CHALMERS

Connecting AUTOSAR VFB to Simulink Environment

Master of Science Thesis in the Master Degree Program, Networks and Distributed Systems

NAVEEN MOHAN

HANNES ZÜGNER

Chalmers University of Technology

University of Gothenburg

Department of Computer Science and Engineering

Göteborg, Sweden, May 2012

Connecting AUTOSAR VFB to Simulink Environment.

NAVEEN MOHAN

HANNES ZÜGNER

Examiner: ARNE DAHLBERG

Chalmers University of Technology

University of Gothenburg

Department of Computer Science and Engineering

SE-412 96 Göteborg

Sweden

Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering

Göteborg, Sweden May 2012

Abstract

This thesis was conducted as a part of the Architecture For Future Electric-vehicles (AFFE) project. AFFE project is a research project funded by VINNOVA to bring out the next generation electric vehicles based on Automotive Open System Architecture (AUTOSAR).

The primary aim of the thesis is to analyze and demonstrate the possibility of connecting model-based design environments to the AUTOSAR Virtual Function Bus (VFB) to make the process of development of complex automotive systems easier. The tools chosen for this purpose were Mecel's Picea suite for the VFB implementation and Mathworks' Simulink as the model-based design environment.

The outcome of the project is that a scalable solution to connect the two disparate programs was created using two different Interprocess Communication (IPC) methods.

Preface

This Master thesis proposal was made by Mecel AB who wished to investigate the possibility of connecting the Simulink environment to the AUTOSAR VFB to test the correctness of the models at an early stage in the development cycle.

We are grateful to Mathias Fritzson, Charles Wahlin and our examiner Arne Dahlberg for their invaluable help in giving us useful feedback and helping us sort out our technical issues during the course of the project.

We also wish to thank Karin Denti, Erik Hesslow and the others at Mecel who helped us during all the stages of our work

Contents

List of abbreviations

AFFE	Architect For Future Electric vehicles
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basic Software
CAN	Controller Area Network
CU	Control Unit
ECU	Electronic Control Unit
HIL	Hardware in the loop
HW	Hardware
IDE	Integrated Development Environment
IPC	Inter-Process Communication
LIN	Local Interconnect Network
MBD	Model-based Development
MCAL	Microcontroller Abstraction Layer
MIL	Model In the Loop
OS	Operating System
RPC	Remote Procedure Call
RTE	Run-Time Environment
SIL	Software in the loop
SW	Software
SW-C	Software Component
VFB	Virtual Function Bus
WN	Wheel Node

1 Introduction

This section describes the purpose, scope and background of the project.

1.1 Background

The AUTOSAR project aims at creating a standard for automotive architecture that will be used to provide basic infrastructure for the function of standard software and applications. It was created by a large group of automotive manufactures together with Tier 1 suppliers to increase co-operation between organizations and to standardize interfaces such that code could be reused more easily [1].

In general cases testing the functionality of the system are of great importance at each stage in the development of AUTOSAR systems. At this point, in the model-based development method, simulations are done by creating models using a MBD tool and code is generated and later integrated with the AUTOSAR platform. This is a time consuming way to work. Because complex systems are hard to get right in one attempt, one has to repeatedly go back to the Simulink model, make the appropriate changes, regenerate the code and finally integrate it in the AUTOSAR environment again to repeat the test.

While the application software components are unique to each project, the rest of the architecture of AUTOSAR can be seen to be more or less the same for each project. These parts can be purchased as a package from organizations such as Mecel. Mecel's AUTOSAR platform is the Picea suite and it is this product that was used in the thesis.

This thesis was conducted as part of the government founded project 'Architecture For Future Electric-vehicles (AFFE). The objectives of AFFE are to create and present control system architecture for electrical hybrid vehicles including a drive-by-wire system with an electric motors and wheels. It is important that the architecture fulfills certain safety regulations as well as functional requirements needed for the integration of different subsystems and components. The final solution requires the giving of a proof of concept as well as guidelines, principles and solutions. It is also important that the system be scalable and adaptable to different platform configurations [2].

1.2 Purpose

The purpose of this thesis is to create a connection between the AUTOSAR Virtual Function Bus (VFB) and the Simulink environment. The aim is to make the Simulink models execute in their natural environment so that, the steps of code generation and (re)integration of the code in the AUTOSAR environment, can be avoided. As of now the debugging is very time consuming and it lacks visibility. This thesis aims at changing this and taking full advantage of the intuitive ease that comes with the design of the system in a MBD environment. This is done shifting the steps of debugging and optimization to the MBD environment and hence reducing the time and the cost to develop the whole system by a significant factor. Another important factor to be taken into account is that the solution should scale well so that it can be adapted to larger

projects and that the modified system should run within the time bounds required by the project.

MBD, as of now, is just used to design components. In a medium to large project this process can be relatively faster than using handwritten code. However using this methodology implies a large reliance on the code generated by the code generator integrated with the model-based. Although the reliance on the generated code is high, it still needs to be verified. If such a reliance on the system exists though, this works quite well. MBD is presently the method of choice depending on the functionality of the application. The correctness of the code depends on the correctness of the model. If the model is correct, then the code generated is so as well. But errors in logic etc. might lead to faults in the functionality of the system itself and this number of errors increases exponentially with the increase in complexity of the system. These faults are inescapable in the early stages of design.

This leads to a fundamental problem; to test the functional correctness of a component, one might have to simulate the whole system and the whole process of doing so from using the code generated from the models and integrating it with the simulator to running the tests is time consuming. Furthermore, the number of licenses required for the code generating tools increases as the number of people using the licenses increases. The process of debugging is slow as while the model itself is relatively easy to understand, the code generated by an automatic code generator might be complex and relatively harder to debug. The solution then, is to find some way so that the developer can see the working of the model in graphical format and make changes to the model without having to generate code and test the functionality for each change.

This is the focus of the master's thesis. The primary aim is to connect two products namely, the Mecel's Picea suite for Windows in the shape of a VFB simulator and Simulink. Picea's VFB simulator encapsulates the functionality of the AUTOSAR components below the application software level in the architecture and Simulink is the most common tool used for the MBD of application SW-Cs. A proof of concept by integrating the two using the AFFE-light system is presented and demonstrating that the system runs within the timing bounds.

1.2.1 Current implementation

Figure 1.1 shows the logical structuring of the systems as is being used today. The abbreviation SW-C, throughout this report, is used exclusively to indicate application software components. In figure 1.1 it is shown how the SW-Cs are connected to the RTE via standard interfaces within Picea. The SW-Cs are built in Simulink and code is generated from them to be integrated into Picea. All communication between the SW-Cs as well as the communication between the SW-Cs and the lower layers of the system goes through the VFB.

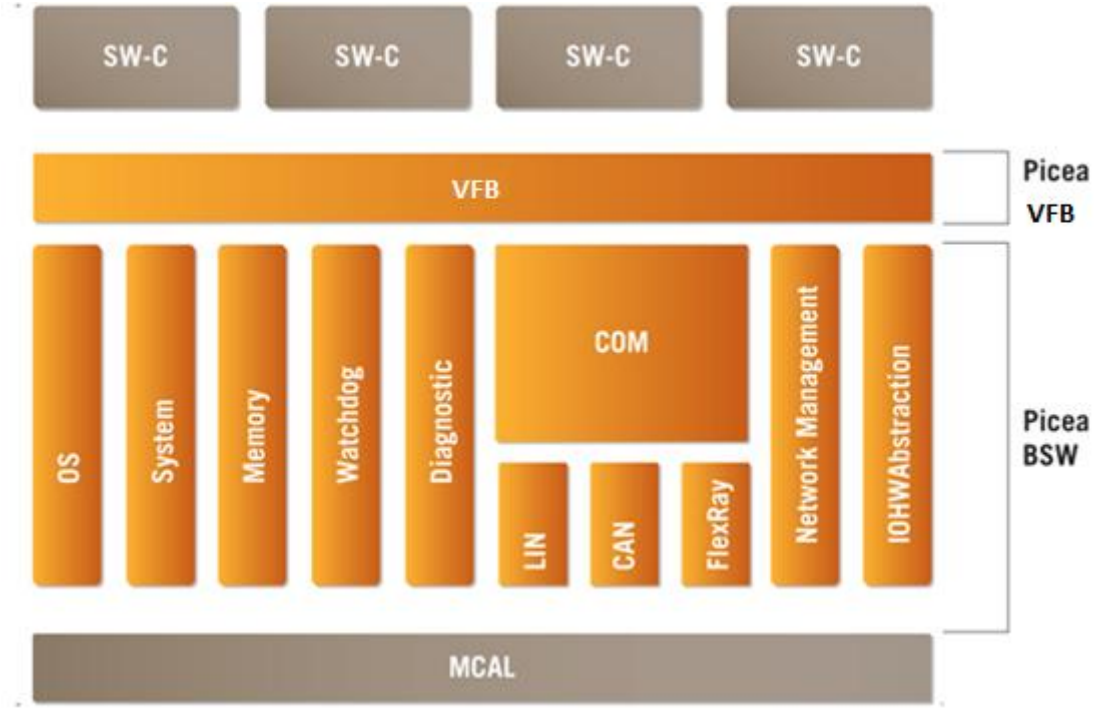

Figure 1.1: Overview of the original system.

1.2.2 Planned method of implementation

Figure 1.2 shows the proposed implementation. Here it is seen that the SW-Cs have been separated from the RTE. They now execute in the environment they were designed in i.e. Simulink. Communication to and from the SW-Cs is still done through the RTE but through the pseudo SW-Cs instead. Since the AUTOSAR architecture allows for a separation between different layers, no major changes were expected to be made in the layers below the VFB.

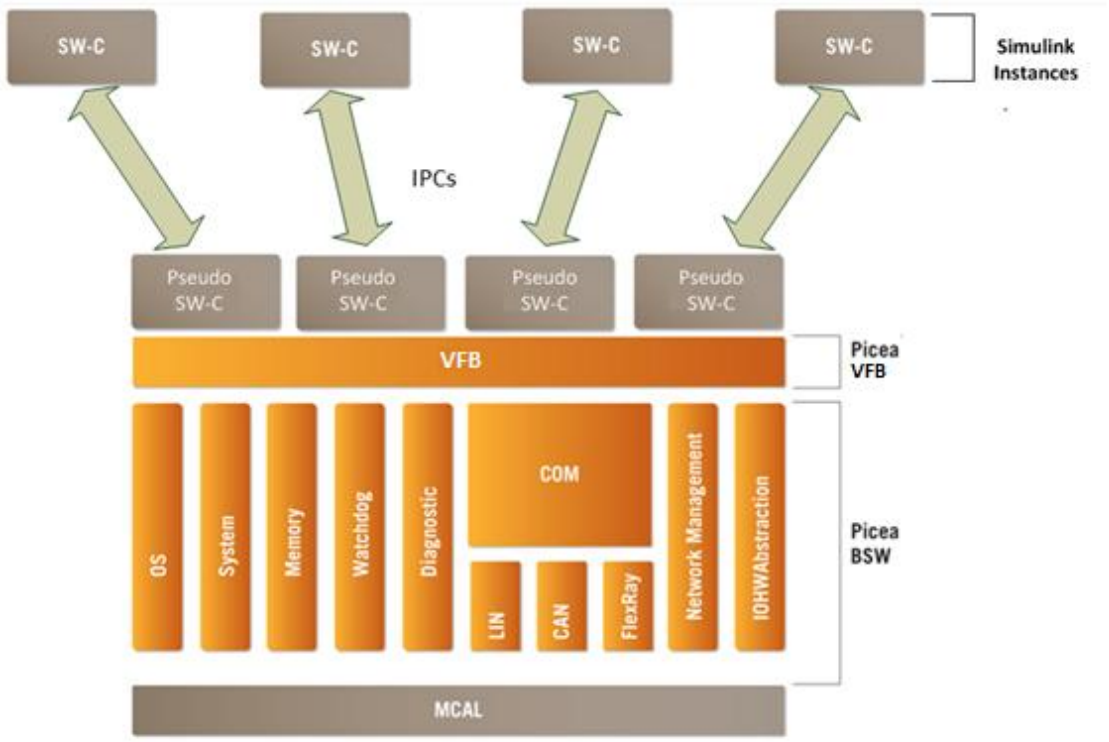

Figure 1.2: Overview of the proposed implementation.

1.3 Problem statement

The problem can be stated as follows:

Use the AFFE-light system provided by Mecel to:

- Investigate which kind of Inter-process Communication (IPC) techniques can be used between Simulink and the in house developed Picea AUTOSAR platform and select two of the most significant ones.
- Compare and contrast performance of different techniques with each other and with the original model which uses generated and compiled code.
- Design a framework that can be can be easily expanded upon in the future with minimal configuration changes.

1.4 Scope and Delimitations

The project focuses on the VFB, Operating System (OS) and Application Layer of AUTOSAR. The other layers of the architecture are not discussed further here. Software Components (SW-C) developed in Simulink as well as a demo application of Picea was made available by Mecel at the beginning of the project. These were used as baselines to create the solution though the solution aims to be more generic.

All the tools and software obtained are assumed to be in working order. No further testing is done with regards to the Picea demo product and the Simulink models obtained. The justification for this is that the AFFE-project is quite mature and issues with correctness are not anticipated.

Only two of the many IPCs possible will be chosen to be implemented. The implementation of communication with other IPCs is out of the scope of the thesis.

2 Theory

In this chapter, a brief history and a review of AUTOSAR and Simulink will be presented along with a few lines on how they are currently used in tandem by the industry to develop software. This section will also discuss the different IPCs available and give an overview of them.

2.1 AUTOSAR

The AUTOSAR platform is an open, standardized automotive software architecture jointly developed by a consortium of automobile manufacturers, suppliers and tool developers. The goals behind doing so were the following:

- Management of E/E complexity associated with growth in functional scope
- Flexibility for product modification, upgrade and update
- Scalability of solutions within and across product lines
- Improved quality and reliability of E/E systems
- Fulfillment of future vehicle requirements, such as, availability and safety, SW upgrades/ updates and maintainability
- Increased scalability and flexibility to integrate and transfer functions
- Higher penetration of "Commercial off the Shelf" SW and HW components across product lines
- Improved containment of product and process complexity and risk
- Cost optimization of scalable systems [1]

AUTOSAR aims to separate the application from the infrastructure. One of the design goals is to provide a common basic infrastructure to all the developers of software so that they are not vendor specific anymore, are more portable and reusable thus allowing different vendors to cooperate on standards but compete on the implementation. Having this basic infrastructure helps reduce costs by reusing parts of code across different projects and if needed, organizations. It uses a component based software design model for the design of a vehicular system. All the application software components interact with each other through an entity called the Virtual Function Bus. The VFB abstracts from the application software all the complexities of communication and provides a standardized interfaces for inter component communication and with the operating system itself. This separation is provided by adding extra layers of abstraction between the components. Figure 2.1 shows the basic structure of AUTOSAR.

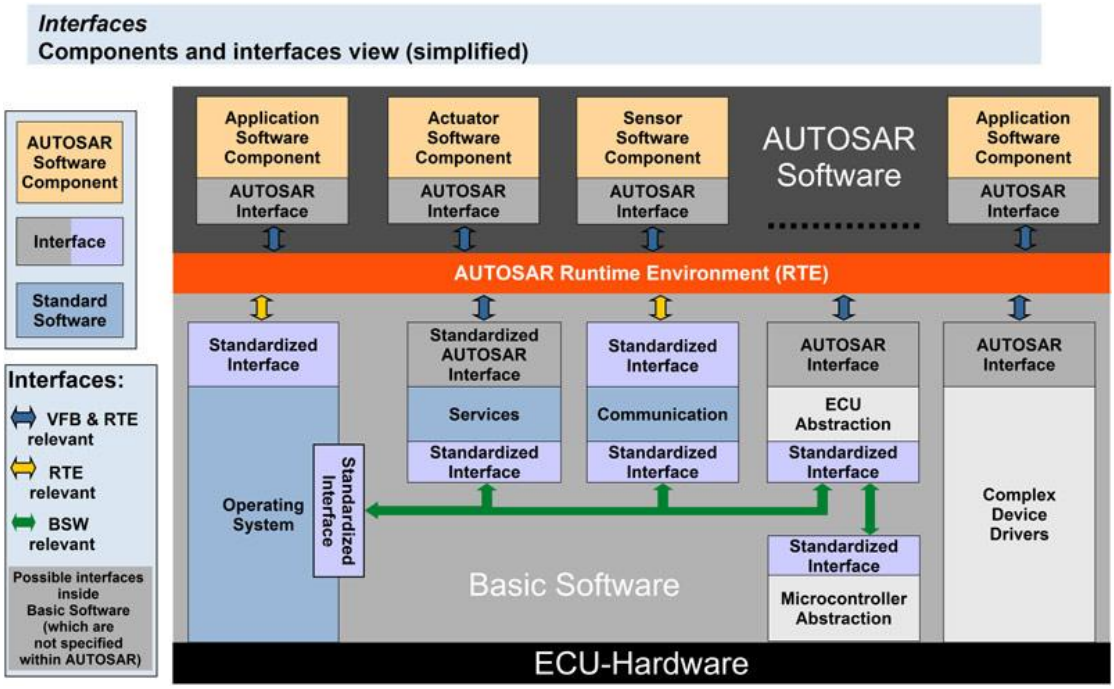

Figure 2.1: Basic AUTOSAR architecture [1].

Above the RTE, the software architecture style changes from layered to component-based from the application layer and downwards. The functionality of the system is mainly encapsulated in application software components (SWCs) [1].

2.1.1 Application software components

A SW-C in AUTOSAR is an application or part of an application that is designed for a specific purpose in a car e.g. to control a windshield wiper. Each component encapsulates part or whole of an application. A large SW-C can be viewed either as comprising many smaller components each designed with a single functionality in mind or as an independent component itself [3].

2.1.1.1 Runnables

A SW-C consists of one or many runnables that perform certain tasks. These runnables are called upon by the RTE. There are different commands, or events, sent from the RTE to trigger a runnable. A timing event can be used to periodically execute a runnable whereas a data received event triggers a runnable based on data received. This behavior is as per the SW-C specification for AUTOSAR [3].

2.1.2 The Virtual Function Bus

In order to fulfill the goal of reliability, the SW-Cs are implemented completely independent of hardware. All the communication in and out of the SW-C goes through the standardized interfaces using the API provided by the VFB. The developer of the SW-C can thus ignore the specifics of the underlying hardware and focus just on the application itself. Thus parts of the work involved in integration can be done much earlier than other comparable development processes in AUTOSAR [4].

2.1.3 Runtime environment

The RTE implements the VFB on ECU level. It provides communication capability between SW-Cs as well as the necessary services for the SW-Cs to execute. It provides capabilities for communication between SW-Cs on the same ECU as well as communication between SW-Cs of different ECUs. Whether inter or intra communication is applied the RTE provides a communication abstraction to the SW-Cs attach to it. This is done by providing the same services and interfaces to the SW-Cs whether using inter communication channels such as FlexRay, CAN, LIN, etc. or simply intra-ECU communication. The RTE is generated on a per ECU basis [5].

2.1.4 Basic Software

The BSW is a layer located below the RTE in the architecture. The BSW allows the RTE to communicate externally with other ECUs over a bus network like CAN, FlexRay or LIN. The BSW contains both standard and ECU specific software components. The standard components include for example the system service such as NVRAM and memory management. It also includes a communication framework with e.g. FlexRay, CAN and LIN. The BSW also includes the operating system and input/output services [6].

2.1.5 Operating system

The AUTOSAR OS is based on the industry standard OSEK OS and shares many of its characteristics. It is responsible for scheduling the tasks which the RTE has mapped runnables to. The resources managed by a specific OS implementation are defined within a configuration file of the OS [7].

2.2 Model-based development and Simulink

Model-based or model driven development is a modern way of developing complex systems in the automotive world. A model-based system at the core increases the visibility and increases understanding of the solution thus helping engineers detect potential problems early on during the development phase. This greatly reduces the development cost and time and increases functional safety. A model is usually in a common format which can be comprehended much easier by any other person working on the project and thus opens up new vistas for knowledge sharing and peer review. The advantage in savings by using model-based development over hand written code can be as much 36% according to some studies [8].

There are a few tools currently used to design models. However, the software used most often by the industry is Simulink. Simulink offers a quick and easy design and an optional licensed code generator that can be used to generate AUTOSAR compliant code [9].

The code generated by Simulink's 'AUTOSAR target production package' is integrated with a simulator so that the system can be simulated on the whole and tested before being implemented it into hardware.

Simulink has a graphical environment and provides many built-in as well as customizable blocks as part of various libraries and toolboxes. It has sets of predefined blocks for communication, signals, inputs/outputs, etc. [9].

With Simulink, it is possible to create different environment models and SW-Cs such as Wheel Nodes (WN) and Control Units (CU). With the help of these models and add-ons such as Simulink's code generator, it is possible to generate C-code and implement into real-time systems.

2.3 IPC

Interprocess Communication (IPC) methods are different ways in which data can be exchanged between processes. The processes can be either on the same computer or on different computers connected by a network. One of the requirements of the thesis was to find one way to connect Picea and Simulink for communication between programs running on the same computer as well as another for the programs to be able to communicate in a distributed system.

2.3.1 File mapping and shared memory

Using file mapping gives the process the ability to look at the content of a file as if it were a block of memory in the RAM. This is done by loading the contents of the file provided using the file handle as an argument for the function that is used to create the shared memory. The file on disk that is referred to by the handle is then loaded into the memory and can be traversed in a way similar to a memory location. This provides for the ability to modify or simply look at what data exists by the use of simple pointer operations. If there is a multitasking environment it is important for the processes to uses some kind of synchronization object such as a semaphore.

A special case of file mapping is called shared memory. In shared memory, a NULL is passed in place of the file handle to the physical location of the file, indicating to operating system that this file exists only on memory. Shared memory is an efficient way of passing data between processes. One process creates a memory location which the other process can then access. Shared memory is a very efficient way of passing information between programs, since the shared memory method only utilizes the RAM of the system. A drawback with this method is that the processes must exist on the same computer [10]. This is one of the methods implemented in this project.

2.3.2 Remote Procedure Call

RPCs give applications the possibility to call function remotely. This IPC can be used between computers in a network or on a single computer. RPCs can be used to connect applications on different operating systems. RPCs have a high performance rate for a network based system as the clients and server are tightly coupled [10].

2.3.3 Pipes

Pipes can be of two distinct kinds, anonymous or named pipes. Named pipes are used when transferring information between processes that are not related or between processes on different PCs. A named pipe server typically spawns a named pipe with a

common name or the name is sent to the clients. The named pipe server process is in charge of access to the pipe. If a named pipe client process knows the name of the pipe and has access to the pipe, it can open the other end and start communication. After a successful connection between the server and the client they communicate using read and write operations.

Anonymous pipes are used when the processes are related. A common application of anonymous pipes would be when they are used to take the standard input or output from a child process and deliver it to the parent process. Since the pipes are one-way by nature, in order to enable duplex communication, two pipes need to be used. The child process reads from one pipe and writes on the other. Similarly the parent process reads and writes on the opposite pipes. Anonymous pipes cannot be used between unrelated processes or over a network [10].

2.3.4 Mailslot

This IPC is a one-way communication method. There are two distinct entities in this method; mailslot clients and mailslot servers. Any application creating a mailslot is a server. The clients send data to the server by writing in its mailslot. Any incoming messages are queued up at the mailslot. They are saved at the mailslot until the server has read them. For this kind of communication to work both ways, a process has to function as both a server and a client i.e. each process should have at least two mailslots open.
It is possible to use this kind of communication in many ways. The client can for example send messages to its local computer or to a computer on a network. It is even possible for the client to send broadcast messages. Broadcast messages are limited in size where the messages to single host only have the restriction decided by the mailslot server [10].

2.3.5 Socket Communication

Another type of IPC is the Socket Communication. A socket is characterized with an IP-address and a port number. Based on this information the socket delivers the data packet to the right thread or process [10].

2.3.6 Clipboard

Using the clipboard is a way to create a central storage for applications sharing data. Whenever a copy or cut operation is done by a process, it stores the data on the clipboard in application-defined or standard format. When any other application wants to fetch data from the clipboard it chooses between the different formats it supports. The applications need only to agree about which formats that are going to be used. This makes the clipboard a very loosely coupled exchange board. The clipboard IPC can be used between applications on the same computer as well as on different computers in a network [10].

3 Tools

This chapter discusses the software tools used in this thesis.

3.1 Simulink

As mentioned earlier, Simulink is a commercial tool for modeling, simulating and analyzing dynamic systems. It has a customizable set of blocks that can be used to simulate a wide variety of systems. Simulink offers a tight integration with MATLAB and can be scripted from it. In the automotive communications domain, Simulink is used to model systems so that AUTOSAR code can be generated from it using the embedded coder add-on [9].

3.2 Visual C++ Express and Mecel Picea Demo

Visual C++ Express is a free Integrated Development Environment (IDE) tool from Microsoft [11]. It is a light weight version of the Microsoft Visual Studio product line. The idea with the express edition is to provide an easy-to-use and easy-to-learn environment. The full version of Visual C++ has to be paid for and offers many more features. However, this was not required for the system. The Picea demo version used in this project was developed and modified in Visual C++ Express.

The Picea demo was built using the Mecel Picea suite including its VFB simulator and the code generated from the AFFE-light models. The final implementation comprised about 13000 objects including all the archives used. The VC++ IDE is used to work with these files to debug and modify the code as required.

3.3 Mecel Picea Testbench

Picea Testbench is a tool used to simulate, at a very basic level, the environment around the vehicle and the actions of the driver. It does so by communicating with the vehicular system itself. This is achieved by sending CAN frames to the Picea VFB simulation. The CAN frame consists of a number of bytes which can specify as inputs. The CAN frame in the AFFE-light system is used to communicate the following attributes of the environment to the system:

- Angle of the steering wheel
- Max torque for the wheels
- Angle of depression of the gas pedal

The information in this frame is used to decide how the car is moving. The system processes these inputs and gives as outputs the calculated values of

- Torque for left and right wheel

Thus though Picea Testbench does not give an elaborate replication of the environment, it provides a basic testing ground for the logic used in the vehicle.

4 System overview

This chapter aims to give a description of the system used for the thesis.

The suggested AFFE project implementation consists of two control units (CU) and 4 wheel nodes (WN) as shown in figure 4.1.

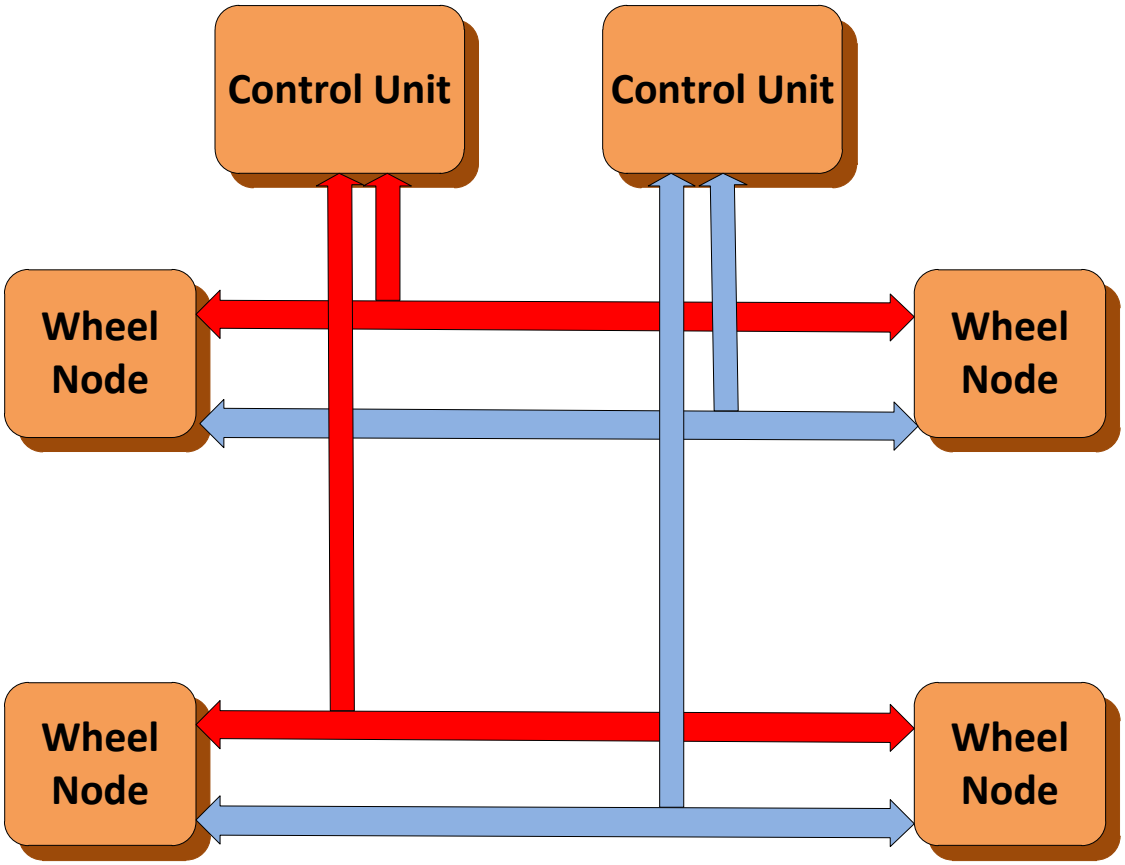

Figure 4.1: The original AFFE system.

However, as the idea was still in the planning stage, the decision was made to keep the solution simple and a smaller version of the system called the AFFE-light system was used instead. The AFFE-light system is shown in figure 4.2.

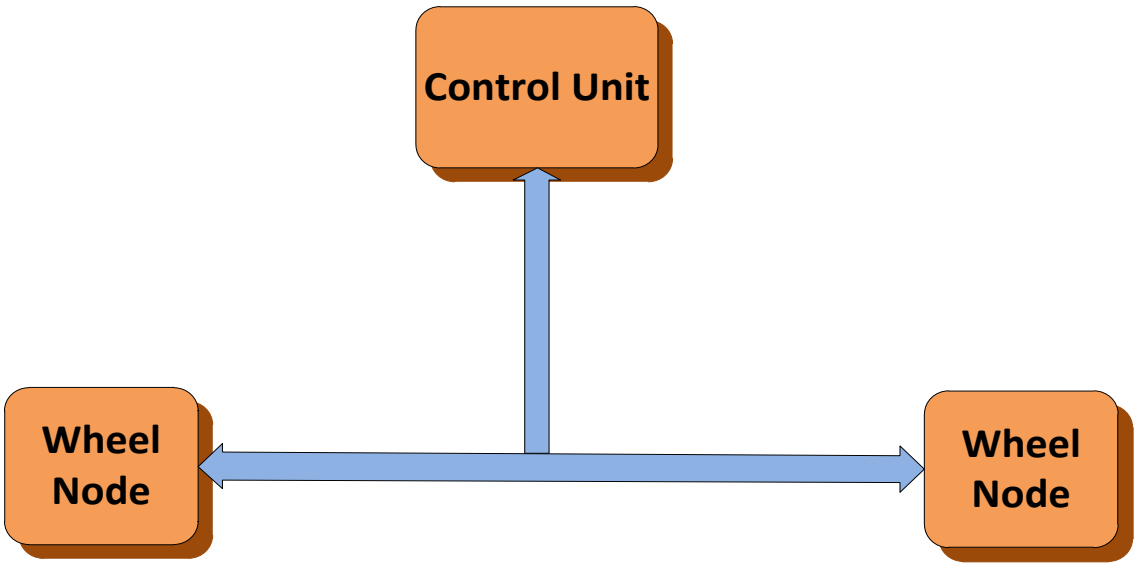

Figure 4.2: The smaller AFFE light system.

The AFFE-light system includes a CU and two WNs supported in software by the VFB in Picea's implementation of the AUTOSAR system. The CU as well as the WNs was provided as is and their internal structure was not changed. The AFFE-light system that was provided had about thirteen thousand files in total but only a few of these had to be modified as most of them dealt with the lower layers of the architecture.

The models that are used in this thesis are discussed in detail in the following sections.

4.1 Control Unit

The Control Unit (CU) takes as input, the angle of depression of the gas pedal, the max torque allowed for the Wheel Unit (WU) and angle of the steering wheel. Based on the gas pedal angle and the max torque, the CU calculates the speed for the wheel and sends it to the WUs. The steering angle is directly passed to the wheels. Inside the CU there are three runnables as shown in figure 4.2. The first runnable checks the status of the engine, if an error has been reported the engine is stopped but if not everything continues as normal. The second runnable, as shown in figure 4.3, takes gas pedal angle and max torque as input. It then uses a lookup table to convert the gas pedal angle to obtain the right value of torque that should be applied. A third runnable uses the steering wheel angle as input and just passes it on to the wheels.

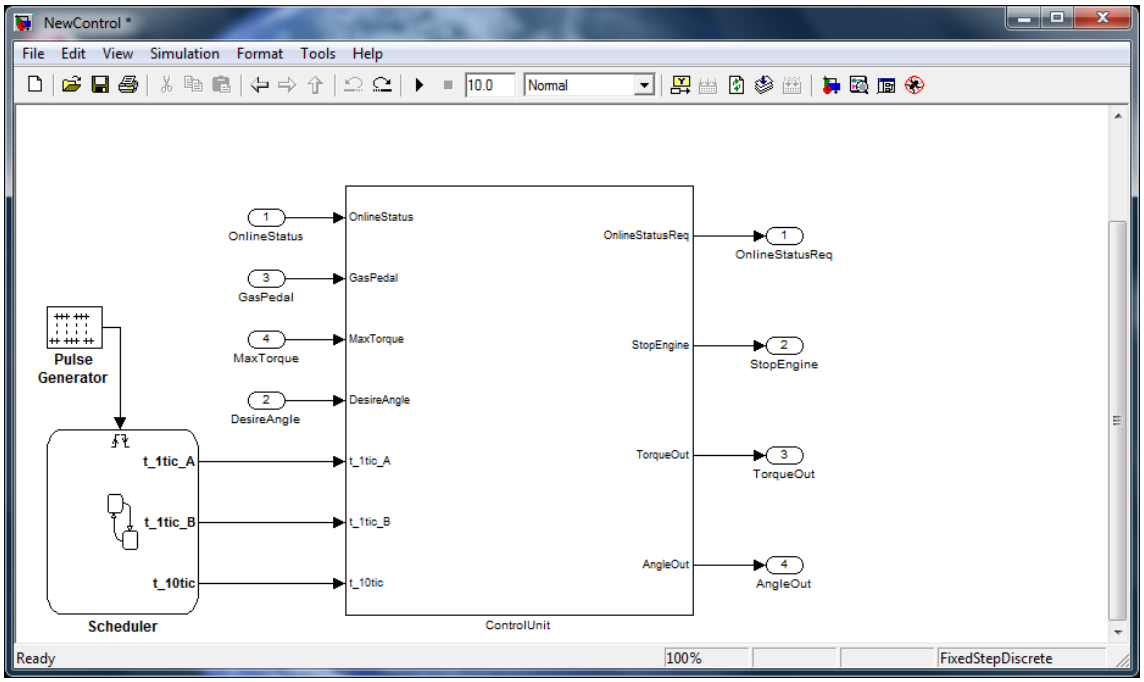

Figure 4.2: Simulink model showing the runnables of the CU.

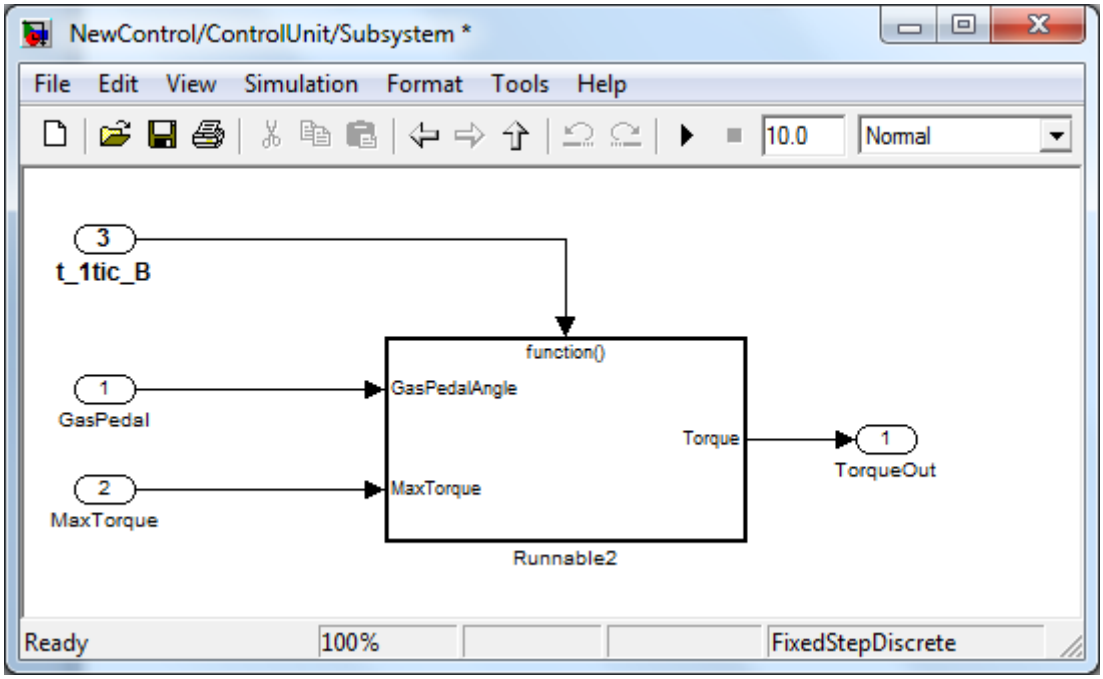

Figure 4.3: Inside a runnable of the CU.

4.2 Wheel node

Inside the Wheel Node (WN) a runnable calculates the torque that the wheel should generate. The wheel node model is shown in figure 4.4. The torque is calculated based on the value received from the CU as well as the inputted steering angle.

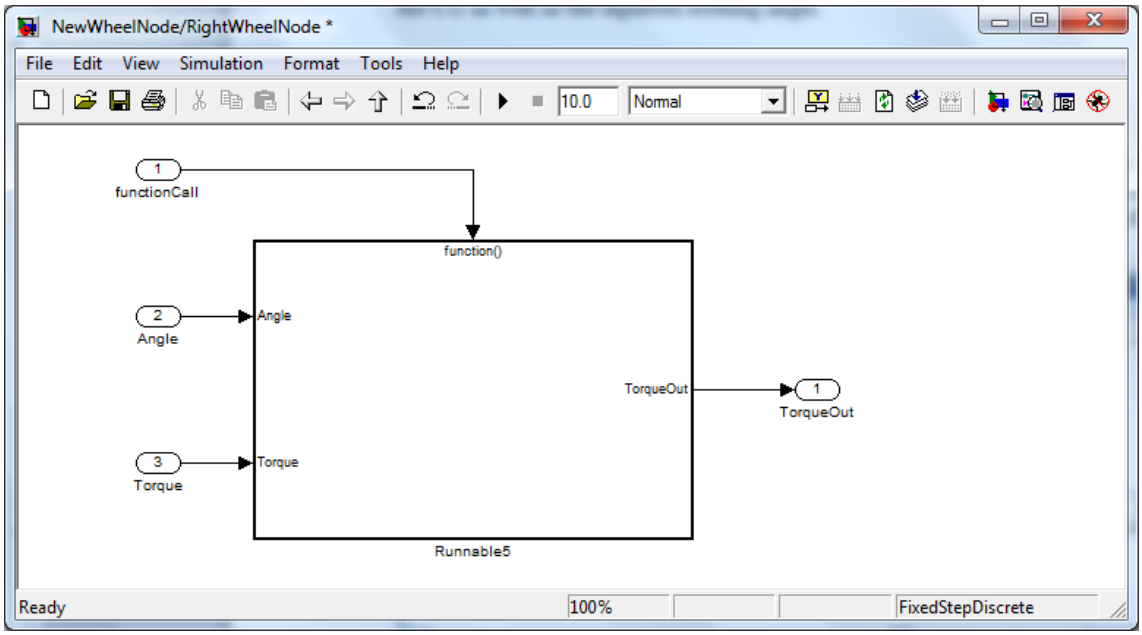

Figure 4.4: Overview of a wheel node in Simulink.

Inside the wheel node there is an algorithm to calculate the actual torque to give out. This algorithm takes steering angle and torque as input. The steering angle is important e.g. if the car is turning left the left wheel should output less torque because of a shorter distance to travel. The runnable can be seen in figure 4.5.

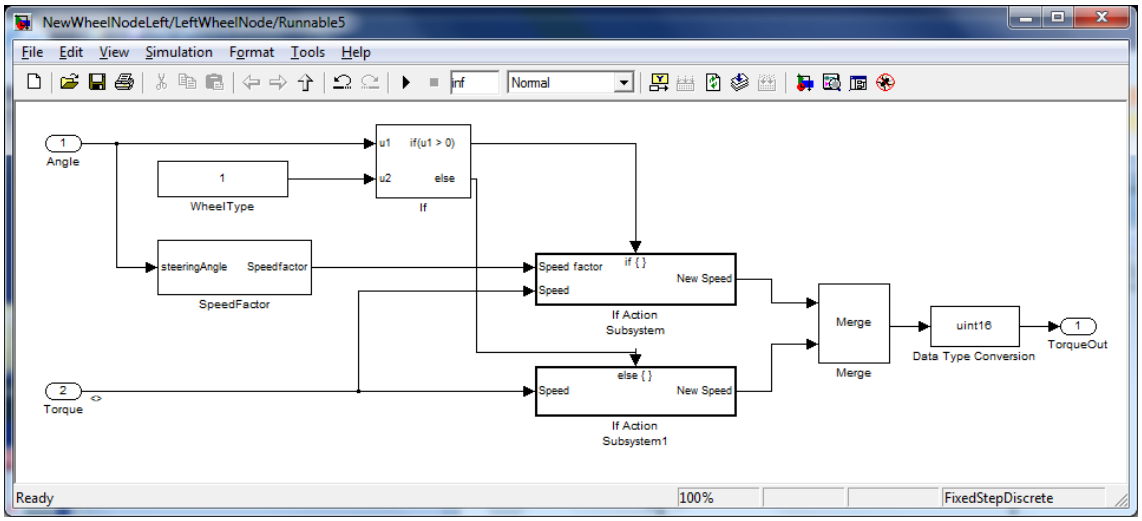

Figure 4.5: Runnable in the wheel node.

5 Design

In this section, the requirements for the final solution are identified and the final communication protocol is discussed.

5.1 Initial investigation

To get an understanding of the project an initial literature study of AUTOSAR, Simulink and the concepts of AFFE was done. Information about AUTOSAR itself, how the AUTOSAR VFB works as well as information about how AUTOSAR uses SW-Cs had to be studied.

Since Simulink is one of the two major applications used in this thesis, a more detailed study about Simulink was needed. Information about how Simulink works and what its capabilities and limitations are was provided by Mathworks [9].

5.2 Requirements

The following requirements were identified for the final solution.

1) Control of the order of execution of the components should remain within Picea.
2) Of the original system, only the runnable files should be changed. No outside modifications to the RTE or the BSW are permitted.
3) Picea and Simulink should execute mutually exclusively. There should be no competition between Picea and Simulink for use of the processor since the simulated system only uses one core.
4) The solution should be scalable and it must be possible to reuse the work done in this project on much bigger projects without heavy configuration and huge changes in the logic.
5) The Simulink models should not be altered in any fashion which may cause the logic in the runnables to change.

5.3 Methods chosen

The requirement was to choose one method optimized for running the whole system in one computer and a distributed method. The two methods chosen to be implemented were shared memory using file mapping and the TCP/IP method.

Shared memory was chosen because it was shown in documentation to be one of the fastest methods of IPCs because the data is never written to disk and stays in the memory.

TCP/IP was chosen as it was the simplest of the distributed method. Some of the other IPCs discussed, used TCP/IP at a lower level themselves and thus would have been slower.

5.4 Communication Protocol

In order for Picea to be able to distinguish between the information and know which functions to call every message from Simulink to Picea includes an ID. The S-functions in Simulink, as will be explained in more detail later, are mapped to specific functions in Picea through unique IDs.

The communication protocol used between the two programs viz. Simulink and Picea is as follows:

Only a single integer of data is sent between the two programs. Pointers are used to address specific bytes of the integer and modify them. The contents of the individual bytes are shown in the table 5.1.

Byte Position	Significance
1	ID of the function in question
2	Used to send data from the block to Picea if applicable.
3	Used to send data from the block to Picea if applicable. (Currently used only by 16 bit data types)
4	Currently unused.

Table 5.1: Content of the different bytes in the frame sent

The first byte is used by all the communication blocks in the Simulink system. The second is used only if it is connected to one of the output ports of the model while the third is used only if the data type that the model outputs is of more than 8 bits.

In the AFFE-light project, a maximum of 16 bits were used by any port. Thus there was no need to use more than 2 bytes of data for carrying the content of the processing done by Simulink. However, this byte can be used in the future, with minimal changes, to carry some extra information e.g. carry error codes or to carry larger data types.

In the protocol used in the initial versions of the system, to test proof of concept etc., character strings were used as they were the most comfortable to work with. The newer protocol was chosen to avoid the overhead of using string functions in C and because characters are not a datatype native to Simulink.

6 Evolution of the Solution

This section describes the phases in the development of the solution and tries to explain the reasoning behind the structuring of the final solution. The routines used in shared memory communication as well as in the TCP/IP solution can be found in appendix C and D.

6.1 Standalone programs

To begin with, two standalone programs using the windows API for C which could interact with each other were created using the selected IPC methods. The idea was to keep the code as close as possible to the final version and reuse it with minor modifications as and when required in the next iteration of development and integration. These programs were used as baselines for testing the correctness of the code integrated with Simulink and the modifications made within Picea.

The next step after the creation of the programs was to use them to send and receive simple data from Simulink and Picea individually. To do this in Picea was relatively easy as the code could be reused with little modifications by editing the rte.c file to call the server function as well. Nonetheless, one problem that was noticed was that Picea could not handle a blocking call in the RTE thread for more than a small period of time. This problem was temporarily solved by using a polling type were RTE periodically checks for signals set rather than a blocking solution.

To test Simulink for data communication however was a little harder. Although there are built-in blocks in the Instrument Control Toolbox of Simulink which can be used for socket communication, they had limited flexibility. Simulink for example does not have any built-in block that can act as a TCP/IP server. Thus it was tested only as a client in the initial stages.

6.2 S-function builder blocks

System-functions, or S-functions, are computer language description of a Simulink block. This block can be used to extend the use of Simulink environment. The S-function can be written in C, C++, FORTRAN, ADA or MATLAB [12].

Writing an S-function is much like writing a normal function in for example C. The S-function includes definitions, initial conditions, input and output, termination conditions as well as normal #includes. Every time the S-function is reached in the block diagram the function inside is called and executed.

The first proof of concept was shown by using Simulink as server for the component for the control unit runnables and as a client for the other runnables in the wheels. This was done by the S-function builder block method and using the outputs field in the GUI for running the logic of the system. Picea was used as a server for the other runnables to contact, and process data.

Though hand written S-function provides the largest feature set to aid the integration of generic C code, it is a relatively complex task to write one. The S-function builder block allows for a smaller set of features but a much faster way of integrating existing C code with Simulink.

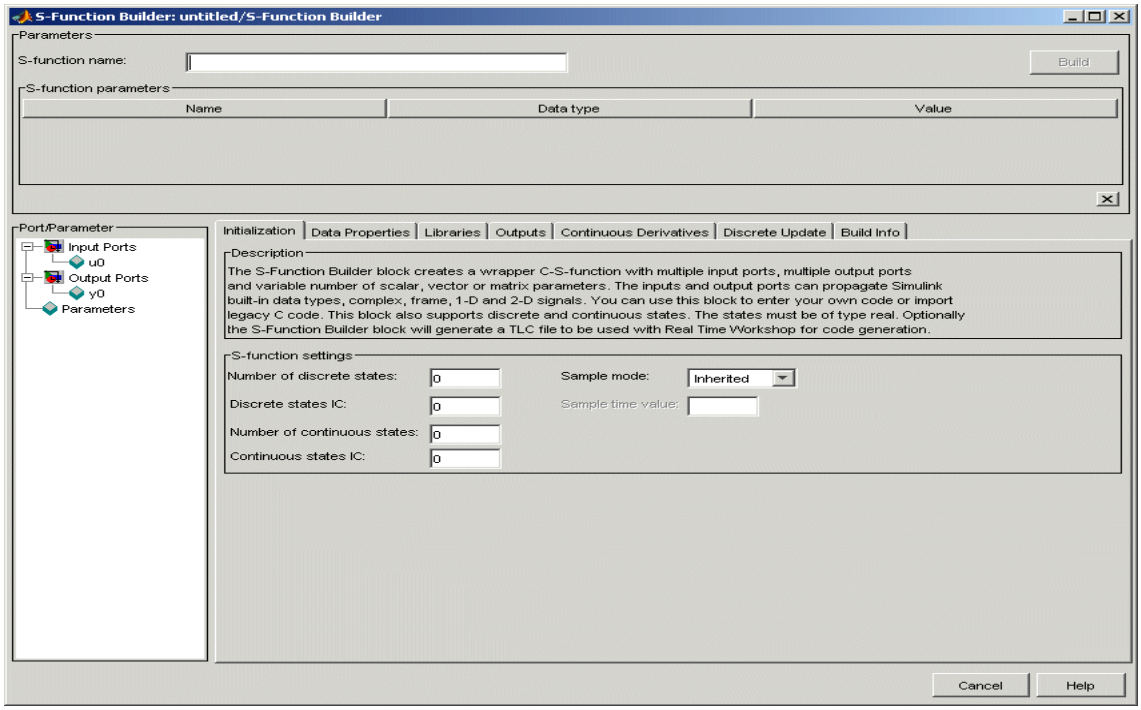

Figure 6.1: Basic S-function builder overview.

The S-function builder block in Simulink allows the user to place a block in the Simulink model that will incorporate normal C code [13]. It also allows the user

1) A place to assign a unique name for the function.
2) To select the number of input and output ports and their datatypes.
3) To select the number of states in a system and their types.
4) A specific place to enter the paths or the names of the header files required.
5) A specific place to enter the logic to calculate the output for the block for each stage of the process.
6) A place to select whether a mdlStart() and mdlTerminate() function should also be generated. These can be edited later by the user directly as required.

The block, when run, would use the logic provided by the user to generate outputs [14].

S-function builder block was by far the easiest method to implement the system. However, it did not give the fine grained control required for optimizing the solution

and the values of the IDs had to be hardcoded into the files. The implementation obtained by using this method was inconvenient to use because the end-user had to manually copy in code per block, choose the data types for each block and compile by clicking on the build button in the block. This step could not be scripted easily and would be very time consuming for larger systems.

There was some work required to ensure that the sockets shutdown immediately as windows was prone to let the port linger for some time after a closesocket() was called and a lack of persistent memory meant that the program could not dynamically assign a new port for itself. This problem was solved by explicitly calling shutdown() with instruction to end both sending and receiving on the socket before the closesocket() call.

Though this successfully demonstrated the objective of the system i.e. to connect Simulink and Picea, the system itself was slow as the program opened and shutdown sockets for each block for each run of the Simulink model. There were also other problems with the requirements set that control of execution should lie with Picea which said that Simulink should at no point act as a server since in the control unit, a server was required in Simulink to communicate which runnable has to be run in this system..

Since the whole system had 14 S-Function Build blocks in total, this gave 14 different files that had to be modified for each change and built at every run, by manually clicking on the 'build' button in the S-function dialog boxes. This way of working got after a while too complicated and was not deemed to be scalable.
An additional problem was that because access to the other functions of the Simulink block was not available, the program had to be designed to be stateless. This meant that at each execution cycle, a variety of steps that should ideally have been only done once per simulation had to be repeated. Hence there was a lot of overhead which considerably slowed down the system.

Thus, though this method was used to implement the system and show sufficient proof of concept, the final system implementation was done using hand written C S-functions for better performance.

6.3 Hand written C S-Functions in C

The slow performance seen required an overhaul of the system as it was at that point. A decision was also made, after a meeting with the supervisor of the project, to change from the S-function builder block method to another method that would not involve manual building of files per block.

The different methods to do so were looked into and the one selected was using pure C S-function files [12]. These allowed access to parameters of the whole Simulink block and allowed doing nearly everything that a native Simulink block would do. It also came up that as this method involved compiling the function into a MEX file at the beginning of the project, this was actually faster to use than the S-function builder block method which involved opening each block and compiling it once. The fact that the same file could be reused for multiple blocks using this method, would greatly enhance maintainability.

It is possible to ‘mask’ the S-function making it more user friendly with definition of a custom interface for the user.

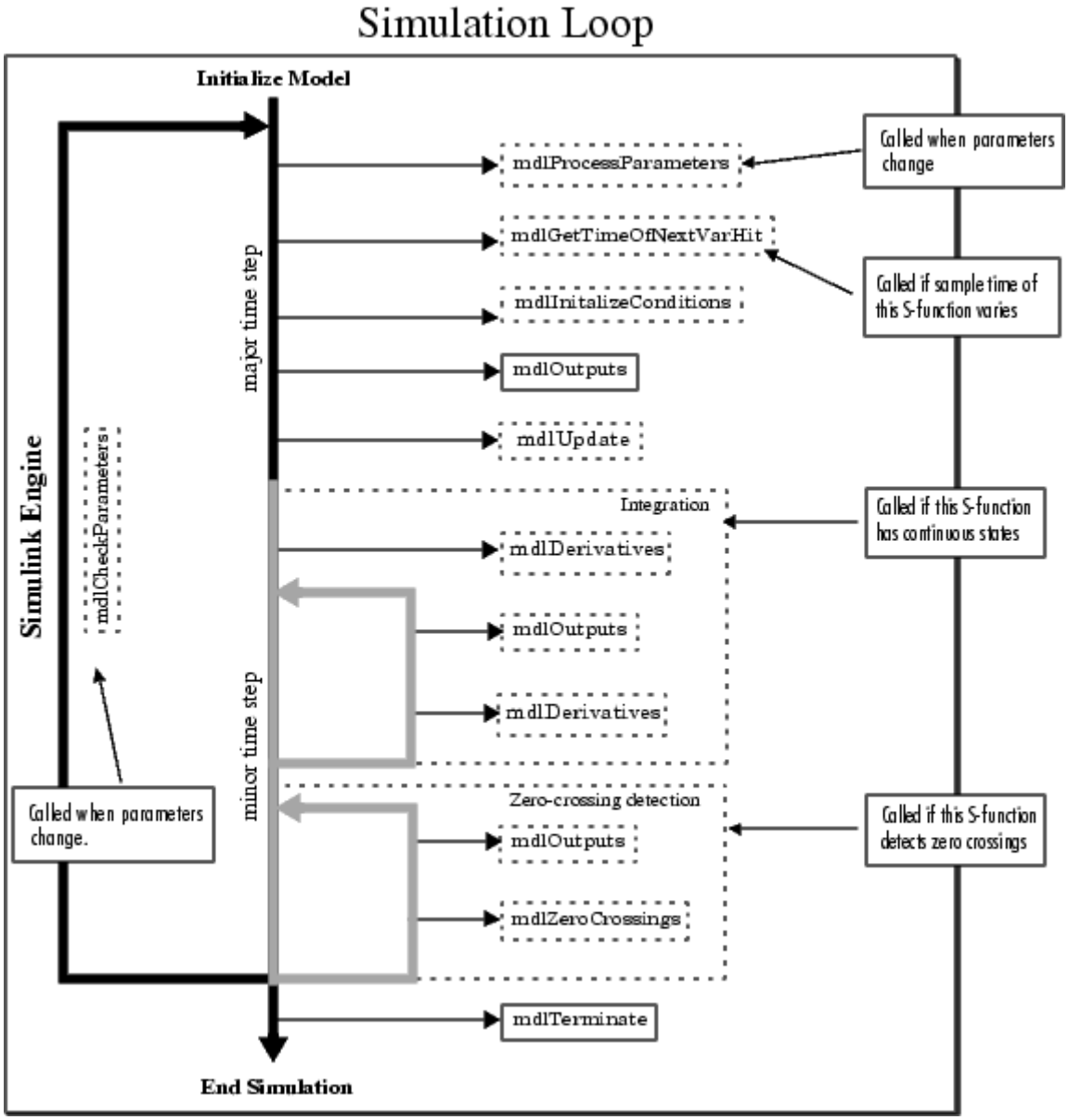

Figure 6.2: Order of execution by the Simulink engine [15].

Figure 6.2 shows the order in which the Simulink engine calls the implemented functions during the course of execution. A Simulink model is executed in stages. The first phase is the initialization phase where the signal widths, data types and sample times are propagated. The block execution order and memory allocation are also performed at this stage. Each simulation step involves executing the blocks in the order determined initially. These simulation steps are performed in a cycle till the simulation is complete. The MEX S-function consists of a set of methods that can be invoked by the Simulink engine directly during the different phases of execution. The S-function

methods have to comply with naming conventions as they are called directly by the engine.

6.3.1 Defines and Include statements

The S-function code starts with the following #define statements.

#define S_FUNCTION_NAME <name of the S function>

This is used to define the name of the S-function.

#define S_FUNCTION_LEVEL 2

This is used to define that the code will use the S-function level-2 format.

#include "simstruc.h"

This header is used to access the SimStruct data structure's and the MATLAB application's API functions.

The headers that are used in the program, such as the *windows.h* for the shared memory block and the *winsock.h* for the TCP/IP block are included after these lines.

6.3.2 Routines used

mdlInitializesizes()

The Simulink engine calls this function to gather information about the number of input and output ports and their datatypes. It is also used to set the number of parameters to be accepted by the function [12].

mdlStart()

The use of this function is optional. It need not be implemented for the block to be compiled or executed. However, it is used in this implementation to perform steps that are ideally done only once. For e.g. in the TCP/IP block, the mdlStart method is used to set up the otherwise expensive connections between the Server at Picea and the blocks at the start of the execution. This step would otherwise have to be performed once per execution cycle in the S-function builder block method [12].

mdlOutputs()

The implementation of this function is mandatory and it is called by the engine at the beginning of each time step to calculate the outputs of the block. This incorporates the core logic of the block and is presumably the most important function in the program. The implementation chosen relies heavily on this function and the most part of all communication between Picea and Simulink is handled here [12].

mdlTerminate()

This function is used to perform tasks at the end of the simulation. The routine can be used to free up resources, break connections and generally restore the system to the pre simulation state [12].

6.3.3 Compiling an S-function based C file.

A complete S-function C file can be compiled from the Matlab interface using the following command once the compiler is set up correctly.

mex <function_name> [13]

6.3.4 Working

Each block in the model acts as a client to the server that Picea has started. Therefore it is essential that Picea is started before the Simulink model.

The mdlStart() method in Simulink is used to initialize the variables and establish the connection to Picea. Since Simulink executes each block of a model in a specific order, there is no danger of any block executing out of turn within a model Simulink but there is no synchronization between the models open in different instances of Simulink. Thus there may be incorrect ordering between blocks from different models. This condition is handled using the server at Picea to synchronize order of execution between the threads.

When each block gets its turn to execute, it calls the mdlOutput() function which executes the bare minimum set of instructions that are required as most of the work is done exactly once when the model's execution starts.

This new general block used is instead simply called S-Function. This block takes the name of the S-function C file as input. With access to the full S-function API, it became possible to homogenize the whole solution and reuse the same C file for each block.
The block was then masked so that the details of the block would be abstracted away from the user. The following section explains the reason behind and the details of the masking mechanism.

Parameters and masks

In the builder block method, the project IDs were hard coded into the S-Function Builder. This was suitable for testing and for verifying the mapping of the IDs to functions in Picea. This way of working could obviously not scale and had to be changed to a more user friendly interface. Simulink masks were used to minimize the configuration by the user and to centralize the location where the changes are to be made.
In both the IPC methodologies selected, to be able to handle the mapping of IDs to Picea the user is prompted to input the ID for the particular block and what data type the block uses as well. There are also specific options to be selected based on the methodology. For example, the shared memory block also has a signal type option that can be used to select which set of signals or memory would be used and the TCP/IP block allows for entering the IP address and the port of the server at Picea.
By selecting what type of SW-C the block is a part of, different settings will be loaded. Figure 6.3 shows the customized user interface of the shared memory blocks.

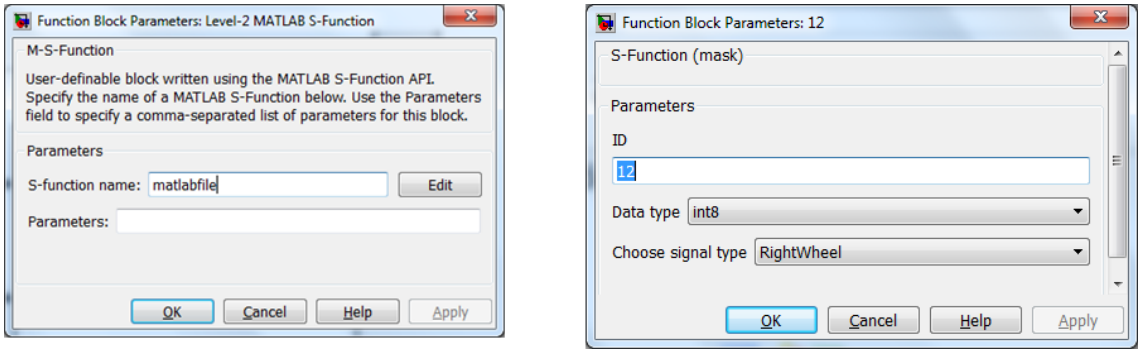

Figure 6.3: Original and modified user interfaces.

Using masks abstracts away details of the block that the user need not modify. It gives one central point for all the changes to the block to be done and makes the configuring of the system much faster [16].

6.4 Final Implementation

The final implementation uses S-functions written in C. Each input and output port in the models are replaced with one of the new Simulink blocks created. These blocks are assigned unique IDs and the datatypes to be used by them are selected from the dropdown menus on their GUIs. Settings specific to the methodologies such as the IP address for the server for TCP/IP or the type of Software component for the shared memory system are then entered into the GUI. This completes the work to be done in configuring Simulink.

In Picea, the following modifications are done.

i. From the StartOS() method or earlier in the execution of the program, the TCP/IP server is started.

ii. The mapping of the IDs used by a runnable is implemented in a separate file for each runnable. These runnables are called by the run function based on the ID received.

iii. All signals used in the system are reset in the StartOS() function. This is to prevent unintended behavior of the system.

iv. Picea was given entire control over the scheduling of the system. The earlier versions of the solution had Simulink acting as a server that the RTE thread communicated with to pass the ID of the runnable to be run. This could not be possible once the Simulink blocks were homogenized and necessitated this change.

v. The threads that were started by the alarms originally used the THREAD_PRIORITY_TIME_CRITICAL priority. However, while the solution was not yet completed, this caused the PC to become non responsive. To counter this, the THREAD_BASE_PRIORITY_MAX priority was use instead during the development phase.

vi. In the initial phases of the project, the RTE was modified to call the communicate files directly. But to get to the goal limiting modification in existing files to the runnable files, this had to be changed to be called from the runnables itself.

vii. Earlier, the memory locations for the shared memory system were kept open by running an external program. This was no longer required in the final implementation as the Simulink model would leave these locations open.

6.4.1 Shared memory

Communication using shared memory was the first method implemented for the project. The motivation behind the choosing of this method was that shared memory is known to be the fastest way for two processes to communicate whilst on the same physical computer. The shared memory location stays on the RAM of the system and utilizes lightweight windows events called signals to synchronize data transfer between the programs. This methodology has some shortcomings in that the processes must be located on the same computer and that this system is platform specific and is native to Windows only. However as said before, the advantage gained is in the speed. There are no protocol stacks to deal with and communication remains very simple and fast.

The shared memory system works in the following fashion. Both Picea and Simulink are started and Simulink creates the memory locations needed to communicate in the initialize phase. Following this, each block in the model executes its output method one by one. In the output method, the Simulink blocks first wait for Picea's signal Signal3 to start executing. Once the signal is obtained, the blocks identify themselves by ID to Picea. In case the executing block is an output block, it also sends the output of the processing of the model. Picea is signaled by using the signal Signal1 while Simulink waits on Signal2. Picea then uses the ID that was given to call the RTE function associated with this ID and sends the value returned by the function to Simulink. It then signals the availability of the data by using Signal2. The Simulink block retrieves the data, passes it to the model and then Signals to Picea using Signal1 that it can continue. The Signal1 indicates the end of the execution of a communication block and signals Picea to read from the memory. Picea knows, based on the ID, if this is the end of the runnable and proceeds.

There are thus, three signals that are used to synchronize the communication. This Signal set is specific per model. This along with the fact that a unique memory location is chosen, ensures that there is no inter-model interference in the system.

To aid this, the Simulink block designed allows for the selection of the SW-C from the GUI. Selecting from the drop down menu allows for a unique memory location and a signal set to be selected.

6.4.2 TCP/IP communication

This was the second method of implementation for the project chosen. Since Windows Signals and shared memory are limited to use on a single computer, a system that could be distributed over multiple computers was deemed to be desirable as well. This would

reduce the processor and memory load that would otherwise have to be handled by a single computer.

As per the requirement set for the project, Picea should remain in control of the system at all times. Thus the server is set on the Picea side of the system. This is done by starting a separate thread for the server which listens on the assigned port and waits for incoming connections. The server thread spawns a new thread for each of the incoming connections from the blocks used in Simulink. The reason for using different threads is that by doing this the management of the system becomes easier. The individual threads make use of blocking calls and sleep till the operating system informs them that there is data to be processed. This makes for an efficient system but leaves the problem of co-coordinating the threads so that the Simulink processes wait and do not get a chance to process their data till the VFB calls for them. A basic level of ordering is received directly from Simulink as it allows only one block of a model to execute at one point of time. However, since different instances of Simulink needed to be run at one time, there would be at exactly the same number of blocks running simultaneously as the number of instances of Simulink. Thus there had to be some differentiation done in Picea to ensure that the order of processing was what the RTE required it to be. To achieve this, windows signals were used from the calls to the run() function of the runnables to control the order of processing. All the Software Components have access to a global signal which is set when there is any runnable running. In addition to this, for every Software Component, there exists a pair of Signals to signal the start and end of the execution of the software component to the RTE. A further set of 3 signals are used within the Servers threads so that the block used for scheduling in the Simulink model gets a different value sent to it from the server. It must be noted that all signals used in the TCP/IP method are relevant and used only at the Picea side. These signals are used only for the purposes of co-ordination between the different threads at Picea. Simulink does not participate in this.

On the Simulink side, during the initialization phase, each of the blocks connects to the server in Picea using TCP/IP sockets. They identify themselves with the unique ID assigned to them then wait for data to be received from Picea to process. The Input blocks use this to provide data to the model while the output data blocks merely send the processed data to Picea.

When each server thread is allowed to execute, it receives the ID from the specific Simulink block and executes the corresponding RTE call, processes it and returns a result to Simulink. The result is then read by Simulink, reflected onto the model and the next block in sequence is allowed to run and receive data from Picea.

7 Testing and Results

This section presents the reasoning behind the development of the test as well as the design and the results.

In order to test and verify the two different solutions that were implemented, a test to measure the performance of the system was designed and incorporated into the solution. The performance of the implemented solutions was then compared to that of the original one.

The system has a total of five runnables, 3 for the CU and 1 for each of the WNs. These are called in order from the RTE.

In the original program the code for the runnables were implemented in Picea which would give none or little process switching and overhead. In the other solutions proposed in the thesis, the runnables run in their natural environment of Simulink. This would force the system to switch process and therefore generate some overhead. Since these would cause the new solutions to run slower, it was decided to use time taken for runnable calls as a measure of performance.

To make the timing as accurate as possible each runnables execution time was measured and then added with weights to the average. A cycle consists of all 5 runnables executing once each. In order to get as accurate results as possible the system ran 30000 cycles for each test run. A total of five test runs for each solution was performed and an average value was calculated. Both the new and the old solutions were tested on two different systems, one DELL Inspiron PC with an i5 processor clocked at 2.40 GHz and one DELL Studio PC with an i7 processor clocked at 1.6 GHz. These systems had 4GB of RAM installed. Figures 7.1 and 7.2 show the outcome of the tests. The results seen are average times of execution for a single runnable.

Run	**Original (time in us)**	**Shared Memory (time in us)**	**TCP/IP (time in us)**
1	4.750243	297.784798	684.335962
2	4.738396	296.058547	635.680373
3	4.835660	297.023139	689.455095
4	4.806832	299.727421	659.746134
5	4.876847	299.504264	654.943754
Average	**4.801556**	**298.019633**	**664.832263**

Figure 7.1: Table of results from DELL Inspiron (intel i5 processor).

Run	Original (time in us)	Shared Memory (time in us)	TCP/IP (time in us)
1	7.282058	527.242404	1699.509549
2	7.168050	533.386413	1675.742360
3	7.122712	510.399304	1695.908078
4	7.468927	528.337446	1698.702238
5	7.481844	536.467410	1660.177948
Average	**7.304718**	**527.166595**	**1687.608034**

Figure 7.2: Table of results from DELL Studio (intel i7 processor).

The tests were implemented in the following fashion, a function call measures the number of processor tics before the runnable starts and the number of processor tics at the point when the runnable returns. The difference is obtained and used in conjunction with the systems clock frequency to calculate the time elapsed. The resolution chosen to display the results is in microseconds.

At certain points extremely high values were seen in the tests. To be able to determine whether these values were a valid reflection of the quality of the solution, an analysis of the values was done in order to see how frequently they showed up. It was seen that they represent less than 2% of the total values. An assumption was thus made that they are results from the PC's internal behavior such as process switching possibly because of other programs being run as well. Hence, they are to be treated as noise and need not be added to the results. Thus, after determining the range of values that could be seen as valid representations of system time, the other outliers were excluded from the final calculations.

The functions used to measure the performance of the system and the code used is documented in appendix A.

Results

As seen in tables 7.1 and 7.2 the average time taken per call is higher than the original version. These results were expected since a lot of process switching is introduced compared to the original system where the execution only contains one process.

8 Discussion

This section discusses the issues faced during development. It also discusses the results from the previous chapter and explains how they were interpreted.

As mentioned during the discussion of the S-function builder block method, one major problem encountered during the initial stages was that Picea could not handle infinite waits and would crash or hang when these were used. This problem was given a temporary solution by using a polling method to check for the signal instead of using infinite waits. The main drawback with polling is that a lot of processor time is wasted; processor time which can instead be more productively used for the running of Simulink while Picea sleeps waiting for Simulink to signal it to start again.

The issue was related to the OS simulator currently used in the Picea demo. As this was designed to simulate a real time system, it expected all the tasks assigned by it to have completed by specific times and did not handle delays or blocking calls well and would cause the system to crash. This was a problem that needed to be solved as both the shared memory and the TCP/IP communication methods require the use of blocking calls. Another factor was the requirement set that only Picea or Simulink would execute mutually exclusively. Thus the implementation of the OS simulator had to be modified to handle blocking calls more effectively.

The solution to the problem was to create two routines as outlined below. The first function called a function created for the purposes of this thesis i.e. EveryoneSleeps() that would suspend every alarm threads in the system except the RTE thread. Signals within the OS simulator were used to ensure that the creation of any new threads would wait while the RTE thread was running. The other function called EveryoneWakes() could be used to reawaken the alarm threads of the system and enable the creation of the new threads as well. This essentially set a lock on the system to stop the creation and scheduling of new threads in Picea while Simulink was being run. The net effect of the steps mentioned above was that the blocking calls required for the communication methods could now be handled by the system.

In order for the system to run in real time, for this project, the execution of one cycle has to be lower than 10ms. The results from the measurements show that the new solutions both run faster than that specified by the timing requirement. This is based on how fast the environment models that will be used for the system will expect a response from the system. The TCP/IP model however comes close to the deadline on the i7 processor.

It is also shown from the results how the i5 PC outperforms the i7 PC. Because of license issues for Simulink it was not possible to test the solutions on dedicated computers and hence, the tests had to be performed on the personal PCs of the authors. These PCs had different types of programs and services running in the background which could not be stopped. The results are therefore a pointer to what performance is to be expected when testing on dedicated PCs. To have any relevance, the timing of the different methods is compared within the same system. This meaning that the comparison is done between the shared memory solution and the TCPIP solution within

either the i7 PC or within the i5 PC. One important thing to notice though is how the performance changes at approximately the same rate on the different PCs. The execution time increases 2.2 times for the TCPIP implementation compared to the shared memory implementation for the i7 PC and with 1.2 times for the i5 PC. Likewise the execution time increases 71 times for the original implementation compared to the shared memory implementation for the i7 PC and with 61 times for the i5 PC. This is consistent with the view of the models. The TCP/IP system being more complex from the system's point of view with more threads etc. would be expected to have higher performance degradation on a busier system.

It is envisioned that using the solutions proposed in this thesis, will lead to software engineers designing the SW-Cs without being delayed by the bottleneck of shared code generator computers. This will lead to engineers being able to act more independently and save man-hours as well. This solution can also enable greater co-operation between different organizations as the Simulink models would be executed on different systems at different organization's local sites; something that was not possible till now.

9 Conclusion and future work

This chapter deals with the conclusions made as a result of the testing and review of the solution and the future work that may be performed in this field.

Conclusions

As has been said before, the solution and its implementation were well within timing requirements of the AFFE-light system as the highest average value was only about 1.6 milliseconds. It must be noted that these were measured on PCs not dedicated for simulating the project but personal PCs with a lot more programs and services that were running that could cause unintended process switches and add to the running time of the simulations. A justification for considering this to be an issue can be shown by the following counterintuitive fact; the PC with the Intel i5 processor and similar configuration as the PC with the Intel i7 processor constantly outperformed the latter although, by nearly every standard of measurement, the i7 processor is superior to the i5. This means that there was a significant effect of the other processes/services running on the system. The logical conclusion of this is that the solution will be much faster on a PC that is used in a dedicated environment.

The original system was used a baseline to check the correctness of the functionality and see if any faults were introduced in the correctness. The new solutions can be reported to be performing identically in terms of correctness. Thus the new solution did not cause any unintended consequences in the system and the delay introduced by the solution was deemed to be well within acceptable limits.

Existing models can be adapted with changes with the AFFE-light system as a reference model. The solution is thus scalable as the number of changes required to implement this solution is within reasonable limits.

Future work

There is some refinement of the solution possible which was not done because of the lack of time. Though the solution clearly demonstrates the advantages of using the new method over the old one, to adapt existing projects to this method can be a daunting task simply because of the sheer size of some of the systems currently in use. Adding the communication blocks to existing project models involves adding them individually to each port and modifying the parameters of each block. An easier way to do this is to automate more of the configuration. It would be handy for e.g. if the blocks could automatically assign the ID of the port in the model that they are connected to. The IDs however have to be unique throughout the system and this would be a problem because Simulink reuses port numbers per model.

Thus, an even better solution would be if the block adds the hash of the name of the model it is placed in to the port number. This would, in most cases make the IDs unique but more importantly, the AUTOSAR side of the system (Picea demo in this case) could be adapted to perform the same calculation and generate the same IDs in the communicate files.

Testing of the system over a network using TCP/IP, was not done mainly because of the lack of availability of systems with the appropriate system licenses at the same time. A greater time for communication will obviously be added by the network latency if the solution is used like this but whether or not this offsets the delays caused by the load on the processor differs per project.

There can also be a better way to implement the tasks in the VFB simulator. For example, it would be more resource friendly to create a threadpool to assign RTE tasks to. The current simulator in use kills the threads after one execution of the RTE task. However, such a modification to the VFB simulator is beyond the scope of this thesis.

The methodologies discussed are in the process of being implemented in the full AFFE project by Mecel. This will be a true test of the scalability of the solution. While the results of that process will not be discussed here, it must be noted that it did make a significant difference in the ease of analysis of the AFFE-light system that was used in this thesis.

References

1. *AUTOSAR Overview.* [Online] [Cited: May 14, 2012.] http://www.autosar.org/index.php?p=1&up=1&uup=0.

2. *Architecture For Future Electric vehicle.* s.l. : MECEL internal documentation.

3. *AUTOSAR Software Component.* [Online] [Cited: April 20, 2012.] http://www.autosar.org/index.php?p=1&up=2&uup=1&uuup=0.

4. *AUTOSAR Virtual Function Bus.* [Online] [Cited: April 17, 2012.] http://www.autosar.org/index.php?p=1&up=2&uup=2&uuup=0.

5. *AUTOSAR Specification of the RTE.* [Online] [Cited: April 19, 2012.] http://www.autosar.org/index.php?p=1&up=2&uup=3&uuup=2&uuuup=0&uuuuup=0.

6. *AUTOSAR Basic Software.* [Online] [Cited: April 26, 2012.] http://www.autosar.org/index.php?p=1&up=2&uup=3&uuup=3&uuuup=0&uuuuup=0.

7. *AUTOSAR Operating System.* [Online] [Cited: April 21, 2012.] http://www.autosar.org/index.php?p=1&up=2&uup=3&uuup=3&uuuup=0&uuuuup=0.

8. *Embedded Market.* [Online] [Cited: April 13, 2012.] http://www.embeddedmarketintelligence.com/2010/07/19/model-based-design-mbd-and-model-driven-development-mdd/.

9. *The Mathworks family of sites.* [Online] Mathworks inc. [Cited: May 12, 2012.] www.mathworks.com.

10. *Interprocess Communications.* [Online] Microsoft. [Cited: March 25, 20212.] http://msdn.microsoft.com/en-us/library/windows/desktop/aa365574(v=vs.85).aspx.

11. *Visual Studio 2008 Express.* [Online] Microsoft. [Cited: March 29, 2012.] http://www.microsoft.com/visualstudio/en-us/products/2008-editions/express.

12. *S-function built with C code.* [Online] Mathworks. [Cited: May 5, 2012.] http://www.mathworks.se/help/toolbox/simulink/sfg/f6-151.html.

13. **The Mathworks Inc.** *Integrating code with Simulink;* . Natick, Massachusetts 01760 USA : Mathworks training services, 2010.

14. *S-function builder.* [Online] Mathworks. [Cited: April 12, 2012.] http://www.mathworks.se/help/toolbox/simulink/slref/sfunctionbuilder.html.

15. *Order of execution by the Simulink engine.* [Online] [Cited: May 7, 2012.] http://www.mathworks.se/help/toolbox/Simulink/sfg/sfunc_c23.gif.

16. *Mask editor.* [Online] Mathworks. [Cited: May 8, 2012.] http://www.mathworks.se/help/toolbox/simulink/gui/f8-3488.html.

Appendix

A. Code and function description for time function

QueryPerformanceCounter()

This function returns the number of "ticks" since the systems was started. It is called before the execution of the runnable and after it is done executing.

QueryPerformanceFrequency()

This function returns the number of "ticks" per second. In order to get the time in seconds the difference between the start and end value in QueryPerformanceCounter() has to be divided whit the frequency.

Code

```
extern struct TIME {
                LARGE_INTEGER start;
                LARGE_INTEGER end;
                LARGE_INTEGER Freq;
}timing;

struct TIME values;

extern double average = 0;
extern int counter = 0;

int timeFunction(int x)
{
                double diff;

                if (x == 1)
                {
                  QueryPerformanceFrequency(&values.Freq);
                  QueryPerformanceCounter(&values.start);
                  return(0);
                }
                else if (x == 2)
                {
                  QueryPerformanceCounter(&values.end);
                  diff = ((double)(values.end.QuadPart - values.start.QuadPart) * 1000000) /
values.Freq.QuadPart;

                if (counter < 100000)
```

```
                {
                if (diff > 8 && diff < 50)
                {
                  average = ((average * counter) + diff) / (counter + 1);
                  counter++;
                }
                }
                else
                {
                  printf("\nAverage - %f", average);
                }
                  return(0);
                }
                  return(0);
}
```

B. List of Signals used

Shared Memory

Nr	Signal name	Used for
1	Signal 1	Used to signal a write operation from Simulink CU to Picea
2	Signal 2	Used to signal a write operation from Picea to Simulink CU
3	Signal 3	Used to signal the next block in Simulink CU that a read operation has been done
4	Signal 4	Used to signal a write operation from Simulink right WN to Picea
5	Signal 5	Used to signal a write operation from Picea to Simulink right WN
6	Signal 6	Used to signal the next block in Simulink right WN that a read operation has been done
7	Signal 7	Used to signal a write operation from Simulink left WN to Picea
8	Signal 8	Used to signal a write operation from Picea to Simulink left WN
9	Signal 9	Used to signal the next block in Simulink left WN that a read operation has been done
10	Signal 123	Used to signal Picea to sleep/wake up threads

TCP/IP

Serial number	Signal name	Use
1	Signal50	Used as a general flag to prevent two runnables from being executed at the same time.
2	Signal51	The RTE uses this to signal the scheduler that one of the control unit runnables is to run.
3	Signal52	This is set to signal to the RTE that one RTE runnable has finished executing.
4	Signal53	This is used to Signal the thread processing the left wheel node to start communicating with Simulink
5	Signal54	This is used by the thread processing the left wheel node to signal the RTE that it has finished executing.

6	Signal55	This is used to Signal the thread processing the right wheel node to start communicating with Simulink.
7	Signal56	This is used by the thread processing the right wheel node to signal the RTE that it has finished executing.
8	Signal57	This signal is used by the RTE to specify that the first runnable of the control unit should be executed.
9	Signal58	This signal is used by the RTE to specify that the second runnable of the control unit should be executed.
10	Signal59	This signal is used by the RTE to specify that the third runnable of the control unit should be executed.
11	Signal123	This signal was used to correct the problems with Picea as mentioned before.

C. Routines used by shared memory

CreateFileMapping()

The communication is initialized by the first program calling this function with a name for the memory mapped object. This procedure returns a handle.

MapViewOfFile()

This function uses the handle returned by the previous function. It is used to create a view of the object in the programs address space. This function call returns a pointer to the file view.

OpenFileMapping()

The second program can get hold of the information in the memory by calling this function with the same name for the object as the first program.

WaitForSingleObject()

To make the processes aware that there is information for them in the memory different events are used. The remote process is waiting for an event to happen by calling the function WaitForSingleObject().

SetEvent()

Once a process is finished writing to the memory it sets a signal telling the remote process that there is something for it in the memory. The process uses the SetEvent() function to do this.

CreateEvent()

An event is created by using the function CreateEvent(). This function call returns a handle to the event object.

UnmapViewOfFile()

This function invalidates the occupied address space and makes the address range available for other allocation.

CloseHandle()

Whenever a process is finished with an object it is good practice to close the handle. Many processes can have handles to objects and after the last handle is closed the object will be removed from the system.

D. Routines used for TCP/IP

Socket()

The Socket function is used to create a socket bound to a specific transport service provider. It can be used to create sockets based on either version of

WSAStartup()

This function is used to initiate use of the Winsock DLL by a process. This has to be used in the beginning of any windows application that uses Windows sockets. An application can access the Winsock socket functions only after this function has been successfully called.

Connect()

The connect function is used to establish a connection to a specified socket. This is used by the client to connect to a server. It needs to be done only once per program but is essential for subsequent send() and recv() functions.

Bind()

A bind function associates a local address with a socket. This is done to associate a socket name with an unnamed socket created as the result of a Socket() call. A socket name consists of an IP address, a port number and the address family.

Listen()

The listen() call places a socket in a state that listens for incoming connections. The maximum number of pending connections is also specified as a parameter.

Createthread()

This is used to create a new thread in the system that can be used to execute another set of instructions independently of the original thread. A function pointer to a function that is to be run is passed with the call.

Accept()

This function permits an incoming connection attempt on a socket. It then returns a handle to the new socket where the connection has actually been made.

Send()

This function is used to send data on a connected socket. The address to the buffer which contains this data is a parameter passed through to the function.

Recv()

This function is used to receive data from a socket. The data received is stored into a buffer whose location is sent as a parameter.

Shutdown()

The shutdown() function is used to disable receiving data, sending data or both. It is called before calling CloseSocket() so that the data transmission can be effectively disabled.

Closesocket()

This function releases the resources used by the socket and closes it.

WSACleanup()

This is the complementary call to the WSAStartup() function and is used to indicate that the WinSock DLL will no longer be used. This frees up the resources allocated by the windows socket implementation for the program.