

MASTER

Automatic target-specific code-generation from Simulink to composable multi-core platforms

Xiao, Liyin

Award date:
2021

[Link to publication](#)

Disclaimer

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Automatic target-specific code-generation from Simulink to composable multi-core platforms

Master Thesis Preparation Report

Liyin Xiao

Supervisors:
Dr. Dip Goswami
Mojtaba Haghi

1.0 version

Eindhoven, August 2021

Abstract

This thesis provides the implementation of feedback control applications which are designed in Simulink model-based environment. The targeted embedded platform for implementation is CompSOC, a predictable and composable system-on-chip. Simulink provides an Embedded coder which can automatically generate the code of a model-based simulation. Then a part or the whole of the simulation is uploaded and executed on CompSOC platform. The target-specific simulations are either processor-in-the-loop or hardware-in-the-loop simulations.

The purpose of this study is to build an automatic code generation tool targeting the predictable and composable multi-core platforms. To this aim, the generated code should be compiled to an executable which has real-time temporal behavior. It also should be able to be divided to separate executables where each part is executed on a separate core of the platform. PIL is a simulation method which compiles the generated code from the feedback control model and then upload and run the code on the embedded platform. HIL or external mode simulation is a method which is used in the development and testing of control systems with complex operations. To implement PIL and external mode simulations on the embedded platform successfully, A framework in Simulink environment is proposed to enable automatic code-generation, compile and execution on the targeted embedded platform.

Preface

I would like to extend my deepest gratitude to my head supervisor Dip Goswami for his patience and thorough feedback on my thesis. Without his consistent and illuminating instruction, this thesis could not have reached its present form. I shall extend my thanks to my project tutor Mojtaba Haghi for all his kindness and help. He is a patient person who led me into the world of research. I would also like to thank my parents. Without their constant encouragement this work may never have been finished.

Contents

Contents	v
List of Figures	vii
1 Introduction	1
1.1 Composable Multi-core Platform	2
1.2 Problem Definition	2
2 The Composable multi-core platform	4
2.1 Virtual processors	4
2.2 TDM schedule policy on CompSOC	4
3 PIL and external mode simulations	6
3.1 Development environment	6
3.1.1 Hardware	6
3.1.2 Software	7
3.2 Control application	7
3.2.1 State-Space Model	7
3.2.2 Considered cases	9
3.2.3 Discrete State-Space Model	9
3.3 PIL Simulations	9
3.4 HIL Simulations	9
3.5 Simulink model and code generation	10
3.5.1 Simulink Model	10
3.5.2 Code Generation	10
3.5.3 Code generation with Real-Time Workshop	11
4 Single-Core simulations implementation	15
4.1 Control Design and Implementation	15
4.2 As-fast-as-possible Scheduling	16
4.3 Real-time scheduling	17
5 Automation for multi-core implementation	19
5.1 The multi-core scheduling	19
5.2 The shared memory block	19
5.2.1 To the shared memory	20
5.2.2 From the shared memory	21
5.3 Choose specific tile and partition slot	22
5.4 The hexFile	23
5.5 CaseI: model-based controller and hard-coded plant	24
5.6 CaseII: model-based controller and code generated plant	25

6	Experimental Results	29
6.1	Single core implementation	29
6.2	Automated multi-core integration	29
6.3	Results and Analysis	30
6.3.1	Conclusion	34
7	Conclusions and future plan	35
7.1	Conclusion	35
7.2	Future plan	35
	Bibliography	38

List of Figures

1.1	V-steps of Model	1
1.2	The composition of CompSoC platform	2
2.1	High-level overview of a possible composition of CompSOC	4
2.2	The TDM schedule policy	5
3.1	PYNQ board overview	6
3.2	The motion system	7
3.3	The feedback control model in Simulink environment	10
3.4	Code generation process	10
3.5	Code generation from Real-Time Workshop document	11
3.6	Configuration Parameters	11
3.7	Solver diagram	12
3.8	Interface diagram	12
3.9	Templates diagram	13
3.10	The File customization template	13
3.11	Code Generation Report	14
4.1	2-DOF controller configuration	15
4.2	As-fast-as-possible Scheduling	16
4.3	The naive composition of rt_OneStep	16
4.4	The specific TLC code and its corresponding generated embedded code for rt_OneStep function	17
4.5	The composition of new rt_OneStep	17
4.6	The specific TLC code and its corresponding generated embedded code for realtime	18
4.7	TDM schedule	18
5.1	The multi-core scheduling	19
5.2	The third level of automation for multi-core code generation.	20
5.3	legacy write	20
5.4	ToSharedMemory	21
5.5	ToSharedmemory block	21
5.6	legacy read	21
5.7	FromSharedMemory	22
5.8	FromSharedmemory block	22
5.9	system target file	23
5.10	Configuration Parameters dialog box	23
5.11	runAvrDude(hexFile)	24
5.12	Generate hexFile	24
5.13	Control task	24
5.14	Plant simulation	25
5.15	Hardware architecture	25

LIST OF FIGURES

5.16	The control model	26
5.17	The plant model	26
5.18	Processor tile and virtual platform	27
5.19	External mode configuration1	27
5.20	External mode configuration2	28
6.1	The naive implementation of feedback-feedforward application	29
6.2	Pulse signal input	30
6.3	Output $x[1]$ with the single core implementation	30
6.4	Output $x[2]$ with the single core implementation	31
6.5	Output $x[3]$ with the single core implementation	31
6.6	Output $x[4]$ with the single core implementation	32
6.7	Output $x[1]$ with the multi-core and hard-coded plant implementation	32
6.8	Output $x[2]$ with the multi-core and hard-coded plant implementation	33
6.9	Output $x[3]$ with the multi-core and hard-coded plant implementation	33
6.10	Output $x[4]$ with the multi-core and hard-coded plant implementation	34
7.1	The final level of automation for multi-core code generation	36
7.2	The model is partitioned to execute concurrently	36
7.3	Multicore target architecture	37

Chapter 1

Introduction

Nowadays, the implementation of digital control applications on embedded platform became a hot research topic since embedded implementation causes several uncertainty in stability and performance. Embedded implementation is widely used in many industrial topics such as aircraft autopilots, mass-transit vehicles, oil refineries, paper-making machines, and countless electromechanical servomechanisms[1].

The process of design and implementation of a digital controller consists of specific steps. Figure 1.1 demonstrates a V-step model of these steps. The control design usually starts from model-in-the-loop (MIL) simulations in model-based simulation environments. In this step the controller is designed in a way to verify control performance requirements. The next step is called processor-in-the-loop (PIL) which has a non real-time nature. In such simulation, the designed controller (and possibly the model of the system under study) is executed on the embedded platform. This simulation enables the designer to verify the functional correctness of control code while executed on the platform. This step consists of platform-specific code generation for the control application, uploading and execution of the code on the platform, and the measurement of the execution times of different tasks. The final step of the implementation is called the hardware-in-the-loop (HIL). In this step, a real-time simulation verifies the temporal behavior of the controller on the platform in aspects such as real-time execution, periodicity, interruptions with other applications on the platforms and so on.

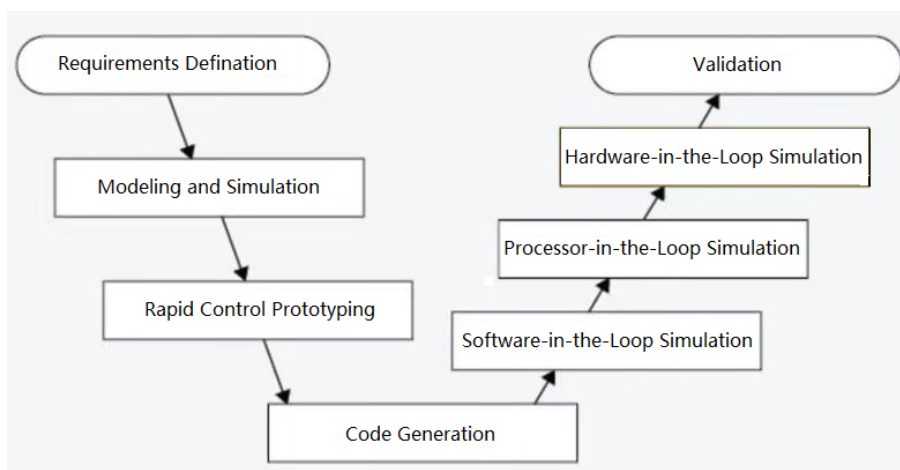


Figure 1.1: V-steps of Model

Embedded implementation of feedback control applications presents several challenges in terms of stability and performance [2]. In the controller design phase, a common assumption is that control tasks/software execute periodically, sequentially and without jitter. On widely used platforms

such as Raspberry Pi, dSPACE and arduino, strictly periodic, jitter-free execution is difficult to achieve due to interference from system tasks with other applications that share platform resources. The tailored embedded platforms for real-time applications are interesting targets for control applications [5]. These platforms offer properties such as determinism in execution times and composability in multi-application scenarios (which guarantees interference free execution of applications). These properties guarantee periodic and jitter-free execution of the control applications. In this thesis we focus on developing a implementation framework for such platforms and specifically on a certain platform called CompSOC. Before we continue further, we first describe the platform.

1.1 Composable Multi-core Platform

The embedded platform used in this project is called CompSOC[6] which is designed for executing multiple embedded applications. By using Time Division Multiplexing arbitration (TDM), this platform meets the requirements of predictability and composability, which guarantee non-interference execution for feedback control applications[5]. For this reason, CompSOC platform is chosen for the control application.

The Figure 1.2 shows the composition of CompSOC platform[5] which is a tile-based architecture. Each tile contain its own processor with unique data memory (DMEM) and instruction memory (IMEM). As a result, the two processor tiles can access to each other using Direct Memory Access (DMA) through its communication memory (CMEM).

TDM scheduling policy on CompSOC guarantees an isolated and non-interference implementation for each application since this platform uses a predictable and composable micro-kernel (CoMiK). This kernel creates multiple virtual processors (VPs) and each VP uses part of hardware resources. Under TDM scheduling manner, the utilization of these VPs will not affect each other. Therefore, the platform is composable and achieves real-time performance. Figure 1.2 shows the TDM scheduling with three partition slots on the first tile.

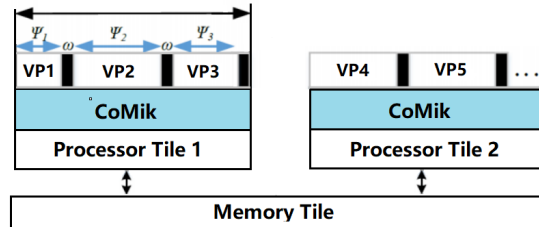


Figure 1.2: The composition of CompSoC platform

1.2 Problem Definition

In this project, we present an HIL framework for model-based simulations targeting composable multi-core platforms. The framework is an add-on code-generation tool to Simulink [6] environment. The framework enables the designer to start from a model-based simulation environment (Simulink) to complete all the steps of the V-model (which are MIL, PIL, and HIL) within the same environment.

The framework is able to generate the target-specific code for both PIL and HIL simulations, build an executable out of the generated code and upload it on the platform. Since the platform is multi-core and is able to run a number of applications simultaneously, the framework enables to choose the specific core and its scheduling, on which the simulation will execute.

The goals of this project is to develop an HIL framework which has the following features:

- It automatically generates the target-specific HIL code for FPGA-based embedded platform CompSOC from any Simulink models (www.mathworks.com).
- It automatically uploads and executes HIL codes on the platform either with a real-time or *as-fast-as-possible* timing fashion.
- It allows for online data logging and parameter tuning while the simulation is running on the platform.
- It Enables the designer to specify the scheduling of the targeted core on the platform.
- It allows the user to divide a single simulation to different tasks and execute them on different cores of the platform.

The rest of this report is organized as the following: Chapter 2 discusses the composable multi-core platform, Chapter 3 describes the PIL and external mode simulations . Chapter 4 indicates single-core simulations implementation. Chapter 5 describes automation for multi-core implementation. Chapter 6 gives the Experimental Results. Chapter 7 imagines the future plan.

Chapter 2

The Composable multi-core platform

The embedded platform considered in this project is CompSOC which has a tile-based architecture. Figure 2.1 illustrates a possible composition of CompSOC, which normally consists of processor tiles and monitor tiles. The processor tile is mainly made up of a MicroBlaze soft-core processor which plays a role in processing.

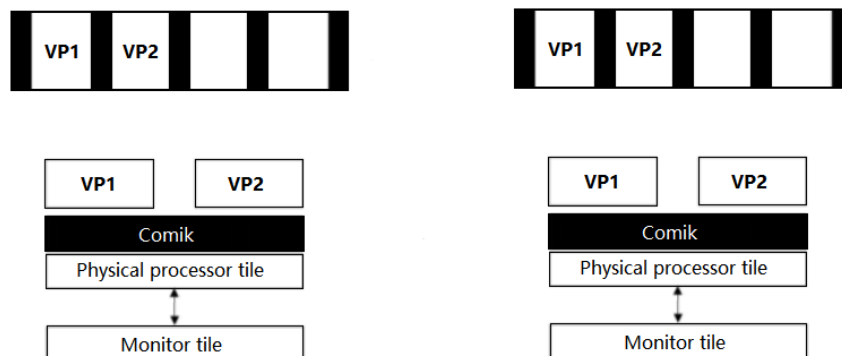


Figure 2.1: High-level overview of a possible composition of CompSOC

2.1 Virtual processors

As illustrated in Figure 2.1, processor tiles consist of a physical Microblaze processor that has an instruction memory (imem) and a data memory (dmem). These memory are tightly coupled. Since real-time applications can share the processor resources with other applications, a composable and predictable micro-kernel is introduced to create multiple virtual processors (VPs) which can be considered as processing resources [9]. CoMik uses TDM schedule policy to divide the processor into TDM partition slots. Each VP takes up part of the processing resource available on the physical processor that is allocated in a TDM schedule policy.

2.2 TDM schedule policy on CompSOC

A periodic time-division-multiplexing (TDM) policy is applied on all processors aiming at achieving real-time performance with cycle accurate time division. This schedule policy help to maintain the

properties of predictability and composability on this platform. Because each VP utilizes part of the processing resource, they will not affect each other using TDM schedule policy. Therefore, the composability on this platform is guaranteed. The application which executed on the platform will be allocated to a single partition slot and waiting for starting a new CoMiK slot. Predictability is also a vital property which is needed to be able to guarantee that worst-case performance and deterministic execution times are met for real-time applications.

The TDM frame include two kinds of slots, the partition slots(φ) and the CoMik slots(ω). Figure 2.2 illustrates a possible schedule of one TDM period with four CoMik slots and four partition slots which could be different.

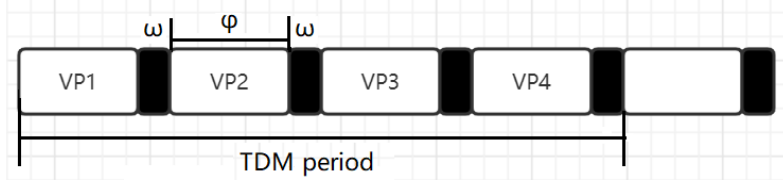


Figure 2.2: The TDM schedule policy

In this instance, the TDM period could be calculated as below:

$$P_{TDM} = \sum_{i=1}^N \varphi_i + N \times \omega.$$

where N represents the number of slots. φ_i and ω represent the clock cycle of different partition slots and CoMik slots. TDM scheduling is achieved by a periodic interrupt that indicates a context swap between two different virtual processors. Then CoMik's interrupt routine is considered to execute the context swap, which preserve the previous VP's related context and arrange the next VP[5].

Chapter 3

PIL and external mode simulations

This chapter explores the inner workings of the MATLAB-CompSOC integration tool and Simulink code generation. This includes an introduction to how this tool handles control tasks, and how Simulink generates code from its model diagrams.

3.1 Development environment

3.1.1 Hardware

The FPGA board chose in this project is PYNQ-Z2 which is cheap, small, and requires minimal power consumption. This board is based on Xilinx Zynq System on Chip (SoC) and designed to support an open-source framework PYNQ (Python Productivity for Zynq). The board uses the 650MHz dual-core Cortex-A9 processor by ARM which, importantly, includes 1G Ethernet and USB 2.0 High-bandwidth peripheral controllers that can be connected with PC through Ethernet or USB[7]. Figure Figure 3.1 shows the composition of this board. This allows generated codes to run on the hardware platform.

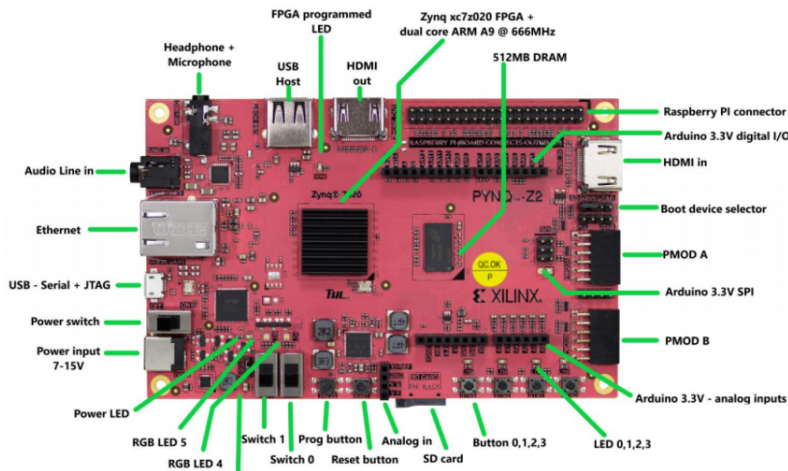


Figure 3.1: PYNQ board overview

3.1.2 Software

The system is modeled in Matlab R2018b, one of the most common tools for mathematical and technical calculations.

3.2 Control application

3.2.1 State-Space Model

The control application considered in this project is a dual rotary fourth-order single input multiple output (SIMO) motion system as depicted in Figure 1.2. It consists of two masses which are connected by a spring. In addition, there is a motor as the input to drive the first mass. The attached sensors are used to measure the angular position of both masses.

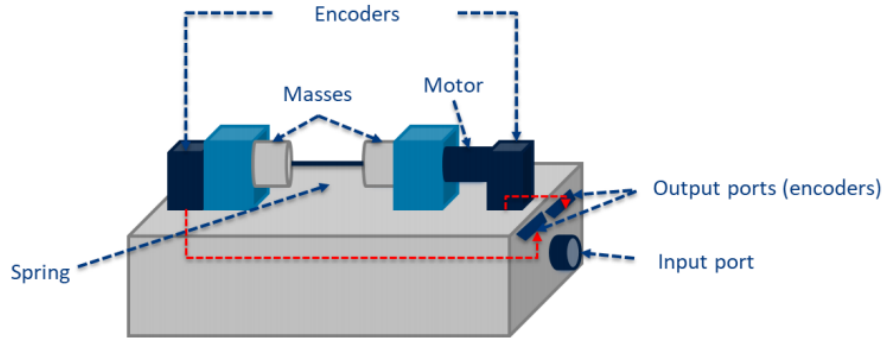


Figure 3.2: The motion system

Based on the description above, the equation of corresponding state-space representation is given below[7]:

$$\dot{X}(t) = AX(t) + BU(t), \quad (1)$$

$$Y(t) = CX(t),$$

where constant matrices A, B and C denote the state matrix, the input matrix and the output matrix respectively. $X(t)$ is the state array of the system and $U(t)$ is the control input and $Y(t)$ is the output of the system. System states X consist of the velocity of two masses and the displacement of two masses: $X(t) = [\theta_1, \theta_2, \omega_1, \omega_2]^t$,

The differential equations for the system are given as

$$J_1 \ddot{\theta}_1 = K_m i_m - k(\theta_1 - \theta_2) - d(\dot{\theta}_1 - \dot{\theta}_2) - b(\dot{\theta}_1 - \dot{\theta}_2)$$

$$J_2 \ddot{\theta}_2 = -k(\theta_2 - \theta_1) - d(\dot{\theta}_2 - \dot{\theta}_1) - b(\dot{\theta}_2 - \dot{\theta}_1)$$

These equations can be simplified to :

$$J_1 \ddot{\theta}_1 = K_m i_m - k(\theta_1 - \theta_2) - (d + b)(\dot{\theta}_1 - \dot{\theta}_2) \quad (3.1)$$

$$J_2 \ddot{\theta}_2 = k(\theta_1 - \theta_2) + (d + b)(\dot{\theta}_1 - \dot{\theta}_2) \quad (3.2)$$

Now, to derive the state-space model of the system, we consider the states vector as

$$x = [\theta_1, \theta_2, \omega_1, \omega_2] \quad \text{where} \quad \omega_1 = \dot{\theta}_1 \quad \text{and} \quad \omega_2 = \dot{\theta}_2$$

Thus, we define our states as :

$$\begin{aligned}
 x_1 &= \theta_1 \\
 x_2 &= \theta_2 \\
 x_3 &= \omega_1 = \dot{\theta}_1 \\
 x_4 &= \omega_2 = \dot{\theta}_2 \\
 \\
 \dot{x}_1 &= x_3 \\
 \dot{x}_2 &= x_4 \\
 \dot{x}_3 &= \ddot{\theta}_1 = \frac{K_m i_m}{J_1} - \frac{k}{J_1}(\theta_1 - \theta_2) - \frac{(d+b)}{J_1}(\dot{\theta}_1 - \dot{\theta}_2) \\
 \dot{x}_4 &= \ddot{\theta}_2 = \frac{k}{J_2}(\theta_1 - \theta_2) + \frac{(d+b)}{J_2}(\dot{\theta}_1 - \dot{\theta}_2)
 \end{aligned}$$

Substituting above values in equations (3.1) and (3.2), we get,

$$J_1 \ddot{x}_1 = K_m i_m - k(x_1 - x_2) - (d+b)(x_3 - x_4) \quad (3.3)$$

$$J_2 \ddot{x}_2 = k(x_1 - x_2) + (d+b)(x_3 - x_4) \quad (3.4)$$

We also know that,

$$\begin{aligned}
 \text{Input to the system} &= u = i_m \text{ and,} \\
 \text{Output of the system} &= y = x_1
 \end{aligned}$$

Thus, we can now write the state-space model for continuous time as,

$$\dot{x} = Ax + Bu$$

$$\Rightarrow \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{k}{J_1} & \frac{k}{J_1} & -\frac{(d+b)}{J_1} & \frac{(d+b)}{J_1} \\ \frac{k}{J_2} & -\frac{k}{J_2} & \frac{(d+b)}{J_2} & -\frac{(d+b)}{J_2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \frac{K_m}{J_1} \\ 0 \end{bmatrix} u \quad (3.5)$$

and for output,

$$y = Cx$$

$$\Rightarrow y = [1 \ 0 \ 0 \ 0] x \quad (3.6)$$

and matrix A, B, C can be expressed as follows,

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -7.08 \times 10^4 & 7.08 \times 10^4 & -1.1 \times 10^6 & 1.1 \times 10^6 \\ 7.08 \times 10^4 & -7.08 \times 10^4 & 1.1 \times 10^6 & -1.1 \times 10^6 \end{bmatrix}, \quad (2)$$

$$B = \begin{bmatrix} 0 \\ 0 \\ 1.173 \times 10^4 \\ 1 \end{bmatrix}, \quad (3)$$

$$C = [1 \ 0 \ 0 \ 0], \quad (4)$$

3.2.2 Considered cases

The calculated values for K and F as described in the table 3.1.

Gains used in $u[k]$		Values
5*Feedback Gain K	k_1	5.822990811552409
	k_2	-5.822992246204845
	k_3	-0.180024323629381
	k_4	0.180023422504676
	k_5	0.516351924157763
Feedforward gain F	-	0.0000014346524431

Table 3.1: Calculated values of Feedback Gain K and Feed-forward gain F

3.2.3 Discrete State-Space Model

Since the embedded platform usually works at discrete time. Hence, the continuous state-space model should be discretized to fit the sampling period of the controller. The discrete state-space model can be defined as:

$$\begin{aligned} x(k+1) &= \phi x(k) + \Gamma u(k), \\ y(k) &= Cx(k), \end{aligned} \quad (5)$$

where,

$$\phi = e^{Ah}, \Gamma = \int_0^h e^{As} B ds, \quad (6)$$

and h is the sampling period between two samples. Our control task is to design $u[k]$ which makes $y[k]$ follow $r[k]$.

3.3 PIL Simulations

PIL is a simulation method which compiles the generated code from the feedback control model and then upload and run the code on the embedded platform. In this control application, the digital controller is realized by a FPGA-based embedded control system. Because traditional model-based simulation is not often sufficient to exactly capture control dynamics. This method increases the realism of the simulation and provide communication with specific hardware platforms. In PIL the target platform is not a real time environment and the communication with the external embedded platform is given by using specific functions installed in a simulation integrated environment.

The progress from model-based simulation to implementation on an platform requires developers to communicate the computer platform with the aimed embedded hardware. The target-specific object code generated in the host PC and is then downloaded to the target embedded platform for compiling and execution. The simulation tool in Simulink environment, running on the host PC, then communicates with the downloaded software.

3.4 HIL Simulations

HIL simulation is a method which is used in the development and testing of control systems with complex operation. With HIL simulation the physical part of the control system is replaced by a simulation, using a mathematical model that fully describes the important dynamics of the physical model. HIL simulation can be performed directly with Real-Time Workshop, which using a computer as a host and a target in simulation.

In this project, both model and controller are compiled and then upload and run the generated code on the embedded platform.

3.5 Simulink model and code generation

The control application is built in Simulink environment according to the previously shown mathematical model. The code generation is done with Real-Time Workshop, which generates the files that are uploaded and executed on the platform.

3.5.1 Simulink Model

Implementing the control application (Eq. 5 and Eq. 6) in Simulink with reference $r[k]$ and states $x[k]$ as inputs, and resulted $y[k]$ and new states $x[k]$ as output, this simulink model can be seen in Figure 3.3:

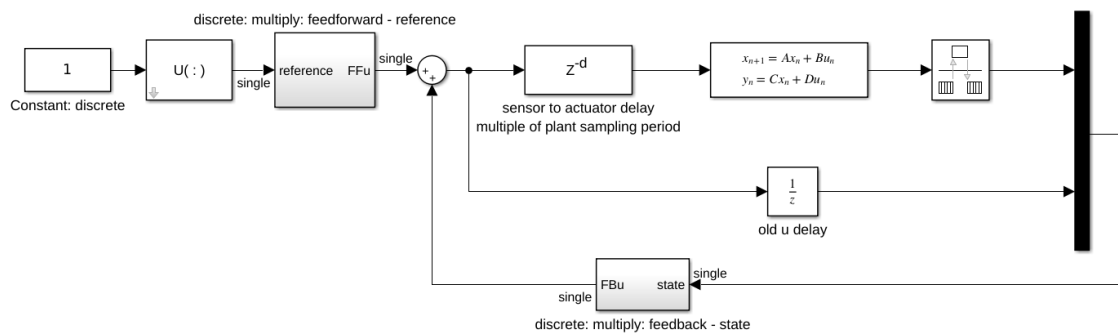


Figure 3.3: The feedback control model in Simulink environment

3.5.2 Code Generation

Code generation is a complex process that involves a large number of intermediate files. The Simulink environment provides a Embedded coder tool which can transform a Simulink model to a targeted platforms' programming code such as C/C++. [8]. After the code is generated, the user can build and run the compiled code by clicking the Build icon. The code generator builds the executable and generates the Code Generation Report.

In order to generate code successfully, the first step is to specifying code generation settings in the Configuration Parameters dialog box. Next is to choose the appropriate solver and code generation target, and checking the model configuration for execution efficiency. The process starts with the user's Simulink model which is converted to the intermediate Real Time Workshop (RTW) document by SFunctions. The RTW document is then converted to C using Target Language Compiler (TLC) files. In Figure 3.4 the process and associated file types are enumerated.



Figure 3.4: Code generation process

The Target Language Compiler (TLC) is a tool originally developed for the Matlab Real-Time Workshop. It then became the integral part of Matlab Embedded Coder. It enables the user to generate embedded C code directly from Simulink model via using the complete RTW document. This embedded code consists of three groups: entry point, tasks, and auxiliary. The transition from the RTW document to embedded C can be seen in the flowchart Figure 3.5.

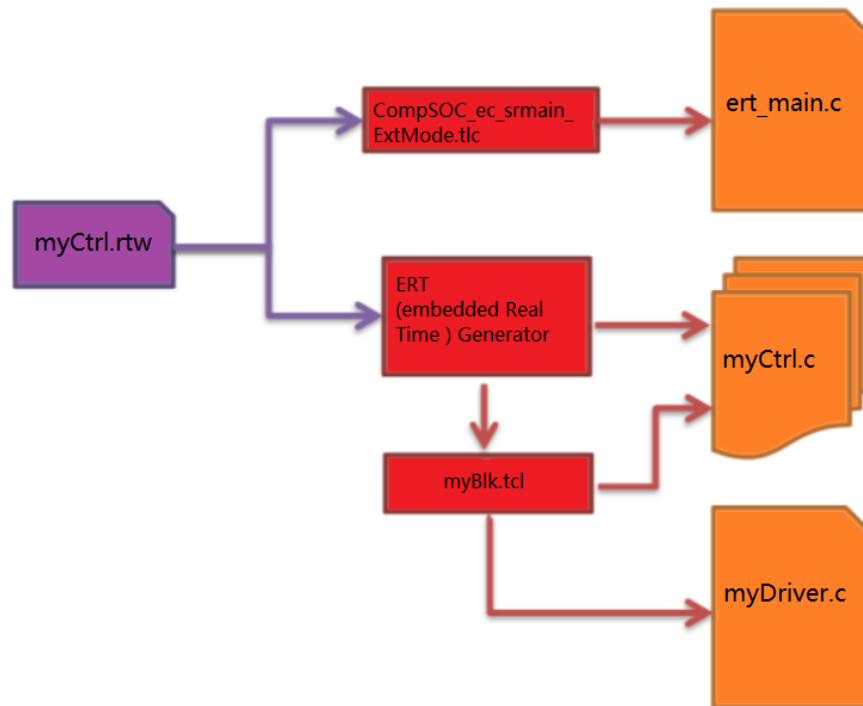


Figure 3.5: Code generation from Real-Time Workshop document

The Embedded Real Time (ERT) generator is a particular set of TLC files specialized to generate codes targeting embedded platforms. It is always used for creating the tasks. This generator calls related TLC files for each block described in the RTW document to generate its specific embedded code. The generated file is a set of files that can be built into an executable. The executable can then be uploaded and executed on the embedded platform. They have the same name as the initial Simulink model, but with a different suffix and file extension [9].

3.5.3 Code generation with Real-Time Workshop

A Simulink model has been built, then the next is to generate the code for the feedback control application. Code generation ensures that the code is generated in a effective way, in order to cope with memory space and speed of the embedded platform.

The Code Generation Options is found under C/C++ Code in the Code menu. Clicking this button will open the Configuration Parameters dialog. A system target file is chose, *CompSOC_ec.tlc*, with the target set for CompSOC embedded platform. The language which is uploaded and executed on the platform is set to C. This can be seen in the Figure 3.6.

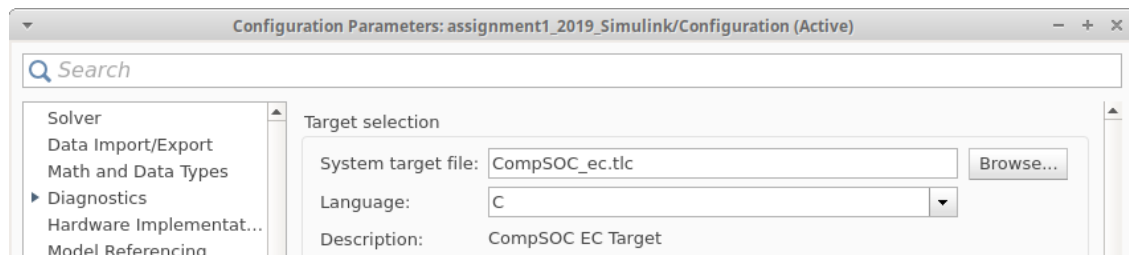


Figure 3.6: Configuration Parameters

The option for solver type is Fixed-step solvers which solve the model at fixed time intervals

from the start time to the stop time of the simulation. The size of time intervals is set in the Fixed-step size option which determines the fundamental sample time. Figure 3.7 depicts these options.

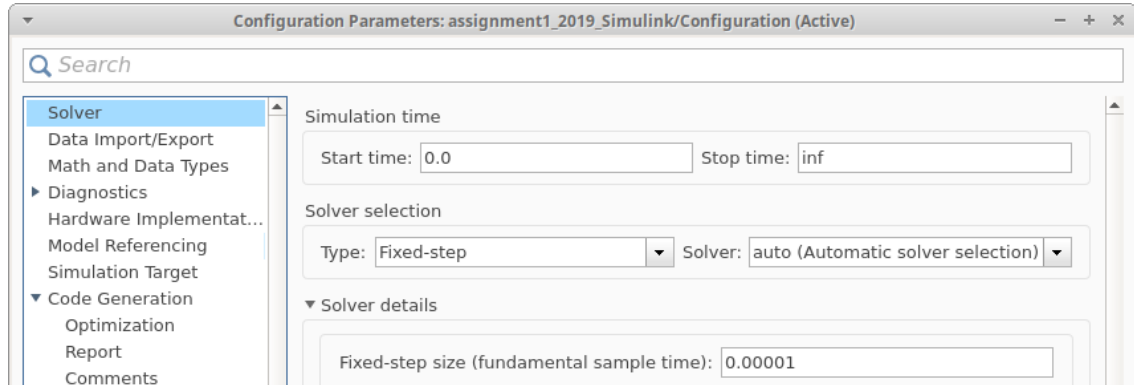


Figure 3.7: Solver diagram

In the interface dialog, the external mode option is selected because the simulation use the I/O drivers to communicate with the embedded platform, the application stores contiguous response data in memory accessible to Simulink until a data buffer is filled. This can be seen in Figure 3.8.

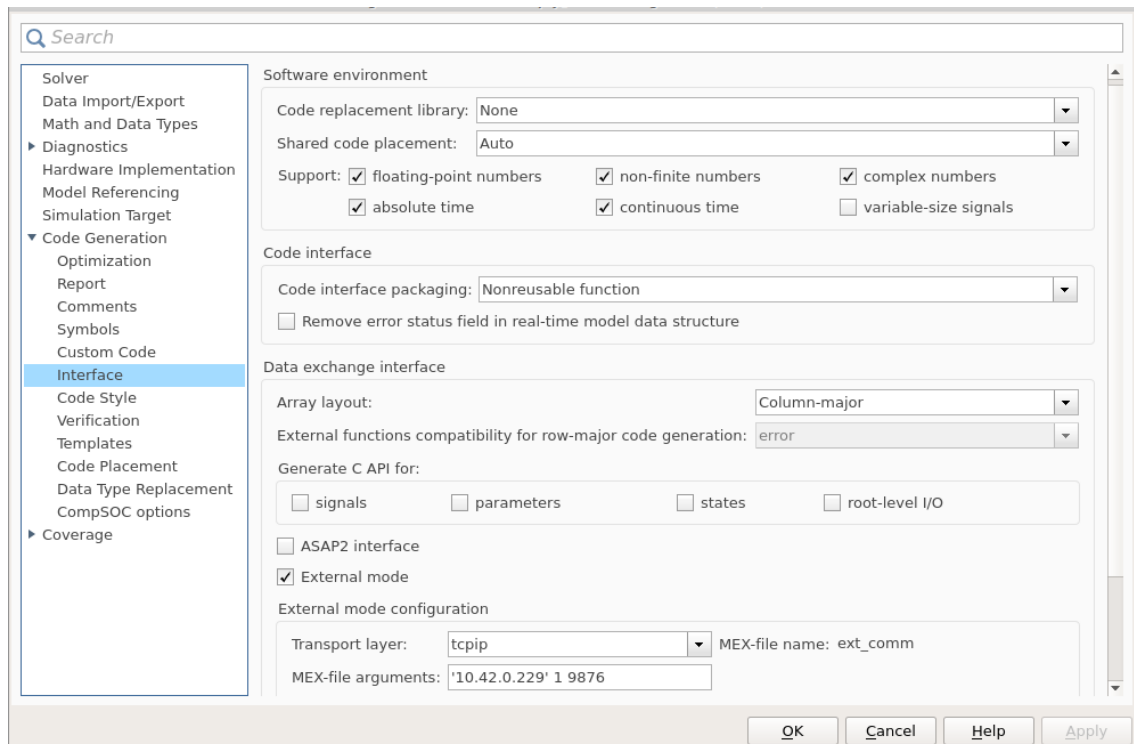


Figure 3.8: Interface diagram

The *ert_code_template.cgt* file is chosen for both header and source templates. File customization template is used to customize the generated code with a CFP template file.

The *CompSOC_ec_file_process.tlc* file is used in this thesis to call a code template API to emit the code into specified sections of generated source and header files. This can be seen in Figure 3.9.

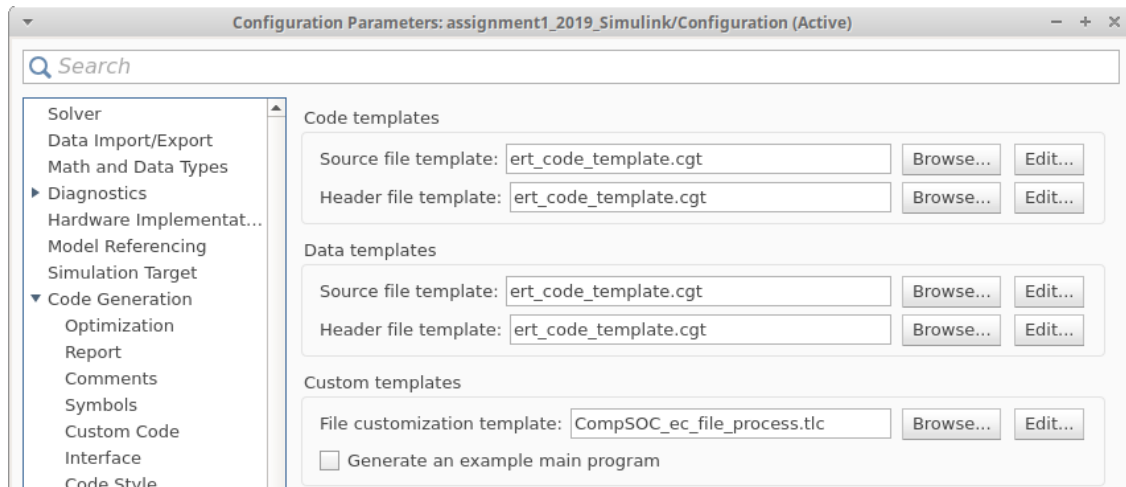


Figure 3.9: Templates diagram

Previously the *CompSOC_ec_file_process.tlc* file is chose to call a code template. From Figure 3.10, the extra file *CompSOC_ec_srmain_ExtMode.tlc* is used to generate the main file which is mean to generate the *ert_main.c* file. This file is provided by Mathworks to be used as a basis for custom modifications, and for use in simulation.

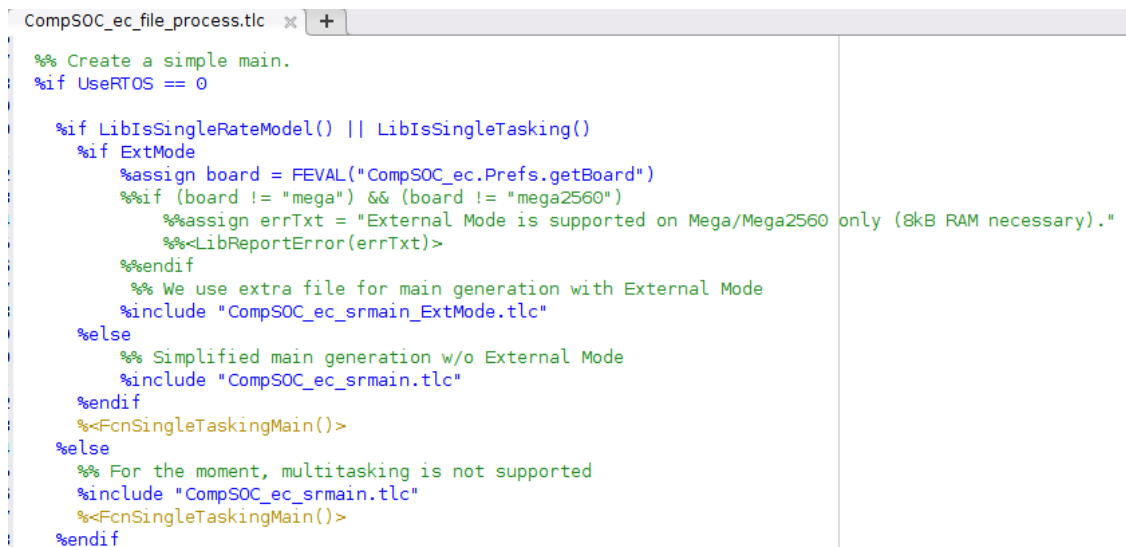


Figure 3.10: The File customization template

Then the next step is to build the Code Generation Report by clicking the Build Model in the tool bar.

The generated report which shows in Figure 3.11 contains header and source-file for the feedback control application, definition files and the file *ert_main.c* which is an example file for developing embedded applications. This file provides a basis for custom modifications, and for use in simulation. *ert_main.c* is generated from *CompSOC_ec_srmain_ExtMode.tlc* file which is modified by the customer. This can be seen in the next chapter.

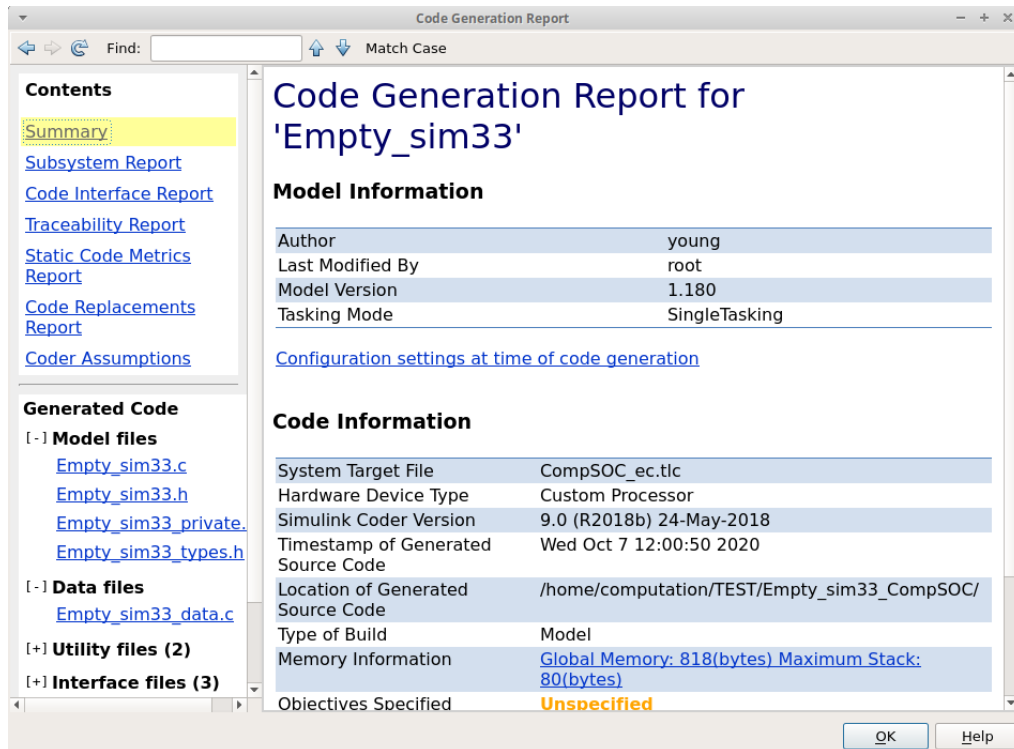


Figure 3.11: Code Generation Report

Chapter 4

Single-Core simulations implementation

The integration tool provides the module *ert_main.c* as a template example for developing embedded applications. It is provided as a basis for custom modifications, and for use in simulation. In our project, *ert_main.c* is generated from *CompSOC_ec_srmain_ExtMode.tlc* file. *ert_main.c* contains two functions, the first one is *rt_OneStep*, which is a timer interrupt service routine (ISR). *rt_OneStep* calls *MODEL_STEP* to execute processing for one clock period of the model. As provided, main function is useful in simulation only when it is modified for real-time execution.

4.1 Control Design and Implementation

In this project, we will concentrate on the 2-DOF controller configuration presented in [8] and displayed in Figure 4.1. This controller can be defined as:

$$u(k) = Kx(k) + Fr(k), \quad (7)$$

where K represents feedback controller which aims to stabilize the outputs for the system based on an available process model, and F is in terms of feedforward controller which helps to improve the accuracy of the system output. According to Figure 4.1, the reference signal is known beforehand and its scaled velocity, acceleration enable a straightforward feedforward tuning.

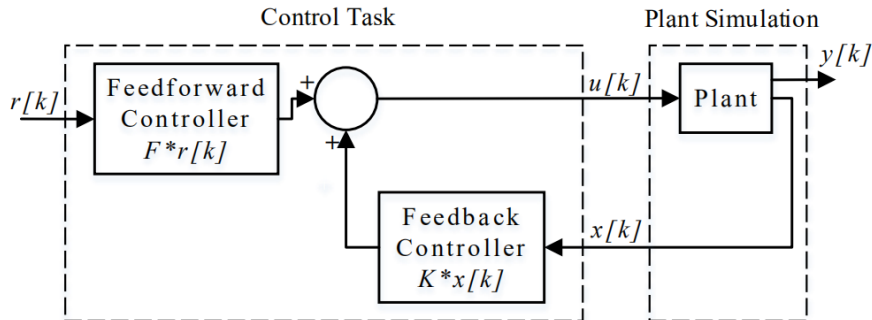


Figure 4.1: 2-DOF controller configuration

4.2 As-fast-as-possible Scheduling

The integration tool enables to generate an executable from the simulink model and performs external mode (HIL) simulations. This executable has a scheduling nature which called "as-fast-as-possible", which refers to the time when the platform reads the reference $r[k]$ and sends the calculated new states $x[k]$ back to Simulink, then it immediately reads the next reference $r[k]$. From the Figure 4.2, the Simulink sends reference $r[1]$ to the platform, then the platform calculate the new states $x[1]$ and sends it back to simulink, then it reads the new reference $r[2]$ from Simulink immediately.

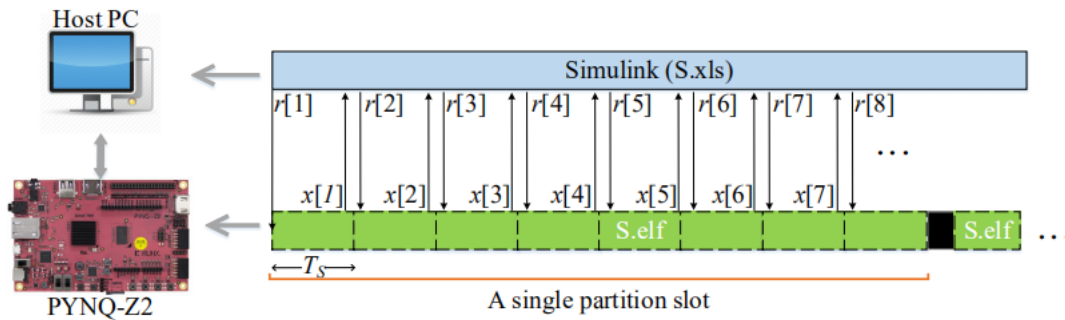


Figure 4.2: As-fast-as-possible Scheduling

The core of the Embedded Coder program is typically the main function. On each iteration, the main function executes a background code and checks for a termination condition. The main loop is periodically interrupted by a timer.

The execution driver, *rt_OneStep*, sequences calls to the *model_step* function. In a single-rate model, *rt_OneStep* simply calls the *model_step* function. Code compilation is controlled by the symbol NUMST, which represents the number of sample times in the model. NUMST is defined to be 1 for a single-rate model; otherwise NUMST is greater than 1. NUMST is defined in the generated makefile *model.mk*. In our project, the model achieve a Single-Rate Operation, The following pseudocode shows the composition of *rt_OneStep* in a single-rate program.

```
void rt_OneStep()
{
  Check for interrupt and other error
  Enable "rt_OneStep" (timer) interrupt
  ModelStep-Time step combines output,logging,update.
}
```

Figure 4.3: The naive composition of *rt_OneStep*

The *rt_OneStep* function is designed to execute the model process within a single clock period. In order to achieve this timing constraint, *rt_OneStep* maintains and checks a timer overrun flag. This means timer interrupts are disabled until the overrun flag has been checked.

The *ert_main.c* is generated from a special TLC file and then it will be uploaded to the platform to do the execution. The left side of Figure 4.4 shows the specific TLC code. The corresponding generated embedded code is represented on the right side. The pseudocode is a design for a harness program to drive the model. The *ert_main.c* program, as shipped, only partially implements this design. We must modify it according to our specifications.

```

%openfile tmpBuf
%assign fcnReturns = "void"
%assign fcnName = "rt_OneStep"
%assign fcnParams = "void"
%assign fcnCategory = "main"
%createrecord fcnRec {Name fcnName; Returns fcnReturns;
    Abstract ""; Category fcnCategory;
    Type "Utility"}
%<SlibDumpFunctionBanner(fcnRec)>
%undef fcnRec

%<fcnReturns> %<fcnName>(%<fcnParams>)
{
    static boolean_T OverrunFlag = false;

    if (OverrunFlag) {
        %<LibSetRTModelErrorStatus("\Overrun\");>
        return;
    }

    OverrunFlag = true;

    %<LibCallModelStep(0)>\

    OverrunFlag = false;

    rtExtModeCheckEndTrigger();
}

void rt_OneStep(void)
{
    static boolean_T OverrunFlag = false;

    if (OverrunFlag) {
        rtmSetErrorStatus(Empty_sim33_M, "Overrun");
        return;
    }

    OverrunFlag = true;

    Empty_sim33_step();

    OverrunFlag = false;

    rtExtModeCheckEndTrigger();
}
    
```

Figure 4.4: The specific TLC code and its corresponding generated embedded code for `rt_OneStep` function

4.3 Real-time scheduling

The first progress we want to achieve is to change the scheduling of the executable on the platform from "as-fast-as-possible" to "real-time" since control tasks/software need to execute periodically, sequentially and in a jitter-free fashion. Figure 4.5 depicts such real-time implementation, where T_s is the execution time of one step of the simulation and h is the execution period. *S.xls* is the simulation model and *S.elf* is the build executable. The Simulink periodically sends reference $r[k]$ to the platform with the execution period h .

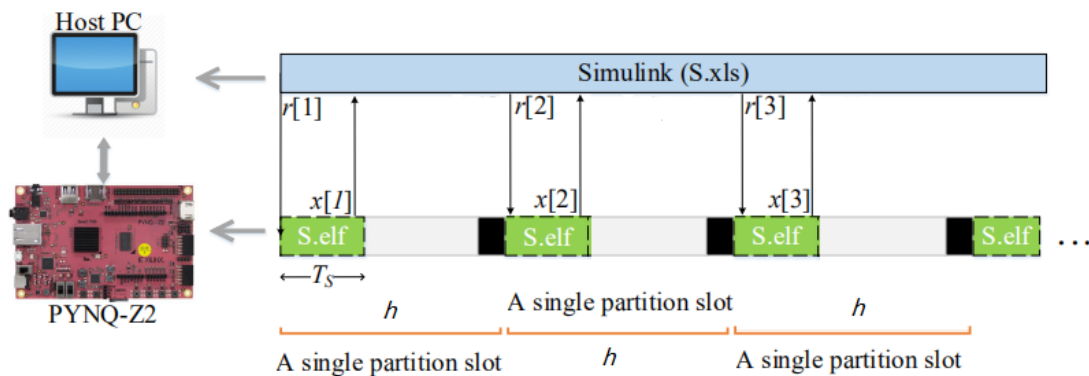


Figure 4.5: The composition of new `rt_OneStep`

The main function which used for performing the control tasks is periodically interrupted by a timer function. The `rt_OneStep` function executes processing for one clock period of the model. What needs to be done is to put a waiting timer *asm("sleep")* of the platform after the `rt_OneStep` and this can force the execution to become periodic. Figure 4.6 shows this situation.

<pre> { boolean_T rtmStopReq = %<GET_TYPE_ID_REPLACEMENT("false")>; rtExtModePauseIfNeeded(%<RTMGet("RTWExtModeInfo")>, ... %<NumRuntimeExportedRates>, ... &rtmStopReq); if (rtmStopReq) { %<RTMSetStopRequested("true")>; } if (%<RTMGetStopRequested()> == true) { %<LibSetRTModelErrorStatus("\Simulation finished\");>; break; } } /* External mode */ { boolean_T rtmStopReq = %<GET_TYPE_ID_REPLACEMENT("false")>; rtExtModeOneStep(%<RTMGet("RTWExtModeInfo")>, ... %<NumRuntimeExportedRates>, ... &rtmStopReq); if (rtmStopReq) { %<RTMSetStopRequested("true")>; } } /*_t_iter = *timer;*/ rt_OneStep(); asm("sleep"); </pre>	<pre> &rtmStopReq); if (rtmStopReq) { rtmSetStopRequested(assignment1_2023_Simulink_M, true); } if (rtmGetStopRequested(assignment1_2023_Simulink_M) == true) { rtmSetErrorStatus(assignment1_2023_Simulink_M, "Simulation finished"); break; } /* External mode */ { boolean_T rtmStopReq = false; rtExtModeOneStep(assignment1_2023_Simulink_M->extModeInfo, 2, &rtmStopReq); if (rtmStopReq) { rtmSetStopRequested(assignment1_2023_Simulink_M, true); } } /*_t_iter = *timer;*/ rt_OneStep(); asm("sleep"); </pre>
--	--

Figure 4.6: The specific TLC code and its corresponding generated embedded code for realtime

This steps requires careful implementation respecting the platform processing power and minimum possible sampling period. The following Figure 4.7 shows the TDM schedule in a single-rate program.

```

volatile TDMScheduleEntry table[] = {
    { .id = 1, .length = 0x100000 },
};

TDMSchedule schedule = {
    .length = sizeof(table)/sizeof(TDMScheduleEntry),
    .index = 0,
    .schedule = (TDMScheduleEntry*) table
};

```

Figure 4.7: TDM schedule

where the length of *TDMScheduleEntry* is the execution time T_s of one step of the simulation, then the simulation goes to sleep until the end of the execution period h which can be seen in Figure 4.5.

Chapter 5

Automation for multi-core implementation

5.1 The multi-core scheduling

According to Figure 4.1, the 2-DOF control application consist of control model and plant model. Our final work is to divide the executable into two parts, and map each part to a specific core. As Figure 5.1 shows, the simulink model is divided into two executable of C.elf (the controller) and P.elf (the plant) to be executed on two different cores. In this way, the two executable enables to strictly execute separately and cannot affect each other.

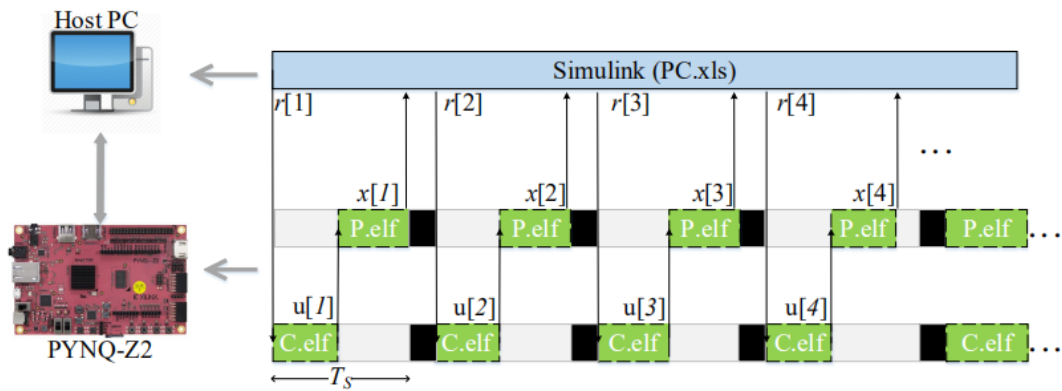


Figure 5.1: The multi-core scheduling

5.2 The shared memory block

According to Figure 5.2, the shared memory blocks is used to implement the connection between separate executable on different VP(virtual processor), the controller reads the states $x[k]$ of system from the shared memory by the "from share memory" block and sends the calculated new control input $u[k]$ to the "To share memory" block. In terms of plant simulation, it reads the new control input $u[k]$ from the same shared memory address using the "from share memory" block and sends the control output $y[k]$ to the shared memory where the controller reads the states from.

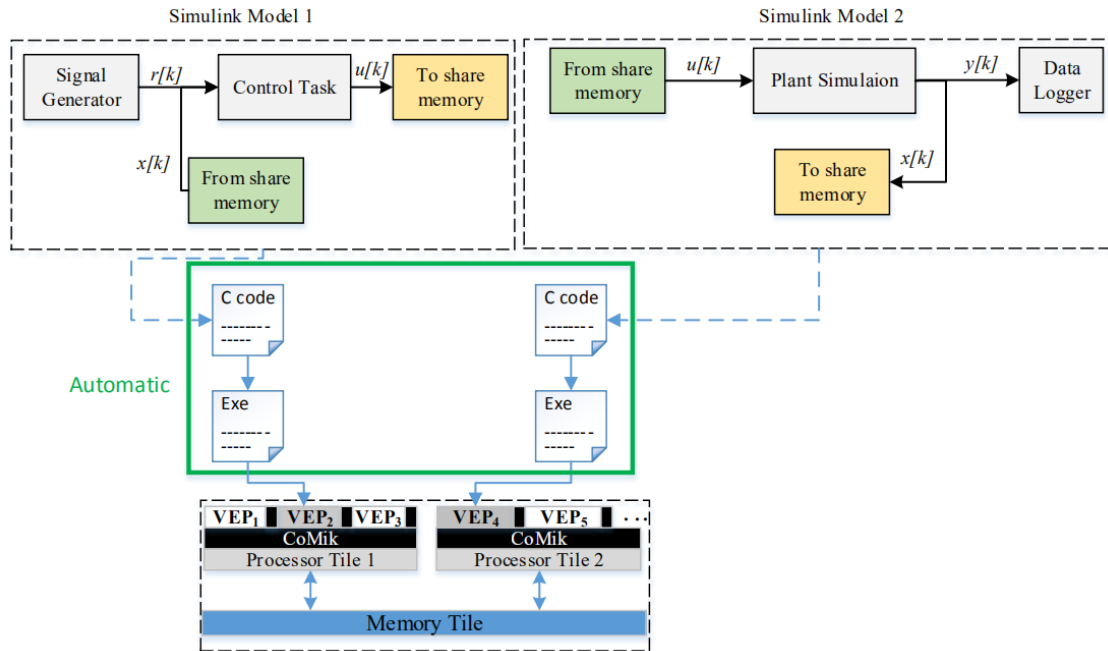


Figure 5.2: The third level of automation for multi-core code generation.

In order to achieve this goal, the Legacy Code Tool is used to generate the shared memory block. This tool can be used to generate fully inlined C MEX S-functions for legacy or custom code. The S-functions are optimized for embedded components and can be used to call existing C or C++ functions. This tool can be used to include these types of S-functions in models for which intend to generate code, use the tool to generate a TLC block file. The TLC block file specifies how the generated code for a model calls the existing C or C++ function.

5.2.1 To the shared memory

According to Figure 4.1, the controller executable(C.elf) calculates new value of control input $u[k]$ and then writes this control input to the shared memory. Figure 5.3 shows the legacy code for writing to the shared memory which generate a masked S-Function block that is configured to call the existing external code.

```
def = legacy_code('initialize');
def.SFunctionName = 'Write_to_Shared_Memory';
def.SourceFiles = {'ToSharedmemory.c'};
def.HeaderFiles = {'ToSharedmemory.h'};
def.LibPaths = {'/home/computation/WRITE/'};
def.OutputFcnSpec = 'write(single u1[], int32 u2, int32 u3)';
legacy_code('sfcn_cmex_generate', def);
legacy_code('compile', def);
legacy_code('slblock_generate', def);
legacy_code('sfcn_tlc_generate', def);
```

Figure 5.3: legacy write

In order to generate the 'Write to Shared Memory' block, using the Legacy Code Tool to transform an existing C function into a C MEX S-function. Figure 5.4 depicts how to integrate an existing C function into a Simulink model using Legacy Code Tool. This is a function that stores the value of its floating-point input to the specified address. The function is defined in a source file named *ToSharedmemory.c*, and its declaration exists in a header file named *ToSharedmemory.h*.

```

1 #include <ToSharedmemory.h>
2
3 void write(float InPut[], int InPut_addr, int Param)
4 {
5
6     int addr = 0x00000000;
7     addr = addr + InPut_addr;
8     float *share_to_2 = addr;
9
10    |
11    int i = 0;
12
13    for ( i = 0; i < Param; i++ )
14    {
15        *(share_to_2 + (10 + i))= InPut[i];
16    }
17
18 }
    
```

ToSharedmemory.c

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <math.h>
4
5 void write(float InPut[], int InPut_addr, int Param);
6 #define AXI_BRAM_CTRL_SH_3_S_AXI 0x00020000
    
```

ToSharedmemory.h

Figure 5.4: ToSharedMemory

Using `legacy_code('slblock_generate', def)` to insert a masked S-Function block into a Simulink model. Figure 5.4 depicts this 'Write to Shared Memory' block.

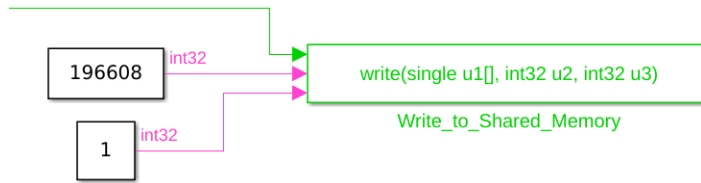


Figure 5.5: ToSharedmemory block

where the first input represents the value which stores in the shared memory, and the second input is in terms of base address of the shared memory, and the third input controls the number of value which write to the shared memory.

5.2.2 From the shared memory

According to Figure 4.1, the plant executable(P.elf) reads control input $u[k]$ from the shared memory and calculates new states value $x[k]$ using the state-space equations ,then writes these states to the shared memory. Figure 5.6 shows the legacy code for reading from the shared memory.

```

1 - lct_spec == legacy_code('initialize')
2 - def.SourceFiles = {'FromSharedmemory.c'};
3 - def.HeaderFiles = {'FromSharedmemory.h'};
4 - def.SFunctionName = 'rd_to_sharedmemory';
5 - def.OutputFcnSpec = 'void read(single y1[numel(u1)],single u1[],int32 u2,int32 u3)';
6 - legacy_code('sfcn_cmex_generate', def);
7 - legacy_code('compile', def);
8 - legacy_code('slblock_generate', def);
9 - legacy_code('sfcn_tlc_generate',def);
    
```

Figure 5.6: legacy read

In order to generate the 'Read from Shared Memory' block, using the Legacy Code Tool to transform an existing C function into a C MEX S-function. Figure 5.7 depicts how to generate a

specified block into a Simulink model using Legacy Code Tool. This is a function that reads the value from the shared memory with specified address. The function is defined in a source file named *FromSharedMemory.c*, and its declaration exists in a header file named *FromSharedMemory.h*.

```

1 #include <FromSharedmemory.h>
2
3 void read(float OutPut[],float InPut[],int InPut_addr,int Param)
4 {
5
6     int addr = 0x00000000;
7     addr = addr + InPut_addr;
8     float *share_to_0 = addr;
9
10    |
11    int i = 0;
12
13    for ( i = 0; i < Param; i++ )
14    {
15        OutPut[i] = *(share_to_0 + i);
16    }
17
18 }
```

FromSharedmemory.c

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <math.h>
4
5 void read(float OutPut[],float InPut[],int InPut_addr,int Param);
6 #define AXI_BRAM_CTRL_SH_2_S_AXI 0x00030000
```

FromSharedmemory.h

Figure 5.7: FromSharedMemory

Using `legacy_code('slblock_generate', def)` to insert a masked S-Function block into a Simulink model. Figure 5.8 depicts this 'Read from Shared Memory' block.



Figure 5.8: FromSharedmemory block

where the first input represents the number of elements in the array, and the second input is in terms of base address of the shared memory, and the third input controls the number of value which read from the shared memory. And this function outputs the same number of elements as the first input.

5.3 Choose specific tile and partition slot

The integration tool generates the code for separate Simulink models and target each executable to different cores. According to Figure 5.2, the first executable runs on processor tile1 and VP2 and the second executable runs on processor tile2 and VP4. How to choose specific tile and partition slot is the point of this chapter.

The *CompSOC_ec.tlc* file is used in this thesis as the system target file which controls the code generation stage of the build process and also control the presentation of the target to the end user. Figure 5.9 shows the general structure of a system target file.

```

rtwoptions(oIdx).prompt      = 'Processor tile for PIL application';
rtwoptions(oIdx).type       = 'Edit';
rtwoptions(oIdx).default    = '0';
rtwoptions(oIdx).tlcvariable = 'PIL_ProcessorTile';
rtwoptions(oIdx).makevariable = '';
rtwoptions(oIdx).callback   = '';
rtwoptions(oIdx).tooltip    = sprintf(['Use this parameter to define the Processor tile to be used on CompSOC to execute PIL']);

oIdx = oIdx + 1;

rtwoptions(oIdx).prompt      = 'Size of the virtual platforms in TDM table for the selected processor tile';
rtwoptions(oIdx).type       = 'Edit';
rtwoptions(oIdx).default    = '[10000]';
rtwoptions(oIdx).tlcvariable = 'Size_VirtualPlatforms';
rtwoptions(oIdx).makevariable = '';
rtwoptions(oIdx).callback   = '';
rtwoptions(oIdx).tooltip    = sprintf(['Define VP values in a bracket. for example [1000,3000,4000]']);

oIdx = oIdx + 1;

rtwoptions(oIdx).prompt      = 'Virtual platform for PIL application';
rtwoptions(oIdx).type       = 'Edit';
rtwoptions(oIdx).default    = '1';
rtwoptions(oIdx).tlcvariable = 'PIL_VirtualPlatform';
rtwoptions(oIdx).makevariable = '';
rtwoptions(oIdx).callback   = '';
rtwoptions(oIdx).tooltip    = sprintf(['Use this parameter to define the virtual platform to be used on CompSOC to execute PIL']);

```

Figure 5.9: system target file

The code defines an `rtwoptions` structure array. The `rtwoptions` structure and callbacks are written in MATLAB code, where they are embedded in a TLC file. The pane displays the options defined in `rtwoptions(oIdx)`. Configuration Parameters dialog box can be seen in the Figure 5.10.

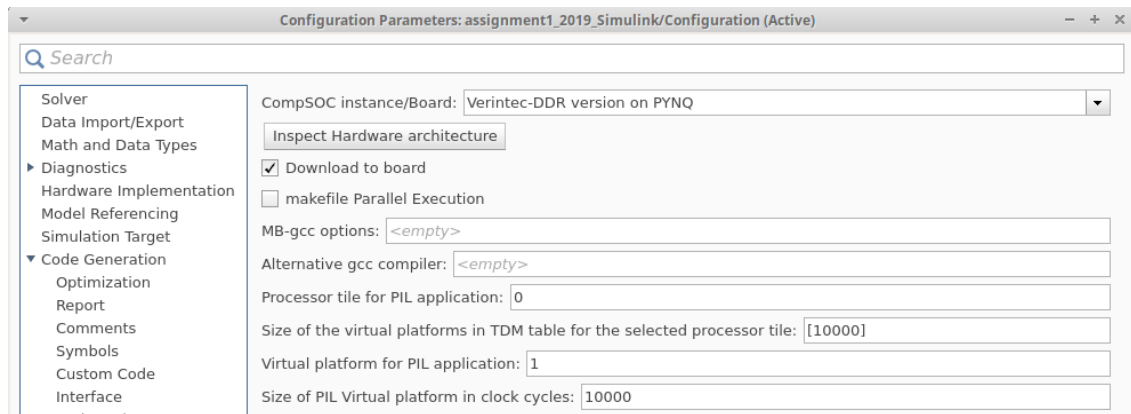


Figure 5.10: Configuration Parameters dialog box

Here using **Processor tile for PIL application** defines the processor tile to be used on CompSOC to execute PIL, using **Virtual platform for PIL application** defines the virtual platform for PIL application to be used on CompSOC to execute PIL. From the Figure 5.10, the processor tile0 and VP1 are chose to execute the application.

5.4 The hexFile

In order to download a program to CompSOC platform, the Real-Time Workshop generates the related hexFile to choose the processor tile and virtual platform for the applications which executed on the platform.

Figure 5.11 shows how to choose the specific processor tile and virtual platform.


```
PT=get_param(gcs,'PIL_ProcessorTile');
VP=get_param(gcs,'PIL_VirtualPlatform');

copy=sprintf('cp %s /home/computation/CompSOC_ec_target/Verintec_v01_SDK/app_tile_%d_%d/out.hex',elf_line,PT,VP)
system(copy)
```

Figure 5.11: runAvrDude(hexFile)

where PT represents the value of processor tile which comes from **Processor tile for PIL application** in the Configuration Parameters dialog box, and VP in terms of the virtual platform which chose by **Virtual platform for PIL application**. Then we build the model and the dialog view can be seen in the Figure 5.12.

```
echo ### Project size
text data bss dec hex filename
24372 620 22784 47776 baa0 Test.elf
echo ### Created Test.hex successfully (or it was already up to date)
hexFile =
'/home/computation/Assignment/Test_CompSOC/instrumented/Test.elf'
### Sending the elf file: /home/computation/Assignment/Test_CompSOC/instrumented/Test.elf
copy =
'cp /home/computation/Assignment/Test_CompSOC/instrumented/Test.hex /home/computation/CompSOC_ec_target/Verintec_v01_SDK/app_tile_0_1/out.hex'
```

Figure 5.12: Generate hexFile

where the **app_tile_0_1** represent the generated code is uploaded on processor tile0 and virtual platform1.

In this way, the generated code can be uploaded on specific processor tile and virtual platform.

5.5 CaseI: model-based controller and hard-coded plant

According to the Figure 5.2, the integration tool generates the code for separate Simulink models targeting different virtual platform on different processor tile. The first step is to divide the control task and the plant.

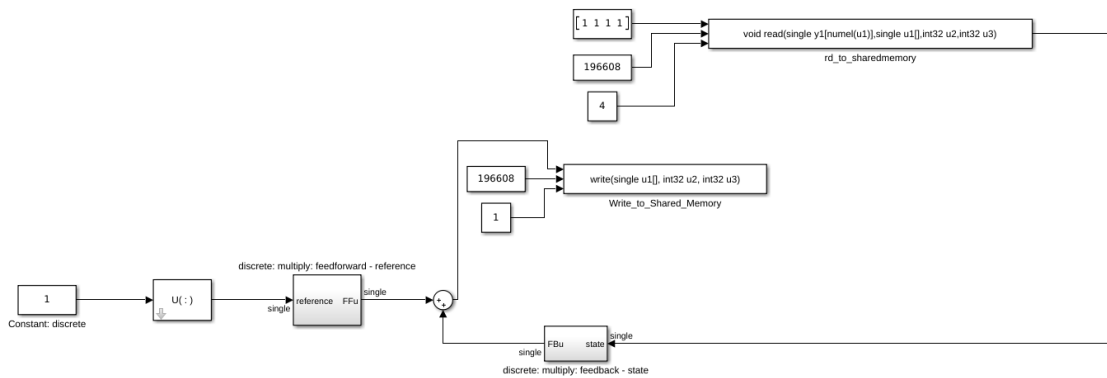


Figure 5.13: Control task

Figure 5.13 shows the control model block which reads the new space-states of the system from a specific point of the shared memory using the "From the shared memory" block and updates the control value using the "To the shared memory" block. In this system, the plant simulation is not generated from the Simulink environment. It is manually written into the **main.c** file which executed on processor tile2 and virtual platform1. From the Figure 5.14, the plant simulation reads the updated control value (**new_control_input[0][0]**) from the same shared memory address using the "from share memory" block and updates the space-states (***(shared02+0) = state[0][0]**) on the same point of the shared memory that the controller reads the sensed value from.

```

float *shared02 = (float *) AXI_BRAM_CTRL_SH_3_S_AXI;
*(shared02+10) = initial_condition_control_input[0][0];
float prev = initial_condition_control_input[0][0];
int flag = 0;

while(1){
    uint64_t t_iter = *timer;

    new_control_input[0][0] = *(shared02+10);

    //update plant state
    set_matrix(N-1, 1, partialA, 0.0f);
    set_matrix(N-1, 1, partialB, 0.0f);

    multiply_matrix(N-1, N-1, N-1, 1, A, state, partialA);
    multiply_matrix(N-1, 1, 1, 1, B, new_control_input, partialB);
    add_matrix(N-1, 1, partialA, partialB, state_next_iteration);

    state[0][0] = state_next_iteration[0][0];
    state[1][0] = state_next_iteration[1][0];
    state[2][0] = state_next_iteration[2][0];
    state[3][0] = state_next_iteration[3][0];

    //if(state[0][0]>= 0.98 && !flag){
    //    xil_printf("reach steady at: %d\n", *timer);
    //    flag = 1;
    // }
    //send 4 states to shared mem
    *(shared02+0) = state[0][0];
    *(shared02+1) = state[1][0];
    *(shared02+2) = state[2][0];
    *(shared02+3) = state[3][0];
}

```

Figure 5.14: Plant simulation

By building the controller model, the Simulink generates the executable of the control task and uploads it on the processor tile0 on virtual platform1. The plant simulation is executed on processor tile2 and virtual platform1. Through the shared memory between processor tile0 and processor tile2 which can be seen in Figure 5.15, the separate executable running on different processor tiles can communicate.

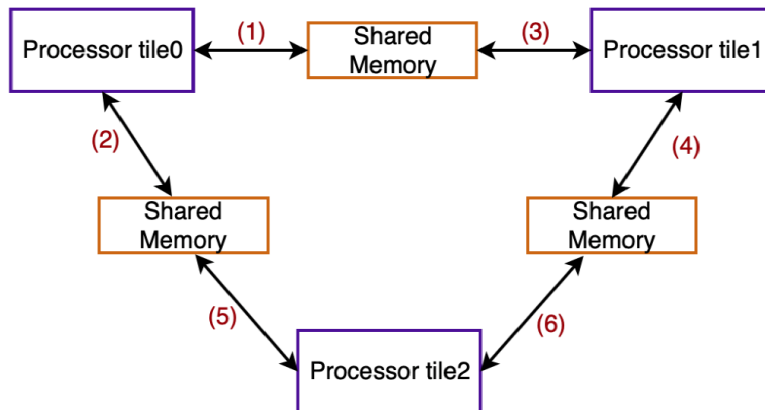


Figure 5.15: Hardware architecture

5.6 CaseII: model-based controller and code generated plant

In this level of the multi-core code generation, the feedback-feedforward control application model is divided into two separate models which shows in Figure 5.2.

Figure 5.16 represents the controller model and Figure 5.17 represents the plant model.

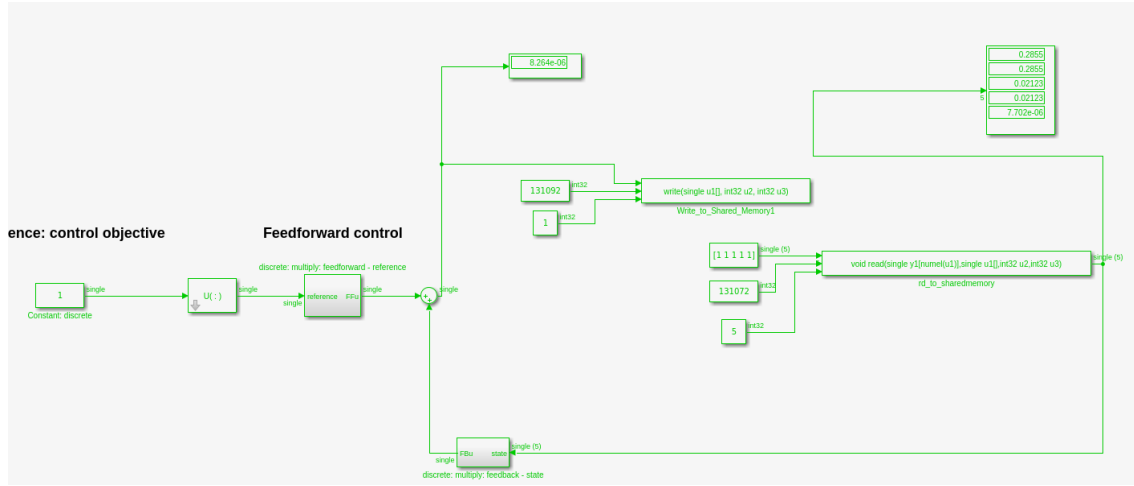


Figure 5.16: The control model

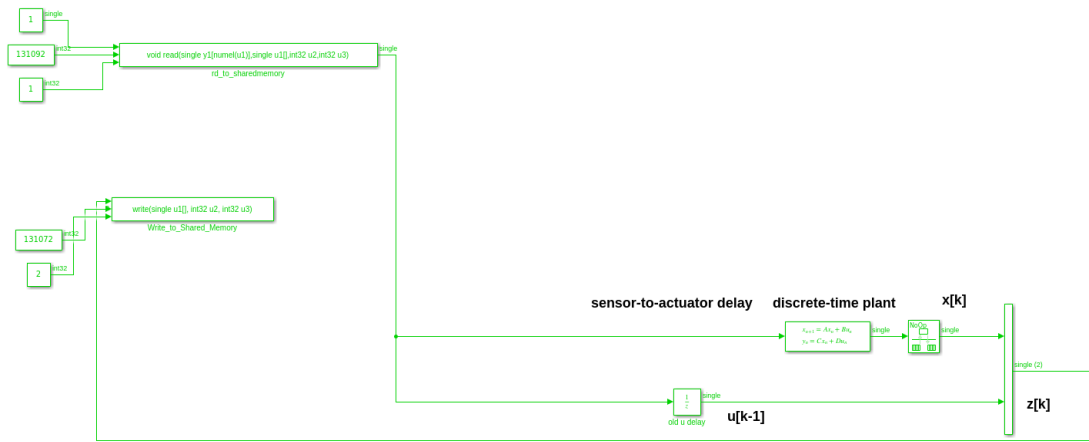


Figure 5.17: The plant model

Each of the separate model uses the generated shared memory block for communication. In the control model, the new calculated control input $u[k]$ is written to the shared memory. The plant model reads $u[k]$ from the same address of the shared memory and then uses the state-space equation to calculate the new states $x[k]$. Then these states are written to the shared memory. The control model reads the new states $x[k]$ from the shared memory with the same address.

From the Figure 5.2, the separate models automatically generate the executable simultaneously. Then the executable uploaded to the specific virtual platform for execution. In this project, we first upload the control executable to the processor tile0 and virtual platform1, then we upload the plant executable to the processor tile1 and virtual platform1. This can be seen in Figure 5.18.

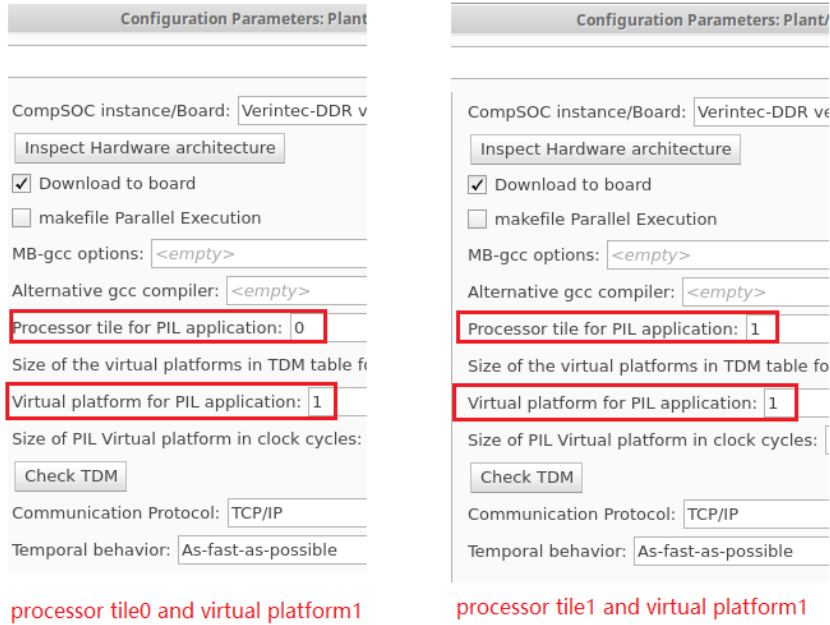


Figure 5.18: Processor tile and virtual platform

After choosing the related processor tile and virtual platform, we need to connect the host PC and the target platform. In this project, TCP/IP is selected to support communication for external mode. TCP/IP is a collective term for a series of network protocols that are used for most of the network communication. The **MEX-file** name field specifies the name of a MEX-file that implements host and target communication on the host side. The default for TCP/IP is `ext_comm`, a MEX-file provided with the Simulink Coder software.

The **MEX-file arguments** let user specify arguments to pass to an external mode interface MEX-file for communicating with executing targets. For TCP/IP interfaces, `ext_comm` allows three optional arguments:

1. **Network name of your target processor:** For example, 'myComputer' or '148.27.151.12'.
2. **Verbosity level:** 0 for no information or 1 to display detailed information during data transfer.
3. **Port number of TCP/IP server:** An integer value between 256 and 65535, with a default of 17725. A port is used to differentiate among different applications using the same network interface. It is an additional qualifier used by the system software to get data to the correct application.

From the figure Figure 5.19, specifying the arguments in the list order. The first argument which used to specify the network name of the target processor is '10.42.0.229'. The second argument is set to 1 which indicates that display detailed information during data transfer. The third argument is 9876 which means the plant application use this number as an additional qualifier.

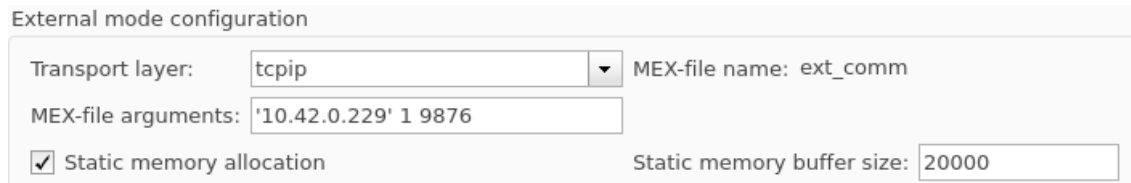
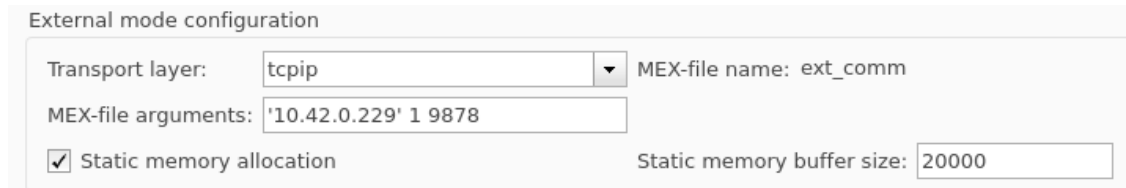


Figure 5.19: External mode configuration1

From the figure Figure 5.20, the only difference is the third argument. In the control application, this argument is set to 9878 which means the control application use 9878 as an additional qualifier.



External mode configuration

Transport layer: tcpip MEX-file name: ext_comm

MEX-file arguments: '10.42.0.229' 1 9878

Static memory allocation Static memory buffer size: 20000

Figure 5.20: External mode configuration2

Chapter 6

Experimental Results

This chapter will discuss the results of automation for multi-core code generation method introduced in Chapter 6.

6.1 Single core implementation

The simulink version used in the project is R2018b. The single core implementation of feedback-feedforward control model in the simulink is illustrated in chapter 4. Implementing the control application in Simulink with reference $r[k]$ and states $x[k]$ as inputs, and resulted $y[k]$ and new states $x[k]$ as output.

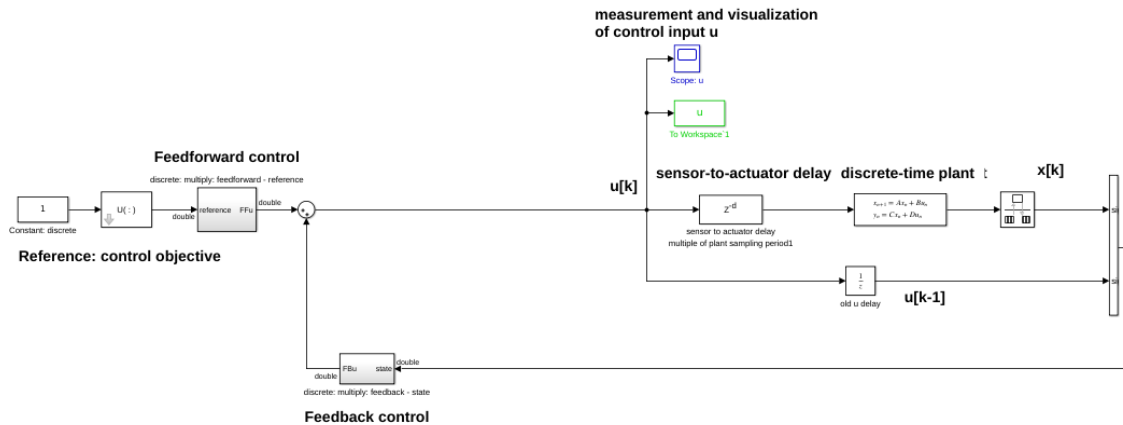


Figure 6.1: The naive implementation of feedback-feedforward application

6.2 Automated multi-core integration

To implement the first case in chapter 5, the first step is to achieve the control task to be code-generated and the plant simulation is still hard coded. Hard-coded data typically can only be modified by editing the source code and recompiling the executable. Data that are hard-coded usually represent unchanging pieces of information. In industry, plant simulation is always hard coded.

In this section, the control model block which reads the new space-states of the system from a specific point of the shared memory using the "From the shared memory" block and updates

the control value using the "To the shared memory" block. The plant simulation is not generated from the Simulink environment. It is manually written into the *main.c* file.

6.3 Results and Analysis

In this thesis, the pulse signal is chose to be as the reference $r[k]$ which can be seen in Figure 6.2.

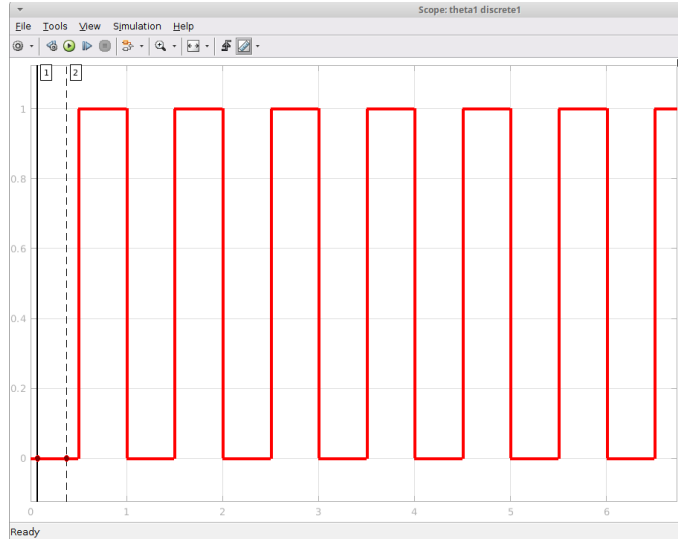


Figure 6.2: Pulse signal input

According to chapter 3, the new states $x[k]$ is the output of this application. Figure 6.3 shows the result of $x[1]$ with the single core implementation. Figure 6.4 shows the result of $x[2]$ with the single core implementation. The purpose of the control task is to design $u[k]$ which makes $x[k]$ follow $r[k]$ as the states $x[1]$ and $x[2]$ represents the velocity of two masses. From the two previous picture, we could find that the output $x[1]$ and $x[2]$ of the control task has the same trend as the reference value after a short period of time.

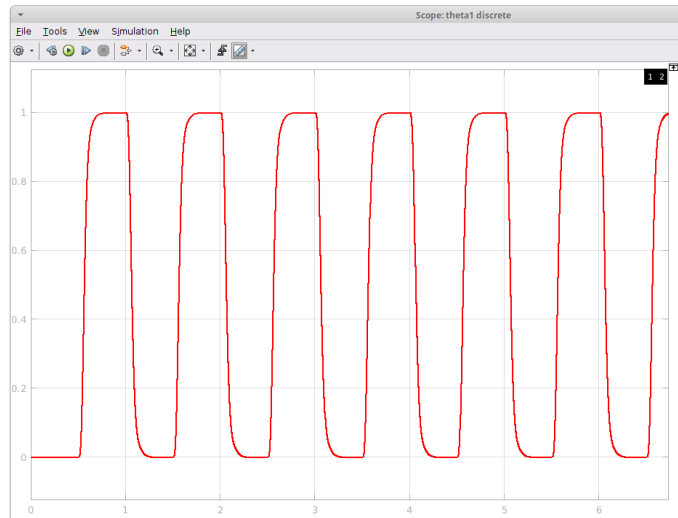


Figure 6.3: Output $x[1]$ with the single core implementation

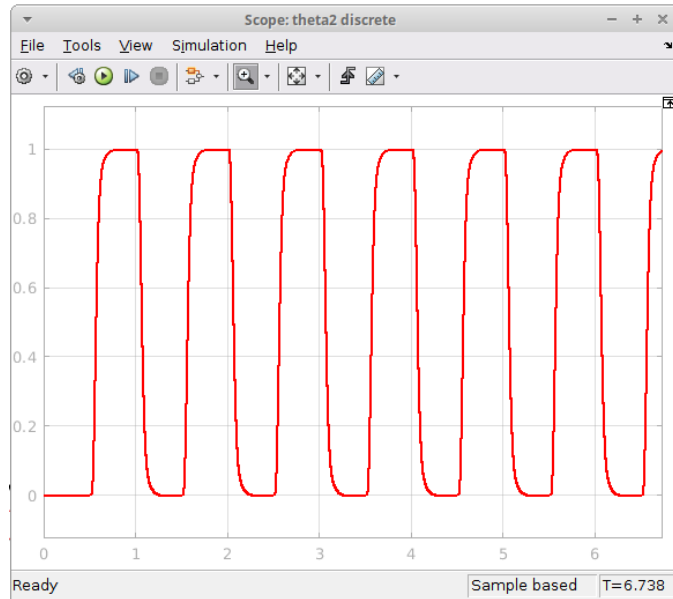
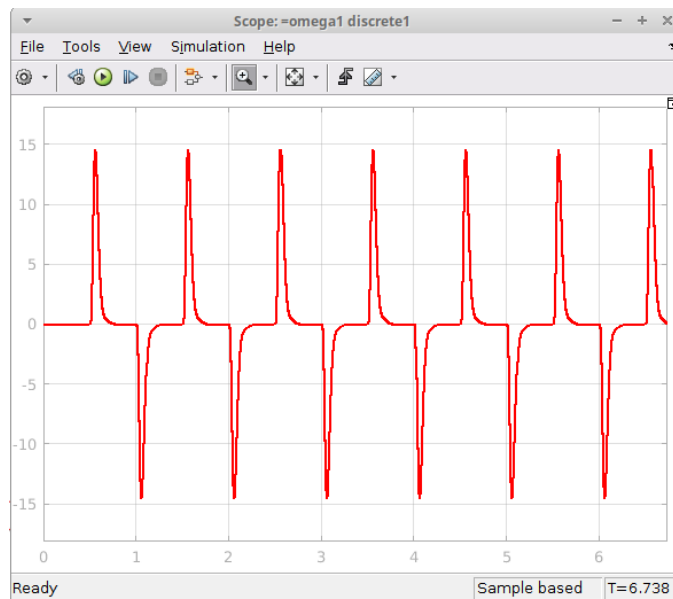
Figure 6.4: Output $x[2]$ with the single core implementation

Figure 6.5 shows the result of $x[3]$ with the single core implementation. Figure 6.6 shows the result of $x[4]$ with the single core implementation. Because the states $x[3]$ and $x[4]$ represents the displacement of two masses. And we can see from the Figure 6.5 and Figure 6.6, most of the time the value of two states is equal to zero.

Figure 6.5: Output $x[3]$ with the single core implementation

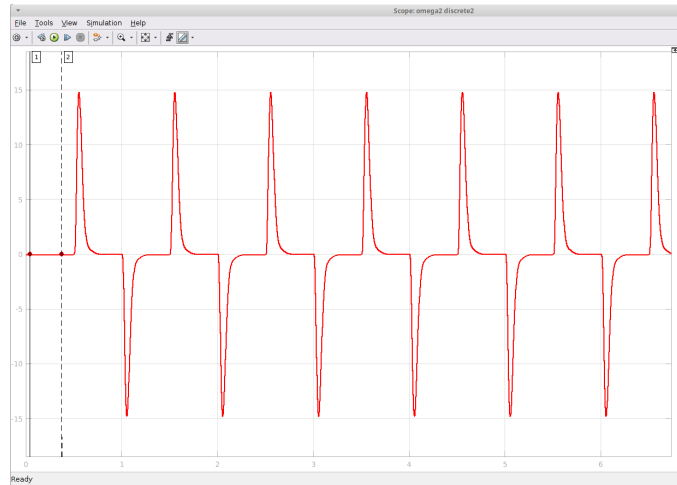


Figure 6.6: Output $x[4]$ with the single core implementation

Figure 6.7 shows the result of $x[1]$ with the multi-core and hard-coded plant implementation. Figure 6.8 shows the result of $x[2]$ with the multi-core and hard-coded plant implementation. From the two previous picture, we could find that the output $x[1]$ and $x[2]$ of the control task has the same trend as the reference value after a short period of time.

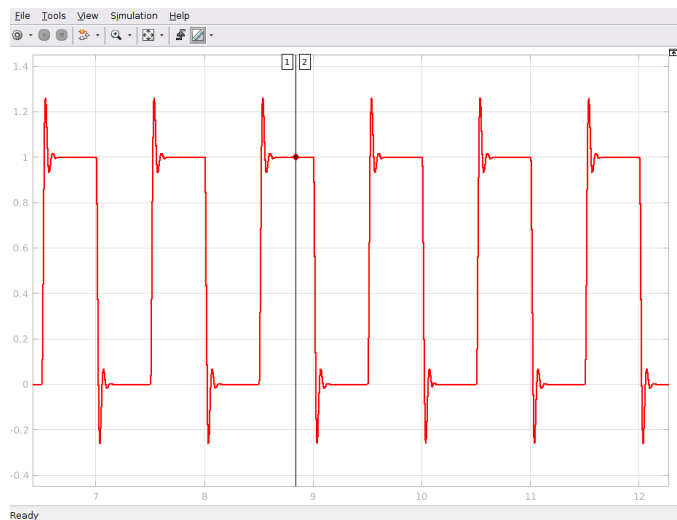


Figure 6.7: Output $x[1]$ with the multi-core and hard-coded plant implementation

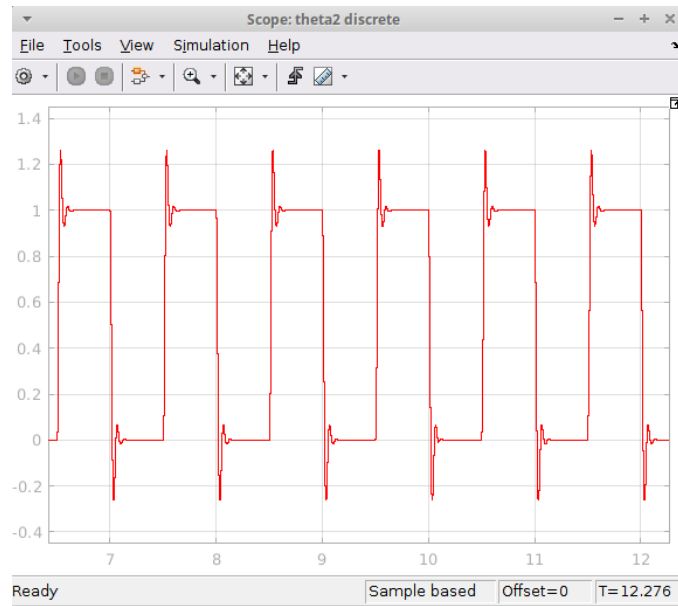


Figure 6.8: Output $x[2]$ with the multi-core and hard-coded plant implementation

Figure 6.9 shows the result of $x[3]$ with the multi-core and hard-coded plant implementation. Figure 6.10 shows the result of $x[4]$ with the multi-core and hard-coded plant implementation. Because the states $x[3]$ and $x[4]$ represents the displacement of two masses. And we can see from the Figure 6.5 and Figure 6.6, most of the time the value of two states is equal to zero.

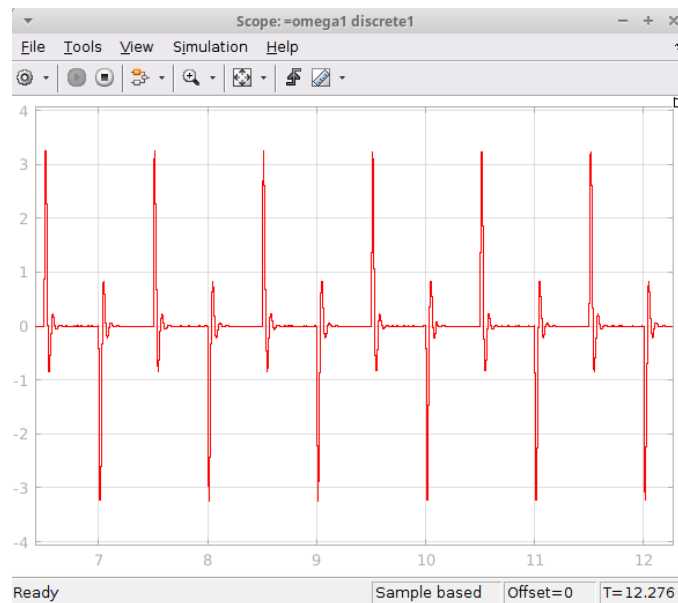


Figure 6.9: Output $x[3]$ with the multi-core and hard-coded plant implementation

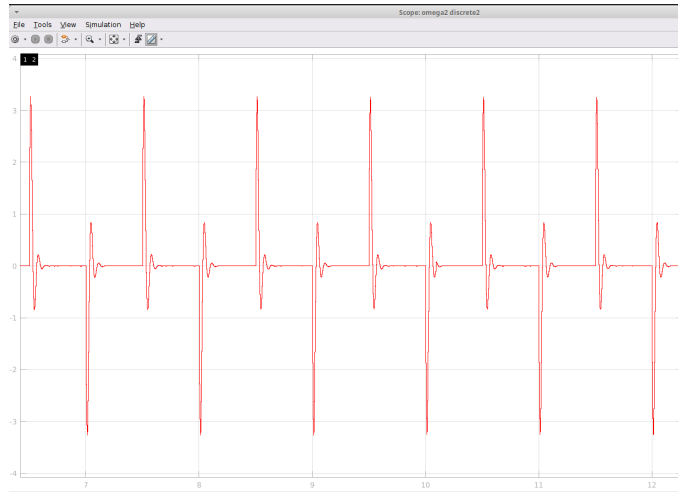


Figure 6.10: Output $x[4]$ with the multi-core and hard-coded plant implementation

6.3.1 Conclusion

In this chapter, two different implementations are discussed. The single core implementation is implementing the control application in Simulink environment with external execution. The automated multi-core integration make use of the shared memory blocks to achieve connection between separate executable on different VP(virtual processor). The output $x[k]$ of these two implementation has the same trend and reach a stable value after a short period.

Chapter 7

Conclusions and future plan

7.1 Conclusion

This thesis focuses on creating a code generation tool which can generate the whole multi-core code automatically and enable code execution on the specific platform which ensures composable and predictable. First, the composable multi-core platform - CompSoC is introduced. We choose ComSoC to implement multiple applications with TDM scheduling policy on CompSOC which can guarantee an isolated and non-interference implementation for each application.

We introduce FPGA board which is PYNQ-Z2 as hardware to explore the inner workings of the MATLAB-CompSOC integration tool and Simulink code generation. First, the process of design and implementation of a digital controller is introduced to emphasize on HIL simulation and PIL simulation which can bridge the gap between simulation and final system construction. Then we introduce the code generation, which generates the files that are uploaded and executed on the platform. Although code generation is a complex process, the Simulink environment provides the code generator which can build the executable with no further interaction between users and the Simulink environment. With the code generator, it enables to generate embedded C code directly from Simulink model that can be uploaded and executed on the embedded platform.

Using this code generator, we build an executable from the simulink model and performs external mode simulations with a nature of "as-fast-as-possible". By modifying the main function in the execution driver, the real-time scheduling of the executable has achieved. To achieve automation for multi-core code generation, the Legacy Code Tool is introduced to generate the shared memory block which is used for implementing the connection between separate executable. Therefore, we create the integration tool generates the code for separate Simulink models targeting different virtual platform on different processor tile.

7.2 Future plan

1. Plant simulation is hard-coded in Chapter 5. This plant code can be replaced by a generated plant code from a Simulink plant model.
2. Fully automated multi-core integration: In this level, the tool can generate the whole multi-core code automatically. As shown in Figure 7.1, there is just one simulink model which consists of two different parts where each part is regarded as different tasks and their target VP(virtual processor) is defined by users.

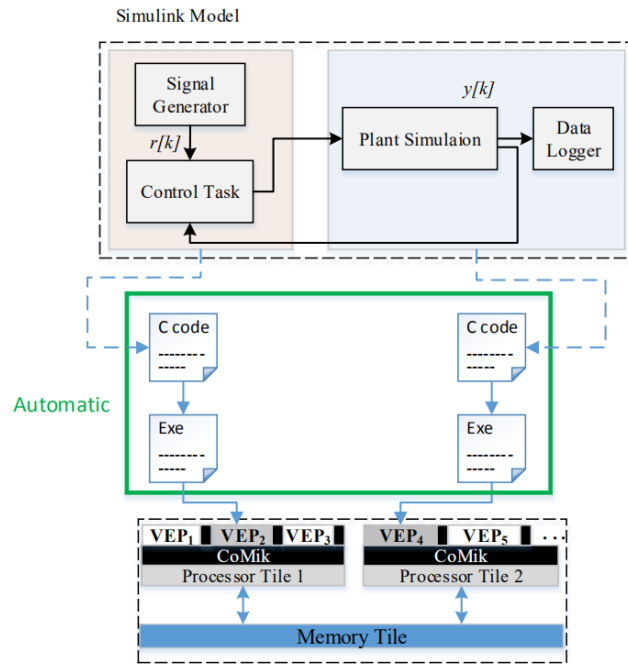


Figure 7.1: The final level of automation for multi-core code generation

In order to achieve fully automated multi-core integration, the first step is to divide the origin simulink model into two parts. As shown in Figure 7.2, the model1 is responsible for reading reference $r[k]$ and states $x[k]$, then calculating the control input $u[k]$, the model2 is responsible for reading the control input $u[k]$, then calculating the new state $x[k]$.

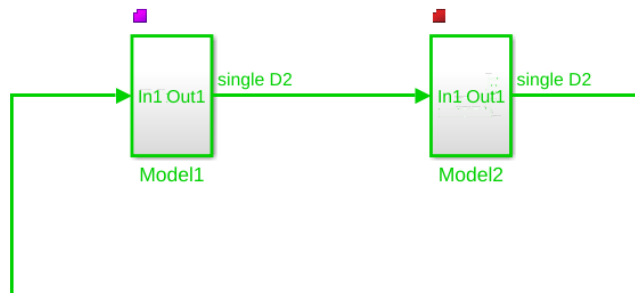


Figure 7.2: The model is partitioned to execute concurrently

After configuring the model for concurrent execution, the multicore is chosen as the target architecture which is used to deploy the partitioned model. As shown in Figure 7.3, there are two cores in this architecture where each sub-model can be mapped to different core.

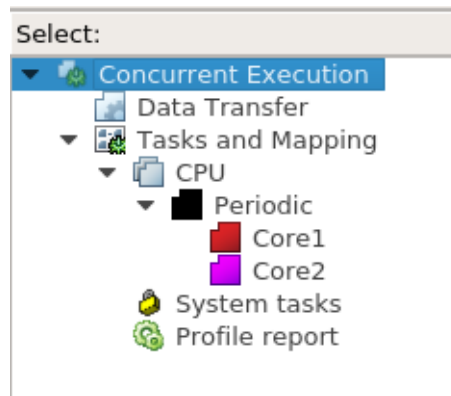


Figure 7.3: Multicore target architecture

3. Further improvement could be focused on a case where the actual plant is in the loop. In this case the results can be compared with the external mode simulation. In order to connect the Simulink model to the actual plant I/O modules for the sensors and actuators drivers should be provided in Simulink. This could be done by creating I/O driver blocks in Simulink using Legacy code generator.

Bibliography

- [1] Albert Thumann D. Paul Mehta, “Handbook of Energy Engineering” in SAMOS, 2008. pp. 305.
- [2] D. Goswami, R. Schneider, A. Masrur, M. Lukasiewicz, S. Chakraborty, H. Voit, and A. Annaswamy, “Challenges in automotive cyber-physical systems design,” in SAMOS, 2012.
- [3] H. Yan et al., “H output tracking control for networked systems with adaptively adjusted event-triggered scheme,” *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, no. 99, pp. 1–9, 2018.
- [4] G. Frehse, A. Hamann, S. Quinton, and M. Woehrle, “Formal analysis of timing effects on closed-loop properties of control software,” in *RTSS*, 2014, pp. 53–62.
- [5] K. Goossens et al., “NOC-based multiprocessor architecture for mixedtime-criticality applications,” *Handbook of Hardware/Software Codesign*, pp. 491–530, 2017.
- [6] M. Haghi, F. Wenguang, D. Goswami, and K. Goossens, “Delay based design of feedforward tracking control for predictable embedded platforms,” in *ACC*, 2019.
- [7] TUL. PYNQ Z2 User Manual. <http://www.tul.com.tw/ProductsPYNQ-Z2.html>, 2018. 33
- [8] Chris Hote Tom Erkkinen. Automatic Flight Code Generation with Integrated Static RunTime Error Checking and Code Analysis. *AIAA Modeling and Simulation Technologies Conference and Exhibit*, pages 1–2, 2006.
- [9] File and Folder Created by Build Process. <http://www.mathworks.com/help/rtw/ug/files-and-folders-created-by-the-build-process.html>, November 2012.
- [10] A. Nelson, A. Nejad, A. Molnos, M. Koedam, and K. Goossens, “CoMik: A predictable and cycle-accurately composable real-time microkernel,” in *DATE*, 2014.
- [11] Andrew Nelson, Ashkan Beyranvand Nejad, Anca Molnos, Martijn Koedam, and Kees Goossens. ”CoMik: A Predictable and Cycle-Accurately Composable Real-Time Microkernel”. 2014. vii, 14, 15
- [12] Juan Valencia. ”Composable Platform-Aware Embedded Control Systems on a Multi-Core Architecture”. 2016. vii, 13, 15, 16, 19