CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Control Enegineering

Bachelor's Thesis

Open Rapid Control Prototyping and Real-Time Systems

Michal Lenc
michallenc@seznam.cz

May 2022
Supervisor: Ing. Pavel Píša, Ph.D.

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Lenc Michal** Personal ID number: **492387**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Control Engineering**

Study program: **Cybernetics and Robotics**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Open Rapid Control Prototyping and Real-Time Systems

Bachelor's thesis title in Czech:

Otevřený systém pro návrh řídicích aplikací reálného času

Guidelines:

The need to develop quickly and user friendly control systems is growing. Education and industry would gain from solutions which allows complete introspection not only of generated code but even of complete tool-chain. Open, extensible and freely accessible solution is win for education, industry and enthusiasts. Combination of NuttX RTOS and pysimCoder project has potential to take this role for constrained MCU based systems and thanks to NuttX POSIX compatibility share most of the tools and code with pysimCoder other targets as Linux and RTEMS.
1) Familiarize with pysimCoder project and NuttX operating system
2) Test and extend device support (ADC, DAC, PWM, encoders, etc.) and system and BSP level support (tick-less) of NuttX supported MCUs used by industry partners (imxRT and SAM70) to extend their use for control applications and number of supported pysimCoder blocks
3) Extend pysimCoder to allow runtime monitoring and tuning of of model parameters (all, or selectable subset, consider silicon-heaven protocol)
and connection of signals in distributed control system
4) Prepare demonstration of achieved results on selected platforms and models (PMSM motor control, ball on beam)
5) Document achieved results with focus on use of pysimCoder in education

Bibliography / sources:

1) Bucher, R.: Python for Control Purposes,
http://robertobucher.dti.supsi.ch/wp-content/uploads/2017/03/BookPythonForContro
l.pdf
2) pysimCoder, GitHub https://github.com/robertobucher/pysimCoder
3) NuttX, GitHub https://github.com/apache/incubator-nuttx
4) NuttX, Documentation https://nuttx.apache.org/docs/latest/
5) Lenc, M.: Google Summer of Code 2021, NuttX Support for Rapid Control Applications Development with pysimCoder https://summerofcode.withgoogle.com/projects/#4867567685992448

Name and workplace of bachelor's thesis supervisor:

Ing. Pavel Píša, Ph.D. Department of Control Engineering FEE

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **19.01.2022** Deadline for bachelor thesis submission: _____

Assignment valid until: **30.09.2023**

Ing. Pavel Píša, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. _____
Date of assignment receipt

_____
Student's signature

Acknowledgement / Declaration

I would like to thank my supervisor Pavel Píša for his mentorship on many projects and for igniting my insterest in computer science during his Computer Architecture bachelor course.

I would also like to thank Roberto Bucher for his mentorship during my project of extending NuttX support for pysimCoder and František Vacek and his work group at Elektroline company for giving me the opportunity to work on their NuttX related projects.

Last but not least my acknowledgements go to my family and friends for their support in my activities.

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague 20. 5. 2022

.....

Abstrakt / Abstract

Kombinace operačního systému reálného času NuttX a nástroje pysimCoder představuje otevřený systém pro návrh řídících aplikací. Cílí primárně na menší a levnější mikrokontroléry, například na rodiny imxRT, STM32 nebo SAM E70. Kombinace NuttXu a pysimCoderu může být využívána jak ve školním prostřední tak i v průmyslových aplikacích.

Schopnost NuttXu a pysimCoderu řídit aplikace v reálném čase byla otestována na systémech jako jsou RCRC systém druhého řádu nebo inverzní kyvadlo, nicméně spousta funkcionalit stále není naimplementována. To zahrnuje například podporu pro ladění parametrů modelu v reálném čase či podporu vektorových signálů.

Tato práce se zaměřuje na implementaci podpory pro ladění parametrů modelu v reálném čase s využitím průmyslem ověřené otevřené infrastruktury Silicon Heaven. Schopnost měnit parametry za běhu aplikace je demonstrována na aplikacích řídicích systém v reálném čase s cílem jednoduché reprodukce ve školním prostředí.

Combination of RTOS NuttX and software tool pysimCoder provides an open source environment for control application design targeting mostly smaller and cheaper microcontrollers like imxRT, STM32 or SAM E70 series. The combinaton can be widely used in educational and possibly even in industrial environment.

Capabilities of NuttX and pysimCoder to produce a real time control system have been tested and demostrated on systems like RCRC plant or inverted pendulum control, howewer the combination still misses some features like runtime monitoring and tunning of model parameters or vector signals.

The thesis documents design and development of an extension that allows to monitor and change selected model parameters at runtime. The open source industry proven Silicon Heaven communication infrastructure is used for the remote methods invocation and tree organized runtime system introspection. The ability is demonstrated on real time applications with focus on easy reproduction in educational environment.

Contents /

/ Figures

Chapter 1
Introduction

Many systems for control application design, both proprietary and open source, can be currently used for educational or professional purposes. While proprietary software like Matlab/Simulink usually offers more functions and provides better stability, open source alternatives, for example SciLab or ROS 2, have the advantage of being free and allowing programmers to explore how the code is written. Usage of control application design tools in education also requires the software to be compatible with affordable hardware. The possibility to design application that can be run on cheap microcontrollers like STM32, imxRT, ESP32 and many others would be a great support for education.

This possibility is partialy provided by software tool pysimCoder[1] and a real time operating system NuttX.[2] PysimCoder, designed by Professor Roberto Bucher from University of Applied Sciences and Arts of Southern Switzerland, is an open source control application development tool that includes a block diagram editor which allows the user to design a control application. A C code can be generated from this design, can be flashed in an embedded system and used for a real time control.[1, chapter 7][2]

PysimCoder currently supports two POSIX compatible open source operating systems: Linux[3] and NuttX. The latter is a real time open source operating system that offers a wide support of smaller and cheaper microcontrollers.[3] PysimCoder also supports various target chips and boards including STM32H7, Raspberry Pi or SAMD21. The support of NuttX in pysimCoder means the same block diagram designed by the user can be compiled and run on any embedded board supported by NuttX without the need for major changes in the application design. This offers the flexibility for both industry and educational organizations and allows them to change their hardware without investing into massive software changes.

While pysimCoder can already be used on many microcontrollers for simplier control applications, it still lacks many functionalities offered by its proprietary alternatives. One of the missing features, discussed and implemented in the frame of this thesis, is the ability of runtime monitoring and tuning of model parameters. Basic monitoring is currently supported via input and ouput blocks for UDP, TCP or communication over serial port, but this aproach is not ideal for more complex systems with several inputs/outputs. It also does not allow the real time changes of block's parameters which can complicate controllers tuning and calibration or requires a complex workaround. Current aproach requires the parameters to be changed in pysimCoder GUI, the application to be compiled and flashed to the board again. The implemented solution allows the programmer to tune model's parameters, constants of PID controller for example, directly at runtime without the need to reboot NuttX.

Silicon Heaven infrastructure[4] was chosed to provide network communication, support for runtime monitoring and tuning of model parameters and model introspection.

[1] `https://github.com/robertobucher/pysimCoder`
[2] `https://github.com/apache/incubator-nuttx`
[3] `https://github.com/torvalds/linux`
[4] `https://github.com/silicon-heaven`

The infrastructure implements a ChainPack, an open standard format for data serialization that aims to take the advantages from XML and JSON formats. Microcontrollers from imxRT series and SAM E70 series were chosen as target hardware platforms for project's results demonstration.

The first chapter of the thesis introduces the reader to NuttX operating system and discusses its source code and drivers organization. The theoretical part of the thesis then continues with the text describing pysimCoder application. The reasons behind selecting microcontrollers from imxRT series and SAM E70 as target hardware platforms are described in chapter 4 and are followed by the implementation of selected drivers to NuttX mainline. The sixth chapter introduces Silicon Heaven infrastructure and discusses its implementation to pysimCoder. This is the key part of the thesis. The thesis is completed by a short introduction of real time control examples.

Chapter 2
NuttX

This chapters describes NuttX real time operating system used in this thesis. The reader is introduced to basic information regarding NuttX source code organization, device driver's system and compilation steps.

2.1 Introduction

NuttX is an open source real time operating system (RTOS) first introduced by Gregory Nutt in 2007. It has come a long way since that and has been undergoing incubation at The Apache Software Foundation since 2019. NuttX is written in C language and offers a POSIX compatible environment.[4] This means the applications written in compilance with POSIX standards (on GNU/Linux for example) can be run on NuttX as well with minimal changes. Furthemore it implements ANSI standards and some further APIs from Unix systems or some other real time operating systems like VxWorks.

As described in the docoumentation, NuttX is scalable from small 8 bits to modern 64 bits microcontrollers.[5] This is allowed by linking from static libraries and using many source files that contain only small number of functions. The disadvantage of this aproach is less clear source code but it allows the build system to link only those functions desired by the user. According to a NuttX documentation, the final executable can than be run for example on only 32 kB total memory (code and data) although typical NuttX build usually requires about at least 64 kB memory.[5]

The desired functions like additional drivers (ADC, CAN, Ethernet), applications or systems features (tickless mode) can be selected using Kconfig system which is taken from Linux Kernel. The configuration system offers a great freedom in choosing which features should be included in NuttX, on the other hand it sometimes lacks proper documentation and thus sometimes it is not user friendly. Some of these configurations also does not work well together and can result in build error.

2.2 Source Code Organization

The NuttX directory structure takes a lot of inspiration from Linux kernel.[6] The most important folders from the perspective of this thesis are subdirectories `arch/`, `boards/` and `drivers/`. These subdirectories contain source code and header files for supported architectures, microcontrollers, boards and drivers. The following sections shortly describe each of those subdirectories. The goal is to provide basic understanding regarding NuttX source code and driver's organization. This knowledge is important for the implementation of driver's to NuttX mainline which is described in chapter 5.

The following sections discuss source code organization only from the device drivers' point of view. NuttX core itself is not taken into account here as this goes beyond the scope of the thesis.

2.2.1 Arch

The subdirectory `arch/` contains folders `include/` and `src/` for each supported microcontroller.[6] Folder `include/` includes basic microcontroller definitions like which peripherals are supported, external interrupts or peripheral identificators. The more important folder from the point of this thesis is `src/` which includes all source files for hardware specific implementation for supported drivers (also called `lower half` in NuttX documentation).[7]

This is where NuttX varies from Linux. While Linux kernel usually implements a single driver for all architectures and machines using the same IP core for given peripheral, NuttX strictly implements those controllers for each microcontroller. This difference can be shown with the following example.

Example: Let's take a FlexCAN controller for the purpose of this example. Linux has driver for this controller located in common file `drivers/net/can/flexcan.c` while NuttX implements `flexcan.c` files in both `arch/arm/src/imxrt/` and `arch/arm/src/s32k1xx/` MCUs subdirectories. Structures and functions from those files are connected to the `upper half` part of the driver which provides a high-level POSIX interface (write, read, etc.).

Every lower half part of the driver is usually divided into three separate files. The consensus in NuttX community is to keep the names in format `mcu_driver.c`, `mcu_driver.h` and `hardware/mcu_driver.h`.

Example: The names for ADC driver for imxRT MCU would be `imxrt_adc.c`, `imxrt_adc.h` and `hardware/imxrt_adc.h`.

The first `.c` file contains source code for the driver and takes care of setting up the peripheral and interface with driver's upper half described later in this chapter. The first header file usually defines just one function which can be accessed from board level. This function takes care of initial setting of the driver and usually also returns the driver instance so it can be registered by the upper half. The latter header file located in `hardware` subfolder contains definitions of peripheral's registers and bit fields.

2.2.2 Boards

The second subdirectory mentioned in this thesis is `boards/`. This includes all necessary source files and header files for board level support such as booting process or parts of the code that call functions from `arch/` section that initialize drivers. Board specific implementation is not discussed in the thesis. However implemented drivers were included in already exising board support packages, the initialization and compilation of a specific board is desribed later in the thesis.

2.2.3 Drivers

The last section of the three mentioned is `drivers/` which contains files for the `upper half` parts of the drivers as named per NuttX documentation. The `upper half` registers itself a device name (`dev/adc0`, `dev/mcan0`) via Virtual File System and implements the high level interface such as `read`, `write` or `open`. The interface between the `upper half` and `lower half` is mediated via callbacks to the `lower half` part of the driver.[7]

2.3 Supported Platforms and MCUs

This section contains a non exhaustive list of platforms and MCUs supported by NuttX. Only platforms and MCUs interesting from the point of view of this thesis are mentioned here, the complete list can be found in NuttX documentation or in NuttX source code.

The most extended support in NuttX is for ARM instruction set architecture (ISA) nowadays used in most of the microcontrollers for embedded systems.[8] This includes versions of STM32 chips, microcontrollers from imxRT series, LPC series or SAM series. Other architectures like Xtensa with support for microcontrollers designed by Espressif company, widely known ISA x86 or open standard architecture RISC-V are also supported but so far not as widely as ARM.

2.4 NuttX Compilation

The following steps are required to compile NuttX. The compilation steps are documented for Linux distribution Ubuntu (version 20.04 and newer) but this should be the same for other Debian based Linux distributions. The compilation requires the kconfig-frontend package to be installed on the system.[1] Architecture specific toolchain (e.g. arm-none-eabi-gcc for ARM targets) is also required. Compilation of NuttX on Windows or Mac OS is also possible and is described in NuttX documentation.

```
git clone https://github.com/apache/incubator-nuttx.git nuttx
git clone https://github.com/apache/incubator-nuttx-apps.git apps
cd nuttx
./tools/configure.sh board:config
make
```

Where `board` represents the name of the board (also the name of the directory where board's files are located) and `config` represents the name for required configuration. The configurations available for certain board can be found in NuttX documentation for that board or by looking directly to the source code.

Example: The basic configuration with serial port console for Teensy 4.1 board would be `./tools/configure.sh teensy-4.x:nsh-4.1`. The configuration file can be found in `boards/arm/imxrt/teensy-4.x/configs/` directory.

The described steps compile NuttX executable `nuttx.bin` or hexadecimal source file `nuttx.hex` based on the target hardware.

[1] `https://packages.ubuntu.com/focal/kconfig-frontends`

Chapter 3
PysimCoder

This chapter introduces a software tool pysimCoder and its block editor. The Linux and NuttX compilation steps of a block diagram designed in pysimCoder are described in following sections as well as pysimCoder code generation process. The understanding of the code generation process is very useful as some changes were made in this area during the Silicon Heaven infrastructure implementation. The source code organization is also mentioned here. While it is not important for this thesis, its knowledge can provide to be useful and helpful. It also becomes necessary for the implementation of future blocks into pysimCoder.

3.1 Introduction

PysimCoder is an open source control application development tool designed by Professor Roberto Bucher from University of Applied Sciences and Arts of Southern Switzerland. The application consists of an extended python-control library and a graphical block editor with a code generator.[1, chapter 7] As mentioned by Professor Bucher at NuttX Online Workshop 2021, the extension of python-control library allows integration of control design and simulaton methods as full and reduced state space observer, anti-windup mechanism and discrete linear quadratic regulator. These methods are highly useful for the implementation of real time controllers.[2] PysimCoder can also perform the simulation of designed block diagram.

The pysimCoder functionality for real time control was tested and demonstrated by Professor Roberto Bucher on systems like second order RCRC plant or inverted pendulum.[9][10] The system is also used at control theory courses at University of Applied Sciences and Arts of Southern Switzerland.[2]

3.2 Block Editor

PysimCoder's block diagram editor GUI is in many ways similar to popular Simulink editor. The editor consists of two separate windows, diagram window and library. Library window includes all available blocks separated to several libraries (Math, NuttX and input for example). These blocks can be moved to diagram window using drag and drop method.[1, chapter 7] Both windows are shown in Figure 3.1.

Most of the blocks have specific parameter options and can support various number of inputs/outputs. The parameter settings is opened by double left click on the block placed in a diagram window. This allows the user to set parameters like controller's gain constants, driver's device name and so on. The selection of number of inputs/outputs can be accessed via single right click on the block. The option for multiple inputs/outputs might not be supported for all blocks (NuttX encoder block offers only one output for example).[11]

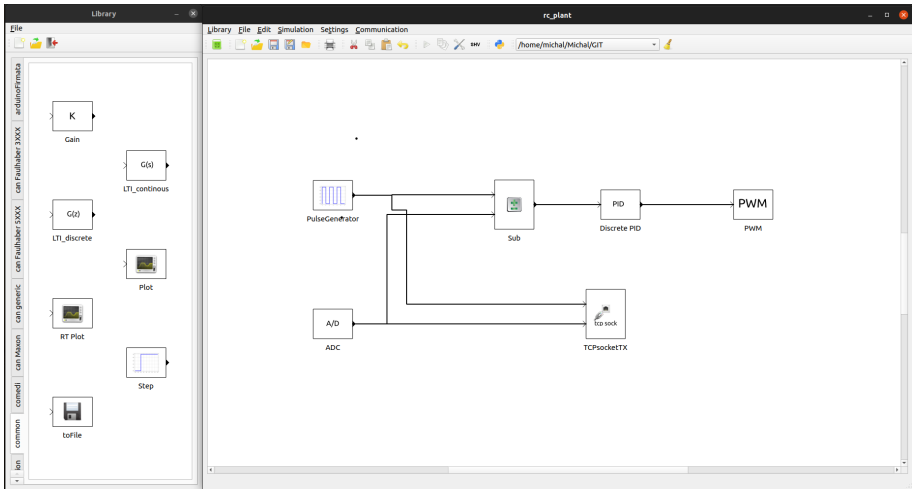

Figure 3.1. PysimCoder's library (left) and diagram window (right) (Source: [11]).

The extensive manual with the steps required to create an own block diagram in pysimCoder can be found in chapter 7 of Professor Bucher's book Python for Control Purposes.[1]

3.3 Source Code Organization

This section provides basic introduction of pysimCoder's source code structure and organization. The goal is to give a basic understanding that would allow the reader to design and create own custom block without major obstacles. Basic organization of folders taking care of code generation is also shortly mentioned but is fully introduced in section 3.4. This focus mainly on code organization, code generation and block's lifecycle is described in section 3.4. It is recommended to remind this section once again after reading the section 3.4 to get the complex picture.

PysimCoder has two blocks related folders in its structure, `resources` and `CodeGen`. The first mentioned folder provides blocks' declaration in JSON format and Python part of the code while the latter contains Makefile templates and C parts of blocks' code for supported targets. Both subdirectories are described in the following sections.

3.3.1 resources

The first declaration of the block is provided in `resources/blocks/blocks/library` folder where library stands for actual library the block belongs to (i.e Math, NuttX, etc.). This folder contains `.xblk` files in JSON format that provides declaration of block's inputs, outputs, parameters and so on. The list of required keys follows.

- lib – the name of the library the block belongs to
- name – the name of the block
- ip – number of inputs
- op – number of outputs
- stin – 1 if number of inputs can be set by user, 0 otherwise
- stout – 1 if number of outputs can be set by user, 0 otherwise
- icon – name of the icon located in `resources/icons` folder
- params – name of block related Python function followed by the list of parameters
- help – user help

Example: Lets take a look into an example of block definiton in JSON format. File `nuttx_PWM.xblk`[1] can be found in `resources/blocks/blocks/NuttX` folder with the following definition.

```
{
 "lib": "NuttX",
 "name": "PWM",
 "ip": 1,
 "op": 0,
 "stin": 1,
 "stout": 0,
 "icon": "PWM",
 "params": "nuttx_PWMBlk|Port:'/dev/pwm2'|channels: [1]
                        |PWM freq [Hz]:1000|Umin [V]:0.0
                        |Umax [V]:100.0",
 "help": "Help text"
}
```

The JSON file described above provides the link to Python function (`nuttx_PWMBlk` in our example). These functions are located in `resources/blocks/rcpBlk/library` where the library naming remains the same as in `blocks/` directory. The following example continues with PWM block for NuttX.

Example: The name of the file located in `resources/blocks/rcpBlk/NuttX` is `nuttx_PWMBlk.py`. The simplified code can be seen below.

```
import numpy as np
from supsisim.RCPblk import RCPblk
from scipy import size

def nuttx_PWMBlk(pin, port, ch, freq, umin, umax):

  blk = RCPblk('nuttx_PWM',pin,[],[0,0],1,[freq,umin,umax],ch,port)
  return blk
```

The order of parameters passed to nuttx_PWMBlk is exactly the same as the order of parameters in `.xblk` files. The order of parameters passed to RCPblk function can be seen in `toolbox/supsisim/src/RCPblk.py`.[2] It is necessary to pass the parameters in the correct order as the placement in RCPblk functon defines the type of the parameter (integer, double or string). It is also worth mentioning the first parameter of the function is the name of the corresponding C function and the fifth parameter is the input to output relation.

3.3.2 CodeGen

While `resources` directory contains files that defines the block, `CodeGen` located files takes care of the actual execution of block's functions. Once again, the example is used to present the structure.

Example: We take a PWM block for NuttX RTOS once again. The corresponding file

[1] https://github.com/robertobucher/pysimCoder/blob/master/resources/blocks/blocks/NuttX/nuttx_PWM.xblk

[2] https://github.com/robertobucher/pysimCoder/blob/master/toolbox/supsisim/src/RCPblk.py

`nuttx_PWM.c`[3] is located in `CodeGen/nuttx/devices/` folder, the largery simplified C code follows.

```
static void init(python_block *block){}
static void inout(python_block *block){}
static void end(python_block *block){}

void nuttx_PWM(int flag, python_block *block)
{
  if (flag == CG_OUT){          /* get input */
    inout(block);
  }
  else if (flag == CG_END){     /* termination */
    end(block);
  }
  else if (flag == CG_INIT){   /* initialisation */
    init(block);
  }
}
```

The enter function `nuttx_PWM` receives an integer flag and a pointer to the block structure. The corresponding function is called based on the receved flag. Function `init` is called only once and as the function name would suggest it initalizes the driver. Flag CG_OUT calls function `inout` that performs the required operation. In our example it updates the PWM duty cycle based on the block input, but it could also read data from ADC or generate some kind of an output (square, triangle, etc.). This function is called repeatedly. Function `end` then takes care of stopping the output and closing the driver. The definition of those functions is not listed here as it is block specific.

These sections desribed the organization of the pysimCoder blocks. The core of the code generation can be found in `toolbox` directory. The principle of the process is described in the following section.

3.4 Code Generation

This section provides the description of code generation process from designed block diagram. It is divided into two subsections, general description regarding the process and more practical point of view on source code organization.

3.4.1 General Description

Every block in pysimCoder can be described by a set of equations representing its internal states and outputs.

$$x_{k+1} = f(x_k, u_k, k) \tag{1}$$

$$y_k = g(x_k, u_k, k) \tag{2}$$

Two equations listed above are well known as state space equations, however they are executed in an opposite order. System firstly updates block's output, described by equation (2) and then internal state per equation (1). Blocks would need to remember

[3] `https://github.com/robertobucher/pysimCoder/blob/master/CodeGen/nuttx/devices/nuttx_PWM.c`

theirs previous x_k state if equation (1) would be executed first, this is not necessary if the order is opposite. The first function is also not mandatory as some blocks do not have an internal state. Apart from those two functions, each block also has an initialization and a termination function. These functions are called throught corresponding flag: CG_OUT, CG_STUPD, CG_INIT or CG_END.[2]

The code generation process takes a designed block diagram and an optional Python script (used for implementation of the controller and variables definitions) and generates a Python script `tmp.py`. This script translates the block diagram into C code from which specific blocks functions are called. The diagram of this process can be seen in Figure 3.2.

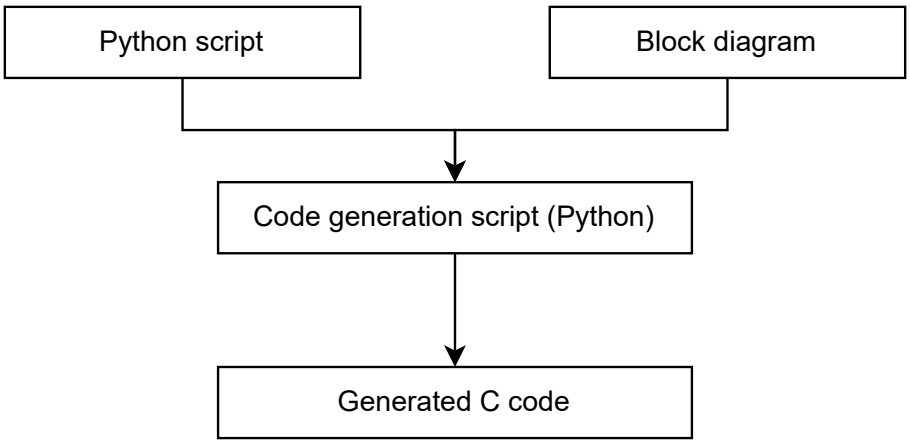

Figure 3.2. PysimCoder Code Generation Process (Inspired by: [2]).

The generated C code is subsequently compiled and linked with the block library and main C file with the real time thread and creates an executable.[2] The code generation process also needs to find the right execution sequence so the blocks are executed in the correct order (as blocks' inputs depending on other blocks' outputs can not be executed first).

3.4.2 Source Code Organization

As mentioned at the end of section 3.3, main source files for Code Generation are located in `toolbox` directory. This directory contains two libraries, `supsictrl` and `supsisim`. The first mentioned implements control methods as state space observer and others mentioned in the introduction, the latter one brings the files related to code generation.

The function `generateCCode` in `scene.py` creates the `tmp.py` file mentioned in the previous sections. The sample of the code from `tmp.py` can be seen below.

```
Const = constBlk([2],  3.2)
Gain = matmultBlk([2],[3],  1)
Print = printBlk([4,5,3,1])

blks = [Const,Gain,Print,]

fname = 'test'
os.chdir("./test_gen")
genCode(fname, 0.01, blks)
genMake(fname, 'rt.tmf', addObj = '')
```

```
import os
os.system("make")
os.chdir("..")
```

The first three lines of the code calls the block specific function that was introduced to the readed in the previous section. This returns the block structure which is subsequently passed to `genCode` function located in `RCPgen.py` in `supsisim` library. This function takes care of C code generation, the third step shown in Figure 3.2. Then corresponding Makefile is generated and required executable is created. Files `scene.py` and `RCPgen.py` are the ones that were changed during Silicon Heaven implementation. These changes are fully described in chapter 6.

3.5 NuttX Integration

This section provides the necessary steps to succesfully run pysimCoder on NuttX. The compilation of NuttX remains the same as described in section 2.4 but requires some additional configuration options. These options are described in NuttX documentation.[11] Once NuttX is compiled, following commands need to be run.

```
make export
cp nuttx-export-xx.x.x.zip /../pysimCoder/CodeGen/nuttx
cd /../pysimCoder/CodeGen/nuttx
unzip nuttx-export-xx.x.x.zip
mv nuttx-export-xx.x.x.zip nuttx-export
cd devices
make
```

Where xx.x.x in nuttx-export-xx.x.x.zip stands for the current version of NuttX. Execution of `make` command in `devices` folder compiles the C files described in previous section. Then pysimCoder can be run either via included script `pysim-run.sh` or via application installed on Linux. The instalation process is described in pysimCoder documentation, the steps required to succesfully create pysimCoder application for NuttX are described in NuttX documentation.[11] The important setting is to select a Template Makefile for the target. This can be done in the top menu by clicking on Block settings icon which is highlighted in the red circle as described in NuttX documentation.[11] The required Template Makefile for NuttX is `nuttx.tmf`.

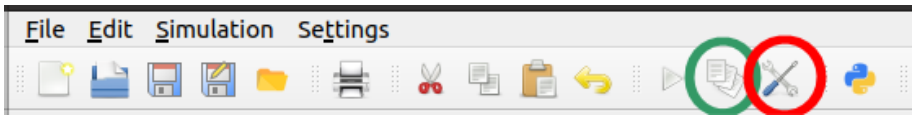

Figure 3.3. PysimCoder Menu Option (Source: [11]).

The loadable executable can be generated by selecting Generate C-code icon highlighted in the green cirle. The executable is a standard NuttX with a terminal, file system and selected applications plus an application called `main`. This is the designed block diagram. The control application can be run from NuttX terminal simply by the following command.

```
nsh> main
```

The application `main` can also be selected as an init function instead of `nsh_main`, providing terminal support, in NuttX configuration. Another option is to modify initialization scripts inside NuttX system to run generated application automatically.

3.5.1 Supported Blocks

Apart from common blocks, usually from mathematic, input or output libary, pysimCoder implements NuttX specific blocks. These blocks provides an interface to device driver peripherals such as ADC, PWM or DAC. This chapter provides the list of supported peripherals at the time of writing this thesis.

- ADC – Analog to Digital Converter
- CAN – Controlled Area Network, with FD support
- DAC – Digital to Analog Converter
- ENC – Encoder
- GPIO – support for both input and output pins
- PWM – Pulse Width Modulation with multichannel support
- serialOut – support for serial output

Some blocks from communication library as TCP and UDP respect POSIX standard and they can be used with NuttX as well.

Chapter 4
Target Hardware Selection

Two microcontrollers, imxRT1060 and SAM E70, were selected as a target hardware for the testing purposes of this thesis. This chapter discusses the reasons behind this selection and introduces these MCUs to the reader. Two microcontrollers, imxRT1060 and SAM E70, both used in the thesis's practical part, are discussed here.

4.1 Introduction

As mentioned in section 2.3, NuttX provides extended support for microcontrollers with ARM instruction set architecture so the logical step is to use an ARM based MCU. The alternative, interesting from educational point of view, would be the selection of open architecture RISC-V based MCU, for example Espressif ESP32-C3. While the open source status of the instruction set architecture is interesting and promising for the future, the development of NuttX support for those chips is still ongoing and the MCU designed by Espressif also does not provide many pin outputs. This resulted in selection of ARM based microcontrollers.

Most of the ARM based microcontrollers provide support of peripherals such as ADC, PWM, Encoder or Ethernet that are needed for the fulfilment of the thesis's goals. This means the decision of particular microcontrollers being based mostly on the faculty's and industry partners' requirements.

The microcontroller from imxRT series, imxRT1060, was already used during my previous projects at the Department of Control Enegineering at the faculty and was tested with pysimCoder on a real time control system. The open Base Board for Teensy 4.1, board designed by Czech company PiKRON[1] and using imxRT1060 MCU, provides connections to required peripherals. That resulted in selecting it as the first target hardware. Using the already tested and supported hardware brings the advantage of not having to worry about NuttX support for peripherals as it was already implemented during my previous projects (PWM driver, FlexCAN driver) and focus solely on pysimCoder part. However the thesis's assignment reguired to extend device driver support for target hardware and contribute to NuttX mainline.

The second target hardware was selected in cooperation with Czech company Elektroline a.s.[2] The company's goal is to build its new smaller IoT devices and systems above NuttX RTOS. Their requirements include peripherals like ADC, Ethernet, Encoder, SPI or CAN. SAM E70 MCU from Microchip Technology was found out to be the best option, also regarding to current market situation during chip shortage, and thus selected for the new boards designed by Elektroline. This MCU was also used to contribute drivers extension during this thesis.

[1] `http://www.pikron.com/`
[2] `https://www.elektroline.cz/`

4.2 imxRT1060

imxRT1060 family of microcontrollers is designed and manufactured by Dutch company NXP Semiconductors. The chips are based on ARM Cortex-M7 core running at frequency 600 MHz and offer data and instruction cache memory of 32 KB size as well as 512 kB of data and instruction tightly coupled memory. The supported peripherals are CAN FD, ADC, PWM, Ethernet, dedicated Encoder driver, QSPI and SPI among others. The MCU also supports crossbar switches that can route inputs to outputs based on user's requirements.[12]

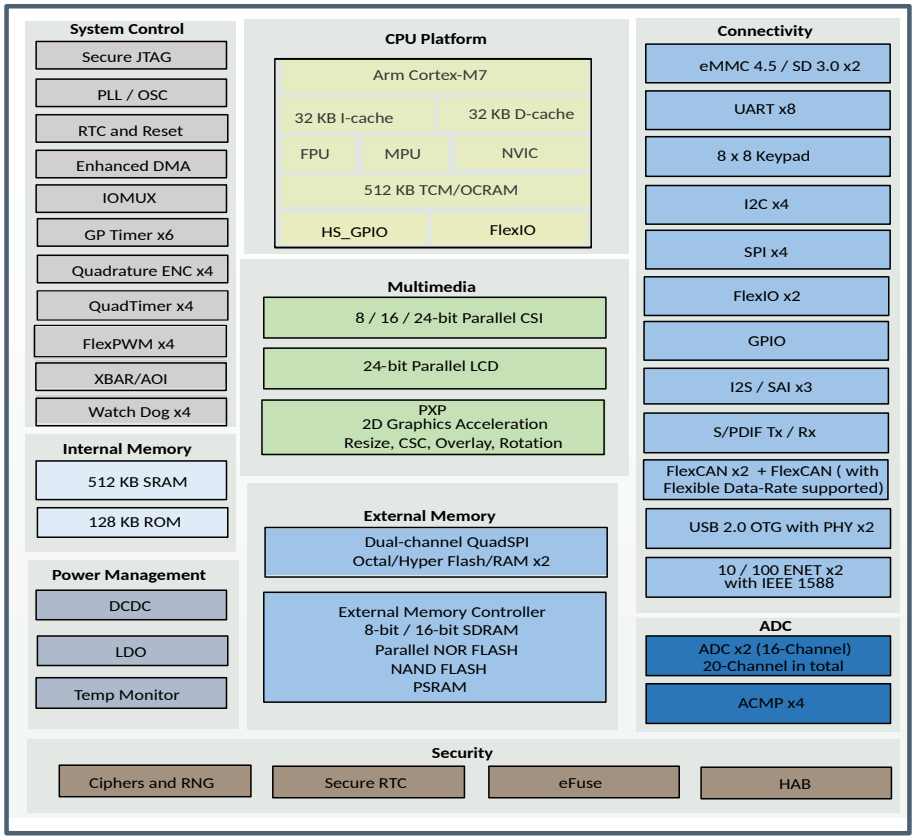

Figure 4.1. imxRT1060 MCU block diagram (Source: [12]).

Teensy 4.1 Development Board manufactured by PJRC campeny can be used as a target hardware.[3] This board provides all peripheral pinouts, such as ADC, PWM or Ethernet, needed for the thesis's goals. The board however does not provide the necessary interfaces for the drivers. This problem can be solved by Base Board for Teensy 4.1 designed by Czech company PiKRON that provides the necessary interfaces including PMSM or DC motor control peripherals.[4] Morover, the design of the board is open and fully fits into the thesis's theme.

[3] https://www.pjrc.com/store/teensy41.html
[4] https://gitlab.com/pikron/projects/imxrt-devel/-/wikis/teensy_bb

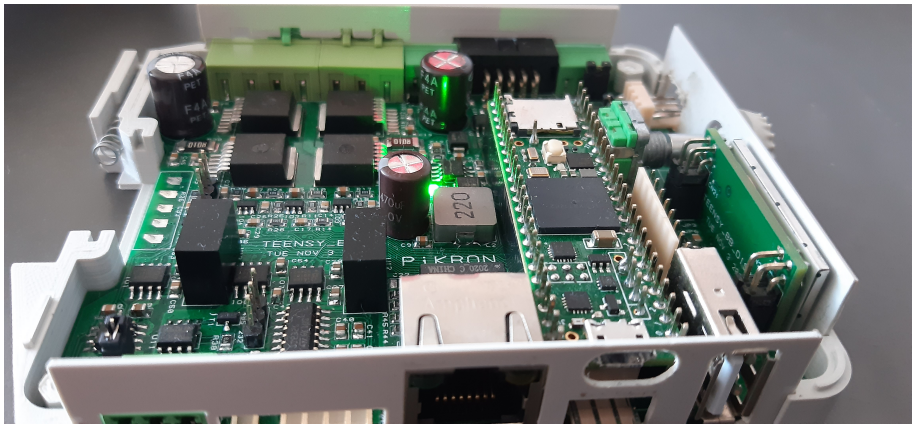

Figure 4.2. imxRT Teensy-4.1 Base Board by PiKRON.

All necessary peripherals for the purpose of this thesis were already implemeneted by other users or during my previous projects (FlexCAN or PWM). Base Board for Teensy 4.1 was also tested and its capability was demonstrated with a simple DC motor position real time control.[13] Having runtime monitoring and tunning of model parameters as primary goal and leaving NuttX as secondary, this brings the advantage of having an already supported and tested peripherals as a backup if some problems occur with another target hardware.

4.3 SAM E70

Microcontrollers from SAM E70 32 bit series are designed and manufactured by American company Microchip Technology. They use ARM Cortex-M7 core running at 300 MHz and can have up to 2048 KB of Flash memory based on selected version of the microcontroller.[14] The MCUs from this series have data and instruction cache memory of 16 KB size. The peripherals offered on SAM E70 are CAN FD, ADC, PWM, Ethernet MAC, UART, QSPI, SPI and USB host and device among others.

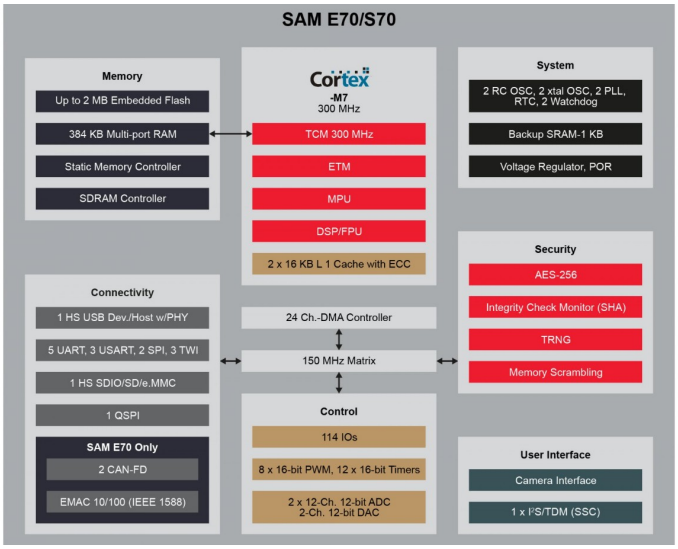

Figure 4.3. SAM E70 MCU block diagram (Source: [14]).

The number of supported peripherals varies by the used version of the chip. SAM E70 microcontrollers are manufactured in 64 to 144 pin package options. The latter

one offers full usage of supported peripherals while options with less pins do not offer some funtions.[14] The package version interferes with the peripheral's programming very rarely and is not taken into account in further sections.

An evaluation kit SAM E70 Xplained from Microchip was used as a target board for the peripherals tests and examples. The documentation and schematics for this board can be found at company's website.[5]

NuttX did not offer support for some key peripherals at the time of the thesis assignment. The most important for real time control were ADC, also required by Elektroline for their projects, and PWM. The support of QSPI in SPI mode was also absent and required by Elektroline but this part is not mentioned in the following chapter since it does not have a direct connection to real time control.

[5] `https://www.microchip.com/en-us/development-tool/ATSAME70-XPLD`

Chapter 5
Drivers Implementation

This chapter is focused on the first practical part of the thesis, the implementation of selected periherals' drivers. All following drivers are implemented for SAM E70 MCU which was discussed in the previous chapter, however the implementation process would be similar and sometimes even the exactly same for other targets. This chapter does not try to provide a step by step manual or whole process of source code creation but rather to introduce the most important steps and parts of driver creation for NuttX. Some key code parts for the periherals' drivers are listed here but a look into the source code is recommended for the complete understandment of the driver's functionality.

There were two major peripherals necessary for the ability of target's real time control: Analog to Digital Converted or shortly ADC and Puls Width Modulation, PWM. The implementation of these drivers is discussed in the following sections.

5.1 Analog to Digital Converter

Analog to Digital Converter (ADC) is a system that performs conversion of an analog signal (e.g. electric voltage or current) to a digital signal. This digital signal is represented by a binary number of a finite number of bits[15, section 9.1, pg. 612] (up to 16 bits in case of SAM E70). Digital Design and Computer Architecture by Harris & Harris can be recommended for further reading about the peripheral.[16]

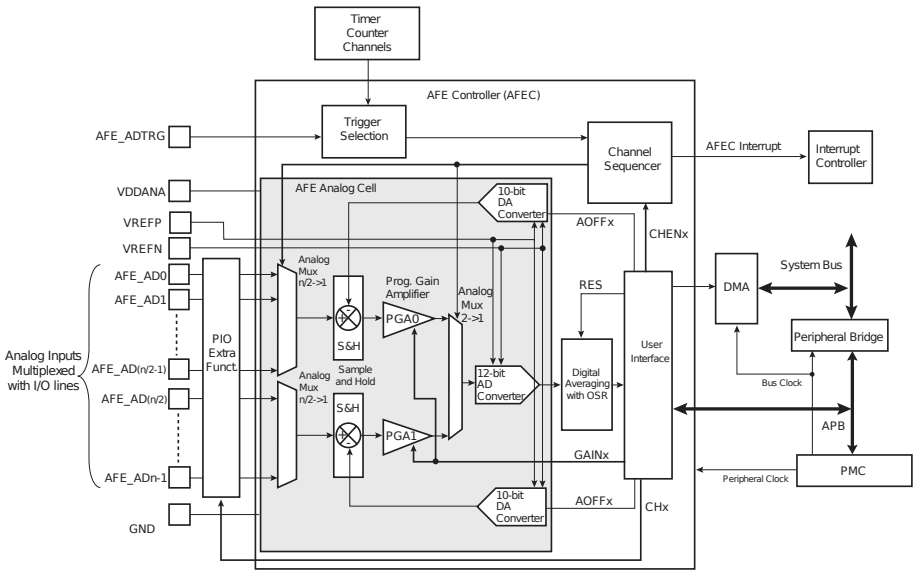

Figure 5.1. Analog Front End Controller Block Diagram (Source: [17, figure 52-1].)

Chips from SAM E70 series implement ADC under peripheral called Analog Front End Controller (AFEC) in chip's datasheet.[17, section 52] Apart from ADC, it integrates a programmable gain amplifier for ADC inputs, two analog multiplexers and

digital to analog converter. This allows the MCU to perform analog to digital conversion either of 12 lines or simuntaniously of two 6 lines. SAM E70 has 12 bit ADC resolution by default but this can be extended to 16 bit by digital averaging.[17, section 52]

The following sections use both ADC and AFEC naming. AFEC is more common when reffering to whole microcontroller's controller while ADC refers to the device driver.

AFEC can be triggered either by software trigger or by external hardware trigger (e.g. PWM output, timer/counter) as can be seen in Figure 4.2. ADC sampling at higher frequencies (for example 10 kHz) needed for proper real time control usually requires Direct Memory Access transfers from AFEC peripheral to chip memory. Direct Memory Access, abbreviated as DMA, provides a peripheral such as AFEC with a direct access to main memory. As a result, the transfer of received data to the memory is not executed by the processor but by the DMA. This allows the processor to execute other tasks during that time and thus increases the system performance.[18]

This resulted in selecting timer/counter as ADC trigger and implementing DMA support for the AFEC driver. The implementation of the driver is described in the following section.

5.1.1 Driver Implementation

As discussed in chapter 2, source code for device drivers is located in `arch/` subdirectory. For SAM E70 MCU family, this would be `arch/arm/src/samv7` subfolder. The lower half part of the driver is divided into three separate files in NuttX, `sam_afec.c`, `sam_afec.h` and `hardware/sam_afec.h`. This once again comes from the NuttX's community consensus as mentioned in chapter 2.

The lower half part of ADC driver communicates with the upper half located in `drivers/` via adc_dev_s structure. This structure requires the lower half of the driver to provide two fields, ad_ops and ad_priv. The first mentioned links architecture specific operations to driver operations as setup or reset while the latter can provide architecture specific logic as resolution or trigger selection. The following sample of the code shows definition of these fields in SAM E70 AFEC driver.

```
static const struct adc_ops_s g_adcops =
{
  .ao_bind     = afec_bind,
  .ao_reset    = afec_reset,
  .ao_setup    = afec_setup,
  .ao_shutdown = afec_shutdown,
  .ao_rxint    = afec_rxint,
  .ao_ioctl    = afec_ioctl,
};

#ifdef CONFIG_SAMV7_AFEC0
static struct samv7_dev_s g_adcpriv0 =
{
  .irq         = SAM_IRQ_AFEC0,
  .pid         = SAM_PID_AFEC0,
  .intf        = 0,
  .initialized = 0,
  .resolution  = CONFIG_SAMV7_AFEC0_RES,
```

```
#ifdef CONFIG_SAMV7_AFEC0_SWTRIG
  .trigger     = 0,
#else
  .trigger       = 1,
  .timer_channel = CONFIG_SAMV7_AFEC0_TIOACHAN,
  .frequency     = CONFIG_SAMV7_AFEC0_TIOAFREQ,
#endif
  .base          = SAM_AFEC0_BASE,
};

static struct adc_dev_s g_adcdev0 =
{
  .ad_ops      = &g_adcops,
  .ad_priv     = &g_adcpriv0,
};
```

The initialization of the driver is done by the public function `sam_afec_initialize` called from board level logic which returns the corresponding adc_dev_s structure. This structure is then registered as a device driver. Functions linked throught adc_ops_s structure are then used to setup registers, interrupts, perform IOCTL operations and other stuffs necessary for driver functionality. These functions are called from the upper hals logic of the driver. The first method is `afec_setup` which is called when the driver is opened for the first time. This function sets up interrupts, trigger and enables and starts DMA if enabled.

The DMA is implemented with so called ping-pong buffers. This means two buffers are used, DMA collects data to the first one while the data from the second one are processed in another thread. Then the buffers switch and worker thread reads from the first one while DMA collects to the second one. This ensures that data are collected by DMA even when CPU needs to process the previously received data.

DMA calls callback function `sam_afec_dmacallback` each times it fills the buffer with the required amount of data. This function sets up the worker thread in which `sam_afec_dmadone` performs data read. The process of data read can be seen below.

```
for (i = 0; i < priv->nsamples; i++, buffer++)
  {
    /* Get the sample and the channel number */

    chan   = (int)((*buffer & AFEC_LCDR_CHANB_MASK) >>
                   AFEC_LCDR_CHANB_SHIFT);
    sample = ((*buffer & AFEC_LCDR_LDATA_MASK) >>
              AFEC_LCDR_LDATA_SHIFT);

    if (priv->cb != NULL)
      {
        /* Give the sample data to the ADC upper half */

        priv->cb->au_receive(dev, chan, sample);
      }
  }
```

The `au_receive` function saves the read data to local FIFO buffer in the upper half section of the driver. User can get the data from this buffer from application level using the POSIX call `read()`. There are two IOCTL calls implemented during my previous projects that helps with the buffer's operations. ANIOC_RESET_FIFO clears the FIFO buffer and ensures all read data are new, ANIOC_SAMPLES_ON_READ returns the number of samples in the buffer.

The implementation without DMA is mostly similar but uses corresponding interrupt that indicates there are data in Last Converted Data Register (AFEC_LCDR). The code shown above would then read directly from this register instead of the DMA buffer.

The ADC can be set up by selecting following configuration options.

```
CONFIG_ANALOG=y
CONFIG_ADC=y
CONFIG_SAMV7_AFEC0=y
CONFIG_SAMV7_TC0=y
CONFIG_SAMV7_TC0_TIOA0=y
```

This configures basic ADC with timer/counter trigger sampling at 1 kHz. The sampling frequency can be change by configuring `CONFIG_SAMV7_AFEC0_TIOAFREQ` and the trigger can be change to software trigger called from application by selecting `CONFIG_SAMV7_AFEC0_SWTRIG`. The channels are not selected in the configuration but it has to be hard coded in board level section. Using DMA requires following setup,

```
CONFIG_SAMV7_XDMAC=y
CONFIG_SAMV7_AFEC_DMA=y
CONFIG_SAMV7_AFEC_DMASAMPLES=10
```

which configures DMA to wait for 10 samples for each channel and then transfer it to memory. The whole source code of the driver is included in NuttX mainline.[1]

5.1.2 Application Usage

The AFEC or ADC peripheral can be accessed from application via standard POSIX calls. The following code shows a simple application that opens the driver and reads the sampled data from it. The code is simplified and for example does not check if error occurs while opening the driver or while performing read operation. The proper application should have these checks.

```
#include <nuttx/config.h>
#include <nuttx/analog/adc.h>
#include <nuttx/analog/ioctl.h>

struct adc_msg_s sample[conf_ch];
int readsize = conf_ch*sizeof(struct adc_msg_s);

int fd = open(block->str, O_RDONLY);

int nbytes = read(fd, sample, readsize);
```

The conf_ch variable provides the number of configured channels. IOCTL calls can be used to get some further information from the driver, for example ANIOC_SAMPLES_ON_READ described in the previous section. The usage of the IOCTL call is also simple and can be seen below.

[1] https://github.com/apache/incubator-nuttx/blob/master/arch/arm/src/samv7/sam_afec.c

```
int ret = ioctl(fd, ANIOC_SAMPLES_ON_READ, 0);
```

5.2 Pulse Width Modulation

Microcontroller also needs some way to control the system. Pulse Width Modulation (PWM) peripheral is used for this purpose. This peripheral generaters a periodic output that is pulsed high for part of the period and low for the remainder. The part of the period for which the pulse is high is called the duty cycle.[16, chapter 9.3.7, page 542.e35] The example of PWM output can be seen in the Figure 5.2.

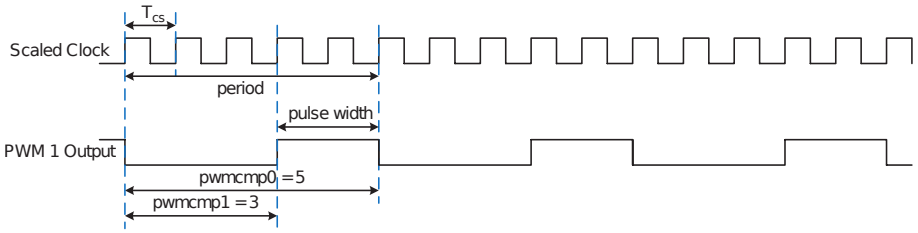

Figure 5.2. Sample of Pulse Width Modulate Signal (Source: [16, figure e9.17]).

PWM can provide an analog output of designed value (voltage) and thus be used for real time syste control. PWM can be also used for other purposes like trigger generator for ADC and other peripherals but these options were not implemented during the thesis. Digital Design and Computer Architecture by Harris & Harris can be recommended for further reading regarding PWM peripheral again.[16]

SAM E70 MCU implements two PWM peripherals, each with 4 indepenent channels that can each control two complementary outputs. While the impelented peripheral provides the option of DMA trasnfers of duty cycle updates,[17, section 21] this option was not implemented to NuttX driver as it was not necessary for the funcionality demonstration. However this might be a place for further work and extension of NuttX support for SAM E70.

5.2.1 Driver Implementation

The logic behind files organization is the same as discussed in the previous section so files `sam_pwm.c`, `sam_pwm.h` and `hardware/sam_pwm.h` were created. The usage of upper and lower half structures is also largely similar to AFEC driver. The communication between lower and upper half of the driver is done with pwm_lowerhalf_s structure. This structure requires the first field to be a pointer to the PWM callback structure pwm_ops_s, the following fields can be architecture specific.

The following sample of the code shows declaration of required structures in `sam_pwm.c` PWM driver.

```
static const struct pwm_ops_s g_pwmops =
{
  .setup      = pwm_setup,
  .shutdown   = pwm_shutdown,
  .start      = pwm_start,
  .stop       = pwm_stop,
  .ioctl      = pwm_ioctl,
};
```

```
#ifdef CONFIG_SAMV7_PWM0

static struct sam_pwm_channel_s g_pwm0_channels[] =
{
#ifdef CONFIG_SAMV7_PWM0_CH0
  {
    .channel = 0,
    .used    = true,
    .pin     = GPIO_PWMC0_H0,
  },
#endif
#ifdef CONFIG_SAMV7_PWM0_CH1
  {
    .channel = 1,
    .used    = true,
    .pin     = GPIO_PWMC0_H1,
  },
#endif
...
};

static struct sam_pwm_s g_pwm0 =
{
  .ops = &g_pwmops,
  .channels = g_pwm0_channels,
  .channels_num = 4,
  .frequency = 0,
  .base = SAM_PWM0_BASE,
};
#endif /* CONFIG_SAMV7_PWM0 */
```

From the code sample can be seen that sam_pwm_s structure is used as a pwm_lowerhalf_s and contains a pointer to PWM operations and a several architecture specific fields. The first initialization and registration of the device driver is done throught public function sam_pwminitialize called from board level section. Similarly to AFEC peripheral, this function returns pwm_lowerhalf_s and takes care of basic initialization like enabling peripheral's clock.

The first function called from the upper half of the driver, when the driver is first opened, is pwm_setup. The functions provides configuration of peripheral registers necessary for pulse generation, however it should not enable the output and generate pulses. The pulse generation itself is done by function pwm_start. The PWM driver has separate 16 bits wide registers for PWM period (SAMV7_PWM_CPRD) and duty cycle (SAMV7_PWM_CDTYUPD) for each channel. However, the option of independent frequency for each channel is currently not supported by NuttX logic and only duty cycle can differ for each PWM channel.

The startup function pwm_setup first need to check whether the set frequency matches with the required one and then can change the duty cycle if needed. Update of frequency by writing to corresponding register is required if freqeuncy does not match. The process of duty cycle update can be see in the code sample below.

```
period = pwm_getreg(priv, SAMV7_PWM_CPRDX + (shift * CHANNEL_OFFSET));
```

```
/* Compute PWM width (count value to set PWM low) */

duty_pct = (duty / 65536.0) * 100;
width = (uint16_t)(((uint16_t)duty_pct * period) / 100);

/* Update duty cycle */

pwm_putreg(priv, SAMV7_PWM_CDTYUPDX + (shift * CHANNEL_OFFSET), width);

/* Enable output */

regval = CHID_SEL(1 << shift);
pwm_putreg(priv, SAMV7_PWM_ENA, regval);
```

The shift variable provides the correct channel offset based on the channel number provided by upper half part of the driver. The code sample above runs in for loop for every configured channel and updates its duty cycle by writing to the corresponding register. Duty cycle update is propagated to control circuits after each period which may result in some channels being updated while others not. This can be solved by setting synchronous update of all channels by writing to the corresponding register as listed below.

```
/* Set sychnronous outputs */

pwm_putreg(priv, SAMV7_PWM_SCUC, SCUC_UPDULOCK);
```

The PWM support can be set up by selecting following configuration option:

```
CONFIG_PWM=y
CONFIG_PWM_MULTICHAN=y
CONFIG_PWM_NCHANNELS=3
```

This is the common driver part configuration that sets PWM and enables multiple channels support if required. The SAM E70 specific configuration can be seen below.

```
CONFIG_SAMV7_PWM0=y
CONFIG_SAMV7_PWM0_CH0=y
CONFIG_SAMV7_PWM0_CH1=y
CONFIG_SAMV7_PWM0_CH2=y
```

It is worth mentioning the difference between PWM and ADC channel selection. While ADC reqeuires the user to select channels in board level section and thus edit NuttX source code, PWM provides an option to set channels via configuration interface.

The source code of the described driver was accepted in NuttX mainline.[2]

5.2.2 Application Usage

The access from application level is also done via POSIX calls similarly to AFEC peripheral. The following code shows a simplified example and once again should not be seen as a complete application but rather as an introduction to the most important parts of possible applications.

[2] https://github.com/apache/incubator-nuttx/blob/master/arch/arm/src/samv7/sam_pwm.c

```
#include <nuttx/config.h>
#include <nuttx/timers/pwm.h>

struct pwm_info_s info;

/* Open the device */

int fd = open(driver_path, O_RDONLY);

info.frequency = frequency;

for (int i = 0;i < n_channels; i++)
  {
    info.channels[i].channel = channel_n;
    info.channels[i].duty = channel_duty;
  }

/* Set frequency and duty cycle and start the output */

int ret = ioctl(fd, PWMIOC_SETCHARACTERISTICS,
               (unsigned long)((uintptr_t) &info));


ret = ioctl(fd, PWMIOC_START, 0);
```

The IOCTL call PWMIOC_SETCHARACTERISTICS can be also used to update the duty cycle during the code running. The PWM logic in NuttX RTOS provides several helps to operate with PWM from the application level. One of them are two defined channels numbers, 0 and -1. The first one means the channel is not used and thus is skipped when calling PWMIOC_SETCHARACTERISTICS or PWMIOC_START and the latter indicates no following channels are used. This breaks the for loop in which channels are set in pwm_start function and thus can save some computation time.

Chapter 6
Runtime Monitoring and Tuning of Model Parameters

The previous chapters introduced two systems used in this thesis, NuttX and pysimCoder, and described drivers implementation to NuttX. This chapter is focused on the main goal of the thesis, the runtime monitoring and tuning of model parameters. Both theoretical overview and implementation of Silicon Heaven infrastructure are discussed in the following sections.

6.1 Introduction

The idea behind runtime monitoring and tuning of model parameters is to allow the user to display and edit individual blocks's parameters, inputs and outputs. Implementation of dedicated blocks for system inputs and outputs would subsequently allow the connection of distributed systems. This basically divides the task into two subtasks: the implementation of runtime monitoring and tuning of model parameters/inputs/outputs and implementation of input and output dedicated blocks.

As already mentioned in chapter 1, Silicon Heaven communication infrastructure[1] was selecteded in order to support runtime monitoring and tuning of model parameters. The infrastructure implements ChainPack, an open remote procedure call protocol (RPC) for data serialization that combines the properties of Extensible Markup Language (XML) and JavaScript Object Notation (JSON). The introduction of Silicon Heaven follows up later in the chapter.

The practical part of the chapter, the implementation of the infrastructure itself, takes a lot of knowledge from the theoretical part of the thesis, especially of chapter 3 describing pysimCoder's code generation process.

6.2 Silicon Heaven Infrastructure

Silicon Heaven infrastructure (SHV) was developed at company Elektroline by Ing. František Vacek and his team. It is used in company's systems on technology control tramway yards in Australia, Belgium and other countries around the Europe.[19] The infrastructure provides core support for many programming languages including C, C++, Python or Rust.

The infrastructure requires running a broker as a server. User applications as pysimCoder control application or GUI designed to interact with the broker are then registered to the broker as clients. Each client can have different rights and settings based on a broker's configuration.

[1] `https://github.com/silicon-heaven`

Apart from its main library, libshv implementing ChainPack RCP, the infrastructure provides GUI tool shvspy[2] and library shvapp[3] with implemented applications including shvbroker. Both svhspy and shvbroker are used in the implementation, their usage is described later.

The following section describes the protocol itself, the ChainPack RPC.

6.2.1 ChainPack RPC

Every ChainPack RPC message consists of three fields: length, protocol and data, respectively. The length field stores an unsigned integer of message length without the length field itself, so the length of protocol field plus the length of data field. Protocol is once again an unsigned integer defining data format (ChainPack RPC, Cpon or JSON). Then data itself follows.[20]

```
+--------+----------+------+
| Length | Protocol | Data |
+--------+----------+------+
```

Data uses the following format. PackingSchema is uint8_t defining the type of data (integer, double, string, map, Imap, MetaMap). The examples of packed data format can be found on Silicon Heaven wiki.[4] The work in this thesis mostly requires the usage of Map, IMap and MetaMap based on the unified message format. Map is basically a dictionary with string keys, IMap has integer keys and MetaMap can support both strings and integers.

Every RPC message data part consinsts of `<meta-data-part>` and `i{data-part}` where i represents the usage of IMap. The example of server request and client reply taken from SHV wiki follows.[21]

```
<1:1,8:17,10:"hello">i{}
```

Where the data in cone brackets represents `<meta-data-part>`. Apart from `1:1`, declaring ChainPack RPC is used, message metadata also contains request ID (`8:17`, where 8 is a tag for ID and 17 is ID itself) and requested method (10 is the tag, method's name is a string). Data part is empty in this case but the IMap still has to be included. The client reply would be.

```
<1:1,8:17>i{2:{"nonce":"1429255113"}}
```

Client replies with the same ID in metadata and then sends the data itself. The number 2 indicates request result follows, in this case it is a Map of some data with a string key. The possible tag keys can be found on Silicon Heaven wiki page.[5]

6.2.2 ChainPack RPC Usage

Silicon Heaven infrastructure brings the support for ChainPack RPC in many programming languages. The core functions are located in libshv library in libshvchainpack subdirectory. The files from this directory are the only ones necessary to support the SHV communication. These files defines functions like `cchainpack_pack_int` or `cchainpack_pack_imap_begin` that are used to pack ChainPack RPC message. This message can then be send or received, for example over TCP.

[2] https://github.com/silicon-heaven/shvspy
[3] https://github.com/silicon-heaven/shvapp
[4] https://github.com/silicon-heaven/libshv/wiki/ChainPack-RPC#chainpack
[5] https://github.com/silicon-heaven/libshv/wiki/ChainPack-RPC#rpc

6.3 Changes to pysimCoder Code Generation Process

Several, mostly minor, changes were required in pysimCoder code generation process in order to get blocks and parameters names to C part of the code and to have a structure that would contain all blocks used in the diagram and additional information about them. Two structures `python_block_name_entry` and `python_block_name_map` were added to `pyblock.h` file. This file also defines the main block structure `python_block` that can access block's parameters, inputs, outputs, dimensions and so on.

```
typedef struct {
  const char * block_name;  /* Name of the block */
  int block_idx;            /* Index in python_block structure */
  int system_inputs;        /* Block has system inputs */
  int system_outputs;       /* Block has system outputs */
} python_block_name_entry;

typedef struct {
  const python_block_name_entry * blocks;   /* Pointer to
                                             * python_block_name_entry
                                             * structure */
  const python_block * block_structure;     /* Pointer to python_block
                                             * structure */
  int blocks_count;                         /* Number of blocks */
} python_block_name_map;
```

The content of `python_block_name_map` is defined during code generation and the structure is then passed as an argument to SHV tree initialization function. Integers `system_inputs` and `system_outputs` defines whether block's inputs/outputs are used as system's inputs/outputs. This is used for the implementation of dedicated input/output blocks.

While getting the name of the block to C code was straightforward, parameters' names were more difficult. Parameter's name is accessable in function `generateCCode` in `scene.py` (refer to section 3.4 for the reminder of pysimCoder code generation process) but the type of the parameter is uknown at that moment. On the contrary, the type is known in `genCode` function from `RCPgen.py` but the names are not available here. Therefore it was necessary to know the parameter's type already in early code generation process in `scene.py`.

I chosed to add parameters' types to xblk files so they could be accessed already in scene.py. Taking the PWM block example from section 3.3.1, the updated key params is:

```
"params": "nuttx_PWMBlk|Port:'/dev/pwm2'|channels: [1]:int
                       |PWM freq [Hz]:1000:double|Umin [V]:0.0:double
                       |Umax [V]:100.0:double"
```

Function `generateCCode` then process these parameters and their names and ensures they are saved to `python_block` structure so they can be accessed from C code. The function also ensures that the generation process does not crash when the type is not defined.

The other option to get the parameter's type is to generate some sort of metadata from RCPblk function that creates the block. This function would generate information regarding what parameters are used, which can be changed runtime and which can be

just read only. This would require the RCPblk functions to be called from the editor during block diagram creation. This aproach would be more benefitial to pysimCoder from long term view, however the implementation would be more difficult and time consuming. Therefore I decided to go with the first option. The metadata approach can be left for some further work on the project.

6.4 SHV Tree Structure

PysimCoder's blocks and their parameters, inputs and outputs are represented by items in a dedicated tree. The tree's format is following.

- project name
  - blocks
    - block 1
      - inputs
        - input1
        - input2
        - ...
      - outputs
        - output1
        - output2
        - ...
      - parameters
        - parameter1
        - parameter2
        - ...
    - block 2
      - ...
    - ...

6.4.1 Supported Methods

Every item needs to support at least two methods according to SHV standard. Those are methods `ls` and `dir`. The first method returns the list of strings with the names of node's children. The list is empty if node does not have children. Method `dir` returns the list of methods supported by the node. In any case those are two already mentioned methods plus item specific methods.

Every parameter's node also supports methods `set`, `get` and `typeName`. As the names already suggest, first two methods send and receive the parameter value. Furthemore `typeName` returns the type of the value. Currently it is only double as integer values are not accessed by SHV but it can be enhanced in the future.

Integer values are often used for channel's numbers (for ADC or PWM for example) or other non-changable parameters like encoder resolution. They are also sometimes used not as a single parameter but rather as an array of integer parameters. Correct representation of integer parameters would most likely require metadata generation as discussed at the end of section 6.3. Therefore only double parameters are currently implemented in SHV tree.

Block's inputs and outputs are read only and thus only support method `get` as well as system dedicated outputs. System inputs support both `get` and `set`.

6.4.2 Nodes' Representation

Two data structures were selected for representation of tree's items, AVL tree (called GAVL in the library) and sorted array (called GSA). Open source uLAN Utilities Library (ULUT)[6] was used for the implementation of mentioned structures to pysimCoder's code. This library by company PiKRON provides implementation of structures and functions commonly used in C, among others AVL tree and sorted array.[22]

Usage of GAVL is prefered when the tree is allocated dynamically during application start since GSA uses more memory reallocation for addition of new items to the array. However GSA can be allocated statically during code generation and all items can be constant. This allows the whole SHV tree to be saved to flash memory and do not waste space in RAM. The implementation of SHV in pysimCoder supports all three options: GAVL, GSA and static GSA.

6.5 SHV Tree Implementation

Previous section discussed the theoretical part of item's representation in a SHV tree. This part introduces the structures and parts of the code taking care of tree initialization. The tree is consisted of `shv_node_t` items. The definition of the `shv_node_t` structure can be seen below.

```
typedef struct shv_node {
  const char *name;          /* Node name */
  gavl_node_t gavl_node;     /* GAVL instance */
  shv_dir_map_t *dir;        /* Pointer to supported methods */
  shv_node_list_t children; /* List of node children */
} shv_node_t;
```

GAVL instance is necessary when using AVL tree. It represents a node in a tree and links left, right and parent node to it. Structure `shv_dir_map_t` is a GSA array of pointers to methods supported by the node. The final field is a structure `shv_node_list_t` used as a list of children. The definition of the mentioned structure follows.

```
typedef struct shv_node_list {
  int mode;                 /* Mode selection (GAV, GSA, static) */
  union {
    struct {
      gavl_cust_root_field_t root;  /* GAVL root */
      int count;                    /* Number of root's chuldren */
    } gavl;
    struct {
      gsa_array_field_t root;       /* GSA root */
    } gsa;
  } list;
} shv_node_list_t;
```

This structure keeps the mode of the node (information what type of tree is used) and GAVL or GSA related root structure. Note that both of those structures are not pysimCoder related and can be used for any item in a tree. Accessing parameters/inputs/outputs in SHV tree reqeuires one more structure. The structure is called `shv_node_typed_val_t`.

[6] `https://gitlab.com/pikron/sw-base/ulut`

```
typedef struct shv_node_typed_val {
  shv_node_t shv_node;          /* Node instance */
  void *val_ptr;                /* Value */
  char *type_name;              /* Type of the value (int, double...) */
} shv_node_typed_val_t;
```

Node instance `shv_node_t` is passed to SHV tree while `val_ptr` and `type_name` variables store a pointer to parameter's value and type name, respectively. This additional structure is only required for parameters, inputs and outputs, other tree's items like blocks use only basic `shv_node_t`.

Apart from `shv_com.c` file implementing communication funtions and described in the following section, the code is implemented in three files: `shv_tree.c`, `shv_methods.c` and `shv_pysim.c`. All files are located in `CodeGen/Common/shv` directory, header files can be found in `include` subdirectory. The code structure is designed so that general tree functions (memory allocation, search, addition of new node and so on) are located in `shv_tree.c`. This file is independend on higher level files `shv_methods.c` and `shv_pysim.c`.

While general functions in `shv_tree.c` and `shv_com.c` could be used for other applications and not only for pysimCoder, the other two files implement pysimCoder related functions. Supported methods are defined in `shv_methods.c` while `shv_pysim.c` provides an entry point and dynamic tree generation if selected.

```
shv_con_ctx_t *shv_tree_init(python_block_name_map * block_map,
                             const shv_node_t *static_root, int mode)
{
  int ret;
  const shv_node_t *root;

  if ((mode & SHV_NLIST_MODE_STATIC) == 0)
    {
      /* Create tree root only if it should be allocated dynamically */

      root = shv_tree_node_new(NULL, &shv_root_dir_map, mode);
      if (root == NULL)
        {
          printf("ERROR: malloc() failed\n");
          return;
        }
      shv_tree_create(block_map, (shv_node_t *)root, mode);
    }
  else
    {
      root = static_root;
    }

  /* Initialize SHV connection */

  shv_con_ctx_t *ctx = shv_com_init((shv_node_t *)root);

  return ctx;
}
```

Function `shv_tree_init`, called from generated C file, is used as an entry point for SHV related operation. This function calls `shv_tree_create()` to create a dynamic GAVL or GSA tree if required and follows up with calling `shv_com_init()` from `shv_com.c`. The `shv_tree_create()` function goes throught all blocks and their parameters/inputs/outputs if supported and adds them to the tree. This process is skipped if SHV tree is generated statically. All tree's nodes are defined in generated C file in that case.

6.6 SHV Communication

Functions securing SHV communication are implemented in `shv_com.c` file. The entry point is a function `shv_init()` that allocates the memory for the SHV container taking care of data sends and receives, initializes TCP connection and performs client login to the server. The login process is standardized and can be seen in SHV documentation.[21]

Data read function is run in a separate thread so SHV operations are executed independently from main control functions. The context of the function can be seen below.

```
static void *shv_process(void * p)
{
  int num_events;
  shv_con_ctx_t *shv_ctx = (shv_con_ctx_t *)p;

  struct pollfd pfds[1];
  pfds[0].fd = shv_ctx->stream_fd;
  pfds[0].events = POLLIN;

  while (1)
    {
      /* Set timemout to one half of shv_ctx->timeout (in ms) */

      num_events = poll(pfds, 1, (shv_ctx->timeout * 1000) / 2);

      if (num_events == 0)
        {
          /* Poll timeout, send ping */

          shv_send_ping(shv_ctx);
        }
      else if (pfds[0].revents & POLLIN)
        {
          /* Event happened on our socket, process TCP input */

          shv_process_input(shv_ctx);
        }
    }
}
```

It is necessary to check for client's timeout set during login process. If server does receive any activity from client for time longer than specified timeout than it aborts

the connection. Function `poll()` takes care of determining whether it is required to ping the server. If `poll()` ends up with a `POLLIN` event it indicates we can read data from socket.

```
int shv_process_input(shv_con_ctx_t * shv_ctx)
{
  int i;
  int j;
  char met[SHV_MET_MAX_LEN];
  char path[SHV_PATH_MAX_LEN];
  struct ccpcp_unpack_context *ctx = &shv_ctx->unpack_ctx;

  i = read(shv_ctx->stream_fd, shv_ctx->shv_rd_data,
          sizeof(shv_ctx->shv_rd_data));
  if (i > 0)
    {
      ccpcp_unpack_context_init(ctx, shv_ctx->shv_rd_data, i,
                                shv_underrflow_handler, 0);

      while (ctx->current < ctx->end)
        {
          /* Get method and path */

          shv_unpack_head(shv_ctx, &j, met, path);

          if (met[0] != '\0')
            {
              shv_node_process(shv_ctx, j, met, path);
            }
        }
    }
  return i;
}
```

Function `shv_process_input()` takes care of TCP read in a blocking mode. SHV function `ccpcp_unpack_context_init()` initialize the `ccpcp_unpack_context` structure and fills it with received data. Function `shv_underrflow_handler()` takes care of yet to be received data. This is used when received message is longer than defined SHV message length (1024 bytes in this case) or not all data are read by initial read because they are not delivered over TCP yet. Then function `shv_unpack_head()` is called to unpack the metadata header part of the message and gets the information about requested method and node location in the tree (i.e. path). Subsequently the node is found in the tree and the method is called.

The file also implements functions `shv_send_double()`, `shv_send_string()` and similar that are used to send a client reply to the server request. This is used when server asks for the block's parameter. The parameter is found in SHV tree, as described in previous two sections, and double value is sent in reply formated as shown in section 6.2.1. The following code shows the definition of function `shv_send_double()`. The creation of ChainPack content is done twice in a loop in order to get the correct lenght of the message. This is something we do not know until we fill the content. Function `shv_overflow_handler` then performs TCP write if `shv_ctx->shv_send` equals 1.

```
void shv_send_double(shv_con_ctx_t *shv_ctx, int rid, double num)
{
  ccpcp_pack_context_init(&shv_ctx->pack_ctx,shv_ctx->shv_data,
                          SHV_BUF_LEN,shv_overflow_handler);

  for (shv_ctx->shv_send = 0; shv_ctx->shv_send < 2; shv_ctx->shv_send++)
    {
      if (shv_ctx->shv_send)
        {
          cchainpack_pack_uint_data(&shv_ctx->pack_ctx,
                                    shv_ctx->shv_len);
        }

      shv_ctx->shv_len = 0;
      cchainpack_pack_uint_data(&shv_ctx->pack_ctx, 1);

      shv_pack_head(shv_ctx, rid);

      cchainpack_pack_imap_begin(&shv_ctx->pack_ctx);
      cchainpack_pack_int(&shv_ctx->pack_ctx, 2);
      cchainpack_pack_double(&shv_ctx->pack_ctx, num);
      cchainpack_pack_container_end(&shv_ctx->pack_ctx);
      shv_overflow_handler(&shv_ctx->pack_ctx, 0);
    }
}
```

Only TCP communication is currently supported but with further time investment SHV could also support communication over serial port or CAN bus. This enhancement can be implemented in some future projects.

6.7 Input/Output Blocks in SHV Tree

The changes to pysimCoder also introduce two SHV dedicated blocks: SHV Input and SHV Output. These simple blocks without parameters and with option of multiple inputs/outputs can be newly found in communication library in pysimCoder GUI. Considering SHV tree, the previously introduced tree structure is expanded by nodes inputs and outputs as follows.

- project name
  - blocks
  - inputs
    - input block 1
      - input 1
      - input 2
      - ...
  - outputs
    - output block 1
      - output 1
      - output 2
      - ...

Opposite to the parameters, individual input and output signals do not use their own names as they do not have any. They use general naming system `input0`, `input1` and so on. Inputs support both methods `get` and `set` while outputs only support `get`. The support of inputs and outputs is only for those two dedicated blocks and can not be used with other blocks from communication library. The addition of other blocks does not make much sence since those blocks are used to generate some signal (Pulse Generator, Step and so on) or receive/send data over TCP and UDP. Dedicated SHV blocks do not intend to replace them but they can be used to connect multiple systems.

Source code for those blocks can be once again found in `CodeGen/Common/shv` directory under names `shv_blk_output.c` and `shv_blk_input.c`. The code structure is as described in section 3.3.2, the block has an entry point function and functions `init()`, `inout()` and `end()` called based on a received flag. Those functions are empty and do not perform any operations since the only purpose of the block is to have selected input or output. Their value is changed throught SHV method `set` and read with `get`.

6.8 SHV Settings in pysimCoder

This section presents compilation and configuration steps to succesfully use Silicon Heaven infrastructure with pysimCoder. SHV support in pysimCoder is compatible with POSIX compliant systems GNU/Linux and NuttX. The following command is required to succesfully compile pysimCoder's source code for NuttX target with Silicon Heaven. Apart from that the compilation steps are identical to those listed in sections 2.4 and 3.3.2.

```
make SHV=1
```

The only change from standard compilation is an additional Makefile parameter `SHV=1`. This parameter ensures the download of required libraries SHV and uLUT. It is also necessary to select support for TCP communication in NuttX configuration before the export is generated. The compilation for GNU/Linux is similar and also requires additional parameter `SHV=1`.

Silicon Heaven options like server's IP and port, user name and password, device name or type of the tree can be selected in pysimCoder's menu under SHV support icon.

6.9 SHV Usage

Apart from ChainPack RPC, SHV infrastructure also offers additional applications and tools as shvbroker and shvspy mentioned earlier in this chapter. This section shows their usage with a pysimCoder application. Package Qt 5.13[7] at minimum is required to succesfully compile shvbroker and shvapp. The latter also requires `libqt5webkit5-dev` package.[8]

Shvbroker, included in shvapp repository, acts as a TCP server for a client (pysimCoder application, some GUI and so on). Following commands show its compilation and usage.

[7] `https://doc.qt.io/archives/qt-5.13/index.html`
[8] `https://packages.ubuntu.com/jammy/libqt5webkit5-dev`

```
git clone https://github.com/silicon-heaven/shvapp.git
cd shvapp
git submodule update --init --recursive
qmake -r
make
cd bin
./shvbroker --config-dir ../shvbroker/etc/shv/shvbroker/ -v rpcmsg
```

Command `-v rpcmsg` is optional and enables printing SHV messages' content to terminal. Directory `shvbroker/etc/shv/shvbroker` contains configuration files with information like server port, host name, accepted users and their passwords and so on.

Another mentioned application, shvspy, is a GUI tool for SHV tree administration. It is connected as a client to shvbroker and allows the user to browse throught SHV tree and call methods. The steps to run the application are following.

```
git clone https://github.com/silicon-heaven/shvspy
cd shvspy
git submodule update --init --recursive
qmake -r
make
cd bin
LD_LIBRARY_PATH=../lib/ ./shvspy -v rpcmsg
```

It is also possible to download shvspy as a build binary.[9] The last command opens a GUI application to which user can add a new server and connect to it. Application designed with pysimCoder performs a mount to the directory selected in pysimCoder SHV settings. Figure 6.1 shows shvspy GUI with an application connected to shvbroker and mounted at `test` directory.

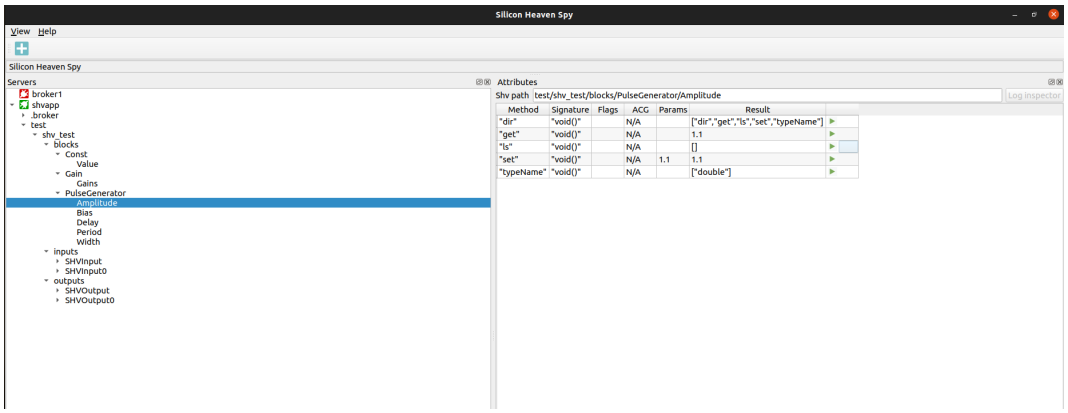

Figure 6.1. Silion Heaven Spy GUI application connected to shvbroker.

To summarize the presented tools, an application created with pysimCoder acts as a client and communicates with a server. Server is represented by shvbroker application. However we also want to interact with the SHV tree and control our model. Application shvspy is used for this purpose. It is a GUI tool in which user can browse throught the tree.

[9] `https://github.com/silicon-heaven/shvspy/actions`

Chapter 7
Examples and Documentation

The penultimate chapter of the thesis introduces the reader with few examples of NuttX, pysimCoder and Silicon Heaven infrastructure combination. Both target hardware microcontrollers, i.MX RT1060 and SAM E70 were used to demonstrate the functionality on a different hardware.

7.1 Introduction

The goal of these examples is to provide an sible entry point for a potencial pysimCoder users. Two examples were selected for this thesis, a simple RC Plant controlled by SAM E70 based board SAM E70 Xplained and more complex control system, permanent magnet synchronous motor (PMSM) control. This was done with Base Board for Teensy 4.1. The examples do not focus on control theory but rather on a practical usage.

7.2 RC Plant Control

RC Plant (resistor and capacitor) serves as an easily replicated example that can be implemented in few minutes and is a good introduction to NuttX, pysimCoder and Silicon Heaven. The example requires two electrical components, resistor and capacitor. The goal is to control capacitor's voltage based on a reference signal. Capacitor is charged throught resistor with PWM output, the actual voltage value is read with ADC. Apart from those two peripherals, support of TCP connection is required to support Silicon Heaven.

7.2.1 Hardware Connection

As mentioned above, SAM E70 Xplained board was used for this example. The following Figure shows electrical connection of board and RC Plant. Note that only used pins are listed in the Figure. This example can also be easily replicated with any other board supporting PWM, ADC and TCP.

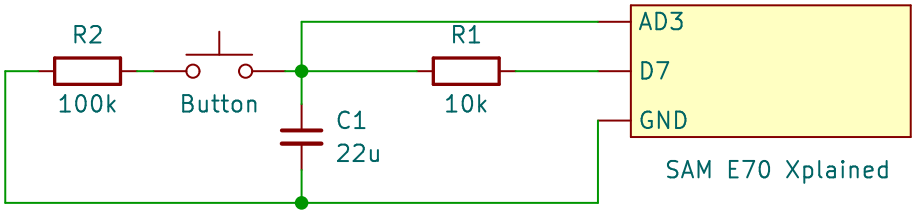

Figure 7.1. Electrical connection of RC Plant and SAM E70 Xplained.

Elements R_1 and C_1 are the RC Plant. Additional resistor R_2 can be used to discharge the capacitor and serves as an external error which controller needs to regulate. SAM E70 Xplained pin 3 is used as an ADC input, D7 is PWM output.

7.2.2 PysimCoder Application

Figure 7.2 shows the block diagram connection in pysimCoder. Pulse Generator block is set to generate a reference signal. The actual value, the capacitor voltage read with ADC, is substracted from the reference and sent to PID controller. The controller's output is then routed to PWM block which generates the PWM based on a duty cycle input.

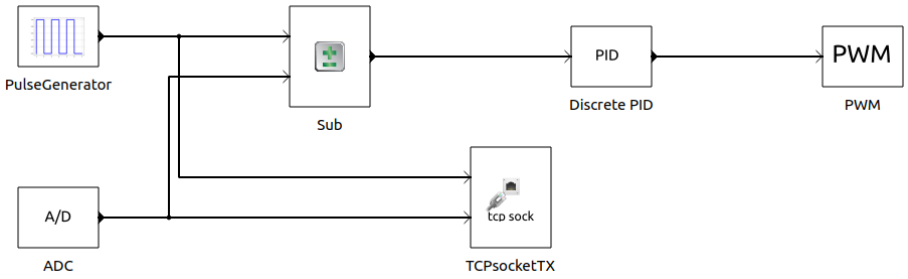

Figure 7.2. RC Plant block diagram in pysimCoder.

Used pin and device driver' name are selected in block's setting which can be opened by left double click on the block. The block diagram with the script providing the compilation of source files can be found in Open Technologies Research Education and Exchange Services (OTREES) GitLab repository.[1]

7.3 PMSM Control

The second example, control of permanent magnet synchronous motor, is a more complex system requiring larger amount of peripherals. Apart from PWM required to drive the motor, periherals like encoder and GPIO are needed. Current sensing with ADC peripheral is requried for vector control (also called field oriented control and abbreviated as FOC) however this type of control is not used in this example. Simplier position control based on a feedback from encoder is done instead. TCP support is of course needed for SHV usage as well.

7.3.1 Hardware Connection

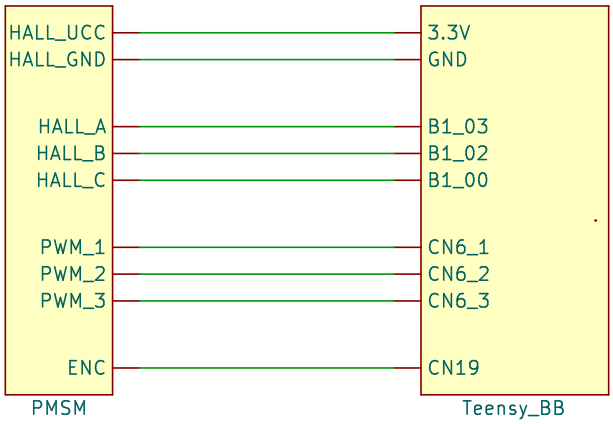

Figure 7.3. Electrical connection of PMSM and Teensy 4.1 Base Board.

[1] https://gitlab.fel.cvut.cz/otrees/nuttx-demos/-/tree/master/platforms/same70/rc-control

Teensy 4.1 Base Board made by Czech company PiKRON was used as a target hardware in this example. Hardware connection in Figure 7.3 is not much complex since the board provides connectors for both encoder input and PWM output. The situation is more complicated with Hall sensors' outputs. The board does not provide a suitable connector so they have to be connected manually to Teensy 4.1 board's pins.

7.3.2 PysimCoder Application

Block diagram designed in pysimCoder from Figure 7.4 can be used to control PMSM position based on an encoder input. Hall sensors are used to get an initial mechanical angle. Once the encoder reaches its index the angle can be determined from the encoder with better precision. The logic taking care of this process is implemented in a block `PMSM Align`. The alignment of mechanical and electrical angle may differ for every motor and needs to be measured and set before the control application is strted.

Blocks for inverse Park and inverse Clarke transformation are used to convert D (set as 0), Q (drived by PID controller) and angle (calculated from Hall/encoder) signals to three phased a, b, c vector that is used to set PWM duty cycle.

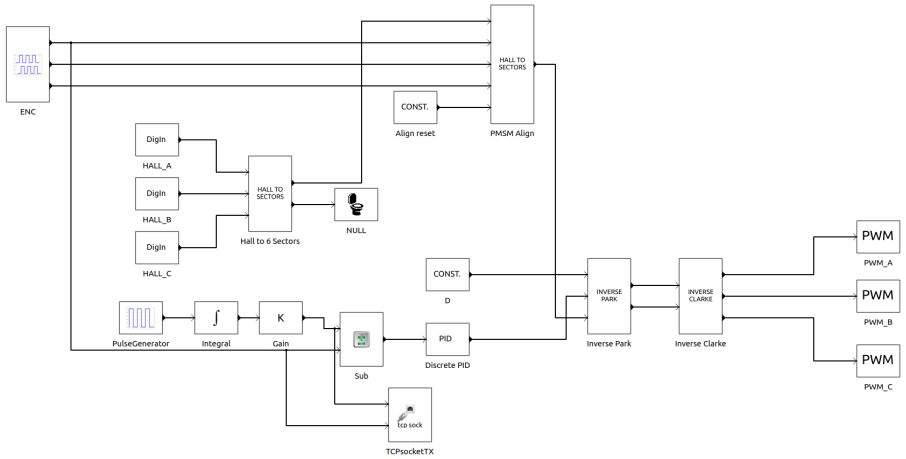

Figure 7.4. PMSM control block diagram in pysimCoder.

Please note this example can currently be used only with non mainline pysimCoder as encoder block uses IOCTL commands to get the index position that are not supported by NuttX mainline yet. These changes are not included in pysimCoder mainline for that reason. The support of these IOCTLs is also only for imxRT MCUs. The block diagram with the script providing the compilation of source files can be found in OTREES GitLab repository[2] as well.

7.4 Documentation

Documentation describing pysimCoder configuration with SHV can be found on pysimCoder's wiki page.[3] The documentation is written from user's point of view and focuses on the usage and configuration rather than on implementation and principles. The reading of this thesis is recommended for a deeper understanding of the implementation process and code organization.

[2] `https://gitlab.fel.cvut.cz/otrees/nuttx-demos/-/tree/master/platforms/imxrt/pmsm-control`

[3] `https://github.com/robertobucher/pysimCoder/wiki/Silicon-Heaven-Support`

Chapter 8
Conclusion

My work done in scope of this thesis enhances the capabilities of an open source tool pysimCoder which in some cases can be used as an alternative to proprietary software like Matlab/Simulink. The pysimCoder integration with a real time operating system NuttX brings the support for many affordable microcontrollers and boards.

Added support of Silicon Heaven infrastructure allows the runtime monitoring and tuning of model parameters and the introspection of designed diagram. This capality was tested and demonstrated on a real time control of RC Plant and PMSM. The possibility to change parameters runtime for example allows the students to easily experiment with a designed controller on a real hardware they can build and connect in home conditions.

The common part of SHV support can also be used in other not pysimCoder related projects as a standalone library. This is demonstraded by remote control of MicroZed APO board's RGB knobs used in Computer Architecture course at CTU FEE.[1] Used SHV communication interface is taken from my implementation to pysimCoder.

While I think all of the thesis's goals were fulfiled, some parts may offer a further involvement. This is a case of getting correct parameters' names to pysimCoder's generated C code. Generation of block's metadata during diagram creation would offer additional possibilites like deciding what parameters can be changed runtime and what can be just read only. Additon of integer parameters is also possible.

Contributions to both pysimCoder and NuttX were approved by systems' maintainers and added to the mainline. Both systems are fully open source and thus other students and users around the world may benefit from the changes and follow up with their ideas and contributions.

[1] `https://gitlab.fel.cvut.cz/b35apo/mz_apo-servo-knobs-shv`

Appendix A
Source Code

This chapter lists my contributions to NuttX and pysimCoder source code done during the work on this project. This includes the implementation of peripheral to NuttX mainline, commits introducing the SHV support to pysimCoder and also some smaller changes that were not directly SHV related but were necessary for succesful implementation. This includes addition of type names to `.xblk` file for example.

A.1 PysimCoder

The contributions to pysimCoder are listed in the following list.

- Option to get parameters names to C code,[2]
- RCPgen: link Inputs and Outputs with Nodes statically,[3]
- Added support for Silicon Heaven infrastructure to pysimCoder.[4]

A.2 NuttX

The following list shows the contributions that were accepted to NuttX mainline.

- samv7: add support for AFEC driver,[5]
- SAMv7: Add DMA and TC support to AFEC (ADC) driver,[6]
- SAMv7: Added support for PWM driver.[7]

[2] `https://github.com/robertobucher/pysimCoder/pull/40`
[3] `https://github.com/robertobucher/pysimCoder/pull/42`
[4] `https://github.com/robertobucher/pysimCoder/pull/48`
[5] `https://github.com/apache/incubator-nuttx/pull/4795`
[6] `https://github.com/apache/incubator-nuttx/pull/4901`
[7] `https://github.com/apache/incubator-nuttx/pull/5471`

Appendix B
Glosary

ADC	Analog to Digital Converter
ANSI	American National Standards Institute
API	Application Programming Interface
BSP	Board Support Package
CAN	Controller Area Network
CAN FD	Controller Area Network Flexible Data-Rate
CTU	Czech Technical University
DMA	Direct Memory Access
FEE	Faculty of Electrical Enegineering
FOC	Field Oriented Control
GAVL	Generated AVL Tree
GSA	Generated Sorted Array
GUI	Graphic User Interface
IOCTL	Input/Output Control
ISA	Instruction Set Architecture
LCD	Liquid-Crystal Display
MAC	Media Access Control Address
MCU	Microcontroller
PID	Proportional Integral Derivative Controller
PMSM	Permanent Magnet synchronous Motor
POSIX	Portable Operating System Interface
QSPI	Quad Serial Peripheral Interface
RGB	Red-Green-Blue
RPC	Remote Procedure Call Protocol
RTOS	Real Time Operating System
SHV	Silicon-Heaven Infrastructure
SPI	Serial Peripheral Interface
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver-Transmitter
UDP	User Datagram Protocol
USB	Universal Serial Bus

References

[1] Bucher, Roberto. *Python for control purposes*. Available from https://robertobucher.dti.supsi.ch/wp-content/uploads/2017/03/BookPythonForControl.pdf.

[2] Bucher, Roberto. *Rapid Control Prototyping with pysimCoder and NuttX*. Available from https://www.youtube.com/watch?v=y7NvFAp3OII.

[3] Carvalho de Assis, Alan. *What is the NuttX RTOS and why should you care?* Available from https://www.embedded.com/what-is-the-nuttx-rtos-and-why-should-you-care/.

[4] Zhang, Mingyang, Martin Timmerman, Luc Perneel, and Toon Goedemé. Which Is the Best Real-Time Operating System for Drones? Evaluation of the Real-Time Characteristics of NuttX and ChibiOS. In: *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*. 2021. pp. 582-590. Available from DOI 10.1109/ICUAS51884.2021.9476878.

[5] The Apache Foundation. *About Apache NuttX*. Available from http://nuttx.incubator.apache.org/docs/latest/introduction/about.html.

[6] The Apache Foundation. *Directory Structures*. Available from http://nuttx.incubator.apache.org/docs/latest/quickstart/organization.html.

[7] The Apache Foundation. *Device Drivers*. Available from https://nuttx.apache.org/docs/latest/components/drivers/index.html.

[8] Güven, Yılmaz, Ercan Coşgun, Sıtkı Kocaoğlu, Harun Gezici, and Eray Yılmazlar. Understanding the Concept of Microcontroller Based Systems To Choose The Best Hardware For Applications. *Research Inventy: International Journal of Engineering And Science*. 12, 2017, Vol. 6, No. 9, pp. 38-44. ISSN 278-4721.

[9] Bucher, Roberto. *pysimCoder - NUTTX - CAN - Control of the inverted pendulum*. Available from https://www.youtube.com/watch?v=iX_hfb6ZoR4.

[10] Bucher, Roberto. *004 Integration pysimCoder - NUTTX - Control application*. Available from https://www.youtube.com/watch?v=y7NvFAp3OII.

[11] The Apache Foundation. *pysimCoder integration with NuttX*. Available from https://nuttx.apache.org/docs/latest/guides/pysimcoder.html.

[12] NXP Semiconductors. *i.MX RT1060 Crossover MCU with Arm Cortex-M7 Core*. Available from https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/i-mx-rt-crossover-mcus/i-mx-rt1060-crossover-mcu-with-arm-cortex-m7-core:i.MX-RT1060.

[13] Lenc, Michal. *[2021] NuttX Support for Rapid Control Applications Development with pysimCoder*. Available from https://cwiki.apache.org/confluence/display/NUTTX/%5B2021%5D+NuttX+Support+for+Rapid+Control+Applications+Development+with+pysimCoder.

[14] Microchip Technology Inc. *SAM E MCUs*. Available from https://www.microchip.com/en-us/products/microcontrollers-and-microprocessors/32-bit-mcus/sam-32-bit-mcus/sam-e.

[15] Northrop, Robert B.. *Introduction to Instrumentation and Measurements*. 2 ed. Boca Raton: Taylor and Francis, 2005. ISBN 0-8493-3773-9.

[16] Harris, Sarah L., and David Harris. *Digital Design and Computer Architecture*. RISC-V ed. Cambridge, MA 02139, USA: Morgan Kaufmann, 2021. ISBN 978-0-12-820064-3.

[17] Microchip Technology Inc. *SAM E70/S70/V70/V71 Family*. Available from https://www.microchip.com/content/dam/mchp/documents/MCU32/ProductDocuments/DataSheets/SAM-E70-S70-V70-V71-Family-Data-Sheet-DS60001527E.pdf.

[18] Ahmed, Altaf, Abdullah Aljumah, and M Ahmad. Design and Implementation of a Direct Memory Access Controller for Embedded Applications. In: 2019. pp. 309. Available from DOI 10.14716/ijtech.v10i2.795.

[19] Píša, Pavel. *Ing. František Vacek*. Available from https://cw.fel.cvut.cz/b192/courses/b35apo/teacher/vacek/start.

[20] Vacek, František. *SHV RPC (ChainPack RPC)*. Available from https://github.com/silicon-heaven/libshv/wiki/ChainPack-RPC.

[21] Vacek, František. *Login example*. Available from https://github.com/silicon-heaven/libshv/wiki/shv-login-example.

[22] Píša, Pavel. *uLan Utilities Library (ULUT)*. Available from https://cmp.felk.cvut.cz/~pisa/ulan/ulut.pdf.