K tomu je však vhodné napřed synchronizovat čas simulace s časem počítače, jinak by byla výsledná data zkreslená rozdílnou rychlostí simulace. Toho lze dosáhnout například za použití Simulink Real-Time toolboxu. Ten obsahuje blok Real-Time Sync, který tuto synchronizaci zařídí. V rámci tohoto bloku lze nastavit, s jakou frekvencí je běh simulace synchronizován a po kolika vypuštěných krocích dojde k přerušení simulace.

# Generování kódu pro distribuované systémy prostřednictvím Simulink Coderu pro platformu BeagleBone

Hlavním cílem byla možnost zahrnutí celého distribuovaného systému so jednoho simulinkového schématu, ze kterého by pak mohl být generován kód pro všechna jeho zařízení. Předchozí řešení tuto možnost nenabízeli, popřípadě šla použít jen na některé systémy.V této části bude tedy představen kód, který toto zvládá. Jelikož tento kód používá veliké množství prvků z knihovny pro BeagleBone Blue, není jeho použití pro jiná zařízení možný bez větších zásahů. Vzhledem k faktu, že tento kód velice rozsáhlý, je jako celek umístěn v přílohách tohoto dokumentu. Nejprve zde bude popsáno jeho použití a současná omezení, následně zde bude rozebráno, jakým způsobem kód funguje a nakonec bude po logických celcích představen kód jako takový. V současné době nese skript označení runFromOne a je obsažen ve stejnojmenném .m souboru.

## Užívánískriptu runFromOne

### Vytvoření kompatibilního Simulinkového schématu

Simulink musí být nakonfigurován tak, aby zvládal generovat kód pro jedno zařízení BeagleBone, jak je popsáno v předchozí části této práce. Cesty k vlastnímu kódu však musí být udány absolutní cestou. Schémata, ze kterých má být generován kód, musí být ve vlastním subsystému pro každé zařízení. Tento subsystém by neměl obsahovat blok enable na své nejvyšší úrovni. Pokud je třeba mít daný subsystém spínaný, stačí ho zabalit o úroveň hloub. Komunikace mezi zařízeními a kořenovým modelem prostřednictvím I/O portů je realizovatelná pro samostatný signál o datovém typu double. Skopy jsou relevantní pouze v hlavním modelu. Hlavní model může osahovat vlastní blokové schéma provádějící výpočty, nicméně to nedoporučuji kvůli zpoždění, které způsobuje komunikace mezi zařízením a modelem. Další nevýhodouje pak nutnost běhu Simulinku. Pro synchronizaci hlavního modelu s časem je třeba využít synchronizačního bloku, například je možné použít blok “Real-Time Sync” z knihovny Simulink Desktop Real-Time. Jelikož se jedná o externí placenou knihovnu, nebyl v tomto skriptu použit.

### Nastavení skriptu

Nastaven skriptu je velice jednoduché. Stačí uložit tři proměnné.

```
rootModel = 'All_In_One';
models = ['M1'; 'M2'; 'M3'];
devices = ['192.168.0.110'; '192.168.0.111'; '192.168.0.112'];
```

První proměnná, `rootModel`, nastavuje jméno simulinkového schématu. Druhá,`models` , určuje, jaké submodely hlavního modelu budou zkompilovány na cílová zařízení. Třetí,`devices` , udává IP adresy

těchto zařízení. Adresy jsou přiřazeny modelům podle pořadí v daném poli. Tím je nastavení skriptu dokončeno.

### Ovládání

Poté, co je jak model, tak skript nakonfigurován, stač pouze spustit požadovanou sekci skriptu. První část zařizuje kompilaci a spuštění modelu na všech cílových zařízeních i běh v rámci Simulinku. Na konci celého souboru jsou pak zbylé dvě části. Ty jsou velice krátké a starají se o zastavení celého systému a jeho opětovného spuštění.

## Principiální popis skriptu

Skript operuje v několika krocích, jejich hranice je uvedena příslušným komentářem v kódu. Jedná se však jen o orientační hranici a faktické rozdělení by mohlo způsobit nefunkčnost skriptu. Nejprve je vytvořena skrytá kopie celého modelu. V něm je pak vymazán obsah kompilovaných subsystémů. Vstupy a výstupy těchto subsystémů jsou dále napojeny na bloky komunikace pomocí protokolu UDP. V případě, že v modelu existují přímá spojení mezi dvěma cílovými zařízeními, není toto spojení tvořeno přes hlavní model. Poté následuje vytvoření modelů, které budou kompilovány a spouštěny na cílových zařízeních. V těchto modelech jsou opět vstupní a výstupní porty nahrazeny boky pro komunikaci přes UDP. Dále jsou přidány komunikační bloky zařizující přímé spojení mezi jednotlivými cílovými zařízeními. Posléze jsou tyto modely sestaveny. Na konec dojde k otevření oken scopův hlavním modelu a je ho následnému spuštění.

## Implementace skriptu runFromOne

Pro orientaci jsou do kódu přidány komentáře, které ve zkratce popisují jakou úlohu plní příslušná část kódu.

### Sekce Top level Simulink model

Zde dojde k vytvoření nového skrytého modelu, jehož jméno je shodné se jménem hlavního modelu, na konec má však přidáno podtržítko. V případě existence takového modelu je tento model vymazán a použit. Dále je sem překopírován celý systém a nastavení původního modelu.

### Sekce Target to target check

V této sekci je implementován algoritmus pro hledání přímých spojení mezi dvěma cílovými zařízeními. Za jediný výstup se zde dá považovat pole `directs`, které tyto spojení ukládá. Pole má jednotlivá spojení ukládána po řádcích, každý o pěti hodnotách. První hodnota obsahuje handler na zařízení odesílající data, následuje handler na příjemce. Další dvě hodnoty jsou čísla výstupního a vstupního portu. Poslední

hodnota udává, zda je toto připojení pouze mezi cílovými zařízeními, nebo zda je použito i v rámci kořenového modelu.

### Sekce nahrazení cílových subsystémů

První série nahrazení IO portů za komunikaci probíhá v této části kódu. Zařízeno je tak rozhraní pro komunikaci pro hlavní model. Implementovány jsou zde i pojistky zajišťující ošetření případů kdy je komunikace mezi modelem a cílem nahrazena komunikací mezi dvěma cíli.

### Sekce target subsystems

Jedná se o větší sekci oddělující generování kořenového modelu a subsystémů pro jednotlivá zařízení. Postup v této sekci odpovídá předchozímu dění. Nejprve je vytvořen prázdný model nesoucí jméno subsystému, následně je do něj tento subsystém nakopírován, poté dojde k nahrazení IO portů za bloky komunikace pomocí UDP. Následně jsou realizována přímá spojení s ostatními cíli. Jelikož desky BeagleBone Blue blokují veškerou komunikaci na broadcast adrese, musí být každé spojení realizováno samostatně. To celou implementaci výrazně zkomplikovalo.

### Sekce Compile and run

V tuto chvíli je již náhradní schéma vygenerované a stačí jej pouze zkompilovat a nahrát do cílového zařízení. To odpovídá již dříve uvedené metodě distribuce kódu.

Nakonec dojde k otevření scopů v top modelu a jejich následnému spuštění. To je provedeno pomocí funkce `set_param`, což umožňuje oddělení běhu simulace od běhu skriptu, čímž jeho exekuce končí.

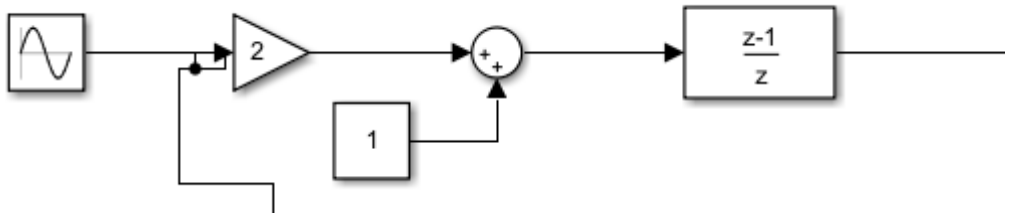
### Sekce stop and run

Zde obsažené dva skripty zařizují zastavení a spuštění všech částí systému. Nedochází zde ke kompilaci, čímž je jejich exekuce rychlá.

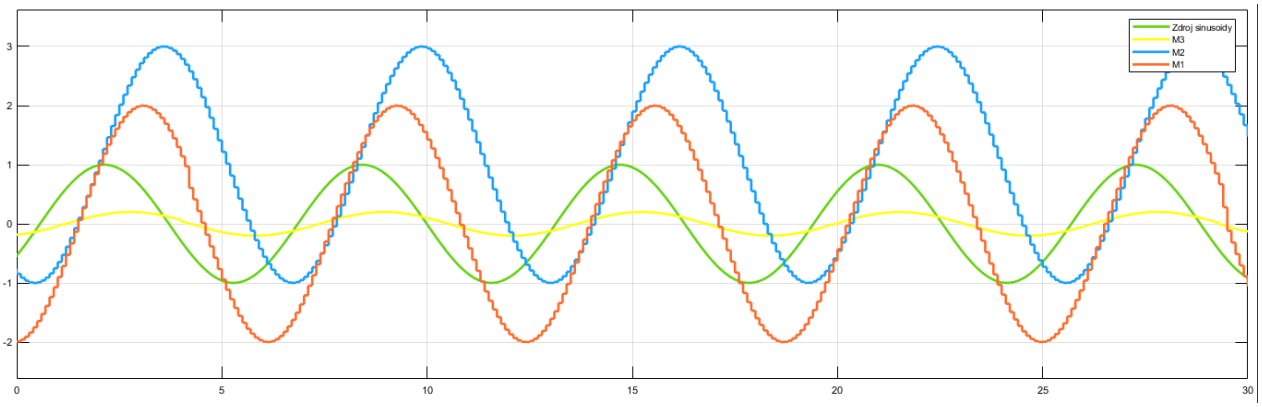
Pro komunikaci se Simulinkem jsou používány porty 25000 a více. Komunikace mezi jednotlivými cíli používá porty s hodnotou nad26000. V případě problému s komunikací je tedy dobrý nápad zkontrolovat jejich dostupnost.

## Experimentální ověření funkčnosti

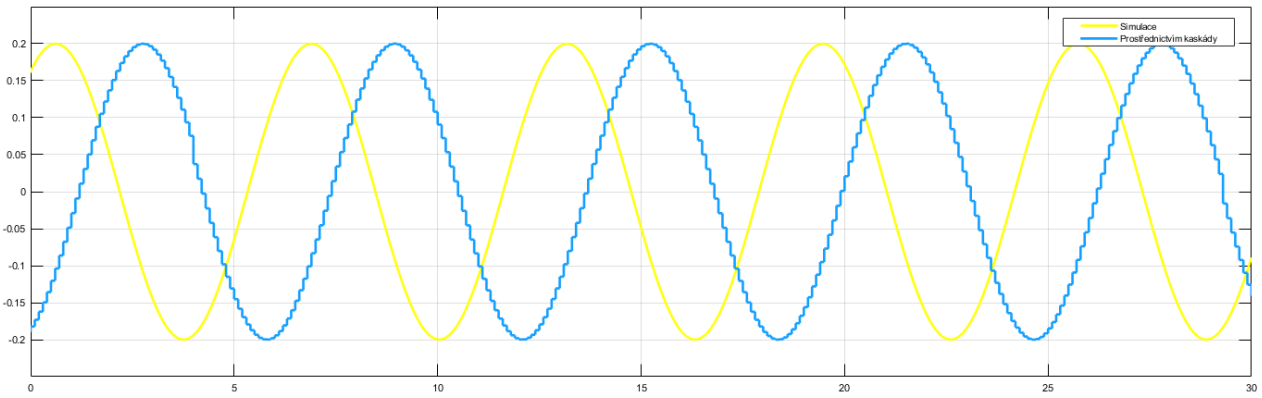
V této kapitole bude na jednoduchém schématu ukázána funkčnost tohoto řešení. Použito bylo jednoduché sériové zapojení cílových zařízení. Extrahuje-li se však pouze efektivní kód bude schéma vypadat následovně.




Obrázek 8: Efektivní schéma testovací aplikace




Obrázek 9: Ukázka postupné propagace signálu jednotlivými zařízeními




Obrázek 10: Porovnání očekávaného a naměřeného průchodu

Oproti očekávanému výsledku má signál po průchodu třemi zařízeními zpoždění asi 2 sekundy. Velkou roli na něj bude hrát zejména vzorkovací frekvence. Nicméně jak grafy ukazují, komunikace probíhá až na drobné zašumění v pořádku.

# Závěr

Tato postupně seznamuje čtenáře s procesem generování kódů ze Simulinku. Dochází k obeznámení se Simulink Coderem, který je pro generování použit. Dále jsou zde získány znalosti o generovaném kódu a ukázány možnosti jeho integrace v rámci větší aplikace. Na jednoduchém příkladu je ukázáno propojení s platformou BeagleBone Blue. Další část je pak již zaměřena na systematické generování kódu pro distribuované systémy. Jsou v ní nastíněny některé možné způsoby realizace. Poslední část se zabývá realizovaným skriptem, který realizuje nejvhodnější řešení pro distribuované systémy. Možnost mít celý systém zahrnut v jednom simulinkovém schématu umožňuje přehlednost řešení napříč jednotlivých uzlů distribuovaného systému. Za velikou výhodu lze také považovat možnost snadné interakce se simulací běžící na vývojovém počítači. Díky tomu lze například nahrazovat nedostupné části systému, popřípadě provést test celého systému na situace, které reálně nastávají jen zřídka.

I přesto, že je v současné době skript funkční a relativně spolehliví, dá se najít celá řada možných vylepšení. Vhodná by byla například úprava zajišťující přizpůsobení nastavení komunikace mezi jednotlivými částmi systému. Dále třeba přidání kontroly změn schématu, které by zajistilo, že jsou kompilovány jen části systému, které byly upraveny. Pro usnadnění užívání by mohlo být vhodné vytvořit konfigurační blok, který by se vkládal do systémů, které by měly být distribuovány.

Provedené testy ukázaly, že je tento skript spolehlivý a že jeho hlavním nedostatkem je čas kompilace, který se pro více zařízení může zdát příliš dlouhý.

## Použité pojmy

Systém – Celkový distribuovaný systém, pro který je generovaný kód vyvíjen

Aplikace–Spustitelný soubor, který běží na jednom uzlu distribuovaného systému

Coder – Simulink Coder

Cílové zařízení – Uzel distribuovaného systému, do kterého je nahrávána aplikace

Handler – Reference na konkrétní Simulinkový objekt

# Literatura


[1] Mathworks, Dokumentace Matlabu, https://www.mathworks.com/help/pdf_doc/matlab/index.html?s_cid=doc_ftr

[2] Mathworks , Dokumentace Simulinku, https://www.mathworks.com/help/pdf_doc/simulink/index.html?s_cid=doc_ftr

[3] Mathworks , Dokumentace Matlab Coderu, https://www.mathworks.com/help/pdf_doc/coder/index.html?s_cid=doc_ftr

[4] Mathworks , Dokumentace Simulink Coderu,https://www.mathworks.com/help/pdf_doc/rtw/index.html?s_cid=doc_ftr

[5] Mathworks , Dokumentace Embedded Coderu, https://www.mathworks.com/help/pdf_doc/ecoder/index.html?s_cid=doc_ftr

[6] Mathworks , Dokumentace Simulink Desktop Real-Time,https://www.mathworks.com/help/pdf_doc/sldrt/index.html?s_cid=doc_ftr

# Přílohy

## Matlabový kód pro generování kódu z jednoho simulinkového schématu

```
rootModel = 'All_In_One';
models = ['M1'; 'M2'; 'M3'];
devices = ['192.168.0.110'; '192.168.0.111'; '192.168.0.112'];


numberOfDevices = length(devices (:,1));
% Create top level Simulink model
try
    top = new_system (strcat(rootModel, "_"));
catch
%fprintf ("%s \n", strcat(rootModel, "_"));

    fprintf ("Exists, deleting content.\n");
    open_system (strcat(rootModel, "_"), 'loadonly');
    Simulink.BlockDiagram.deleteContents(strcat(rootModel, "_"));

end
%open_system (top);
subsys = add_block('built-in/Subsystem', strcat(rootModel,'_/top'));
root = load_system (rootModel);

Simulink.BlockDiagram.copyContentsToSubsystem(root, subsys);
Simulink.BlockDiagram.expandSubsystem(subsys);


target_handles = ones (1, numberOfDevices);

for i = 1:numberOfDevices
    target_handles(i) = getSimulinkBlockHandle(strcat(rootModel, '_/',
models (i, :)));
end


% Chck for direct Target to target connection

directs = [];

for i =1:numberOfDevices
    model = strcat (rootModel, '_/', models(i,:));
    my_handle = getSimulinkBlockHandle (model);
    m = find_system (model);
    pc = get_param (m{1}, 'PortConnectivity');

for ii = 1:numel(pc)
        onlyToTarget = 0;
        j = 1;
for dst = pc(ii).DstBlock
```

```
for th = target_handles
if dst == th
                    directs(end + 1, :) = [my_handle, dst,
str2double(pc(ii).Type),pc(ii).DstPort(j) + 1 , 0]; %#ok<*SAGROW>
                    onlyToTarget = onlyToTarget + 1;
end
end
            j = j + 1;
end
if onlyToTarget > 0
if onlyToTarget == length (pc(ii).DstBlock)
                directs (size (directs, 1) - onlyToTarget + 1:size
(directs, 1), 5) = 1;

end
end

end

end


%Replace subsystem content with comunication blocks
port = 25000;
for i =1:numberOfDevices

    model = strcat (rootModel, '_/', models(i,:));
    my_handle = getSimulinkBlockHandle (model);

    lines = find_system(model,'FindAll','on','type','line');
    ip = devices(i,:);

for ii = lines
        delete_line (ii)
end
    allblocks = find_system(model);
    in = find_system(model,'BlockType','Inport');
    out = find_system(model,'BlockType','Outport');



    toRemove = setdiff(allblocks,in);
    toRemove = setdiff(toRemove,out);

for ii = 2: numel(toRemove)
        delete_block (toRemove{ii})
end

for ii = 1: numel(in)
        isDirect = false;
for iii = 1: size(directs, 1)
if (directs (iii, 2) == my_handle) && (directs (iii, 4) == ii)
                isDirect = true;
end
end
```

```
if isDirect == false
            bh = add_block ('beaglebonebluelib/UDP Send', strcat(model,
'/Send_', string(ii)));
            set_param (bh, 'remotePort', string(port));
            set_param (bh, 'remoteUrl', strcat ("'", ip, "'"));
            tmp = extractAfter (in{ii}, model);
            tmp = extractAfter (tmp, '/');
            add_line (model, strcat(tmp, '/1'), strcat('Send_', string(ii),
'/1'));
            port = port + 1;
end
end


for ii = 1: numel(out)

        isOnly = 0;
for iii = 1: size(directs, 1)
if (directs (iii, 1) == my_handle) && (directs (iii, 3) == ii)
                isOnly = directs (iii, 5);
end
end

if isOnly == 0
            bh = add_block ('beaglebonebluelib/UDP Receive', strcat(model,
'/Receive', string(ii)));
            set_param (bh, 'localPort', string(port));
            set_param (bh, 'signalDatatype', 'double');
            set_param (bh, 'dims', '1');
            tmp = extractAfter (out{ii}, model);
            tmp = extractAfter (tmp, '/');
            add_line (model, strcat('Receive', string(ii), '/1' ),
strcat(tmp, '/1'));
            port = port + 1;
end
end


end

% Create target subsystems

port = 25000;
portDirect = 30000;

for i =1:numberOfDevices
    tic ();
    model = strcat (rootModel, '/', models(i,:));
    my_handle = getSimulinkBlockHandle (strcat (rootModel, '_/',
models(i,:)));
% Copy subsystem to new model

try
        sys = new_system (models(i,:));
catch
```

```
        sys = load_system (models(i,:));
        Simulink.BlockDiagram.deleteContents(sys);

end
%open_system (sys);
    Simulink.SubSystem.copyContentsToBlockDiagram (model, sys);

%Copy configuration of parent model
    rootConfig = getActiveConfigSet (rootModel);
    config = attachConfigSetCopy (sys, rootConfig, true);
    setActiveConfigSet ( sys, config.name);

%Replace I/O ports with UDP send/receive blocks


    in = replace_block (sys, 'Inport', 'beagleboneblue lib/UDP Receive',
'noprompt');
    out = replace_block (sys, 'Outport', 'beagleboneblue lib/UDP Send',
'noprompt');

for ii = 1:numel (in)

        isDirect = false;
        portOffset = 0;
for iii = 1: size(directs, 1)
if (directs (iii, 2) == my_handle) && (directs (iii, 4) == ii)
                isDirect = true;
if iii > portOffset
                    portOffset = iii;
end
end
end
if isDirect == true
            set_param (in{ii}, 'localPort', string(portDirect +
portOffset));
else
            set_param (in{ii}, 'localPort', string(port));
            port = port + 1;
end

        set_param (in{ii}, 'signalDatatype', 'double');
        set_param (in{ii}, 'dims', '1');

end

for ii = 1:numel (out)

        count = 0;
        ips = [];
        isDirect = false;
        isOnly = 0;
        portOffset = [];
for iii = 1: size(directs, 1)
if (directs (iii, 1) == my_handle) && (directs (iii, 3) == ii)
                isDirect = true;
                isOnly = directs(iii, 5);
```

```
if target_handles(iiii) == directs (iii, 2)
                        portOffset(end + 1) = iii;
end
end
                count = count + 1;
                name = get_param (directs (iii, 2), 'Name');

for iiii = 1: size(models, 1)
if (models (iiii, :) == name)
                        ips(end  + 1) = iiii;

end
end

end
end

if isDirect == true
            k = 0;
for j = 1:count
                pc = get_param (out{ii}, 'PortConnectivity');
                name = get_param (pc.SrcBlock, 'Name');
                bh = add_block ('beagleboneblue lib/UDP Send',
strcat(models(i,:), '/SendDirect_', string(ii), '_', string(j)));
                set_param (bh, 'remotePort', string(portDirect +
portOffset(j)));
                set_param (bh, 'remoteUrl', strcat ("'", devices(ips(j),
:), "'"));

                add_line (sys, strcat(name, '/1'), strcat('SendDirect_',
string(ii), '_', string(j), '/1'));
end
end

if isOnly == 0
            set_param (out{ii}, 'remoteUrl', "'255.255.255.255'");
            set_param (out{ii}, 'remotePort', string(port));

            port = port + 1;
else
           delete_block (out{ii});
end
end

% Compile and run created model
%open_system (sys);
    ip = devices(i,:);
    b = beagleboneblue (ip, 'debian', 'temppwd');
    slbuild (sys);
%runs = true;
    runs = isModelRunning(b, sys);
if runs
        fprintf("Running at %s\n", ip);
end
```

```
    toc ();
end



% Run
% Open all scopes

scopes = find_system (strcat (rootModel, '_'), 'BlockType', 'Scope');
for i = 1:numel (scopes)
    open_system (scopes{i});
end
% Run top-level model
set_param(strcat (rootModel, '_'), 'StopTime', 'inf')
set_param(strcat (rootModel, '_'),'SimulationCommand','start')


%% Stop all executables

set_param(strcat (rootModel, '_'),'SimulationCommand','stop')

for i =1:numberOfDevices
    ip = devices(i,:);
    b = beagleboneblue (ip, 'debian', 'temppwd');
    runs = isModelRunning(b, models(i,:));
if runs
        fprintf ("Stopping at %s\n", ip);
% psmisc must be installed at target to perform this kkoperation
        stopModel(b, models(i,:))
end
    toc()
end

%% Run no compile

set_param(strcat (rootModel, '_'),'SimulationCommand','start')

for i =1:numberOfDevices
    ip = devices(i,:);
    b = beagleboneblue (ip, 'debian', 'temppwd');
    runs = isModelRunning(b, models(i,:));
if runs
        fprintf ("Starting at %s\n", ip);
        runModel(b, models(i,:))
end
    toc()
end
```

# Dodatek

## Obbsah přiloženého cd

Přiložené CD obsahuje:

- Tex této zprávy ve formátu PDF
- Matlabové kódy použité v této práci
- Použité Simulinkové modely