CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Control Engineering

Bachelor's Thesis

Open Rapid Control Prototyping, Education and Design Tools

Dion Beqiri
beqirdio@gmail.com

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

Acknowledgement / Declaration

Thanks to...

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

AbstractCZ

AbstractEN

Contents /

/ Figures

Chapter 1
Introduction

Chapter 2
PysimCoder

2.1 Introduction to PysimCoder tool

2.1.1 Control systems and their design tools

"A control system is an interconnection of components forming a system configuration that will provide a desired system response."[1, chapter 1]

We as a species have thrived upon the idea of controlling the processes around us. Of course, there are many natural occurrences that we cannot prevent nor avoid (yet), however we still maintain an increasing ability to control and automate all types of processes that can be used to our benefit. This concept of automated control systems has become the engine behind the rapid development of human technology, and concurrently has changed the way humans live and function as a society.[1] Therefore, as our technology advances further and faster, the necessity for automation solutions and engineers will only grow with it.

Due to the wide range of industries and fields which rely heavily on automation, there are various solutions and tools which offer the ability to design, simulate, and control real time systems. The software which is used all depends on the application and hardware which is being targeted. In an industrial setting, one would probably use TIA Portal [2] or CODESYS [3] to program the controllers of some production line in a factory. For designing and simulating complex electrical circuits, AutoDesk Fusion360 [4] would be a perfect environment for high accuracy and modern solutions, while LTspice [5] is more useful for simpler analog circuits.

For many fields of engineering, there is a corresponding software which is used for control design of a certain type of system, however for the purpose of this thesis I will focus on solutions which are meant for mathematical simulations, and code generation for microcontroller units. These include programs such as MATLAB and Simulink, which when used together make up a very elaborate tool for designing a system controlled by a microprocessor unit, and for visualizing the data coming from the physical system in real-time. In essence, Simulink is a graphical tool for creating block diagram schematics which represent a particular system using mathematics. Furthermore, users of Simulink can create their own blocks, not only for mathematical calculations, but even for control of physical hardware. Using this feature, one can combine the mathematical blocks with hardware-related blocks in order to generate binary code for execution on a MCU, which can then control different types of hardware connected to it. [6]

2.1.2 PysimCoder: Control systems design and code generation tool

Another such tool is PysimCoder, which has been the focal point of my diploma work. Started and developed by professor Roberto Bucher from the University of Applied Sciences and Arts of Southern Switzerland, PysimCoder is an open-source graphi-

cal tool used for real-time code generation, typically for microcontrollers and PCs which are controlling some physical peripherals. Additionally, it can also be used as a simulation tool for simple control schemes. Target operating systems include GNU/Linux with or without preemptive-rt kernel, and NuttX RTOS usually for constrained MCU based systems. At its core, it is a Rapid Prototyping Control application, similar in style to programs like Simulink and XCos.[7] This application is still under development, and it has nowhere near the amount of investment or resources that its alternatives use. Nevertheless, this application has some advantages which should be considered for a control system tool.

Firstly, it is open-source and freely accessible, which allows complete introspection of the generated code, as well as the source code of the application itself. This way, if there is an error or an issue during usage, anyone with a good knowledge of programming can go inside the source and fix the bugs as needed. Another great advantage is that it takes much less memory space, which results in faster installation and overall usage. At the current moment, the whole project takes up about 100MB of space on my hard drive, which is negligible compared to the many gigabytes of space which Simulink requires. This is immediately noticed when comparing loading times between the two.

Probably the best thing about this program is that it uses Python as a programming language for most of the core functions. Using many Python libraries which are specialized in control systems mathematics, the application is able to design a schematic on a source code level. The program can translate this design into C programming language, which can ultimately be compiled and deployed to a MCU as machine code. The combination of Python and C languages is perfect for easy deployment in most environments, both for the host running PysimCoder, and for the target running the generated code. [7, chapter 8]

PysimCoder is originally set up for installation in a Linux OS environment, however due to the many virtualization and containerization techniques used in computers today, the program can even be run in Windows OS, or Mac OSX [7, chapter 8]. The program is so light and accessible that you can really run it from any consumer-level operating system, with barely any strain on the CPU. This, in combination with the free and open-source nature of the program, makes it perfect for the software requirements of the future.

It is obvious by now that the paid and closed-source alternatives will have more features, stronger software stability, higher precision, and overall better performance. Nevertheless, pysimCoder still has a chance to become just as good, and still remain free and open-source. This will depend on students and researchers such as myself who will choose to develop further features and abilities to the program. One feature which pysimCoder would highly benefit from is being able to generate and use vector signals. This is something that Simulink already has full support for, therefore having it here will be one step further to a professional-level software, with virtually no financial cost.

In the next chapter, I will be explaining my journey to the extension of pysimCoder for support of vector signals and blocks with vector operations. Before that however, I must explain how the program works internally, and show the code-generation path from Python to C-code. This explanation will help to better understand the logic behind my implementation of the vector feature, as well as the other features added based on the requirements of this thesis.

2.2 Code generation path of pysimCoder

In order to fully understand the implementation of the features which I have added in the pysimCoder project, I must initially explain how the source code is organized and how it functions. When working on an open-source project, the greatest advantage is that you have complete access to the whole source of the application, however this comes with great risk as well. Any small change in the core functionality of the program can lead to unexpected results and errors. Therefore, the developer must always carefully analyze and understand the source code before they start making changes and adding new functions.

2.2.1 Graphical User Interface of pysimCoder

From the front-end perspective, pysimCoder is a graphical tool used for designing block diagram schematics for the purpose of simulation and real-time code generation. Knowing this, it is only natural that I give a small preview of the GUI of the application. As seen in the image below, the pysimCoder tool is very similar to its larger and more costly counterparts such as Simulink or xCos as far as the graphical editor is concerned. In fact, the editor itself is based on the `PySimEd` project [8] and the `qtnodes-develop` project [9], in combination with a lot of common blocks from PyEdit. [7, chapter 7]

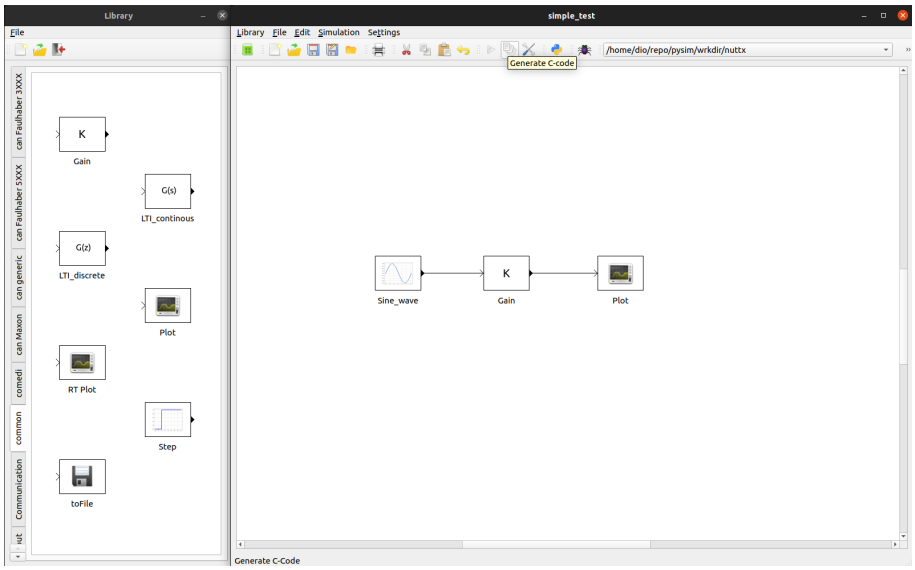

Figure 2.1. Simple preview of PysimCoder's graphical tool

The application is divided in two windows: the library and the diagram canvas. On the left, we have the library window which contains all the blocks which are supported on pysimCoder. From here the user will be able to drag and drop different types of blocks onto the canvas, depending on their control needs. Then on the right, there is the canvas window where the user will be able to interconnect all the blocks that were added there from the library. On the top of this window there is a toolbar and a menu bar from which the user is able to perform a handful of operations necessary for their design. The most note-worthy buttons used are: `Generate C-code`, `Simulate`, and `Settings`. These operations are essential to the code-generation and simulation functions of the application.

Regardless of the complexity of the diagram, the work flow (set of operations) will always be the same getting from design to code generation. The blocks are dragged

to the canvas from the library, then interconnected based on the user's design. Once the blocks are nicely organized and connected, the user will then have to tune any block parameters that must be changed for their particular system. After verifying the correctness of the diagram, the user will have to choose a target-specific template Makefile which will tell the program which compiler to use for the current project. Finally, the user must save the project, and the `Generate C-code` button will be pressed in order to build a binary executable file. In case of a simple simulation running on the host computer, the `Simulate` button can also be pressed in order to execute the file in the same terminal where the pysimCoder application is running. [7, chapter 7]

Every instruction given above will be related to the next part of this section, where I will explain in simple terms how the source code is reacting to the different operations we are performing in the graphical tool, and what is the final result of the code generation performed by the application.

2.2.2 Source code of pysimCoder

The core functionality of the pysimCoder project is enabled by three main languages: Python, C , and Makefile. `Python` is the engine behind the graphical tool, and the back-end processes related to the operation of the tool, and also handles the C-code generation. The generated `C code` will contain the whole system design with all of its blocks, and their respective parameters and interconnections. `Makefile` is used for organizing the compilation of the .c files of the blocks, installing all modules necessary for use of pysimCoder, as well as for producing the final executable file which is the end result of the design made in the GUI. Figure 2.2 gives a more simplistic view of the code generation path, from GUI to the executable generation.

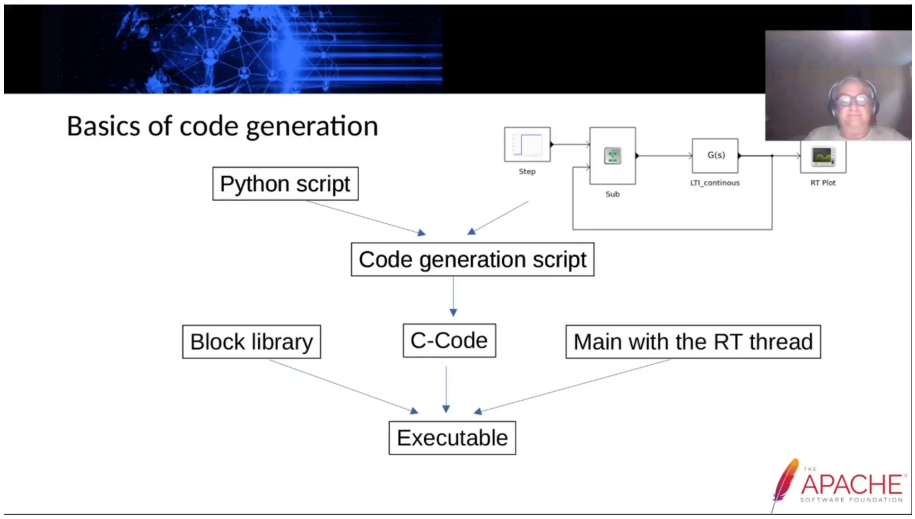

Figure 2.2. Code path from GUI to executable file

Each block from the Library window has three main components which complete its functionality in the code. Firstly, before any code generation takes place, there is the .xblk file which holds all the info of the block which the graphical interface can interpret. These include necessary information such as number of inputs and outputs, assignment of editable parameters along with their default values, name of the block, and also name of the .py file which will be essential to the further processing of the block.

The reference to the .py file specified in the .xblk file is the next step once we want to move towards code generation. This .py file is responsible for calling a python class

named RCPblk, whose purpose is to create an object from the current state of the block to be used in the main code connecting it to the other components of the schematic.

Last but not least, the block needs a C-code function that will be used in the main C code generated by Python. This function will contain the actual code necessary for achieving the desired output of the block.

Now I will break down the path of the source code based on the work-flow presented earlier in the GUI. In the initial phase, the design of the schematic is taking place. This includes everything from dragging and dropping blocks onto the editor, making connections between them, setting their parameters and choosing the simulation settings. During the design phase, various Python tools from the path `/toolbox/supsisim/src` are being utilized to run the graphical tool, and to ultimately save the state of the current project into a file with .dgm extension. This .dgm file will contain everything known about the design before code generation takes place, which mainly includes the edited .xblk of each block, the connections between the blocks, and the simulation settings.

Once the design of the schematic is completed and saved into a .dgm file, the code generation can finally take place. In this phase, the same tools from the path `/toolbox/supsisim/src` will be used. When the `Generate C-code` button is pressed in the toolbar, the supsisim toolbox will first create a file called `tmp.py` and store it in the working directory of the project. This .py file will be responsible for collecting all the C-functions of each block, and using them to generate the main C-code. Then it will create a folder in the same path, and it will store there the main C-code that was generated, and also a Makefile. The last step of the `tmp.py` file will be to call the Makefile, which will finally compile the generated C-code into a binary executable. This binary file is the end-product which is obtained from the simulation design, and is the only file necessary for the execution of the simulation on the target-hardware.

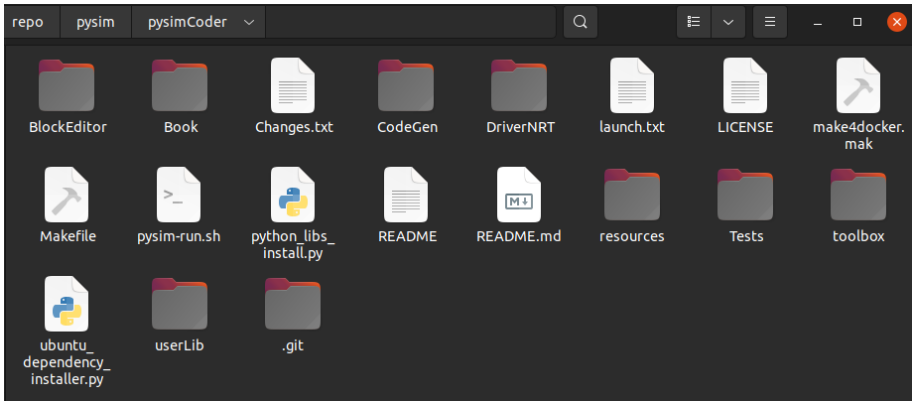

Figure 2.3. Source code organization in main repository of pysimCoder

Above we can see the root filesystem of the pysimCoder project. Most of the core work being done by the program is happening at the `toolbox` folder. In the `resources` folder there is all the .xblk and .py files of each block, as well as all the images used for the blocks' thumbnails. For all code generation purposes of the program, the folder `CodeGen` will be used. Here lies all the C files containing the functions specific to each block (including their dependencies), as well as all the template Makefiles used for the various targets that are supported. These three directories are essential to the pysimCoder application, and work closely together during the path from the GUI to the binary executable generation.

Although there are a few parts of the source code I have not explained, there is already enough information in this chapter to be able to clearly understand the different implementations and changes that will be made in the forthcoming chapters.

Chapter 3
Vector support

3.1 Vectors in Control Systems

As remembered from any elementary mathematics or physics class, a `vector` is a directed line segment. Usually a vector can be visualized on a line, a plane, or a space, otherwise known as the three spatial dimensions. In other words, a vector is a set of numbers, which represent a line with a direction in an N-dimensional space (see Fig. 3.1). Although vectors can have a finite or infinite number of dimensions, they have already played a significant role in mathematics and physics even with only the three spatial dimensions that we are closely familiar with. The uses of vector mathematics reach wide across many fields of science, such as classical mechanics, astrophysics, electronics, and many others.[10] Included here is also control engineering, which uses vectors for many of its mathematical operations.

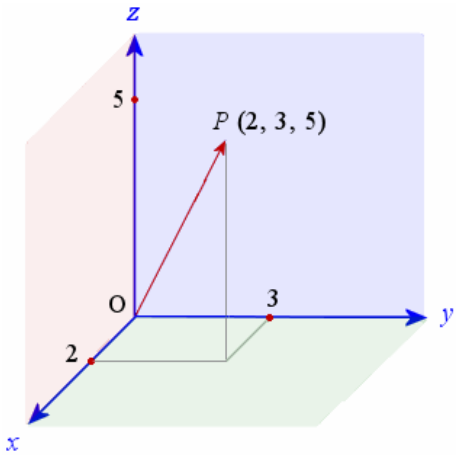

Figure 3.1. Visualization of a vector in a 3-dimensional space (x,y,z axes). The point P represents the direction of the vector, and contains the magnitude as well.

In order to design and analyze control systems, quantitative mathematical models of these systems must be obtained. This condition is true independently of the type of dynamic system in question. The mathematical model of the dynamic system is initially expressed as a set of differential equations, and then usually (but not always) linearized for further use with the Laplace transform. The Laplace mathematical tool is then able to obtain a solution of these linearized differential equations, and ultimately describe the operation of the system.[1, chapter 2.1]

Once the mathematical model is simplified with the use of transfer functions (obtained from Laplace), the input-output relationship can be described. This function is especially useful when wanting to represent the system as a block diagram schematic (see Fig. 3.2). The blocks are unidirectional and operate based on the transfer function of the variables to be controlled.[1, chapter 2.6]

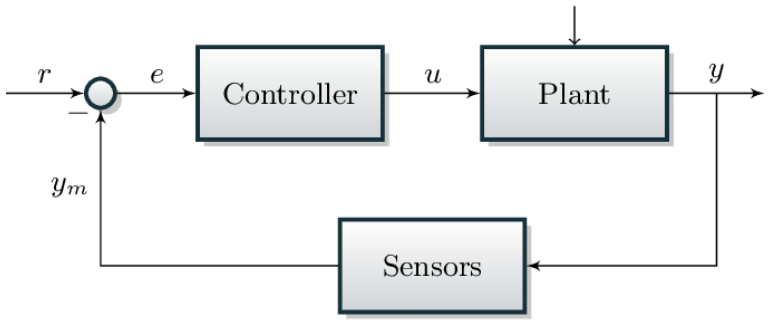

Figure 3.2. Example of usage of block diagram schematics for representation of a closed-loop feedback control system.

Nowadays all the graphical tools for control systems are based on this concept, and many of them use vector signals for operation of their blocks. PysimCoder also uses vectors (and matrices) for many of the mathematical operations necessary for the simulation of a control system, however before my implementation covered in the next sections of this chapter, it did not support the use of vector signals as inputs or outputs of a block. This is a feature that would greatly improve the application, since it can can simplify the diagram appearance, and it could better utilize many of the mathematical tools needed for controlling a dynamic system.

Simulink already supports vector signals both in the graphical editor and in the C-code generation tool (SimulinkCoder). The `Constant` block has the ability to produce a single number, but also a vector or matrix. This block can be used to generate a pure vector signal which is constant on its output. There is also a `Mux` block [11] which can combine input signals into a vector, and also its inverse `Demux` [12] which extracts each element from the vector signal and outputs them individually. These blocks can be used in combination with other blocks which may have vectors as an input or output of their operation. Simulink only has a handful of `pure` vector blocks, including `Mux` and `Demux`. Most of the other blocks support single-data signals, as well as vectors on top of that, if the input is given as such.

Many of the same blocks mentioned above that are used in Simulink can be emulated in pysimCoder. Therefore, for the purpose of testing and experimenting with vector signals for the first time in pysimCoder, I have used the same concept of these blocks. I started with a `Constant Vector` block, which could create a pure vector signal, together with a `Print Vector` block to print the signal on the terminal. After that I proceeded with creating the `Mux` and `Demux` blocks, since they will be able to convert vector signals into its individual elements, and back. This can create an interface between the vector blocks and the `normal` blocks. The approach to this will be covered in the next sections of this chapter.

3.2 Basic implementation of vectors in pysimCoder

3.2.1 Method of approach

Although pysimCoder did not have vector support before this implementation, it did have a framework for enabling the dimensions of a block. By creating new pysimCoder blocks which utilize this ability to carry input and output dimensions, I will be able to test the processing of vectors in the C-function level. In order to achieve this I will have to make changes in the source of the code generation tool, as well as

adding blocks in a new `Vectors` library. For initial simplicity, the dimensions will be set as a block parameter from the user. The automatic setting of dimensions will be covered later, in the next section of this chapter.

3.2.2 Changes in Python source and generated C-code

Before trying to create the new blocks for testing vector signals, first the source of the pysimCoder project must have the ability to enable vectors for these blocks. By making some minor changes to the block representation object, as well as the code generation process, the new blocks will then be able to read their own dimensions in order to properly process the input data.

To better understand where to make these changes, I will refer to the main elements of a pysimCoder block from chapter 2: the .xblk file, .py file, and .c function. In fact, these elements follow a strict hierarchy of representation of a block, as seen in the image below.

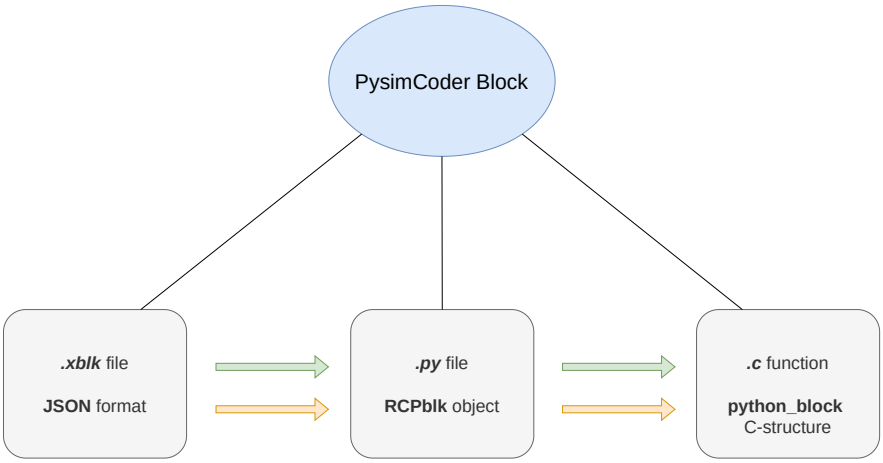

Figure 3.3. The supporting files for a pysimCoder block are shown. The green arrows represent which file contains the specification of the next file or function to be called. The orange arrow shows the switching of block representations. The general direction of both arrows show the hierarchy of the code dependencies of the block.

On the GUI level, the block is initially represented in the same format as the .xblk file, otherwise known as the JSON format (although there is no JavaScript involved). The corresponding Python function specific to the block is then called with all the information contained in the JSON, and will ultimately create a Python class called `RCPBlk`. This class is the next level of representation, and it will be used while pysimCoder is generating the C-code. When Python is generating the code, it will use each RCPBlk object to initialize the block on the C-code level. To do this it will pass all of the block information to a C-structure called `python_block`, which will enable all the written C-functions of the block to utilize its own properties for correct functioning of the code. This C-structure is the last level of representation, and is the only format needed for running of the code on the target. In other words, the other formats only serve to generate this final one. [7]

So all these representations seem to be carrying the block information, and the last two already have some form of support for dimensions of input/output signals. They are seen in the code below, where both of them have the variables already made to carry dimension data.

```
...
typedef struct {
  int nin;              /* Number of inputs */
  int nout;             /* Number of outputs */
  int * dimIn;       /* Port signal dimension */
  int * dimOut;      /* Port signal dimension */
  int *nx;              /* Cont. and Discr states */
  void **u;             /* inputs */
  void **y;             /* outputs */
  double *realPar;      /* Real parameters */
  int *intPar;          /* Int parameters */
  char * str;           /* String */
  void * ptrPar;        /* Generic pointer */
}python_block;
```

For `python_block` there is nothing to change, however in the RCPBlk object, all of the dimension information seems to be defaulted to `1` for all inputs and outputs. This is the first change to make, so that when the RCPBlk is called from the block's Python code, the dimension will pass from the user's input to the `python_block`. Otherwise it would just default to `1`, making vectors impossible to use here.

```
...
        self.name = None
        self.fcn = fcn
        self.pin = array(pin)
        self.pout = array(pout)
        self.dimPin = ones(self.pin.shape)
        self.dimPout = ones(self.pout.shape)
        self.nx = array(nx)
        self.uy = array(uy)
        self.realPar = array(realPar)
        self.realParNames = []
        self.intPar = array(intPar)
        self.intParNames = []
        self.str = str
...
```

After that is settled, the block can then take a dimension as an input parameter from the user, and pass it on to the other representations. This is key to the whole feature, since now I will be able to show how the source code can use this information in order to create actual memory space for the signal.

In pysimCoder, the main file which deals with code generation is `RCPgen.py`. In that file is where all the blocks of the simulation are gathered and interconnected, and then generated as C-code. The tool uses the `RCPblk` objects of each block in order to propagate the information to the `python_block` structure. The blocks are interconnected using `nodes`. A `node` in terms of pysimCoder is a connection between an output of a block, and the inputs of one or many other blocks. The node will carry the values from one block to another, however the source only generates one value per node. This is another barrier preventing the program from enabling vector support, and it is the next change to be made to prepare the blocks for using vector signals.

In the current version of RCPgen.py, it seems that there is only one value given for each node. This setup will not allow any vectors to be generated or used, therefore a short algorithm will be developed to set the correct array space for each node. Generally the nodes must be as large as the inputs and outputs it is connected to, so the array length of each node can be set by either one. Considering this fact, the algorithm will simply go through each block's output, find the dimension of that output, and then append this value to a list which will hold all the nodes' dimensions. Then later on when the nodes are being initialized in C-code, the array length will be set based on the dimension found in the list. The application of this small algorithm will allow the generated C-code to propagate vector values, since it now has memory space to do so.

3.2.3 New blocks for vector testing

After the changes made to the source as explained above, the new vector blocks will finally be able to pass vector signals and use their dimensions for the correct functioning of the C-code. The first vector block to create will be the `Constant` block, as seen in Simulink. This will be able to create a vector output signal based on the input array given from the user. Then to test if the vector signal is actually passing correctly, there is a need for a `Print` block which will be specific to printing vector signals as an array on the main terminal of pysimCoder. These two blocks will become the first experiment on vector signals.

The next blocks to be created will be related to mathematical operations on vectors. The `Gain` block takes each element of the input vector and multiplies it by the factor given by the user. In fact the `Gain` block already existed in pysimCoder, however it has never been used with vectors as its input. Then two very simple arithmetic operations will be implemented for vectors: summing and subtracting. Describing what they do is somewhat redundant, however it's worth noting that the input vectors of both blocks must be of equal dimension for correct operation. By refining these blocks, I will be able to prove that vectors can be passed, and processed in a mathematical way.

Finally there are two very important blocks which have a special role with vectors in a block diagram schematic. These are the `Multiplexer` and `Demultiplexer` blocks, otherwise known as `Mux` and `Demux`. `Mux` is responsible for generating a vector from many single inputs, while `Demux` breaks down the vector into many single outputs. This functionality will be especially useful in the future, when `normal` blocks want to interact with `vector-able` blocks.

These are all of the vector blocks that have been crafted for the purpose of testing vectors in pysimCoder. It is worth mentioning that some blocks are able to know their dimension even without any user input. If the dimensions can be found using some mathematical relation, then it can be passed automatically from the Python file which calls RCPBlk. Otherwise, the dimension will have to be given explicitly from the input parameters of the block, therefore it will pass first from the .xblk representation. In the next section I will explain my extended implementation of vectors, where all the dimensions are set automatically by an algorithm, and no user input is needed.

3.2.4 Demonstration of basic vector support (static dimension setting)

To demonstrate the functionality of the first vector feature in pysimCoder, I have prepared some short examples which are using vector blocks. The first figure below showcases the usage of blocks with mathematical operations (subtraction, sum, gain) on vectors. The vector is summed with itself, and then subtracted by itself, and finally

amplified by a gain of 10. The output is shown in the terminal, found in the bottom right corner of the figure. The result should be the same vector, multiplied by 10 (the gain). On the left of the terminal, there is also a small window where the setting of the dimensions can be seen. Further to the left is the Library, where all the vector blocks can be dragged and dropped to the diagram.

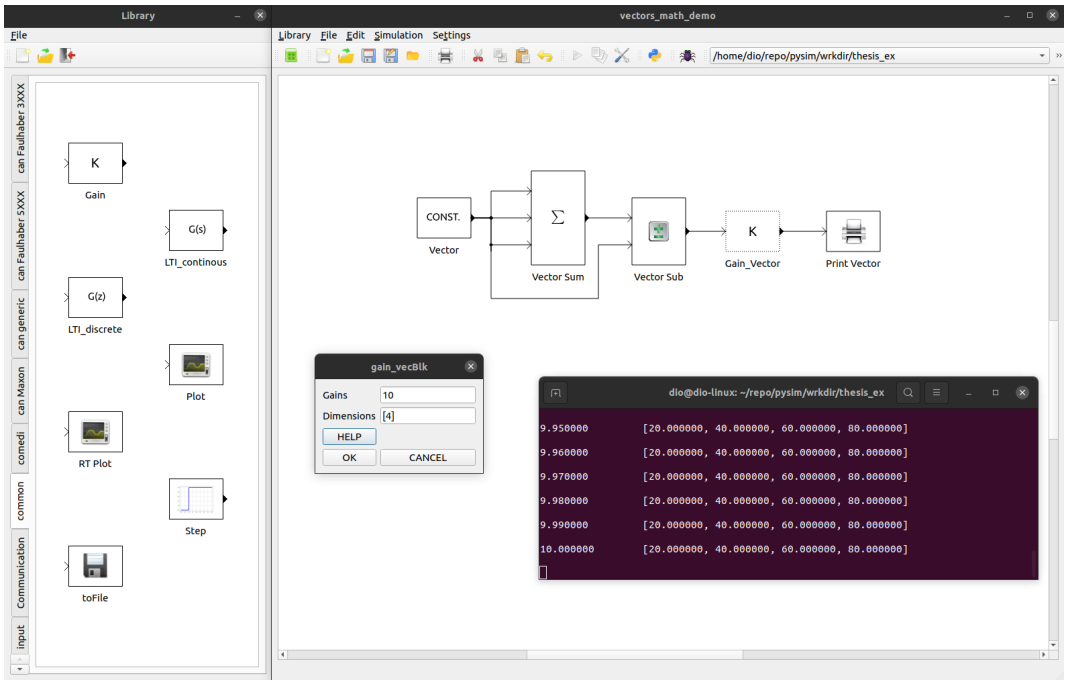

Figure 3.4. Diagram demonstrating usage of mathematical operations on vectors in pysimCoder. Dimensions here are set statically from the user input, as seen in the small window to the left of the terminal output.

In the next figure, there is a very insightful example, which involves the Mux and Demux blocks. Since the two are an inverse to each other, it is possible to connect them to each other, and get the same result from which it started with. Although this seems redundant in a real-life engineering situation, it is in fact a helpful test to see if the two blocks work correctly. As seen in the figure below, the diagram starts with a vector, gets broken down by Demux, and then recollected with Mux. Then the signal is printed on the terminal, and the vector appears to be the same as the input given for the Constant block. This result means that everything is working correctly. The diagram can also be reversed (Mux-Demux), but then the general Constant and Print blocks would be used instead of the new vector versions.

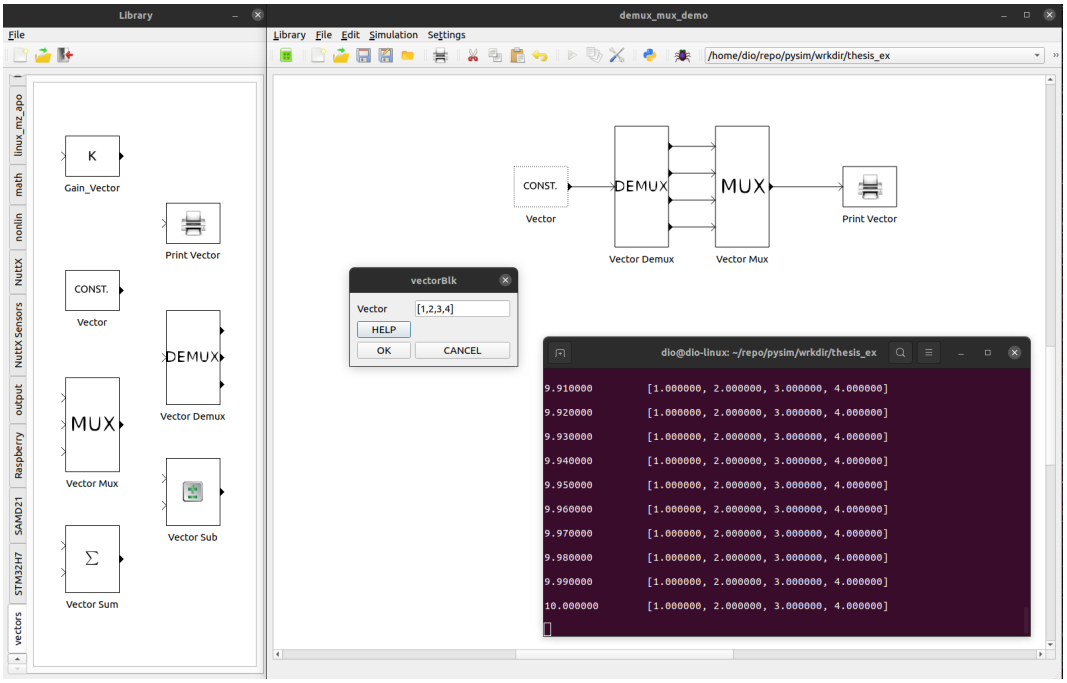

Figure 3.5. Example using Mux and Demux blocks to test each other, as seen from the pysimCoder GUI and terminal. The result of the Print block should be the same as the vector from the Constant block.

3.3 Vectors with dynamic dimension-setting algorithm

3.3.1 Method of approach

In the initial functionality made for vector support, all the dimensions were either set manually, or set automatically from their internal properties. In this section I will cover the automatic setting of dimensions for all blocks which cannot know their dimensions by their own.

To do this, I will create an algorithm which will find each of these blocks, and set their dimensions based on the output of the preceding block. The detection and setting of dimensions must happen during the code generation process. By implementing this algorithm, the dimensions will not be passed as integer parameters anymore, but will use the appropriate variables for dimensions of input and output seen in the `RCPBlk` object class, and the `python_block` C structure.

It is worth to be noted that there are many approaches to setting dimensions automatically in a block diagram schematic which supports vectors. However, for the purpose of showcasing an experimental version which works with the previously working setup of the vector feature, I have chosen the above methodology. In the future I hope that pysimCoder developers including myself will use this proof as a motivation for a more general and organic solution for this feature.

3.3.2 Development of Algorithm

As seen in the last section, some vector blocks seem to know their dimensions internally, such as `Constant`, `Mux`, and `Demux`. The first two of these also seem to be generating a vector output. For the purpose of the explanation of this dimension setting

algorithm, I will denote these two blocks as `Source blocks`. It is important for this implementation that the `source blocks` always know their dimensions internally, since their dimensions will be passed on down the line.

Other blocks, which previously needed their dimensions to be specified, will then be split into two groups. The `middle blocks` are vector blocks with both input(s) and output(s), while the `destination blocks` are those with only vector input(s). Since the latter only have input signals, then they can simply inherit the dimension from the output of the preceding block. However for the `middle blocks`, there is also the setting of the output signal dimensions, which will need further processing.

To find a way to set the output dimensions of a `middle block`, each input/output relationship must first be investigated. In the best case scenario, the output signal would inherit the dimension from the corresponding input signal, assuming a 1:1 mapping of dimensions. In this case, all the middle vector blocks which were previously created (see section 2 of this chapter) have a 1:1 ratio of input/output dimensions, which will make the processing much easier. Nevertheless the chance of having a different ratio when working with vectors in the future is quite high, therefore I have ensured to leave a framework for making blocks which have a more complicated I/O relationship.

To describe the I/O relationship of a block on a mathematical level, I will use the notion of the `dimension ratio`. In other words, this is the ratio between the input dimension and the output dimension. Just earlier it was mentioned that the current `middle` vector blocks all have a 1:1 ratio, which means that the value of the input dimension will be equal to the value of the output dimension. In other cases, the output dimension would have to be multiplied by some factor in order to achieve a natural number. This is the exact reason why I have decided to make a new parameter in the RCPblk class by the name of `dimRatio`. This parameter will hold the information of the I/O dimension ratio, ultimately guiding my new algorithm on how to set its output dimensions.

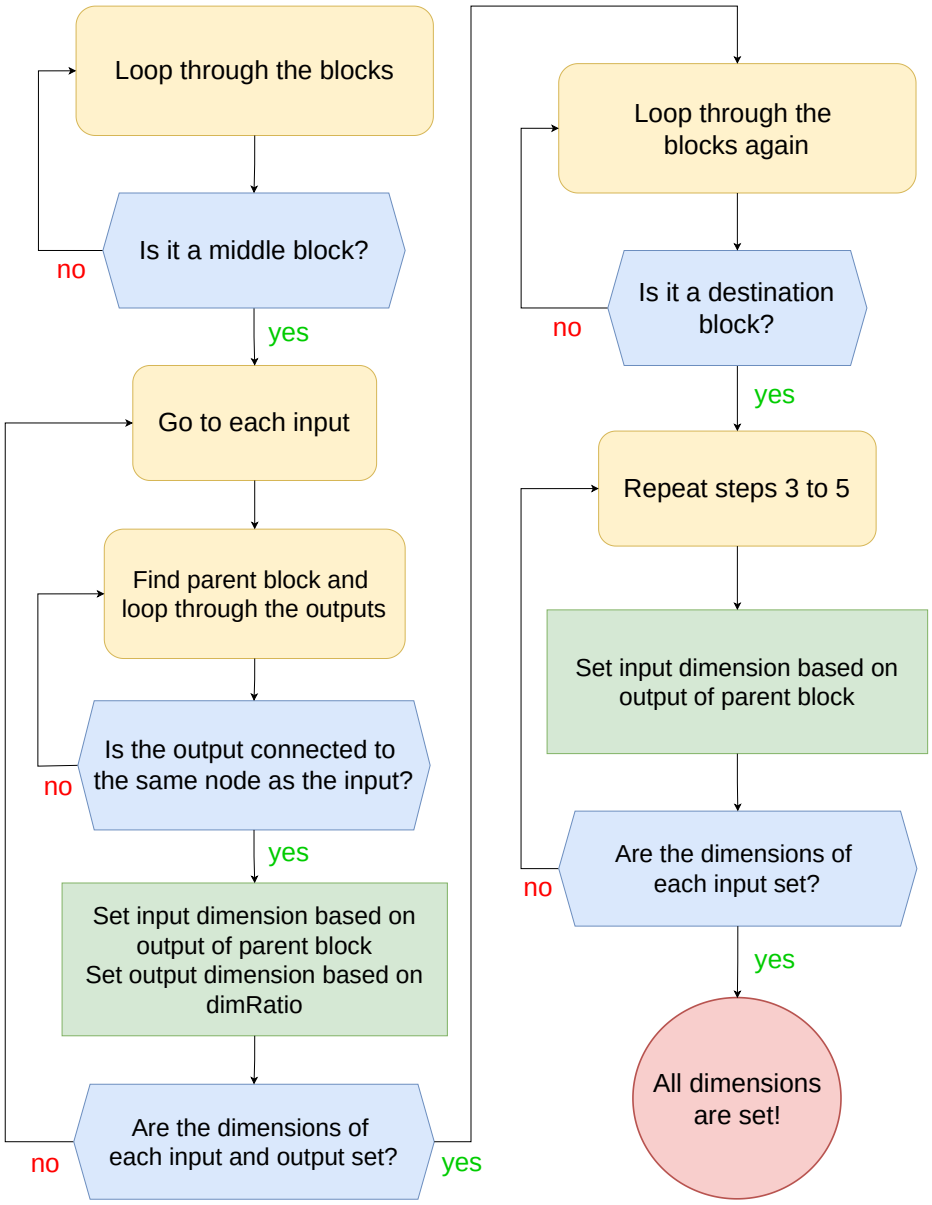

Figure 3.6. Flowchart explaining the path to setting dimensions automatically for vector blocks which previously needed input. Chart is divided in two parts: the left shows the setting of middle block dimensions, while the right is for the destination blocks. The first element of the flowchart is on the top left.

Now I will proceed to explain the inner workings of the dimension setting algorithm for vector blocks, using the flowchart given above (see Fig. 3.4). The first part of the algorithm will be dealing with the middle blocks, as seen in the left half of the diagram. Once that part is resolved, the algorithm can then continue to process the destination blocks. This has to happen in this order since the destination blocks usually get their dimensions from a middle block.

Initially the middle blocks must be found and accessed (Steps 1 and 2). Then the program must iterate through all the inputs of that block, and proceed to find the `parent` block which contains an output connected to the current input selected (Steps

3,4,5). This input will then inherit its dimension from the corresponding output of the parent block. The output signals of the middle block on the other hand, will be based on the dimension ratio parameter contained in the block (Step 6). If all the inputs have been dealt with, the algorithm will then move to the next phase, involving the destination blocks.

For the destination blocks, the same idea will be applied as for the middle blocks. The main difference here, will be that for destination blocks there are only inputs. This means there is no need for any dimension ratio to help select an output dimension, since there are no outputs. Therefore the inputs of the block will simply inherit their dimension from the parent block's output, and the algorithm will be done (Steps 8-13). Most of the steps are identical as for middle blocks, except for Step 9 and 11.

Although the flowchart seems a bit complex, the idea behind the algorithm is not so hard to understand. The goal is to propagate all the vector dimensions, from a `source block`, through the `middle blocks`, and onto the `destination blocks`. To do this the algorithm simply goes to each middle block, sets the dimensions based on the block preceding it, and then continues to do the same for all the destination blocks. Also, it is very important that there is a source block preceding all the other ones, and it must have static dimensions which were set internally.

3.3.3 Changes in source code

After understanding the core logistics of the dimension setting algorithm, a decision must be made on where to apply the algorithm, and what other changes must be made. The main changes in the source code will be related to the code generation process, and also to the block definitions of the vector-enabled blocks. Before I explain my choice, it is worth noting that there are many places in the source where the algorithm can be implemented. The only condition is that it is done before the variables of the dimensions are initialized in the code generation process.

Most of the C-code writing that pysimCoder does is occurring in `RCPgen.py`, from the `supsisim` toolbox (see chapter 2.2.2). Therefore this file in a way would be the first place to put the algorithm, right before the variables of the dimensions and the nodes are initialized in the C-code. Although this placement works and is more direct in terms of implementation, other methods should be considered for the future. For example it can also be written as a function, and imported externally for this feature, in order to shorten the code in `RCPgen.py`. Another consideration would be to apply the algorithm on the GUI level. This could be handy for seeing all the dimensions before the code is even generated. Nevertheless, the first option has been chosen as a testing ground for the more advanced versions to come.

In order for the algorithm to function, it needs to be able to interact with a parameter used for the dimension ratio, which describes the I/O relationship of the vector block. This parameter will be called `dimRatio`, and it will be placed in the `RCPblk` class. The reason for this placement in the block representation hierarchy, is the fact that the algorithm itself is placed in the RCPgen.py file, as mentioned above. The dimRatio will be a list with two elements, representing the ratio. The first element is the numerator, and the last is the denominator. In future implementations, the algorithm must be able to use this parameter for all types of ratios that could occur. However, for the current blocks which are contained in the `Vectors` library, there are only 1:1 ratios used (as seen in last subsection). For the `destination blocks`, who don't have any outputs at all, I will use the 0:0 ratio, so that the algorithm can know how to process it.

For the blocks to be able to adapt to these changes in RCPblk and RCPgen, all of the block file dependencies must be changed. First the input parameter must be removed from the .xblk definition, so that the user will not be able to input any dimension manually. Then the `dimRatio` variable must be enabled in the .py file of the block, and set to the corresponding ratio involved. Then the C function will also be changed, so that all the functions of the vector blocks will be using the `dimIn` and `dimOut` variables defined in the python_block structure. Previously the blocks were using the `intPar` to pass the dimensions in the C-code level, however after the next changes that will be mentioned below, all the blocks will be able to read their dimensions from the newly enabled variables.

I will finish this subsection by mentioning the other changes that were made in RCPgen.py. After the algorithm achieves its goal, the variables in the main C file which will hold all the dimension values will then be initialized. The nodes variables will also be initialized with the correct length of array (as seen in section 2 of this chapter), however this time it will simply read the dimension from the node output and directly use that value. In the first implementation of vectors, these values were buffered through a Python list. Last but not least, the python_block structures given for each block in the diagram will then take the values from the aforementioned dimension variables, and pass them onto the structure's internal dimension variables (dimIn/dimOut).

3.3.4 Demonstration of automated dimension setting

Since most of the changes in this section had no effect on the GUI, I will show a very similar demonstration to the first one seen in the basic vector implementation (from previous section). The difference here will be that there is no input dimensions given. If the output on the terminal is correct mathematically, and there are no other errors, then the algorithm has proven itself to work. This proof can be seen in the figure below, however this is a best-case scenario situation. In more complex diagrams there could be some issues.

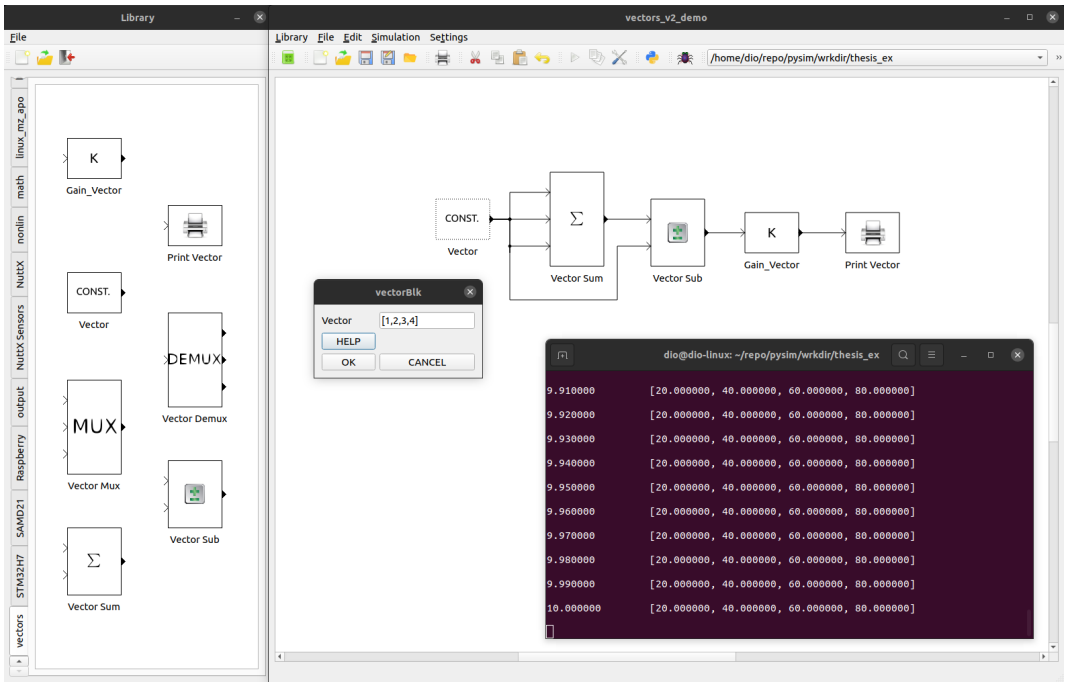

Figure 3.7. View of diagram with mathematical blocks which previously needed dimensions as input parameter, but is now solved automatically from the new algorithm. The result seen on the terminal output is correct, therefore the algorithm has succeeded (since no dimensions were given before).

Chapter 4
Extending support of LinuxOS/GNU targets in pysimCoder

As mentioned in chapter 2, pysimCoder can generate code for many different real-time operating systems. One of the main goals of this thesis is to add support for different boards which support Linux/GNU OS. Therefore this chapter will cover all the support added for the Xilinx Zynq based boards, and also the extension of support for the RaspberryPi boards.

4.1 Zynq based board on pysimCoder

In the following section I will describe how the first implementation of a Zynq target has been achieved. Most of the code is based on an existing Simulink project, and then adapted for deployment in pysimCoder.

4.1.1 Hardware introduction

The MZ_APO educational kit is a microcontroller unit based on the Zynq-7000 SoC family, designed by Ing. Petr Porazil at PiKRON sro. The board offers many peripherals and interfaces, which is why it is used as an educational tool for many courses in CVUT such as Computer Architectures, Advanced Computer Architectures, and Real-Time Systems Programming. It uses the ARM architecture, and supports Linux/GNU as its operating system. The details of the board are presented below (full details found at [13]).

- Base Chip: Xilinx Zynq-7000 All Programmable SoC
- Type: Z-7010, part XC7Z010
- CPU: Dual ARM Cortex ™ -A9 MPCore ™ @ 866 MHz (NEON ™ Single/Double Precision Floating Point)
- 2x L1 32 kB data + 32 kB instruction, L2 512 KB
- FPGA: 28K Logic Cells (430K ASIC logic gates, 35 kbit)
- Computing units in FPGAs: 100 GMACs
- FPGA memory: 240 KB
- Memory on MicroZed board: 1GB
- Operating system: GNU/Linux
- GNU LIBC (libc6) 2.19-18 + deb8u7
- Linux kernel 4.9.9-rt6-00002-ge6c7d1c
- Distribution: Debian Jessie

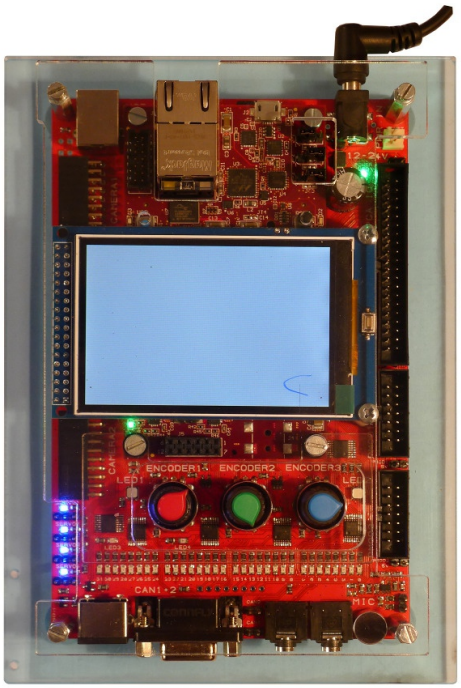

Figure 4.1. The hardware design of the MZ_APO kit is presented. Some main peripherals that can be noticed easily are the LCD screen, three rotary knobs, GPIO pins, LEDs, and the interfaces on the edges of the board. It is commonly used as educational kit for some courses in the Czech Technical University and was designed by Ing. Petr Porazil [13]

To interface with the device, many common slots are supported such as USB-A, Gigabit Ethernet, UART1 serial port and even a slot for microSD card storage. The peripherals included on the board are various, as can be seen in the list below (full list at [13]).

- Small 16-bit parallel bus connected LCD display (480× 320, RGB 565)
- 32 LEDs for direct visualization of 32-bit word (SPI connected)
- 2× RGB LED (SPI connected, 8-bit PWM)
- 3× incremental encoder rotary knob (RGB 888, SPI connected)
- 1× 40 pin FPGA IO connector, 36 FPGA 3.3 VDC signals, jumper enables +5 VDC power, signals match Altera DE2 kits
- 2× PMOD connectors extended by optional +5 VDC power, each provides 8 FPGA signals shared with FPGA IO connector
- 2× parallel camera interface, one 10-bit and one 8-bit
- audio output by simple PWM modulator, on-board speaker and JACK available
- audio input to Xilinx integrated ADC, on-board microphone and JACK

Many of the peripherals above can be implemented in pysimCoder as blocks, however for the scope of this thesis I will be working with the rotary knob `incremental encoders`, as well as the `PMOD connectors`. These two peripherals are already implemented as Simulink blocks, as it will be presented in the next subsection.

The three incremental encoders are physical rotary knobs which can change values in an 8-bit field (from 0 to 255). These can be useful in pysimCoder for adding variability to a signal using some human input. The implementation will be covered in the next parts of this section. [13]

The PMOD connectors are also of high interest for this thesis, as they will be the interface between the board and some DC motor. There are two of these slots, and they will in fact be connected to some DC motor driver, which will handle the actual output of the voltage on the motor. [13]

4.1.2 Existing Simulink project

Since pysimCoder and Simulink are very similar in the aspect of code generation, it is sometimes possible to translate blocks from one program to another. Thankfully, there is already a Simulink project which can generate real-time C code for controlling the peripherals of the MZ_APO educational kit. The source of these blocks can be found in a public online repository [14], and has been developed while using the `ert_linux` target for Simulink.

`Ert_linux` is a code generation target for Simulink which is specifically tailored for `preempt_rt` Linux distributions. The project offers an alternative to the existing Linux code generation targets offered by MathWorks which seem to have some shortcomings. The main advantage `ert_linux` has over the default targets is that it uses `clock_nanosleep` instead of POSIX timers, which results in better real-time performance in the `preempt_rt` Linux versions [15]. The source found in Github [16] consists mainly of the main program template and the Makefile template, both of which are analogous to the templates of pysimCoder used for generating real-time code meant to run in a Linux OS environment.

The `zynq-rt-control` project (also found in Github [14]) contains Simulink blocks and diagrams which are used for controlling the peripherals of the MZ_APO Zynq-based board. In order to use these resources to generate code however, it must use the `ert_linux` target for Simulink, since it was designed using this template. Before translating these blocks into the pysimCoder environment, it is important that the Simulink diagrams are tested and verified on actual hardware. In this case there are only two peripheral blocks used: knob input (incremental encoders), and DC motor (PMOD connectors). The rotary knobs are already embedded on the board, so that can be tested out of the box. For the testing of the other block, a DC motor and driver which connects to a PMOD connector is required. The proper connection of such a device will be presented at the end of this section.

From the `ert_linux` repository, it will be helpful to extract which cross-compiler must be used for generating code on the Zynq board. Then most of the C code needed for the peripheral control will be contained in the `zynq-rt-control` repository. By combining elements of these two projects, a similar implementation can be applied in the pysimCoder application. Therefore the work done on the projects above will be the base of my work on adding support for the MZ_APO educational kit.

4.1.3 Integration with pysimCoder

In this subsection I will be explaining my approach to translating and integrating the Simulink projects mentioned above, into the pysimCoder environment. This will be the first implementation of a Zynq-based board, therefore a new template Makefile must be created, as well as new blocks with all their supporting files.

To construct the template Makefile for a Linux target such as the MZ_APO kit, the same cross-compiler specified in the `ert_linux` project should be used. The rest of the structure of the Makefile will then be taken from the existing real-time Linux targets supported in pysimCoder (found in source path /CodeGen/templates/rt.tmf). The compiler specification will be switch from `gcc` to `arm-linux-gnueabihf-gcc`, which

ensures that the host will be able to compile binary code compatible with the Zynq based board.

Next the existing Simulink blocks must be translated into pysimCoder blocks. Fortunately, the C code functions which the two use are very similar. Although in Simulink there are many more functions, some of them are not related to the target manipulation, but are rather concerned with the simulation properties. The most relevant functions are presented below.

- mdlCheckParameters()
- mdlInitializeConditions()
- mdlStart()
- mdlOutputs()
- mdlUpdate()
- mdlTerminate()

On pysimCoder there are only four functions used [7], as presented in the list below. The first function (INIT) is related to the initialization of the block, and is analogous both to `mdlInitializeConditions()` and `mdlStart()` as seen above. The periodic task functions (OUT and STUPD) are then naturally related to `mdlOutputs()` and `mdlUpdate`. Most of the core computation of the block will be implemented in these two functions. At the end there is the termination function (END), very obviously corresponding to `mdlTerminate()`, which will handle the proper closing of the block functionality.

- initialization function (INIT)
- periodic task functions (OUT, STUPD)
- termination function (END)

Considering the relations shown above, the source code included in the `zynq-rt-control` project can now be used for translation. The code from each function in Simulink, will be copied to the corresponding function in pysimCoder. Then the main changes which will occur will be related to the usage of parameters in the code. Both the `python_block` C structure, and the `RCPblk` class have variables for these parameters, as can be seen in chapter 3. Each parameter variable consists of a different data-type, therefore whatever parameter is used in the Simulink code, will have to be converted based upon its own data-type. Parameters in pysimCoder can be expressed as integers, real numbers, pointers, and strings. The last part of the translation will simply be to copy all the file dependencies which are specified at the header of the C file.

With the C functions all translated, a new folder will be created for the MZ_APO support, located in the `pysimCoder/CodeGen/` directory. Inheriting the structure from the other targets, all the .c files will be contained in a `devices` folder, the .h files in `include`, and the compiled libraries in `lib`. All the file dependencies for this implementation can be found in the `zynq-rt-control` source [14], and don't need any modifications.

To complete the integration of this project in pysimCoder, the supporting .xblk and .py files must also be created for the new blocks. The properties of these block representations will be set based on similar blocks in pysimCoder, while the parameters themselves will be almost identical to the parameters used in Simulink.

Two new blocks will be created using this approach, based on the blocks given in the `zynq-rt-control` [14] project. One will read the input given from the rotary

knobs, and the other will control a DC motor. The results of this integration will be showcased in the following subsection.

4.1.4 Demonstration of results

To prove the functionality of both of the new blocks in one diagram, I have prepared a demonstration in pysimCoder. There will be two motors used, and one of them will be following the position of the other. The three rotary knobs embedded on the board will then be used to tune the values of the PID controller. All the hardware necessary for this demonstration is listed below, while the source code for running the example can be found in an online repository [17].

- Host computer with pysimCoder installed
- 1x MZ_APO educational kit
- 2x DC motor for MZ_APO educational kit
- 1x Ethernet Cable
- 1x DC power supply (12-24)V
- 2x Coaxial male-male cables

First the MZ_APO kit must be connected to two DC motors using the PMOD slots. Then the three devices must be connected to the power supply in a daisy-chain pattern, using the two coaxial cables. Last but not least, the Zynq board must be connected to the same network as the host computer using an Ethernet cable. This will enable communication between the board and the host. The hardware setup can be seen in the image below.

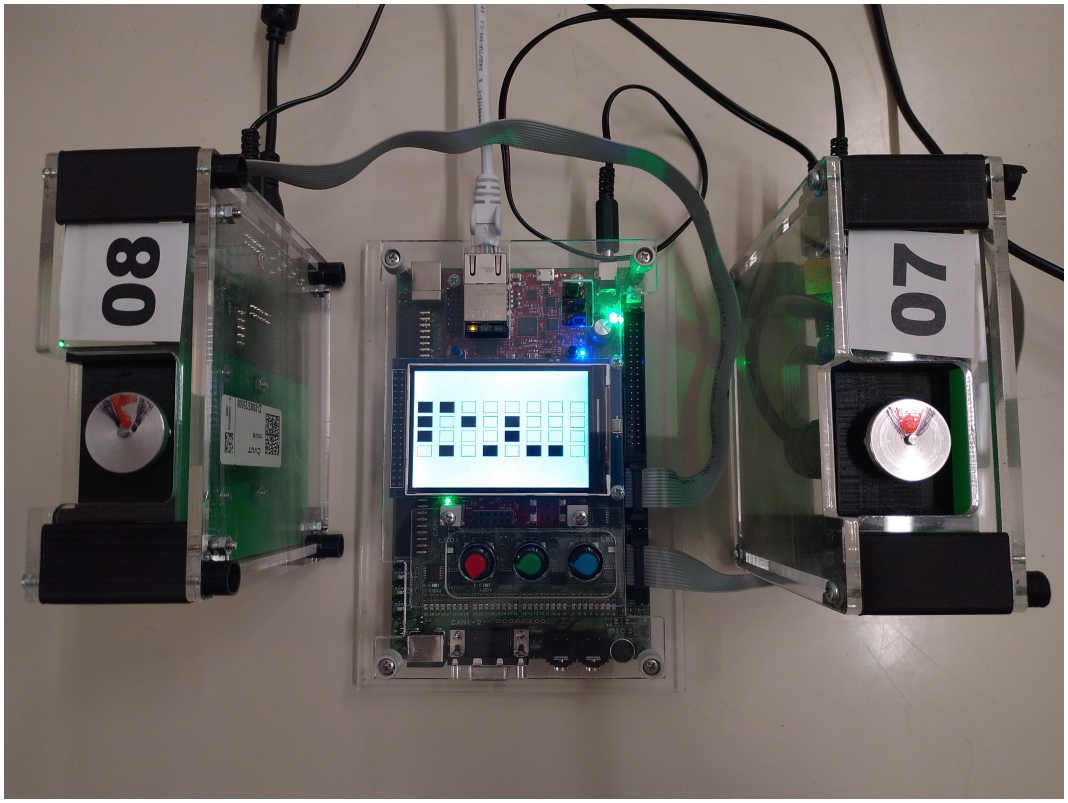

Figure 4.2. The two motors are connected to the Zynq board, while the knobs are used to tune the PID controller. Both the motors are in the same position, which is the desired output of this diagram.

With the hardware all set up, the diagram of the example can be opened (see Fig 4.2), and all the parameters can be edited as needed. There are three encoder blocks, and each one of them controls a different part of the PID controller. Although there exists a PID block in pysimCoder, the parameters can not be variably controlled by another block. Therefore each individual element of a PID controller has been deconstructed, and enabled for control by the knobs. One DC motor block is fed by a constant value, which will make it the controller motor. Another DC motor is connected on the input by the PID controller, and on the output as a negative feedback loop. Naturally it will be seen as the controlled motor. The TCP block at the right most part of the diagram will then be used to propagate these signals to a TCP listener with graphing capabilities.

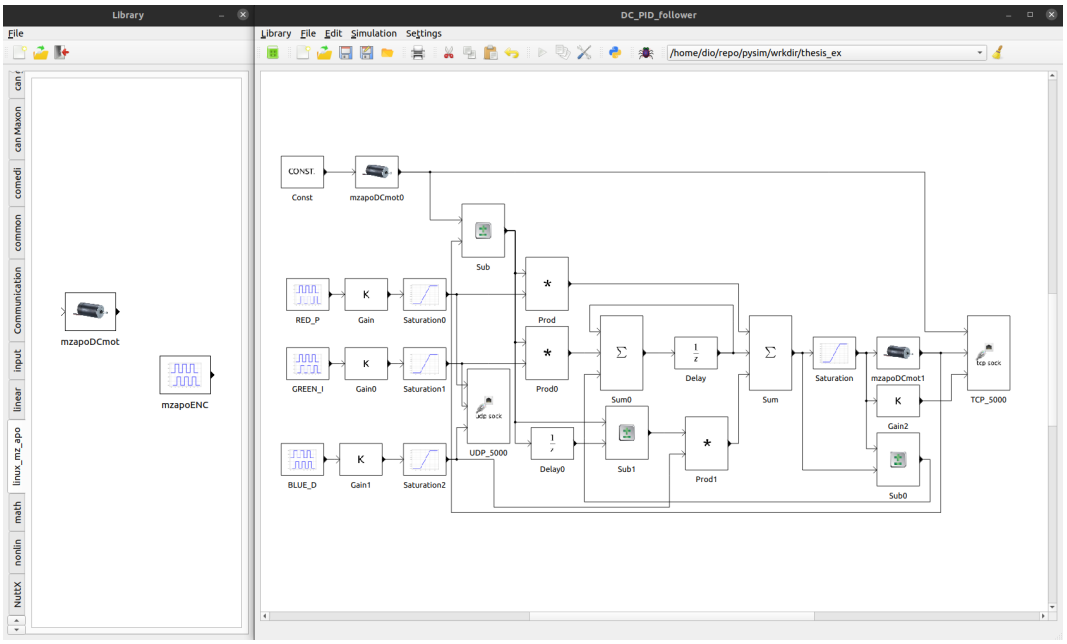

Figure 4.3. Three knob blocks control the PID values. One DC motor is fed by some constant value, while the other is connected in a negative feedback loop and controlled by a PID controller. The signals are sent through TCP for recording and graphing.

Finally the code can be generated, while the sending and execution of the binary on the Zynq board will be handled by the shell script file found in the online repository [17]. Inside this file are two commands as shown below, where the `HOST` and `TARGET` environment variables are referenced. These variables can be set through a terminal, or can be edited in the shell script itself.

```
scp DC\_PID\_follower root@\$TARGET:/tmp
ssh -t root@\$TARGET HOST=\$HOST /tmp/DC\_PID\_follower
```

Then the `RTScopy.py` program will be started, and the above script can be executed. Using the TCP block in pysimCoder, some select signals will be transmitted to the host computer, and visualized in a graph. The terminal commands for achieving this are below.

```
\$ ./RTScope.py
\$ ./DC\_PID\_follower-run.sh
```

4.2 RaspberryPi board on pysimCoder

Raspberry Pi support on pysimCoder in fact already exists, however the translation of an existing Simulink project [18] can be used as a stepping stone for demonstrating the usage of pysimCoder on an ESP32C3 board with NuttX RTOS (covered in the next chapter). Due to this fact, the steps for implementing extended support for a RaspberryPi target will be evidently shorter than the one for the MZ_APO kit.

4.2.1 Hardware introduction

In the example from the `rpi-rt-control` project (available on Github [18]), a RaspberryPi board can control a 3-phase permanent magnet synchronous motor (PMSM) using code generated by SimulinkCoder. The hardware necessary to test this example will be covered in the following text.

RaspberryPi is essentially a small computer or a microcontroller, which can interact with hardware peripherals for various types of projects. It's easy operation allows people of all ages and backgrounds to use it, making it a great asset for educational purposes [19]. The main technical specifications of the device used in this project can be found in the list below, taken from the official RaspberryPi website [20], while the hardware itself can be seen in Fig.

- A 900MHz quad-core ARM Cortex-A7 CPU
- 1GB RAM
- 100 Base Ethernet
- 4 USB ports
- 40 GPIO pins
- Full HDMI port
- Combined 3.5mm audio jack and composite video
- Camera interface (CSI)
- Display interface (DSI)
- Micro SD card slot
- VideoCore IV 3D graphics core

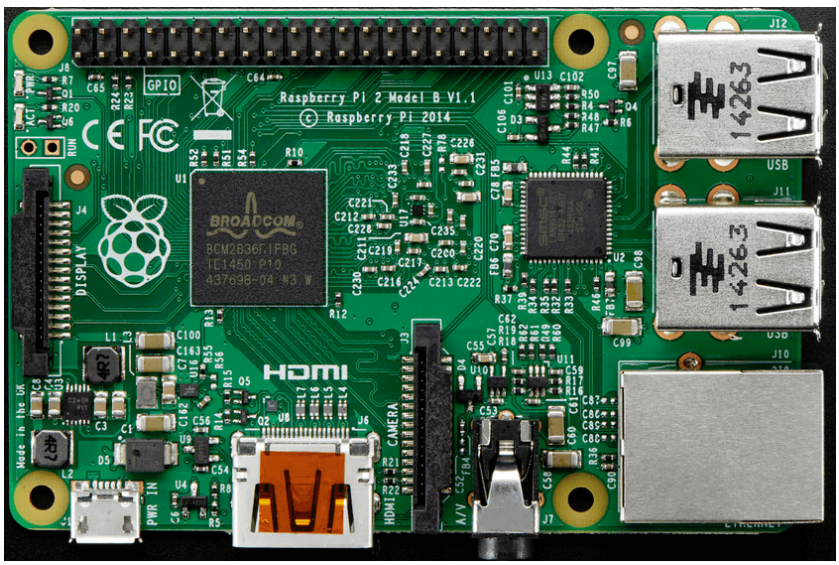

Figure 4.4. The physical appearance of the specific RaspberryPi used in the PMSM control demonstration for Simulink/pysimCoder. Produced in 2014, it is a Model 2B (v1.1).

Unfortunately a 3-phase PMSM cannot simply be plugged into this board and used out of the box. For this reason, additional hardware will be required, such as a 3-phase motor driver, and a FPGA expansion unit. Luckily, these devices have already been produced and provided by a Bachelor's thesis work done by Martin Prudek [21]. Therefore this hardware can in fact simply be connected and used, especially for the purpose of motor control (see Fig 4.5).

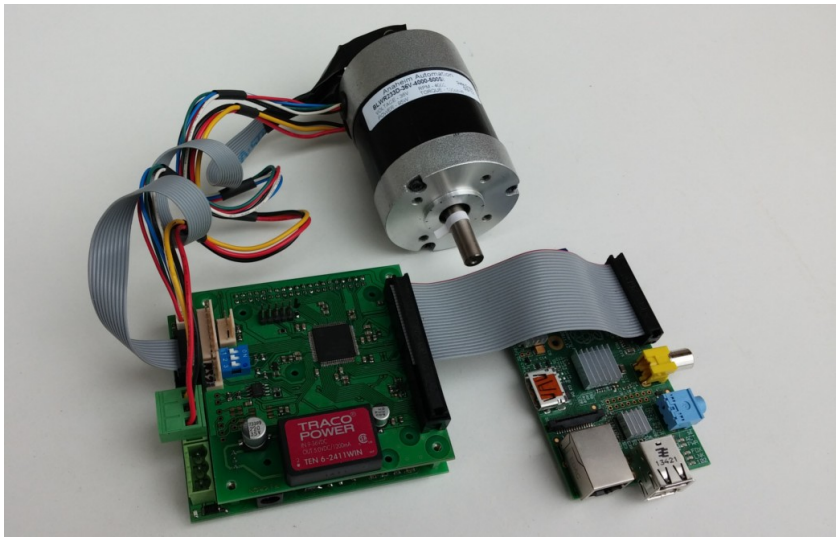

Figure 4.5. Hardware setup for motion control of 3-phase motor (PMSM) using RaspberryPi board and Simulink. The motor is connected to the motor driver, the FPGA board is fitted on top of the driver, and the RaspberryPi is connected to the expansion board with a HAT cable module. Note that the board in this picture is one model older than the one used in my implementation.

The motor driver board's main task is to connect all of the I/O cables of the motor to the expansion unit. This includes connections for the IRC, the Hall effect sensors, and the 3-phase windings of the motor. It will also serve as the power source of the motor and the boards attached to it, once it is connected to a DC power supply. Other components include an ADC, half-bridges, and current measurement signals.

Even with the current amount of hardware, the microcontroller unit still cannot be attached and used. This is due to the fact that the specific RaspberryPi board used for this example does not have any IRC input, and only contains one PWM output, making it unfit for motion control situations [15]. Fortunately, the FPGA expansion unit was specifically designed to act as a middle-man between the motor driver, and the pins of the RaspberryPi board.

Field programmable gate arrays (FPGAs) are semi-conductor devices which can (re)program configurable logic blocks in order to achieve a certain logical system design. The specific FPGA design for this example can be used in a few different modes. For example, one mode is used for motion control of a DC motor, where it converts the IRC signals, and switches PWM to be usable in both voltage directions. Then there is the mode relevant to this project, which will enable the motion control of a 3-phase motor. The fully described FPGA design can be found in Martin Prudek's Bachelor thesis [21].

To communicate with the FPGA expansion unit, the RaspberryPi board will use its SPI pins. Then the source of the FPGA will be able to send and receive signals from the MCU to the motor driver successfully. In addition to this, the expansion unit also holds a DC voltage converter which will be able to give a stable 5 Volts that the MCU

needs for operation, while the 3.3V pins of the MCU will route back to the expansion kit for operation of the FPGA circuitry. More details about this hardware setup and the PMSM Simulink example can be found online [22].

4.2.2 Existing Simulink project and translation to pysimCoder

As mentioned previously, I will use an existing project made for Simulink, in order to make a new block for PMSM control in pysimCoder. The files of interest for the following implementation can be found in the `rpi-rt-control` online repository [18]. Only the files related to the PMSM motion control will be used, which include the block C file and its dependencies, as well as the Simulink diagram used for the control system.

For the conversion of the Simulink block to a pysimCoder block, the same approach from the previous section will be utilized. The blocks of code will be extracted from each Simulink function and copied onto the corresponding pysimCoder function. Then all the parameters must be adapted to the python_block structure, and the supporting header files specified on the top of the C file. All the dependencies mentioned must also be copied onto the `devices` and `include` folders of the RaspberryPi target, in order to enable the proper compilation of the block's C code.

In this particular Simulink block, there are also vector inputs and outputs. To avoid possible issues, I have simply split all these signals into individual inputs and outputs, rather than using the experimental vector feature in pysimCoder. This decision will ensure that the cause of some error message will more likely be related to the new code which was added.

A RaspberryPi target for cross-compilation is already supported in pysimCoder, and in fact there is already a whole library of blocks to generate code for these boards. Therefore, all the block related files (.xblk, .py, .c) which were adapted for the new PMSM block can simply be added to their corresponding folders. The C file in particular should also be compiled in its folder using the `make` command in a terminal, so that the code generation tool will be able to use the new block. In other words, there is no need to create any folders, or to construct any Makefile templates.

There is also a Simulink diagram which can be used to test the functionality of the block on real hardware. The same principles of this diagram will be emulated in the pysimCoder environment, as it has all the blocks necessary to do so. The implementation will be covered in the following subsection.

4.2.3 Demonstration of results

To test whether the adaptation of the block has succeeded, a real-life example should be tested. Therefore all the hardware must initially be set up (as seen in Fig. 4.5), and a diagram in pysimCoder should be constructed to demonstrate the motion control of a 3-phase motor.

Fortunately, most of the blocks used in the Simulink diagram, can also be found in the pysimCoder library. Nevertheless the diagram will be simplified and cut down so that only the necessary functions will be used. Some parts of the diagram in Simulink have a lot to do with graphing, however for the purpose of this demonstration only the signals of higher interest will be visualized.

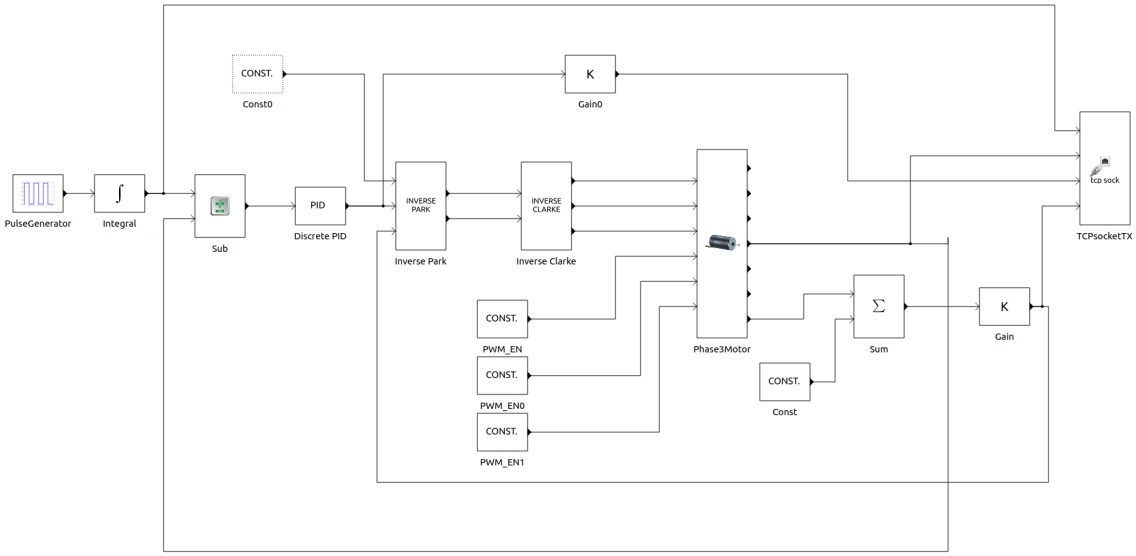

Figure 4.6. PysimCoder diagram for control of 3-phase PMSM using RaspberryPi target. Design has similarities to the original Simulink diagram, however it has been reduced only to the necessary control signals and blocks.

The new RaspberryPi block has 6 inputs, and 7 outputs. The first three inputs will be the PWM signals being fed to the motor, while the last three inputs are `enable` signals (which allow usage of a certain PWM signal). Then on the first three outputs are measurements of the current output, one for each phase. The next three are outputs related to the IRC, and the last output will be for the HAL sensor measurements.

The first three inputs of the PMSM block will be supplied by an `Inverse Clarke` math operation block, which is previously fed by an `Inverse Park` block. Into the latter, there will be one constant signal (zero for this case), one PID controlled signal, as well as a feedback signal from the processed HAL measurement. The pulse wave generator seen on the far left, will be the input for this control system, and it is subtracted with the IRC output before being fed to the PID controller. At the very right there is also a TCP block, which will transfer select signal data through a TCP/IP connection with the host computer. This connection can then be utilized by the RTScope.py program in order to graph all the incoming data points (same as done in last section).

With the diagram all set up, the C code can finally be generated. The binary will simply be transferred and executed on the RaspberryPi, considering all the hardware is connected properly. Although it is possible to interact with the board using the serial terminal, it is recommended to use an SSL connection instead. Therefore the RaspberryPi and the host must be connected to the same network (same as for the MZ_APO demonstration). On the host, the RTScope.py program must be started to enable the receiving and visualization of the incoming data. A preview of this example can be seen in Fig 4.7.

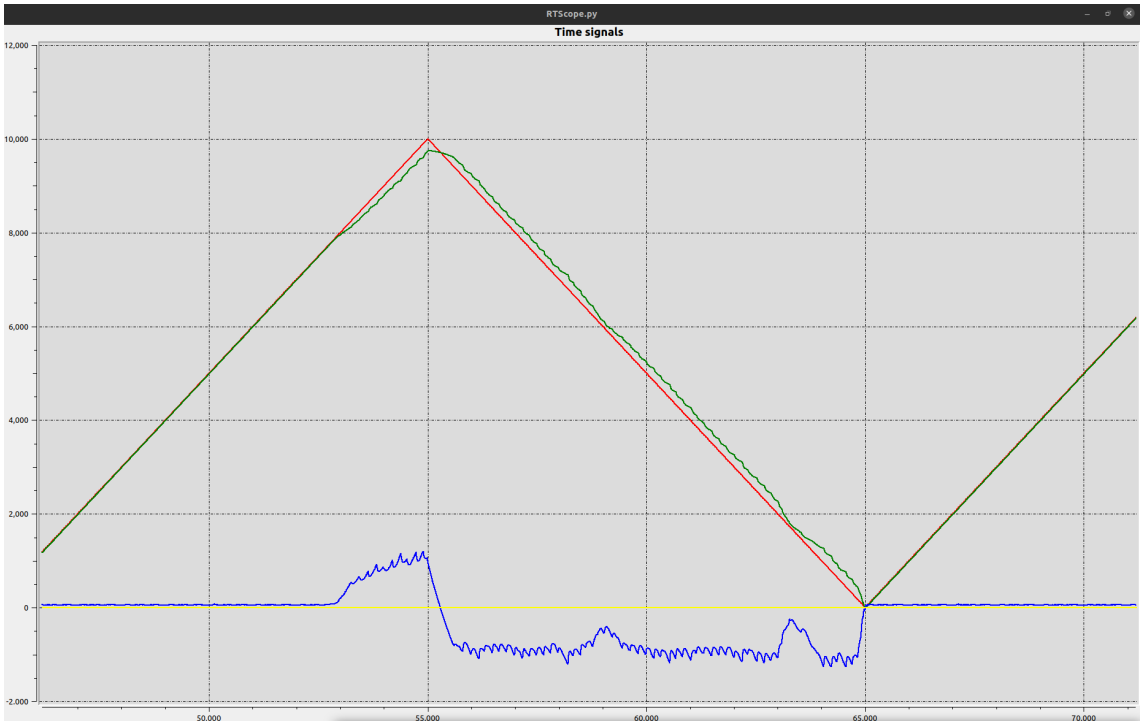

Figure 4.7. Graph showing select signals from the PMSM motion control example using the RaspberryPi. The vertical axis will be dimensionless, since the signals graphed do not have the same units or range of values for that matter. The horizontal axis represents time in seconds. This graph is a screenshot taken from the RTScopy.py program, and is only for testing purposes.

The graph above is the result of the control system applied in the RaspberryPi board for motion control of a PMSM. In red is the input signal being fed to the system, while in green is the IRC position. It is clearly seen that the two signals are in sync when there is no disturbance to the system. However, as seen from the blue signal, once a disturbance is applied the controller will try its best to compensate. The IRC output seems to be slightly shifted to the right, however it still follows the same rate of change as given from the input signal. Once the disturbance is removed, the IRC immediately synchronizes with the input again. This is in fact the desired system response for this setup, therefore it has been proved that the new PMSM block has been successfully integrated into pysimCoder.

Chapter 5
Testing and extending NuttX RTOS support in pysimCoder

5.1 NuttX RTOS

5.1.1 Introduction

NuttX is a real-time embedded operating system (RTOS) designed for high scalability, small footprint, and standards compliance. It supports architectures from 8-bit to 64-bit, and is compliant with the POSIX and ANSI standards. Although NuttX tends to be very small, it is still very rich with features. This is possible since the final build of NuttX will only import the configured features, leaving all the unused ones out. [23]

The main supported CPU architectures can be seen below, while the full list and details can be found online in the NuttX documentation [24]. RISC-V will be of high interest for the scope of this thesis, therefore its support on NuttX will be crucial to the rest of this chapter.

- ARM
- Intel
- RISC-V
- Atmel AVR
- Freescale
- Microchip
- Xtensa LX6

As can be noticed from the NuttX documentation online, there is a very long list of supported features. This raises the question of how it can be so small, and still have all these features available. Although there are a lot of files in the NuttX source code, the code contained inside is usually very short and compact (sometimes just a few lines). This in combination with linking from `static libraries` can ensure that NuttX can simply exclude the code which is not necessary for the current build, which ultimately makes the OS much smaller. [23]

Another trick to keep NuttX tiny is by using `configuration files`. In this file all the features can be enabled or disabled depending on the project requirements and limitations. Additionally, the GNU toolchain supports `weak symbols`, which help to keep the size of NuttX down. [23]

It is worth mentioning that NuttX OS is a completely free software, shared online with a non-restrictive Apache license. The source code can be found on a Github repository [25], and the true open-source nature of NuttX even allows for anyone to add their own support on the project, either locally on their computer, or uploaded to the online repository.

5.1.2 Existing support in pysimCoder

In pysimCoder, there is already existing support for NuttX targets. This means that any microcontroller running NuttX, can also execute binary code generated from the pysimCoder application. That is an amazing feature, since for most use cases pysimCoder has hardware blocks which are specific to a certain microcontroller or architecture. In the NuttX blocks, the code will be identical no matter what CPU brand is used.

NuttX has many features involving the manipulation of hardware peripherals, therefore these functions can be used as pysimCoder blocks. In fact there is already some blocks made for NuttX, such as an ADC, DAC, Digital Input/Output, Encoder, PWM, and a DHTXX sensor. Some general pysimCoder blocks can also be used, including the TCP, UDP, UART, and CAN communication protocols. [26]

Although the NuttX blocks will generally work for any NuttX supported microcontroller, there is still some specification involved in the `configuration file`. As mentioned earlier, the RTOS uses configuration files for setting up the system so that it doesn't include any redundant code in the final compilation. The configuration of NuttX should be set up not only to specify the hardware being used, but also to set certain features that will make the compilation compatible with pysimCoder [26].

NuttX will use this configuration which was set up by the user, in order to compile only the necessary files. This compilation can then be exported to the pysimCoder source, and then utilized by the NuttX blocks appropriately. So the various hardware support of NuttX can in fact be propagated to pysimCoder, however this must be configured by the user externally. The full configuration and steps can be found in the online documentation for pysimCoder with NuttX [26].

5.2 ESP32C3 board with NuttX

It would be of good interest to the pysimCoder project to somehow test a microcontroller which uses the RISC-V architecture. NuttX already has support for a board with this architecture, therefore the hardware can be easily implemented in pysimCoder. The following section will describe the simple testing and configuration of the selected hardware, both in NuttX and pysimCoder.

5.2.1 Hardware introduction

One of the RISC-V boards which is supported in NuttX is the ESP32C3 produced by the Espressif company. Fortunately, the Espressif representation in Brno (Czech Republic) was kind enough to provide my supervisor with some of these boards for the purpose of student research.

The specific hardware provided is the `ESP32-C3-DevKitM-1` (see Fig. 5.1), which is an entry-level development board popular for its small size and vast IoT capabilities. It is based on the `ESP32-C3-MINI-1`, a module which contains a whole system, including the CPU, the memory, the WiFi and Bluetooth antenna, and many more supported peripherals. The specifications of the hardware can be seen below, while the full details can be found online. [27] [28]

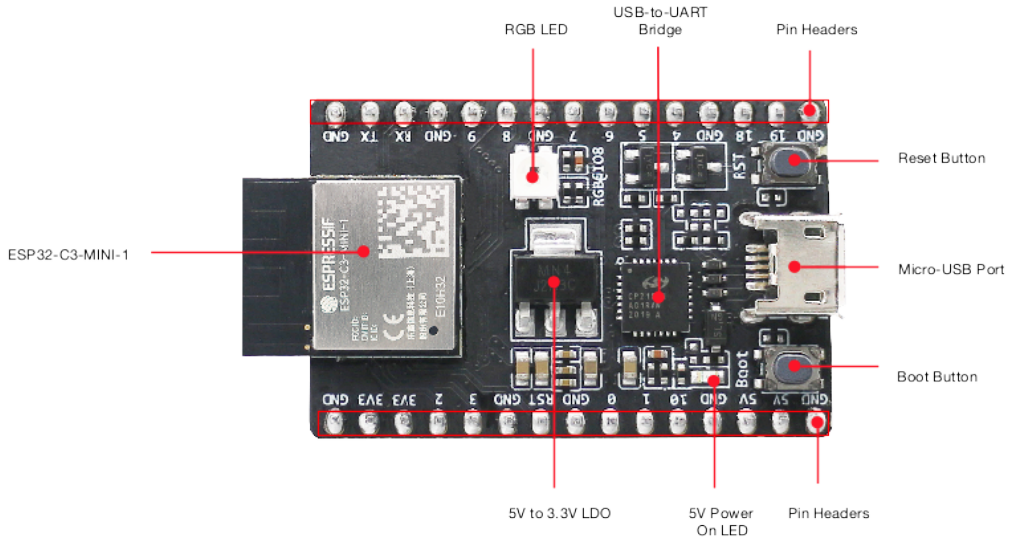

Figure 5.1. The different components of the ESP32C3-DevKitM-1 board are presented with labels. On the left is the main chip, while on the right is the micro-USB connection. The pin headers can be found on the top and bottom. Other components include RGB LED, Reset and Boot buttons, 5V to 3.3V LDO and more.

- ESP32-C3FH4 or ESP32-C3FN4 embedded, 32-bit RISC-V single-core processor, up to 160 MHz
- 384 KB ROM
- 400 KB SRAM (16 KB for cache)
- 8 KB SRAM in RTC
- 4 MB embedded flash
- WiFi - IEEE 802.11 b/g/n-compliant, 2412 2484 MHz
- Bluetooth® - Bluetooth LE: Bluetooth 5, Bluetooth mesh, up to 2 Mbps
- 40 MHz crystal oscillator
- GPIO, SPI, UART, I2C, I2S
- Remote control peripheral
- LED PWM controller
- General DMA controller
- TWAI® controller (compatible with ISO 11898-1)
- USB Serial/JTAG controller
- temperature sensor
- SAR ADC

Considering the very small size of the board, its extensive support of various peripherals is impressive. This makes the board very useful in different types of applications, such as for home automation, industrial automation, smart agriculture, consumer electronics, health care etc. I hope that by implementing this board in pysimCoder, it will become apparent that my project will fall in the category of educational automation. This is one field that was not yet mentioned as an example in the datasheet of the device. [28]

5.2.2 Installing NuttX RTOS on board

Before trying the ESP32C3 on pysimCoder, it will be necessary to load and test the NuttX RTOS on the board. To do this, the project must first be cloned from the existing Git repository [25] using the following command:

```
git clone git@github.com:apache/incubator-nuttx.git NuttX
```

Considering there is already support for the ESP32C3 on NuttX, some examples can be directly loaded. For each board on NuttX there is a corresponding directory in the source, which is full of ready-made configuration files for that specific hardware. These setup files can be loaded using a shell script, and can even be modified after based on the desired features.

There are a few different ways to approach the configuration of NuttX. The best way to initialize the setup is to run a ready made script who's purpose is to load a pre-made configuration file (`defconfig`). Once that is loaded, there are two graphical applications which can help to edit the initially loaded configuration, in order to add or remove features based on the project needs. The first graphical tool can be called by `make menuconfig` command, which will bring up an interactive menu in the terminal. Here the features are presented in a more user-friendly way. There is also another tool called by `make qconfig` command, however this will actually load every single feature possible in the NuttX compilation, therefore it is mostly used for searching through the variables and setting them manually. Once finished editing, the configuration can actually be saved into a `defconfig` file, which can then be loaded by the aforementioned shell script as an initial setup.

Another possibility can be to find all the configuration variables which are needed, and their properties, and write them into a `defconfig` file manually. Then theoretically this file can in fact be loaded by the shell script, although it is not recommended by NuttX to do so. A more clean way will always be the method discussed in the last paragraph. Therefore, that method will be used to initially test NuttX on the RISC-V board, as seen in the following steps given.

```
./tools/configure.sh esp32c3-devkit:nsh
make menuconfig
make
```

For the purpose of initial simplicity, the example which has been chosen to be pre-loaded is a NuttX application called `Nuttshell`. This is a terminal console for NuttX devices, which can be accessed usually over a serial connection (UART or USB). If this tiny shell console can be opened in the host computer, then it proves that the NuttX RTOS has been successfully loaded to the board. Before this is verified, the graphical tools can be used to add more features to the compilation (as seen in second command above). If the desired configuration is ready and saved, the `make` command can be executed in order to perform the compilation of the whole NuttX RTOS in one binary file. In the case that the configuration needs to be saved for future use, it can be compiled into a defconfig file using the following command.

```
make savedefconfig
```

It seems that there were no issues while loading the RTOS in the ESP32C3 board. The command line terminal did in fact succeed to connect to a serial connection on the host computer, and even some other examples were added, although not tested. Now that the installation of NuttX has been verified and described, the actual hardware peripherals will be tested in pysimCoder. The integration to pysimCoder will be covered in the remaining parts of this chapter.

5.2.3 Enabling and testing pysimCoder on board

As mentioned beforehand, there is already a full documentation on how to integrate NuttX with pysimCoder [26]. Using that guide, the existing configuration set up for ESP32C3 (in previous subsection), can be modified so that it can also deploy pysimCoder code. In other words the two configurations will somehow be merged, so that they are compatible with each other.

The current ESP32C3 configuration should be opened, preferably with `qconfig` tool, and each `CONFIG` variable seen in the online documentation should be set manually. Once the changes are saved, the source should be compiled, and ultimately exported to pysimCoder in the `/CodeGen/nuttx` directory. When the `CodeGen/nuttx/devices` folder is recompiled, pysimCoder will be ready to generate code for the ESP32C3. [26]

With the new configuration, the ESP32C3 board will be able to execute some common blocks of pysimCoder, mostly of those with mathematical functions. However, to be able to test the NuttX blocks themselves, which are mostly hardware related, some additional features must be enabled in the configuration which are specific to the board. To find which variables should be set for this, the purpose of the block should be reviewed. For example for the ADC block, the term `ADC` should be searched in the `qconfig` tool. If the variable is found and it is specific to the board, then it should be set to ensure the proper functioning of the block. This example should be followed for all the NuttX blocks which are used in the system. Some of these variables can be seen in the code below, taken from the `defconfig` file. Certain features below are not set for an entire NuttX block, however they may be related to the proper functioning of a block.

```
...
CONFIG_ESP32C3_ADC1=y
CONFIG_ESP32C3_ADC1_CHANNEL0=y
CONFIG_ESP32C3_GPIO_IRQ=y
CONFIG_ESP32C3_LEDC=y
CONFIG_ESP32C3_LEDC_TIM0=y
CONFIG_ESP32C3_SPI2=y
CONFIG_ESP32C3_SPI2_CLKPIN=2
CONFIG_ESP32C3_SPI2_MISOPIN=6
CONFIG_ESP32C3_TWAI0=y
CONFIG_ESP32C3_TWAI0_CLKOUT=y
CONFIG_ESP32C3_TWAI0_RXPIN=5
CONFIG_ESP32C3_TWAI0_TXPIN=4
CONFIG_ESP32C3_UART1=y
...
```

Considering that all of the configuration has been finally set, and there are no changes to be made in the NuttX environment anymore, a pysimCoder-generated application can be tested on the ESP32C2 board. During my testing of these blocks I encountered barely any issues at all, since the NuttX blocks are general and the only debugging can be done in the configuration file of the RTOS. By verification of these fairly simple blocks, it is possible to move on to a larger demonstration using this RISC-V board. The details of this demo will be covered in the next (and last) section of this chapter.

5.3 Demonstration of control using ESP32C3 with NuttX and pysimCoder

One of the main goals of this thesis is to demonstrate the usage of pysimCoder and NuttX by controlling some peripheral. In the previous chapter, an example of a RaspberryPi controlling a 3-phase PMSM was demonstrated. In this section it will be described how that board can be switched for an ESP32C3, and still manage to control the 3-phase motor, which can be achieved using NuttX and pysimCoder together.

5.3.1 Hardware requirements

The general idea of this demonstration is not too complicated. Motion control of a 3-phase PMSM has already been proven to be possible using pysimCoder with the RaspberryPi board. Fortunately, the ESP32C3 board already has SPI support, therefore it would be able to communicate with the FPGA expansion unit. It also has a clock generator with a frequency of 40 MHz, which can be fed into the FPGA board. There is even 3.3V outputs from the ESP32C3, which will be able to power the FPGA chip. All of this means that the RaspberryPi can in fact be switched with this RISC-V board, at least in terms of hardware connections.

In the RaspberryPi version of this demonstration, the 40 external pins of the board fit directly into the 40-pin connector of the FPGA board, even though many pins were unused. This was designed in such a way, so that the implementation of the hardware will look more clean. In the case of using an ESP32C3 board, instead these pins will have to be routed using jumper cables. In addition to that, this version will also use the CAN protocol in order to transmit signals from pysimCoder, therefore some additional hardware will be used. The full list of required hardware for this demonstration is listed below.

- ESP32C3-DevKitM-1
- FPGA expansion unit
- 3-Phase Motor Driver
- 3-Phase Permanent Magnet Synchronous Motor
- UART-to-USB cable
- USB2CAN converter
- WCMCU-230
- Jumper cables
- Power supply (12-24)VDC

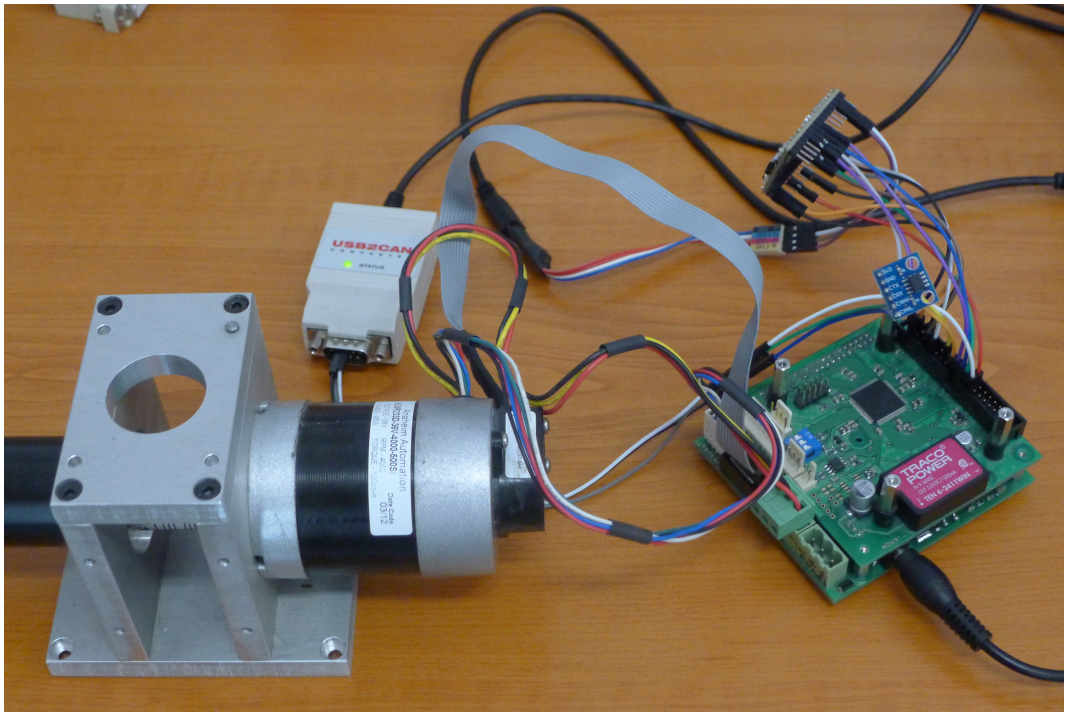

Figure 5.2. Hardware setup for 3-phase PMSM motion control using ESP32C3 with Nuttx RTOS. Code for controlling the motor was generated by pysimCoder. Hardware connections are based from same demonstration for RaspberryPi.

When doing the same demonstration for the RaspberryPi, the microcontroller could simply be placed on top of the FPGA expansion board (see Fig. 4.5), since both of them have 40-pin connectors. In the case of the ESP32C3, there are only 30 pins and they are designed differently from the FPGA connectors. As seen in figure 5.2, the connection between the two devices will be made using jumper cables instead. Only the necessary pins of the expansion board will be connected to the corresponding pins on the microcontroller, using the mapping (available online [29] which is specific to the FPGA mode that controls a 3-phase motor.

Even though the ESP32C3 contains all the pins necessary to connect to the FPGA, some communication with the host computer still needs to be established. This will help the user to be able to read, and hopefully visualize the data coming from the device, as well as to access the NuttShell terminal which runs the main program. For accessing the NuttShell, a serial connection (UART) must be established with the host. This can be achieved easily using a UART-to-USB cable, which can connect to most computers.

For receiving data from the microcontroller, the CAN protocol has been selected. A driver for the CAN communication in this particular board has been recently implemented by another student for their thesis, therefore we have taken this opportunity to test their work on my demonstration. The additional hardware necessary for CAN communication on the ESP32C3, as well as their respective pin mappings, have been provided by this same student. This includes a WCMCU-230 CAN module, a small peripheral which enables connection to a USB2CAN converter (also included). These two devices will complete the CAN connection from the microcontroller to the host computer.

5.3.2 Translation of RaspberryPi project

From the previous subsection it is clear that the interaction between the FPGA expansion unit and the microcontroller will ultimately be the same, as long as the FPGA is not set to a different mode or reprogrammed. Therefore, the C functions which are related to this hardware should also remain the same, with minor changes specific to the operating system running on the microcontroller unit. For the current application in question, the code used in the RaspberryPi PMSM block (described in chapter 4) can be used as a base for the new NuttX implementation.

In this case, there will be changes even to the dependencies of the block source code. Considering this, a deeper understanding of the RaspberryPi code must be sought. In chapter 4, it was mentioned that the C file which was translated from a Simulink block to a pysimCoder block must specify the same header files. Additionally, it was also necessary for those files to be copied onto the pysimCoder repository so that they can be compiled together. Most of the header files are standard C libraries, however the last three are specific to the RaspberryPi (as seen in the code fragment below).

```
...
#include "rpi_gpio.h"
#include "rpi_spimc.h"
#include "rpi_gpclk.h"
...
```

For the ESP32C3 implementation, the clock generator is set up by the NuttX configuration, while the GPIOs are not needed at all. Only the SPI functionality will be taken in consideration for this adaptation, and since these are not common RaspberryPi libraries, the code must be analyzed thoroughly. In the `rpi_spimc.h` file, there is a C structure `spimc_state_t` which holds all the values related to the SPI interaction, including some related to the current, PWM, IRC, and HAL sensors. Besides that there are also some constants defined at the top, and two C functions are referenced at the bottom.

The two C functions are `spimc_init()` and `spimc_transfer`, which are located in the `rpi_spimc.c` file. Most of the time these functions are operating on the structure elements from the .h file, and then sometimes an `ioctl()` request is made, which is common for SPI communication. The header files should also be taken account of in this file, since some of them are specific to Linux. In the main C functions of the block, there are also a lot of operations on the `spimc_state_t` structure, as well as some calls to the two functions described above.

After some short analysis of the RaspberryPi demonstration, it is clear that some changes will need to be made for the following conversion to happen. First, all the relevant files should be duplicated in the NuttX folder in pysimCoder, and their names should be changed to reflect some connection to NuttX. For the most part, the main C file of the block will be identical, with the only difference being in the header files and some RaspberryPi specific commands. As previously discussed, the ESP32C3 clock can be setup in the NuttX configuration and the GPIOs are not used at all, therefore only the SPI related header file will remain in the code, and it should carry the new name that was made for it. Then any of the functions used in this file which would be referenced by the removed header files, should also be removed. The SPI header file itself can be copied without any changes, since the C structure within will be used in the same way.

The most notable changes in fact will be happening in the `rpi_spimc.c` file. Just earlier it was mentioned that this file imports some native Linux libraries. These libraries are directly related to the SPI specifications of the host operating system, therefore they can be switched for the NuttX SPI header files instead (can be found in the source). Most importantly, the `ioctl()` function parameters must be edited to reflect the host OS specifications for SPI transfers that use this function. These specifications can be found in `spi_transfer.h`, and have been used to convert the `ioctl()` request to be compatible with NuttX (seen in the code below). The first parameter remains unchanged. The second parameter is an OS-specific constant, while the last one is pointing to a C structure which contains all the information regarding the SPI transfer taking place. Both of these have been found in the `spi_transfer.h` file, and can be used as NuttX equivalents for this specific application. The result is the following `ioctl()` request.

```
ret = ioctl(spimcst->spi_fd, SPIIOC_TRANSFER, &tr);
   if (ret < 0)
      return -1;
```

These are all the changes regarding the C files which are necessary for the new NuttX block. The .xblk and .py components of the block can be copied directly from the Raspberry Pi block, only changing the names of the files and the descriptive comments. As a last step, the configuration of NuttX must also be slightly extended for the demands of this block. Some features that must be added include SPI, TWAI, and CAN related configuration variables. Then the same steps as given in the previous section should be followed in order to save this new configuration to the pysimCoder source.

5.3.3 Example of real usage

It should be known by now that a new block can only be tested when placed in a new diagram. Considering that the block is identical to the RaspberryPi block (see chapter 4), then in theory the same diagrams can be used. During prior testing on this new NuttX block on the ESP32C3, it has been found out that designs using a PID controller cannot be used. Apparently the tiny RISC-V board cannot handle the sampling frequencies necessary for proper PID control. For this reason a similar control diagram without a PID block will be used as a base. The TCP blocks must also be replaced with CAN blocks, since the communication method to the host has changed.

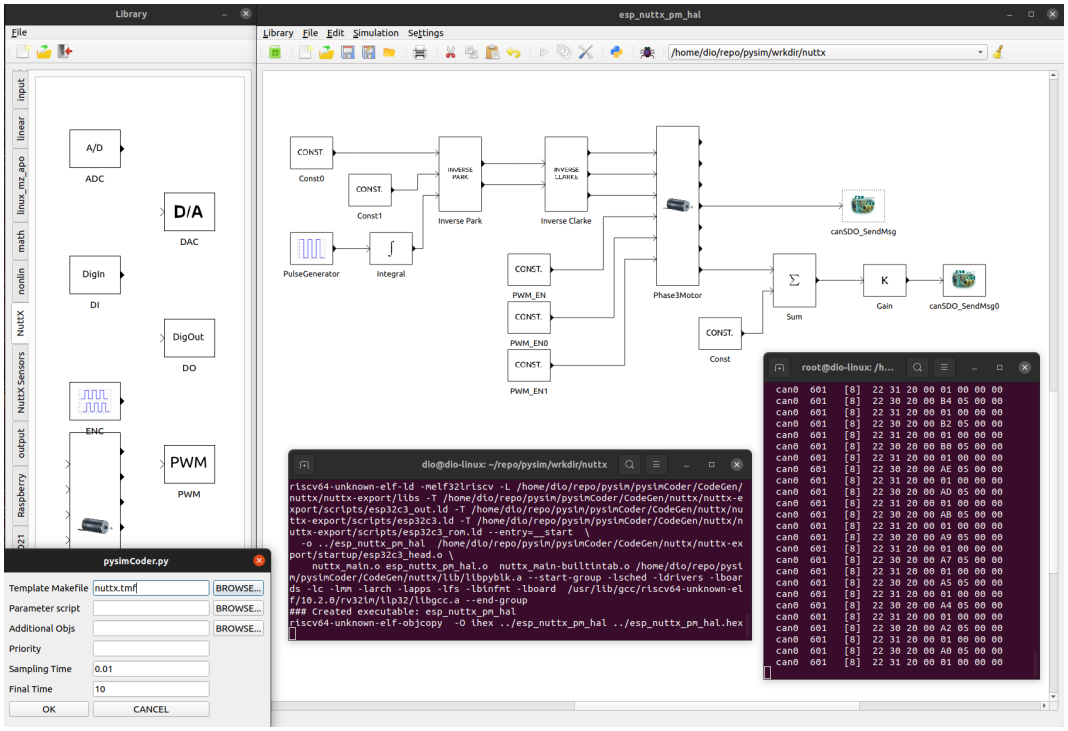

Figure 5.3. Screenshot demonstrating motion control of 3-Phase PMSM using ESP32C3 with NuttX RTOS. Terminal on right shows a CAN receiver printing in hexadecimal format, while the one in middle is for pysimCoder terminal output. Diagram is similar to RaspberryPi PMSM motion control demo, but uses CAN instead of TCP for communication.

As seen in figure 5.3, the terminal on the left is receiving data using CAN protocol, and printing it in hexadecimal format. By a basic inspection of these hexadecimal numbers, it can be noticed that the values are changing as expected from the signal it's connected to in pysimCoder. The 3-Phase motor on the other hand is also reacting as expected from the design, therefore it can be said that the demonstration was successful. Again, it must be noted that there were some issues when attempting to use a PID controller, due to a sampling frequency limitation of the ESP32C3 board. This is one issue to be explored in the future, in order to understand the full limitation of the hardware.

Chapter 6
Conclusion

Appendix A
Source Code

Appendix B
Glossary

MCU ■ Micro Controller Unit

References

[1] Dorf, Richard C.. *Modern Control Systems*. 6 ed. Menlo Park: Addison-Wesley, 1992. ISBN 0-201-60701-8.

[2] Siemens. *Totally Integrated Automation Portal*. Available from `https://new.siemens.com/global/en/products/automation/industry-software/automation-software/tia-portal.html`.

[3] CODESYS Group. *CODESYS - THE COMPREHENSIVE SOFTWARE SUITE FOR AUTOMATION TECHNOLOGY*. Available from `https://www.codesys.com/the-system.html`.

[4] Autodesk. *Integrated CAD, CAM, CAE, and PCB software.* Available from `https://www.autodesk.com/products/fusion-360/overview`.

[5] Analog Devices Inc. *LTspice*. Available from `https://www.analog.com/en/design-center/design-tools-and-calculators/ltspice-simulator.html`.

[6] The MathWorks Inc. *Simulink Coder*. Available from `https://www.mathworks.com/products/simulink-coder.html`.

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

[8] Fabo P., Gašpar G.. pySimEd - universal framework for visual programming. In: *Proceedings of the 16th International Conference on Mechatronics - Mechatronika 2014*. 2014. pp. 560-563. Available from DOI 10.1109/MECHATRONIKA.2014.7018320.

[9] Bülter, Christoph. *qtnodes*. Available from `https://github.com/cb109/qtnodes`.

[10] Shafarevich I., Remizov A.. *Linear Algebra and Geometry*. 1 ed. Berlin: Springer, 2013. ISBN 978-3-642-30993-9.

[11] The MathWorks Inc. *Mux Block*. Available from `https://www.mathworks.com/help/simulink/slref/mux.html`.

[12] The MathWorks Inc. *Demux Block*. Available from `https://www.mathworks.com/help/simulink/slref/demux.html`.

[13] Píša, Pavel. *MicroZed APO*. Available from `https://cw.fel.cvut.cz/b202/_media/courses/b35apo/en/semestral/mz_apo-datasheet-en.pdf`.

[14] Cerny, Lukas. *Simulink files for Zynq RT control*. Available from `https://github.com/aa4cc/zynq-rt-control`.

[15] Sojka M., Píša P.. *Usable Simulink Embedded Coder Target for Linux*. Available from `https://rtime.felk.cvut.cz/publications/public/ert_linux.pdf`.

[16] Sojka M., Gurtner M., Píša P., and Hamacek L.. *Minimalist Simulink Coder Target for Linux*. Available from `https://github.com/aa4cc/ert_linux`.

[17] Beqiri, Dion. *DC Motor follower with PID for MZ_APO kit*. Available from `https://github.com/robertobucher/pysimCoder-examples/tree/main/Linux-mzapo/DCmotor`.

[18] Píša, Pavel. *Raspberry Pi Real-Time Control Support and Experiments*. Available from `https://github.com/ppisa/rpi-rt-control`.

[19] Raspberry Pi Ltd. *What is a Raspberry Pi?* Available from `https://www.raspberrypi.org/help/what-%20is-a-raspberry-pi/`.

[20] Raspberry Pi Ltd. *Raspberry Pi 2 Model B*. Available from `https://www.raspberrypi.com/products/raspberry-pi-2-model-b/`.

[21] Prudek, Martin. *Brushless motor control with Raspberry Pi board and Linux*. Available from `http://hdl.handle.net/10467/62036`.

[22] Píša, Pavel. *GNU/Linux and FPGA in Real-time Control Applications*. Available from `https://installfest.cz/if17/slides/so_t2_pisa_realtime.pdf`.

[23] The Apache Software Foundation. *About Apache NuttX*. Available from `https://nuttx.apache.org/docs/latest/introduction/about.html`.

[24] The Apache Software Foundation. *NuttX Supported Platforms*. Available from `https://nuttx.apache.org/docs/latest/introduction/supported_platforms.html`.

[25] The Apache Software Foundation. *NuttX Source Code on Github*. Available from `https://github.com/apache/incubator-nuttx`.

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

[27] Espressif Systems Co., Ltd. *ESP32C3-DevKitM-1*. Available from `https://docs.espressif.com/projects/esp-idf/en/latest/esp32c3/hw-reference/esp32c3/user-guide-devkitm-1.html`.

[28] Espressif Systems Co., Ltd. *ESP32-C3-MINI-1 Datasheet*. Available from `https://www.espressif.com/sites/default/files/documentation/esp32-c3-mini-1_datasheet_en.pdf`.

[29] Píša, Pavel. *RPi and rpi-mc-1 Board Pins*. Available from `https://gitlab.com/pikron/projects/rpi/rpi-mc-1/-/blob/master/doc/rpi_gpio_use.md`.