RSAR: A LAYERED SOFTWARE ARCHITECTURE FOR CYBER-PHYSICAL SYSTEMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ERHAN AKAGÜNDÜZ

IN PARTIAL FULFILLMENT OF REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

FEBRUARY 2015

Approval of the thesis:

RSAR: A LAYERED SOFTWARE ARCHITECTURE FOR CYBER-PHYSICAL SYSTEMS

submitted by **ERHAN AKAGÜNDÜZ** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Gülbin Dural Ünver
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Gönül Turhan Sayan
Head of Department, **Electrical and Electronics Engineering** _____

Assoc. Prof. Dr. Şenan Ece Schmidt
Supervisor, **Electrical and Electronics Engineering Dept., METU** _____

Assoc. Prof. Dr. Halit Oğuztüzün
Co-supervisor, **Computer Engineering Dept., METU** _____

Examining Committee Members:

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı
Electrical and Electronics Engineering Dept., METU _____

Assoc. Prof. Dr. Şenan Ece Schmidt
Electrical and Electronics Engineering Dept., METU _____

Prof. Dr. Ali Doğru
Computer Engineering Dept., METU _____

Prof. Dr. Semih Bilgen
Computer Engineering Dept., Yeditepe University _____

M. Sc. Koray Taylan
Embedded Software Systems Dept., Roketsan Missiles Inc. _____

Date: 13.02.2015

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: Erhan Akagündüz

Signature :

ABSTRACT

RSAR: A LAYERED SOFTWARE ARCHITECTURE FOR CYBER-PHYSICAL SYSTEMS

Akagündüz, Erhan

M. S., Department of Electrical and Electronics

Supervisor: Assoc. Prof. Dr. Şenan Ece Schmidt

Co-Advisor: Assoc. Prof. Dr. Halit Oğuztüzün

February 2015, 63 Pages

Cyber-physical systems integrate the computation and physical processes. Embedded computing systems are used to control and monitor physical processes. Development of software for cyber-physical systems requires deep knowledge on different engineering areas and physics. Embedded software for a cyber-physical system requires software architectures to separate works done by software developers and dynamic model developers. AUTOSAR is a layered software architecture to develop automotive systems. Main purpose of the AUTOSAR layered software architecture is to separate software development process from hardware details. FMI, on the other hand, is a tool-independent dynamic model interface standard. In this thesis, a layered software architecture RSAR is proposed for cyber-physical systems development. RSAR adheres to the layered software architecture concept of AUTOSAR and supports dynamic model usage with FMI standard.

Keywords: Functional Mockup Interface; Layered Software Architecture for Cyber-Physical Systems

ÖZ

RSAR: SİBER-FİZİKSEL SİSTEMLER İÇİN KATMANLI YAZILIM MİMARİSİ

Akagündüz, Erhan

Yüksek Lisans, Elektrik Elektronik Mühendisliği Bölümü

Tez Yöneticisi: Doç. Dr. Şenan Ece Schmidt

Ortak Tez Yöneticisi: Doç. Dr. Halit Oğuztüzün

Şubat 2015, 63 Sayfa

Siber-fiziksel sistemler fiziksel işlemlerin kontrolünü ve gözlenmesini gerçekleştiren gömülü yazılım içeren sistemlerdir. Siber-fiziksel sistem geliştirmek üst seviye farklı mühendislik alanından bilgi ve fizik bilgisi gerektirmektedir. Siber-fiziksel sistem gömülü yazılımları, yazılım geliştiricileri tarafından yapılan işler ile dinamik model geliştiricileri tarafından yapılan işleri birbirinden ayıran yazılım mimarilerine ihtiyaç duymaktadır. AUTOSAR otomotiv sistemlerinin geliştirilmesinde kullanılmak üzere katmanlı yazılım mimarisi sunmaktadır. Bu mimarinin ana amacı yazılım geliştirme süreçlerini donanım detaylarından ayırmaktır. FMI geliştirme araçlarından bağımsız dinamik model arayüz standartıdır. Bu tez kapsamında siber-fiziksel sistem yazılımları geliştirilmesinde kullanılmak üzere RSAR katmanlı yazılım mimarisi önerilmiştir. RSAR, AUTOSAR katmanlı yazılım mimarisi kavramlarını ve FMI standartında arayüz sunan dinamik modelleri kullanmaktadır.

Anahtar Kelimeler: İşlevsel Model Arayüzü; Siber-fiziksel Sistemler için Katmanlı Yazılım Mimarisi

To My Family

ACKNOWLEDGEMENTS

I would like to thank my supervisor Assoc. Prof. Dr. Şenan Ece Schmidt for her guidance, advice, criticism, encouragements and insight throughout this research.

I also wish to thank a lot to my co-advisor Assoc. Prof. Dr. Mehmet Halit Oğuztüzün for all the valuable knowledge, technical support, academic assistance, innovative ideas.

I also wish to thank a lot to my manager Koray Taylan for all the valuable efforts to ease the procedural processes, to find financial contribution, to provide moral support, to enhance the quality of this thesis work.

I would like to thank to Turkish Ministry of National Defense, Under-secretariat for Defense Industries which gave the team financial and moral support [Project Name: MOKA].

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

AUTOSAR	Automotive Open System Architecture
CAS	Control Actuator System
CPS	Cyber-physical System
DIO	Digital Input Output
ECU	Electronic Control Unit
FMI	Functional Mock-up Interface
FMU	Functional Mock-up Unit
HAL	Hardware Abstraction Layer
IMU	Inertial Measurement Unit
I2C	Inter Integrated Circuit
MCU	Microcontroller Unit
MPU	Microprocessor Unit
PC	Personal Computer
RSAR	Real Time Software Architecture
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver/Transmitter

CHAPTER 1

INTRODUCTION

Cyber–physical systems (CPSs) integrate the computation and physical processes. The control and monitoring of the physical processes are carried out by embedded computing systems. There is an interaction between these physical processes and the computation that involve feedback loops [1].

Development of software for cyber-physical systems requires deep knowledge from different disciplines iterative processes [2]. Developing real time embedded software for such a system demands software architectures for separate works done by software developers and dynamic model developers. The complexity of software design increases with more features included in the cyber-physical system. To develop such software on a custom designed infrastructure will ease design and development phases of software.

Modeling such CPSs is followed by simulations for verification of the operation and performance measurements. The diverse and heterogeneous components of the CPSs require a standardized interface for the simulation models to facilitate model exchange and co-simulation of dynamic models.

The motivation of this thesis work is developing a layered software architecture RSAR (Real-time Software ARchitecture) for CPSs which enables the real-time operation of the software in embedded environments and incorporates the standardized simulation interfaces.

To this end, we adopt the architecture of well-established but proprietary AUTOSAR standard combined with Functional Mock-up Interface (FMI) standard to construct RSAR. AUTOSAR offers layered software architecture standard for automotive systems. Its main purpose is to separate hardware details while developing software using standardized interfaces between layers of software. The Functional Mock-up Interface (FMI) is a tool-independent dynamic model interface standard. Its main

purpose is model reuse and interoperability between various modeling tools and environments throughout the systems development phases.

RSAR aims to minimize migration costs of software development with using current tools used in the company and aims to break dependency on AUTOSAR tool vendors that would be problem to purchase tool licenses to develop real time embedded software for CPSs.

The contributions of this thesis are as follows:

The layered RSAR architecture with detailed description of the interfaces and components.

The demonstration of the correctness and performance of RSAR with a case study of missile avionics software which was previously developed in a defense industry company.

The remaining chapters are organized as follows:

- Chapter 2 provides related literature and background information required for understanding of the subsequent chapters.
- Chapter 3 presents an overview of the related works.
- Chapter 4 explains the details of proposed layered software architecture.
- Chapter 5 explains code generation process in the RSAR.
- Chapter 6 presents a case study and evaluation.
- Finally, Chapter 7 discusses the accomplishments and draws conclusions.

CHAPTER 2

BACKGROUND

2.1 Automotive Open System Architecture (AUTOSAR) Standard

Automotive Open System Architecture (AUTOSAR) is standardized software architecture developed by car manufacturers with the aim to decouple software development details from hardware details.

AUTOSAR provides a common software infrastructure for automotive systems. AUTOSAR aims to achieve technical goals of modularity, transferability and re-usability for software used in automotive systems.

AUTOSAR has been formed with the goals of [3]

- Standardization of basic software functionality of automotive ECUs
- Scalability to different vehicle and platform variants
- Transferability of software
- Support of different functional domains
- Definition of an open architecture
- Collaboration between various partners
- Development of highly dependable systems
- Support of applicable automotive international standards and state-of-art technologies

2.1.1 History

AUTOSAR has been founded by three original equipment manufacturers in 2003. Today, 9 core partners has been involved AUTOSAR: BMW, Bosch, Continental, Daimler, Ford, General Motor, Peugeot, Toyota and Volkswagen. It has 50 premium members, about 90 associated members and about 20 development members [4].

AUTOSAR is broadly used in Europe and is emerging in Asia and North America.

AUTOSAR has 5 different specifications since 2003 which are the releases 2.0, 3.0, 3.1, 3.2, 4.0 and 4.1.

2.1.2 Definitions

Virtual Function Bus (VFB)

To separate software design process from hardware details AUTOSAR defines Virtual Function Bus model. The Virtual Function Bus is a software component interconnection model that strictly separates the domain of application design and implementation of software for a specific hardware. It provides generic communication services that can be consumed by any existing AUTOSAR software component. All the services of the VFB are virtual. The virtual services are implemented for the underlying hardware in a later phase by AUTOSAR tools [6].

Run Time Environment

Run Time Environment provides the actual implementation of the VFB interconnection model [6] .

2.1.3 Software Architecture

To decouple software components from hardware AUTOSAR provides layered software architecture standard. AUTOSAR includes four main layers as shown in Figure 1. To decouple software development from hardware AUTOSAR offers standard interfaces for the layers of the architecture. Thanks to interfaces between the layers, application layer components are developed without knowing the hardware details. Layered architecture offers reuse of the application layer components on different cars, different ECUs.

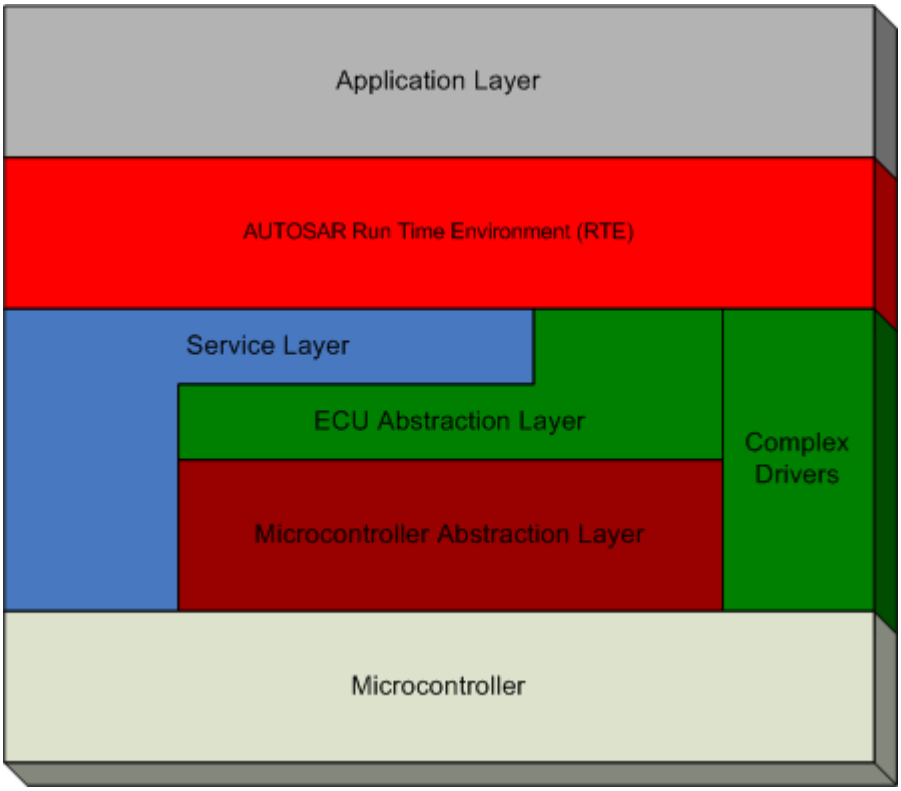

Figure 1 Layered Software Architecture of AUTOSAR [3]

As show in the Figure 1, *Application layer* contains application specific software components. *Runtime Environment layer* isolates *Application layer* from *Basic Software layer* which contains *Service layer*, *ECU Abstraction layer*, *Microcontroller Abstraction layer* and *Complex Drivers*. At the bottom *Microcontroller layer* is located.

The layers have different responsibilities. The *Basic Software layer* and the *Runtime Environment layer* are responsible for the abstraction between the hardware and the application software. Therefore the Basic Software layer contains ECU specific modules as well as general AUTOSAR modules [7].

Basic Software layer is divided into different parts according to their functionality as shown in Figure 2.

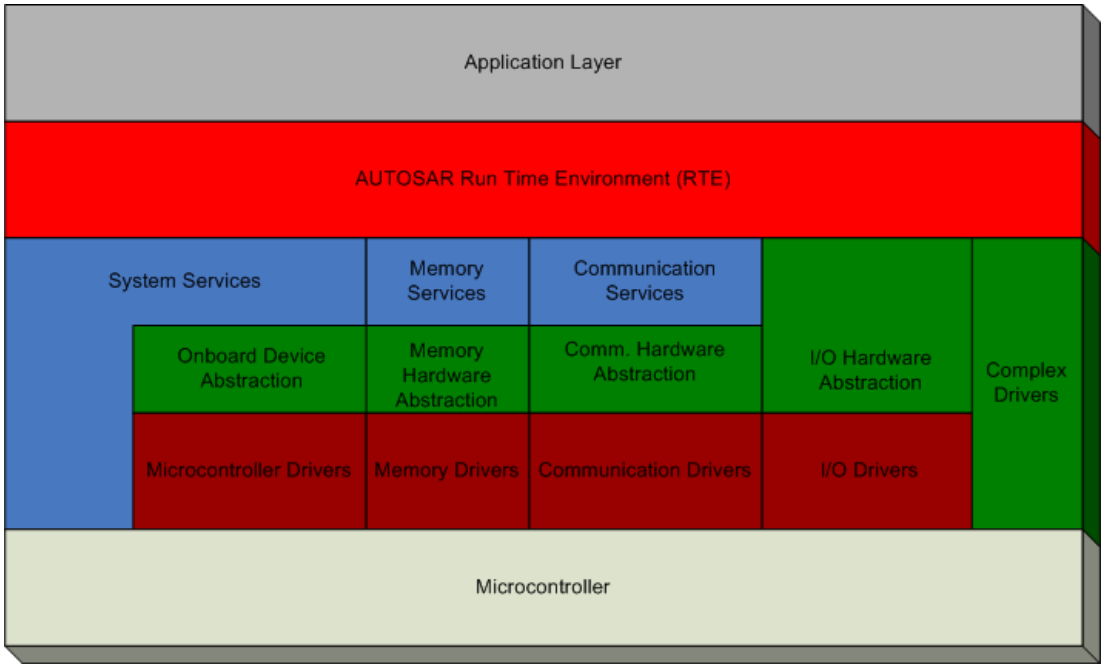

Figure 2 Divided Basic Software Layer [3]

The System stack, consisting of Microcontroller Drivers, Onboard Device Abstraction and System Services, provides standardized services and library function for example for timer operations and operating system functionality [4].

The Management stack, consisting of Memory Drivers, Memory Hardware Abstaction and Memory Services, provides standardized access to non-volatile memory [7].

The Communication stack, consisting of Communication Drivers, Communication Hardware Abstraction and Communication Services, provides standardized access to the vehicle network system [7].

The I/O stack, consisting of I/O Drivers and I/O Hardware Abstraction, provides standardized access to sensors, actuators and other ECU on board peripherals[7].

2.2 Functional Mock-up Interface (FMI)

Functional Mock-up Interface (FMI) is a tool independent standard to support both model exchange and co-simulation of dynamic models using a combination of XML-files and compiled C-code [8]. The models are independent from the tools and any

model can be exchanged in any simulation environment. The models can be reused on any modeling tool supports FMI.

2.2.1 History

The first version, FMI 1.0, was published in 2010. The FMI development was initiated by Daimler AG with the goal to improve the exchange of simulation models between suppliers and OEMs. As of today, development of the standard continues through the participation of 16 companies and research institutes [8].

2.2.2 Definitions

Functional Mock-up Interface (FMI)

Functional Mock-up Interface defines a standardized interface for simulation of cyber-physical systems [8]. A Cyber-physical system is an embedded system that controls physical entities by using software and hardware elements. Cyber-physical systems are used in many areas which are aerospace, automotive, energy etc. [2].

Functional Mock-up Unit

Functional Mock-up Unit is a software component which uses Functional Mock-up Interface. FMU is provided as a compressed file which includes a model description file and source codes, libraries and documentation.

2.2.3 FMI for Co-simulation

FMI for Co-simulation is a standard interface for the model that contains its solver inside. The user of the model does not need to know which integration methods are applied to solve differential equations.

To use a given FMU three steps are needed, which are instantiation and initialization, running and termination.

In the instantiation and initialization step, an instance of FMU is created and initialized. In this step, memory allocation needed by the FMU is done and parameters used in the FMU are initialized to their default values.

FMI defines setter, getter and execution methods to run model. Setter methods are used to set parameters which are used by the model execution. To solve model's differential equations, *doStep* method is defined by the FMI. The output parameters generated by the model are reached by using getter methods.

In the running step firstly, input parameters are set by calling setter methods (*FMUSet*…(…)). Then *doStep()* method is called to solve the model's differential equations. By calling getter methods (*FMUGet…(...)*), output parameters of the solved model can be accessed by the caller.

In the termination step, resources used by the FMU are freed.

CHAPTER 3

RELATED WORKS

In the related work [9], it is proposed that Functional Mock-up Interface standard for model exchange can be utilized in the context of AUTOSAR software component development. Automatic transformations between the XML schemas of the two standards are utilized to convert FMI models to AUTOSAR [9].

In the related work, mapping and conversion scheme between FMI and AUTOSAR is shown and after mapping and conversion FMI is used as AUTOSAR software component. FMI and AUTOSAR use XML definitions to define models. The *Altova MapForce* program is used to map XML schemas in the graphical user interface [9].

In the related work [10], a solution is proposed to use AUTOSAR and FMI together in the Software-in-the-loop simulations. A powertrain co-simulation is demonstrated to facilitate AUTOSAR and FMI standards together.

CHAPTER 4

RSAR: A LAYERED SOFTWARE ARCHITECTURE FOR CYBER-PHYSICAL SYSTEMS

In this chapter the proposed RSAR layered software architecture is described. The details of the layers defined in the architecture have been explained in detail. Dynamic model utilization in the architecture has been covered as well.

4.1 Aim of RSAR

The physical processes related with cyber-physical systems are handled by embedded software systems. A conventional embedded system development approach contains software which is particular to hardware. This tied approach brings some problems along. The software is dependent to the hardware and if the hardware changes in some time, the particular software must be changed also. To eliminate this problem in the domain of embedded software systems, software must be divided in well described layers which break the hardware dependency somehow.

Another problem in the domain of cyber-physical systems is dynamic model utilization in the software. Dynamic models are generally developed tools which generate code from the model descriptions. There exist many dynamic modeling tools for different engineering branches. Thanks to FMI, usage of the models is standardized with a tool-independent common interface. RSAR enables dynamic model utilization using FMI.

The RSAR briefly aims to

- divide software into layers to eliminate/minimize hardware dependency
- shorten design and development phases of the software project by supplying layered software architecture and dynamic model inclusion with FMI
- provide code re-usability between various software projects
- break modeling tool dependency by using FMI

- reduce implementation errors by providing layered software models which contain template C/C++ classes and automatic glue code generation to these template models according to software system requirements

by offering a layered software architecture which utilizes FMI for cyber-physical system software development.

4.2 RSAR Layers

The RSAR solution includes five layers as shown in Figure 3. Between all the layers, the communication is handled over interfaces shown in Figure 3.

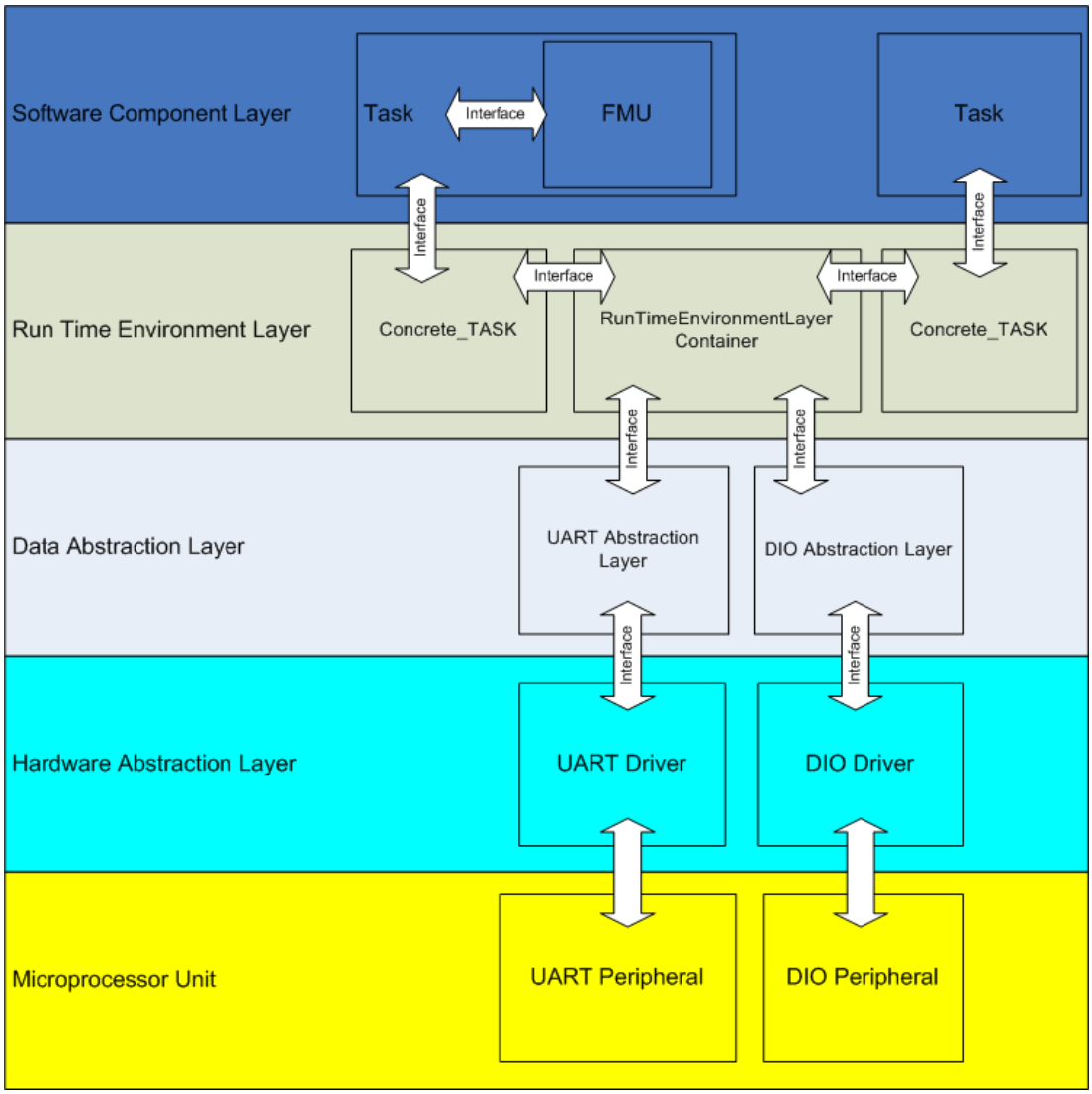

Figure 3 Proposed Layered Software Architecture For Cyber-Physical Systems

A brief introduction to the layered architecture is given here. The details of every layer are given in the following sections.

Hardware Abstraction Layer (*HAL*) provides a standardized way to communicate between the peripherals of microprocessor unit. Peripherals include UART, SPI, DIO and GPT etc. For the *Hardware Abstraction Layer*, some peripheral drivers have been designed and implemented for the thesis proposal which are

- Serial Peripheral Interface Driver
- Universal Asynchronous Receiver/Transmitter (UART) Driver
- General Purpose Timer (GPT) Driver
- Digital Input/Output Driver (DIO) Driver

For each of the drivers given above, an interface has been designed to standardize development of driver. With the standard interface, the user does not need to know about hardware details of peripherals. The user uses only the Application Programming Interface (API) given by the standardized driver interface. In AUTOSAR, *Microcontroller Abstraction Layer* is used for same purposes.

Data Abstraction Layer is used above the *HAL* to convert peripherals data into meaningful data to be used in the *Software Component Layer*. Peripheral data must be converted to logical data that will be used in the *Software Component Layer*. The main purpose of this layer is to parse the incoming data from *HAL* into named signals and to format the named signals into appropriate outgoing data to HAL. The models proposed are explained in 4.6.

The *Software Component Layer* components include virtual connections to get data or to set data which should be separated from hardware communication details. These virtual connections are converted into concrete connection in the *Runtime Environment Layer*. A *Software Component Layer* entity has no interaction with the hardware. It has virtual connections to reach and modify the needed signal. A *Software Component Layer* entity can include a dynamic model which has Functional Mock-up Interface in the proposed solution. By using FMI, software

developers and dynamic model developers will have a common interface to communicate each other in a standard way. The dynamic models can come from various domains including mechanical, hydraulics, pneumatics, thermodynamics, flow dynamics, electrical, software etc.

4.3 Scheduler Design

Real time embedded systems are assumed to work with periodic tasks. In a period, these systems do some tasks which should be completed by a certain time. These tasks are executed by a scheduler in certain times.

In proposed design, *Scheduler* class has been developed to run tasks which should start to run a certain time in the main loop of the embedded software. *Scheduler* is responsible to run tasks using a predefined runnable model. The runnable model in the proposed design is *Task* class. All the runnable entities in the design use *Task* as base class.

Scheduler object needs time information to run tasks. *SystemTime* class has been developed to provide necessary timing information to *Scheduler*. *SystemTime* uses a *General Purpose Timer Driver (GPT)* which resides in the HAL to get timing information from timer peripheral. The object of the *Scheduler* class uses the *SystemTime* object to run tasks periodic. *Scheduler* object knows *Task* interface and runs the *Task* objects by calling their *run* method. The *run* method is a virtual method which is realized by derived classes. Thanks to inheritance, all *Task* objects can do different tasks. The *run* method of a *Task* object must be called in appropriate time in the main loop according to embedded system requirements. To able to run tasks in appropriate time in the main loop, the *Scheduler* object needs one more timer to measure time in the main loop. This timer must have more resolution. In proposed design for this purpose, *Time Base Register* of e300 core is used [1]. *Time Base Register* is sampled at the beginning of the main loop and tasks are executed in the loop when their time comes with respect to sampled main loop time.

The class diagram of the proposed scheduler is shown in the Figure 4.

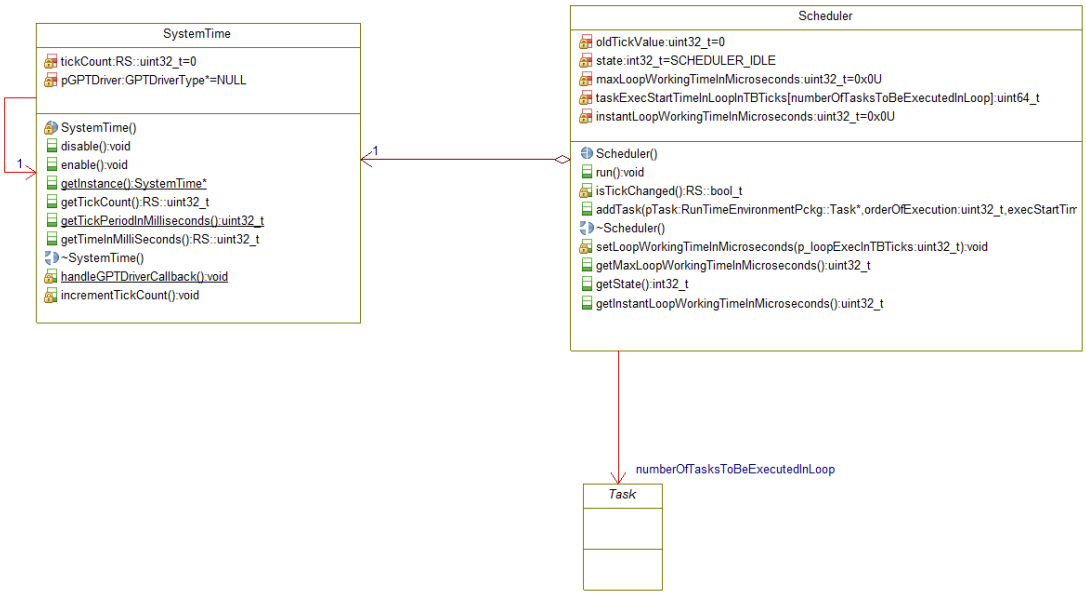

Figure 4 Scheduler Relations

The properties of the scheduler design are listed here.

- Tick period of the *SystemTime* is determined according to embedded system requirement. This tick period determines the main loop start time of *Scheduler*. For example, *Scheduler* starts to execute *Task* objects every 2 milliseconds.

- *Scheduler* knows a number of *Task* objects to be executed. This number changes according to design of embedded system.

- *Scheduler* keeps timing information for debugging purposes. Total execution time of *Task* objects are tracked by the *Scheduler*. Maximum and instant execution times of the main loop are stored inside *Scheduler*. Thanks to this information, developer can observe if total work is completed inside determined main loop time.

- *Task* objects are executed in succession by the *Scheduler*. *Task* objects are introduced to the *Scheduler* with order of execution and time of execution information by the developer. Time of execution information is used by the *Scheduler* to start execution in a defined time in the main loop. When a *Task* object execution is finished, *Scheduler* firstly check the time of execution

information of next *Task* object in the sequence, then it starts execution or waits for the determined time to come.

- *Task* objects keep information of their own execution time. Thanks to this information, developer can observe execution time of any particular *Task* object.

Task objects are registered to *Scheduler* object by using the *addTask* method which is given in Figure 5.

```
82          void addTask(Task* pTask,
83                  const uint32_t& orderOfExecution,
84                  const uint32_t& execStartTimeInLoopInMicros = 0x0U
85                  );
```

Figure 5 addTask Method Definition

The *orderOfExecution* parameter defines which order task will be executed by the *Scheduler* object. The *execStartTimeInLoopInMicros* parameter defines the start of execution time in microseconds in the main loop.

4.4 Software Component Layer Design

In RSAR, *Software Component Layer* is the top layer. A *Software Layer Component* has no dependency with hardware. It needs some input signals and by using these input signals produces some output signals with logic located inside. In the proposed design, a *Software Component* is an abstract class which is derived from *Task* class. These tasks are run by the *Scheduler* object.

In the design process, firstly signals are determined as needed by the software component. Then these signals are grouped together to define an interface. To generate a *Software Component*, all the interfaces are determined by the user using the determined signals needed by the interface. These interfaces show that which data from/to *Run Time Environment Layer* is needed by the component. In Figure 6, there is a *Software Component* which is a *Task* with two interfaces.

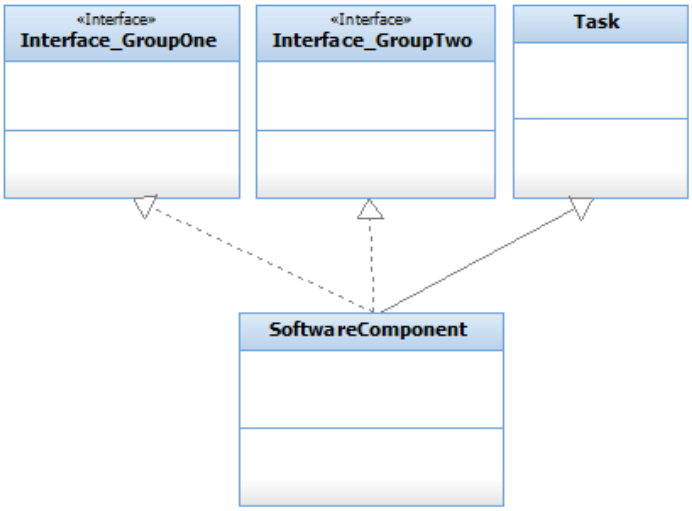

Figure 6 Software Component Design

To generate such a *Software Component*, firstly signals are determined. Then signals are grouped together to define an interface. Lastly, *Software Component* is derived from these interface classes and the base *Task* class. By doing that, developer is free to develop logic inside the *Software Component* without knowing where the data comes from and where data goes to.

In RSAR, a *Software Component* can include a FMU inside. The FMU is connected to *Software Component* by using FMI methods which are getter, setter and execution. The inputs of the *Software Component* are connected to FMU and outputs of the FMU are connected to *Software Component.*

In the RSAR, interfaces and *Software Component*s are generated automatically from defined XML files. The detail of the automatic code generation process has been introduced in section 5.2.

4.5 Run Time Environment Layer Design

The *Run Time Environment Layer* stands between the *Software Component Layer* and the *Data Abstraction Layer*. It is used to connect *Software Component Layer* and *Data Abstraction Layer* together.

To develop *Run Time Environment Layer*, a class has been determined to store data which is used by *Software Component Layer* and *Data Abstraction Layer* and to

direct data from *Software Component Layer* to *Data Abstraction Layer* or from *Data Abstraction Layer* to *Software Component Layer*.

As explained in the section 4.4, *Software Component*s have only virtual methods to use and modify data. These virtual methods are realized within *Run Time Environment Layer*. To realize virtual methods of a *Software Component*, a new class is derived from the *Software Component* class. The derived class is named with a prefix. The prefix word defined in the RSAR is "Concrete". If name of a *Software Component* is "SoftwareComponent", name of the derived class becomes "Concrete_SoftwareComponent". The "Concrete_SoftwareComponent" uses the methods of *Run Time Environment Layer* class. The methods of *Run Time Environment Layer* class can

- set the data to attributes of itself,
- get the data from the attributes of itself,
- direct the data to a *Data Abstraction Layer* component,
- fetch the data from a *Data Abstraction Layer* component.

The *Run Time Environment Layer* class has concrete methods to be used by the *Software Component*s and by the *Data Abstraction Layer* components.

In RSAR, *Run Time Environment Layer* class and concrete *Software Component*s are generated automatically using defined XML files. The detail of code generation process has been introduced in 5.3.

4.6 Data Abstraction Layer Design

Data Abstraction Layer stands between *Run Time Environment Layer* and *Hardware Abstraction Layer*. The aim of this layer is to process data into appropriate format for upper and lower layers. To send *Run Time Environment Layer* data to hardware or to get hardware data to *Run Time Environment Layer*, determined data abstraction models within RSAR are used. In the following sections, two *Data Abstraction Layer* models have been introduced.

The *UART Protocol Abstraction Layer* is a model for asynchronous serial communication. The model is a reference model which can be used to design a model for any other serial communication hardware. The given model is a RSAR *Task* which is executed by the *Scheduler* to handle communication with hardware. The model has responsibility to connect hardware to *Run Time Environment Layer*. The details of the *UART Protocol Abstraction Layer* are given in 4.6.1.

The *DIO Data Abstraction Layer* is a synchronous model given in detail in 4.6.2. The *Run Time Environment Layer* uses the *DIO Data Abstraction Layer* to make connection to DIOs. This process is performed by calling the methods from *Software Component Layer.* A *Software Component Layer* entity calls a method and over the *Run Time Environment Layer*, DIO connection is made synchronously to method call using *DIO Data Abstraction Layer.*

4.6.1 UART Protocol Abstraction Layer Design

UART peripherals are commonly used in embedded systems to communicate with each other. Generally a messaging protocol is used over UART communication. The message coming from UART hardware must be parsed into meaningful data to make process with. A protocol message generally has header, data and CRC areas. The header area introduces the message to receiving side with message id, message length etc. sub areas. The data area of the UART message consists of the raw data including one or more meaningful data. The CRC area is used to detect undesired changes in the message which is transferred.

In RSAR, to generate and parse UART protocol messages that are used in the company, *UART Protocol Abstraction Layer* model has been designed. To implement this model some common classes has been designed as template to model. Then from the XML definition of a UART protocol message, glue code is generated automatically by the implemented python script. The detail of the automatic code generation has been introduced in the section 5.4.1. In RSAR, to send and receive UART protocol messages *Task* and *Avionic* classes are used as base class to derived automatically generated class. The automatically generated class is used as glue code to communicate with *Run Time Environment Layer*. This class converts the raw data

into meaningful data and sets the meaningful data to *Run Time Environment Layer*. In same way, this class gets meaningful data from *Run Time Environment Layer* and converts meaningful data into raw data.

In RSAR, *UART Protocol Abstraction Layer* uses flags to control receiving and sending messages. A *Software Component* sets a flag to trigger UART message sending. *UART Protocol Abstraction Layer* controls the flag over *Run Time Environment Layer*. If a sending message flag is set, *UART Protocol Abstraction Layer* prepares the message and sends it. In same way, a *Software Component* can detect if there is new data by checking a flag. This flag is set by the *UART Protocol Abstraction Layer* over *Run Time Environment Layer*.

In the Figure 7, there is an example for the abstraction layer. *RTE_MC_IMU_Task* class is generated automatically from the XML file definition. It is the glue code between the *Run Time Environment* class and the abstraction layer template class which is named *Avionic*. The *Avionic* and *Task* classes are used as base class to automatically generated class. The *Avionic* class controls the *AvionicPort* object to send and receive *AvionicMessage* object which is abstraction model of UART message in RSAR.

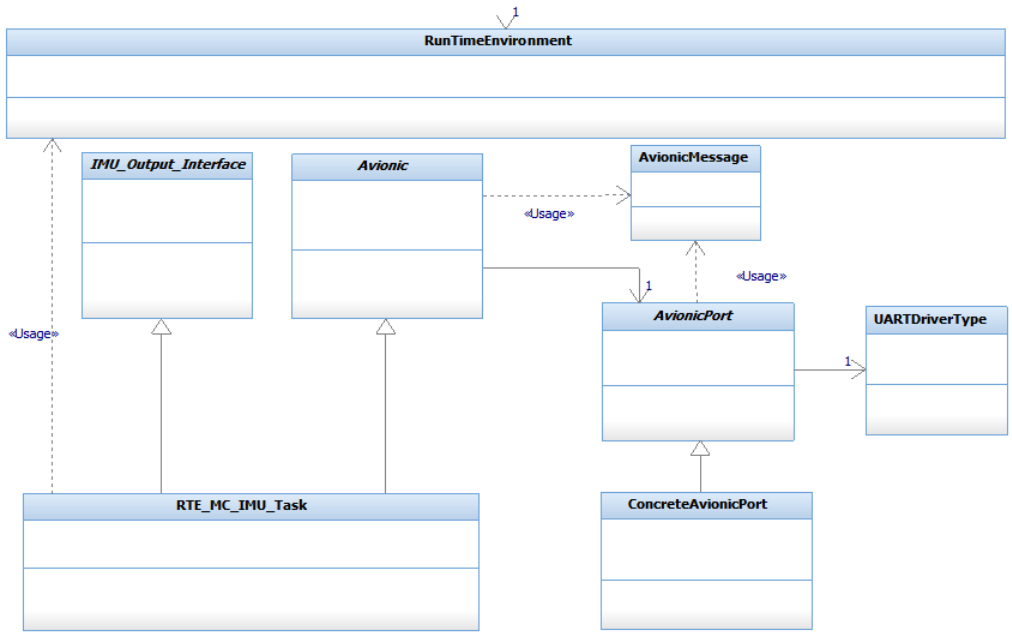

Figure 7 UART Data Abstraction Layer Example

The *AvionicPort* has some properties to control UART protocol communication. Details of the properties are given below.

- *AvionicPort* has a map to identify receiving UART messages. This map creates a relation between message identification number and message length. Message identification number and message number reside in the header section of the message. *AvionicPort* receives only the messages introduced to itself by the message identification number – message number map. If an unknown message is received, it is dropped by the *AvionicPort*.

- *AvionicPort* checks the CRC value of the incoming message. If the CRC value is incorrect, the message is dropped by the *AvionicPort.*

- *AvionicPort* calculates the CRC value of the outgoing message.

- *AvionicPort* has a timeout value for incoming UART messages. The message is dropped by the *AvionicPort*, if *AvionicPort* can't receive message within timeout time.

To understand how the design works, sequence diagrams are given in the Figure 8 and Figure 9.

Figure 8 Receiving UART Protocol Message

To receive a message, the *Scheduler* object calls run method of the abstraction layer *Task* object. The object firstly uses the base class methods to receive UART message. The *rxMsgAsyncMainMethod* of the *AvionicPort* object is used to communicate with UART driver which resides in HAL. If there is a completed UART protocol message, the *receiveMessageAsync* method returns true. Then the *handleRxAvionicMessage* method which is implemented in the auto-generated code is called. According to UART message definition which is given in the XML file, message is parsed and the values are set to *Run Time Environment Layer*. To indicate there is a new message, flag attribute is set. In the example, *IMU_Rx_Flag* is used.

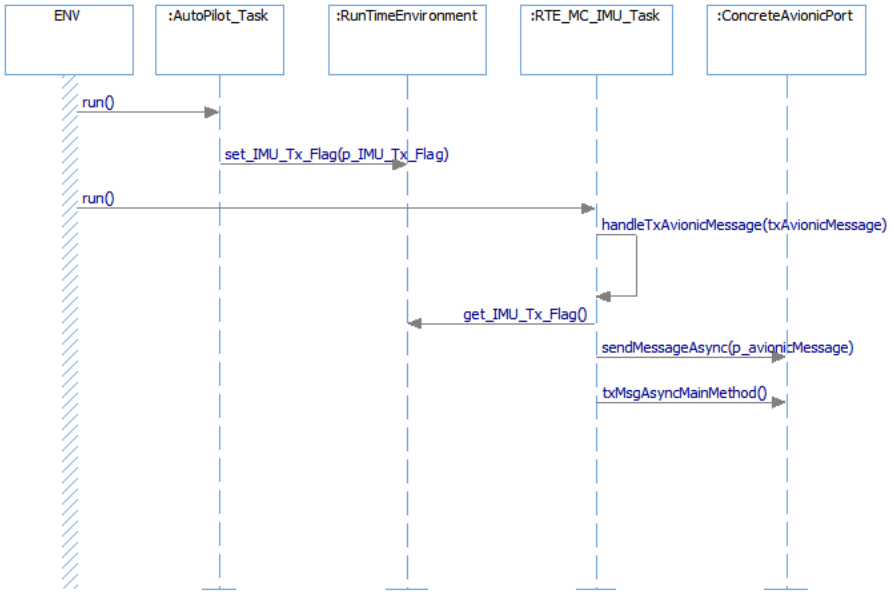

Figure 9 Transmitting UART Protocol Message

To send a message, firstly a *Software Component Layer* object should trigger the *Data Abstraction Layer* object by setting a flag which is *IMU_Tx_Flag* in the example. The flag is set by the calling method of the *Run Time Environment Layer*. Then the *Scheduler* object calls the *run* method of the *Data Abstraction Layer* object. The *UART Protocol Abstraction Layer* object checks if the flag is set. If the flag is set, the UART protocol message is prepared. By calling the *sendMessageAsync* method, the message is sent to *AvionicPort*. The message is sent to UART peripheral by the *AvionicPort* object with calling *txMsgAsyncMethod*. The *AvionicPort* object uses UART driver which resides in the HAL to send raw data.

4.6.2 DIO Data Abstraction Layer Design

DIO Data Abstraction Layer is used to abstract hardware details of discrete input and output ports. A *Software Component* might control or read discrete ports by using Boolean flags in the proposed RSAR design. If the Boolean value is true, it means the discrete port has a value of active state. If the Boolean value is false, it means the discrete port has a value of inactive state. The *Run Time Environment Layer* uses this layer to make connection with DIOs. The DIO is set as *logic 0* or *logic 1* according to defined active/inactive state in the design. The logical value read from a DIO is

converted to Boolean value according its defined active/inactive state in the design by this layer.

4.7 Hardware Abstraction Layer Design

In this part, the detail of the thesis study of HAL design is given. All drivers were developed for the Freescale MPC5200B microprocessor unit. MPC5200B is a PowerPC based microprocessor unit with its e300 core [11].

For all drivers given below, an interface has been defined and using these interfaces the drivers have been developed for the RSAR study. A defined driver interface uses a C language *struct* which contains method pointers and defined attributes to offer a common way to interact with hardware. By doing that, the driver interface could be used for another hardware platform.

The driver interface for the hardware peripheral defines a standard way to communicate with hardware. The implementation of the driver is dependent on the MCU/MPU hardware platform. By using a driver interface, usage of the hardware peripheral is standardized for different MCU/MPU hardware platforms. For the thesis study, GPT, UART and DIO driver interfaces have been defined. For the defined interfaces, drivers have been implemented for the Freescale MPC5200B MPU.

4.7.1 General Purpose Timer (GPT) Driver Design

A timer peripheral of a microcontroller/microprocessor is used to measure time or to count time by interrupt. In thesis work, an interface has been developed and MPC5200B timer driver has been implemented.

```
60  // Defines timer base.
61  // It is used by driver to calculate
62  // tick count for milliseconds or
63  // microseconds.
64
65  typedef enum
66  {
67      GPTDRIVER_TIME_IN_MICROSECONDS = 0,
68      GPTDRIVER_TIME_IN_MILLISECONDS = 1
69  }GPTDriverTimeBaseType;
```

Figure 10 GPT Driver Time Base Type

In GPT interface, a timer can be configured in two ways. First configuration value is *GPTDRIVER_CONTINUOUS_MODE*. This value is used to generate periodic interrupt by the timer peripheral. GPTDRIVER_ONE_SHOT_MODE configuration value is used to generate interrupt only once.

```
71  // Defines mode of timer.
72  // In continuous timer mode timer generates
73  // periodic interrupt. User callback function
74  // is called periodically.
75  // In one shot mode, timer generates interrupt
76  // once then it is disabled. User callback
77  // function is called once.
78  typedef enum
79  {
80      GPTDRIVER_CONTINUOUS_MODE = 0,
81      GPTDRIVER_ONE_SHOT_MODE = 1
82  }GPTModeType;
```

Figure 11 GPT Driver Mode Type

When a timer is elapsed, an interrupt occurs and in some way user needs to know about that event. For this purpose, a user callback method has been defined in GPT interface. The GPT driver handles the interrupt and calls the user callback method to inform user about timer elapse event. The definition of callback method is given in the Figure 12.

```
49  // GPT User call back function type.
50  // When timer interrupt occurs, driver
51  // handles it and then calls user
52  // function if it is registered to driver.
53  // "!!!ATTENTION BELOW!!!"
54  // User function implemenation must be short
55  // as possible. Because driver calls user function
56  // from ISR!!! ISR must be finished quickly.
57  typedef void (*GPTCallbackFuncType)(void);
```

Figure 12 GPT Driver Callback Method Type

```
typedef struct GPTDriver GPTDriverType;

struct GPTDriver
{
    ERROR (*open)(GPTDriverType* pDriver);
    ERROR (*startTimer)(GPTDriverType* pDriver, GPTValueType timeValue);
    ERROR (*stopTimer)(GPTDriverType* pDriver);
    ERROR (*enableNotification)(GPTDriverType* pDriver, GPTCallbackFuncType pUserFunction);
    ERROR (*disableNotification)(GPTDriverType* pDriver);
    ERROR (*close)(GPTDriverType* pDriver);

    // Attributes.
    // User must set hardware configuration before
    // calling open function.
    GPTHardwareConfigType hwConfig;

    // Handle used by driver
    GPTHandleType handle;

    // Stores timer specific driver error.
    // It could be checked if any function returns
    // "GPTDRIVER_ERROR"
    GPTDriverErrorType error;
};
```

Figure 13 GPT Driver Interface Methods

GPT interface consists of the methods given in the Figure 13.

The *open* method is used to initialize the GPT peripheral. The *startTimer* method is used to start GPT peripheral to count. The *stopTimer* method is used to stop GPT peripheral to count. The *enableNotification* method is used to register user callback method to GPT driver. By using this method, driver learns which user callback method will be called. The *disableNotification* method is used to delete record of the user callback method. The *close* method is used to stop GPT peripheral and tells the driver GPT will not be used again until the *open* method calling.

4.7.2 Universal Asynchronous Receiver/Transmitter Driver Design

A Universal Asynchronous Receiver/Transmitter is a hardware which translates parallel data to serial form to send, and incoming serial data to parallel form. UART peripherals are used with external circuits to form electrical signal into standard RS-232, RS-422, RS-485 etc. forms.

In RSAR, an interface has been developed for UART driver and two different kind of UART driver has been implemented for two different peripherals which are MPC5200B PSC and D16950 UART IP.

To set baud-rate of the communication *UARTBaudrateType* which is shown in Figure 14 is defined.

```
// Defines baud rate type.
typedef uint32_t UARTBaudRateType;
```

Figure 14 UART Driver Baud-Rate Type

To set stop bit count of the UART communication *UARTStopType* which is shown in the Figure 15 is defined.

To set bit count of a character *UARTDataBitType* which is shown in the Figure 15 is defined.

To set parity of the UART communication *UARTParityType* which is shown in the Figure 15 is defined.

```
// Defines possible stop bits count configuration
typedef enum
{
    UART_STOP_BITS_ONE = 0,
    UART_STOP_BITS_ONE_AND_HALF = 1,
    UART_STOP_BITS_TWO = 2
}UARTStopBitType;

// Defines type for
// possible data bits count configuration
typedef enum
{
    UART_DATA_BITS_EIGHT = 0,
    UART_DATA_BITS_FIVE = 1,
    UART_DATA_BITS_SIX = 2,
    UART_DATA_BITS_SEVEN = 3
}UARTDataBitType;

// Defines type for
// possible parity bit configuration
typedef enum
{
    UART_PARITY_NONE = 0,
    UART_PARITY_ODD = 1,
    UART_PARITY_EVEN = 2,
    UART_PARITY_STICK_LOW = 3,
    UART_PARITY_STICK_HIGH = 4
}UARTParityType;
```

Figure 15 UART Configuration Parameter Types

The UART driver object definition is given in the Figure 16. The interface has methods to manage UART. The developed MPC5200B PSC and D16950 UART IP drivers use the same interface.

```c
// Defines UART driver type
typedef struct UARTDriver UARTDriverType;

struct UARTDriver
{
    ERROR (*open)(UARTDriverType* pDriver);
    ERROR (*read)(
            UARTDriverType* pDriver,
            uint8_t* pBuffer,
            uint32_t numberOfBytesToRead,
            uint32_t* pNumberOfBytesRead
            );
    ERROR (*write)(
            UARTDriverType* pDriver,
            const uint8_t* pBuffer,
            uint32_t numberOfBytesToWrite,
            uint32_t* pNumberOfBytesWritten
            );
    ERROR (*reset)(UARTDriverType* pDriver);
    ERROR (*close)(UARTDriverType* pDriver);
    ERROR (*enableTxNotification)(UARTDriverType* pDriver, UARTTxCallbackFuncType pUserFunction);
    ERROR (*enableRxNotification)(UARTDriverType* pDriver, UARTRxCallbackFuncType pUserFunction);
    ERROR (*disableTxNotification)(UARTDriverType* pDriver);
    ERROR (*disableRxNotification)(UARTDriverType* pDriver);
    ERROR (*getNumberOfBytesReceived)(UARTDriverType* pDriver, uint32_t* pNumberOfBytes);
    ERROR (*getTxCompletionStatus)(UARTDriverType* pDriver, UARTTxStatusType* pTxStatus);
    ERROR (*setRxNotificationLevel)(UARTDriverType* pDriver, uint32_t numberOfBytes);
    ERROR (*setDirectionOfExternalBuffer)(UARTDriverType* pDriver, UARTExtBufDirType direction);
    UARTHardwareConfigType hwConfig;
    UARTHandleType handle;
    UARTDriverErrorType error;
};
```

Figure 16 UART Driver Interface Methods

4.7.3 Digital Input/Output Driver (DIO) Driver Design

To set or get the value of any discrete data in the RSAR, a DIO driver interface has been offered.

This simple interface defines the logic level type named *DIOLevelType*.

```c
// Defines io level type.
typedef enum
{
    LOGIC_LOW = 0,
    LOGIC_HIGH = 1
}DIOLevelType;

// Defines driver type.
typedef struct DIODriver DIODriverType;

// Defines DIO driver struct.
struct DIODriver
{
    // Function pointer to initialize driver.
    // !!! Must be called before using other functions!!!
    ERROR (*initialize)(DIODriverType* pDriver);
    // Function pointer to write gpio.
    ERROR (*writeChannel)(DIODriverType* pDriver, DIOChannelType channelId, DIOLevelType level);
    // Function pointer to to read gpio.
    ERROR (*readChannel)(DIODriverType* pDriver, DIOChannelType channelId, DIOLevelType* pLevel);
    // Stores last occured error.
    DIOErrorType error;
};
```

Figure 17 DIO Driver Interface

The interface has a method pointer named “initialize” to start driver. The *writeChannel* method pointer is used to set a DIO. The *readChannel* method pointer is used to get value of a DIO. The interface definition is given in Figure 17.

```
// Creates DIODriverType
// Object construction is made with this function.
// \param id : defined "DIODriverIdType" id in "DIODriverIdDef.h"
// \return NULL or place of created DIODriverType.
DIODriverType* createDIODriver(DIODriverIdType id);
```

Figure 18 DIO Driver Constructor

To use different kind of hardware within same interface methods, a constructor method map was implemented. The example constructor map implementation according to driver identification is shown in the Figure 19. To generate different DIO driver objects, a common constructor is used.

```
static const DIODriverCtorMapType sc_DIODriverCtorMap[] = \
{
    {
        CPU_DIO_DRIVER,
        &createDIODriverMPC5200B
    },

    {
        FPGA_DIO_DRIVER,
        &createDIODriverMPC5200B
    }
};
```

Figure 19 DIO Driver Constructor Map

4.8 Design Overview

To understand the software architecture easily, some parts of a missile computer software design is given in Figure 20. The design uses the proposed layered architecture, the RSAR. All the related layers with the architecture are shown in the figure. The design in the figure is responsible to execute auto-pilot algorithm within a specific time in the main loop of schedule. Required data to execute the FMU is received using *UART Protocol Abstraction Layer* over UART driver interface. This example is specific for communication over a UART peripheral. For different hardware peripherals, a new *Data Abstraction Layer* model is required. The detailed information about using different hardware peripherals are given in the section 6.2.3.

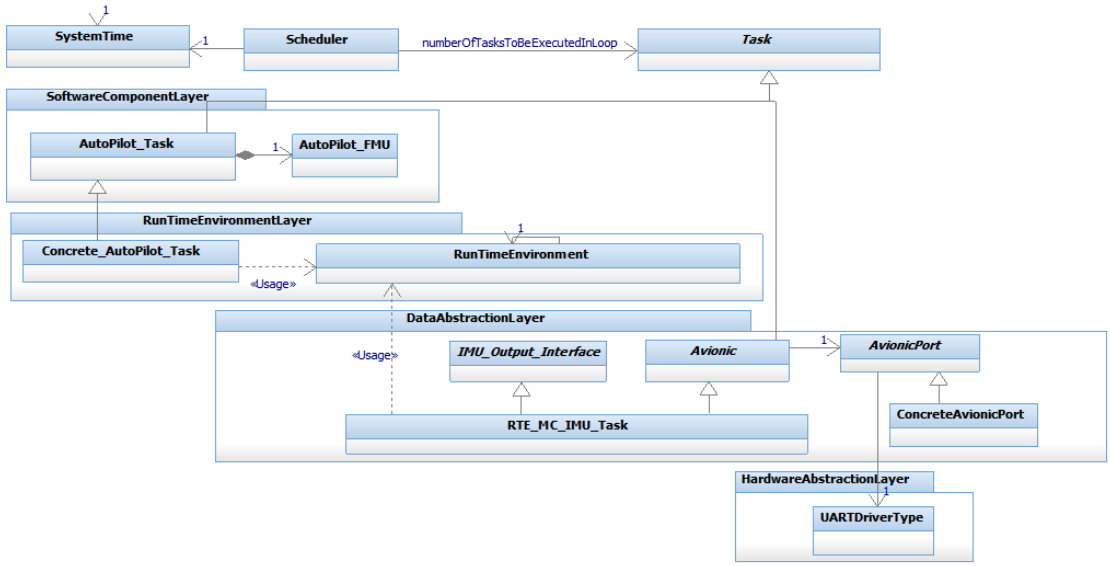

Figure 20 Example Software Design in RSAR

Scheduler class is responsible to execute tasks in given order and time. *Task* class is the base class for all other *Software Component Layer* classes.

The *Avionic* class is derived from the *Task* class. It is the main class for all other avionic classes. It is used as template class to other *Avionic* classes. Here the name *Avionic* is used for the avionics that communicates over UART with a protocol. All derived *Avionic* classes communicate with hardware over an interface named *AvionicPort*. *AvionicPort* class is responsible to check protocol details of the communication. The UART protocol messages have message identification number, message direction, message data and CRC fields. *ConcreteAvionicPort* implements the *AvionicPort* interface. It uses *UARTDriverType* interface to communicate with peripheral hardware. *UARTDriverType* interface is an actually standardized *Hardware Abstraction Layer* driver which communicates with UART peripheral.

AutoPilot_Task class is also derived from *Task* class. It includes auto-pilot algorithm as Functional Mock-up Unit with the Functional Mock-up Interface. When the *Task* instance (here *AutoPilot_Task* instance) is executed by the *Scheduler* instance, *AutoPilot_Task* instance runs the FMU.

SystemTime class provides time information to *Scheduler* class. *Scheduler* class uses *SystemTime* to run its tasks on their time.

CHAPTER 5

SYSTEM INTEGRATION PROCESS

In this chapter, software system integration process for the RSAR is described. The RSAR defines layered software architecture as introduced in the section 4.2. To generate these defined layers some scripts are used to generate code automatically. These scripts are named as the RSAR tools. Automatic code generation properties of the RSAR have been introduced in detail in this chapter.

5.1 System Build Steps

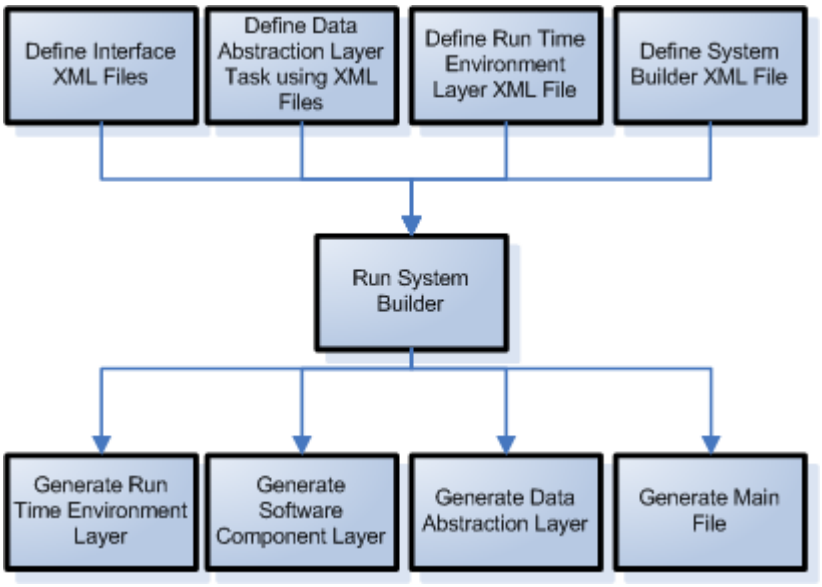

Figure 21 System Build Steps

If a software developer wants to develop a software project in the RSAR, some steps are required to follow. In the Figure 21, the steps of the software system generation are shown. To generate the RSAR software, some XML files must be created by the developer. The details of the XML files are given in the following sections. These XML file definitions are used by the RSAR tools to generate layer code automatically.

First step in the design is to define interfaces of *Software Component Layer* entities. These entities communicate with *Run Time Environment Layer* using the defined interfaces. To define an interface, an XML definition is used. The detail is given in

the section 5.2. *Software Component Layer* must communicate with hardware using *Run Time Environment Layer* and *Data Abstraction Layer*. For this purpose, *Run Time Environment* and *Data Abstraction Layer* XML files are defined by the designer. The detail is given in the sections 5.3 and 5.4.

After defining layer XML files, it is needed to define *Task* execution times. The *Scheduler* must be configured to run these *Tasks.* To configure *Scheduler, System Builder* XML file is defined by the developer. The detail is given in the 5.5.

After defining all XML files, the *System Builder* script is executed. The layers are generated by calling specific scripts of that layer. The main file is also created by this script. The main file contains created *Task and Scheduler* objects and configuration code for the objects. Every layer in the RSAR design is generated by a script automatically. To generate a layer, the responsible script is called by the *System Builder* script. The RSAR tools are these scripts to generate interface and layer code. For *Software Component Layer, Run Time Environment Layer* and *Data Abstraction Layer* code generation different scripts have been implemented.

The detail of the automatically code generation is given in the following sections.

5.2 Software Component Layer Code Generation

As given in the section 4.4, the *Software Component Layer* entities need some input and output signals. These signals are grouped into to define an interface. The interface is defined using a XML definition. This XML definition is used to generate interface code automatically. An example interface definition XML file is given in the Figure 22. As seen in the example XML file, interface name and the attribute types are defined. The interface has a name which is *AutoPilot_IMU_Input_Interface* in the example file. The attribute definitions in the XML file are used to generate interface methods in the generated interface source file.

```xml
<?xml version="1.0"?>
<interface_definition>
    <interface name="AutoPilot_IMU_Input_Interface">
        <attribute name="IMU_Tx_Flag">
            <type>bool_t</type>
            <direction>output</direction>
            <dimension>1</dimension>
        </attribute>
        <attribute name="IMU_Rx_Flag">
            <type>bool_t</type>
            <direction>input</direction>
            <dimension>1</dimension>
        </attribute>
        <attribute name="a_raw">
            <type>float_t</type>
            <direction>input</direction>
            <dimension>3</dimension>
        </attribute>
        <attribute name="euler321">
            <type>float_t</type>
            <direction>input</direction>
            <dimension>3</dimension>
        </attribute>
    </interface>
</interface_definition>
```

Figure 22 Interface Definition XML

To generate interface source file, a python script named *InterfaceGenerator* has been developed. *InterfaceGenerator* script is used seen as in Figure 23. Here, five interfaces have been given whose names can be seen in the figure.

```
interfaceNameList = ["AutoPilot_CAS_Output_Interface"];
interfaceNameList += ["AutoPilot_IMU_Input_Interface"];
interfaceNameList += ["AutoPilot_Missile_T0_Input_Interface"];
interfaceNameList += ["AutoPilot_PN_Input_Interface"];
interfaceNameList += ["AutoPilot_PN_Output_Interface"];
for interfaceName in interfaceNameList:
    interfaceGeneratorObj = InterfaceGenerator(interfaceName);
    interfaceGeneratorObj.generate();
```

Figure 23 InterfaceGenerator Script Usage

The generated interface source example can be seen in the Figure 24.

```
#ifndef AutoPilot_IMU_Input_Interface_H
#define AutoPilot_IMU_Input_Interface_H

#include "MOKA_Type_Definitions.h"

namespace MOKAPckg
{
    class AutoPilot_IMU_Input_Interface :
    {
    public:
        AutoPilot_IMU_Input_Interface();

        virtual ~AutoPilot_IMU_Input_Interface();

        virtual void set_IMU_Tx_Flag(bool_t& p_IMU_Tx_Flag) = 0;

        virtual bool_t get_IMU_Rx_Flag() = 0;

        virtual float_t get_a_raw(uint32_t arrayIndex) = 0;

        virtual float_t get_euler321(uint32_t arrayIndex) = 0;

    };
}
```

Figure 24 Generated Interface File

Interfaces are used by *Tasks* in the RSAR. An abstract *Task* can be derived from one or more interfaces. To generate abstract *Software Component*, an XML definition has been developed. In the XML file, all the interfaces needed by the component are written. An example XML file of abstract *Task* definition is given in the Figure 25.

```
<?xml version="1.0"?>
<task_definition>
    <task name="AutoPilot_Task">
        <interface name="AutoPilot_PN_Output_Interface"/>
        <interface name="AutoPilot_PN_Input_Interface"/>
        <interface name="AutoPilot_Missile_T0_Input_Interface"/>
        <interface name="AutoPilot_IMU_Input_Interface"/>
        <interface name="AutoPilot_CAS_Output_Interface"/>
    </task>
</task_definition>
```

Figure 25 Task Definition XML

To generate abstract *Task* source files, a python script named *AbstractTaskGenerator* has been developed. In the Figure 26, it can be seen how to use the script.

```
taskNameList = ["AutoPilot_Task"];

for taskName in taskNameList:
    abstractTaskGeneratorObj = AbstractTaskGenerator(taskName);
    abstractTaskGeneratorObj.generate();
```

Figure 26 AbstractTaskGenerator Usage Example

The generated source file is shown in the Figure 27. As seen, task is derived from all the interfaces and *Task* class. The only method which should be implemented by the

user is the method named *execute*. All the data is needed by the *execute* method is provided by the interface methods.

```
#ifndef AutoPilot_Task_H
#define AutoPilot_Task_H

#include "MOKA_Type_Definitions.h"

#include "Task.h"
#include "AutoPilot_PN_Output_Interface.h"
#include "AutoPilot_PN_Input_Interface.h"
#include "AutoPilot_Missile_T0_Input_Interface.h"
#include "AutoPilot_IMU_Input_Interface.h"
#include "AutoPilot_CAS_Output_Interface.h"

namespace MOKAPckg
{
    class AutoPilot_Task :
        public AutoPilot_PN_Output_Interface,
        public AutoPilot_PN_Input_Interface,
        public AutoPilot_Missile_T0_Input_Interface,
        public AutoPilot_IMU_Input_Interface,
        public AutoPilot_CAS_Output_Interface,
        public Task
    {
    public:
        AutoPilot_Task();

        virtual ~AutoPilot_Task();

    protected:
        virtual Task::TaskStateType execute();
    };
}
#endif
```

Figure 27 Generated Task Header File

Inside the *execute* method, the software developer can get input signals and can write to output signals using interface methods. By doing that, logic inside the *execute* method becomes independent from the other layers. Abstract methods are realized in the *Run Time Environment Layer*. The detail of method realization in the *Run Time Environment Layer* is given in the following section.

```
25  {
26      fmiReal tmpValue = get_a_raw(0);
27      const fmiValueReference valueIndex = 0;
28      fmu.setReal(component, &valueIndex, 1U, &tmpValue);
29  }
30
31  {
32      fmiReal tmpValue = get_a_raw(1);
33      const fmiValueReference valueIndex = 1;
34      fmu.setReal(component, &valueIndex, 1U, &tmpValue);
35  }
36  {
63
64  fmu.doStep(c, fmuTime, h, fmiTrue);
```

```
65  {
66      const fmiValueReference valueIndex = 0;
67      fmiReal delta_fin_comm_0;
68      fmiFlag = fmu.getReal(component, &valueIndex, 1U, &delta_fin_comm_0);
69      set_delta_fin_comm(delta_fin_comm_0, 0);
70  }
71
72  {
73      const fmiValueReference valueIndex = 1;
74      fmiReal delta_fin_comm_1;
75      fmiFlag = fmu.getReal(component, &valueIndex, 1U, &delta_fin_comm_1);
76      set_delta_fin_comm(delta_fin_comm_1, 1);
77  }
```

Figure 28 FMU – *Task* Integration

To use a FMU inside the RSAR *Software Component Layer*, a python script has been developed also. This script is used to generate code to glue FMU and *Software Component Layer* entity methods. An example code piece has been given in the Figure 28. Inside the code, firstly FMU inputs are set using FMI. Then the FMU is executed. Lastly, outputs of the FMU are set to *Run Time Environment Layer* by using interface methods of *Software Component Layer*.

5.3 Run Time Environment Layer Code Generation

The *Run Time Environment Layer* is generated automatically. To generate *Run Time Environment Layer* automatically, a python script has been developed. The script uses the interface definition XML files of software components to generate the layer.

To use python script, it is needed to have software component XML files. In the example given below, *AutoPilot_Task* is defined. *AutoPilot_Task* definition XML file includes all the interfaces which are needed by the task. The script gets the name

of the *Task* class and opens the XML file and starts to search interface definition XML files. Using interface XML files, all the data needed to generate *Run Time Environment Layer* is extracted. After completion of data extraction, the layer is generated.

The sample python code is given below to generate the layer.

```
from RunTimeEnvironmentGenerator import RunTimeEnvironmentGenerator

taskNameList = ["AutoPilot_Task"];

runTimeEnvironmentGeneratorObj = RunTimeEnvironmentGenerator(taskNameList);
runTimeEnvironmentGeneratorObj.generate();
```

Figure 29 *RunTimeEnvironmentGenerator* Usage Example

After calling the *generate* method *RunTimeEnvironment.h* and *RunTimeEnvironment.cpp* files are generated automatically. Some parts of the generated header file look like given below. Here, the data needed by the *Run Time Environment Layer* is generated from the task definition XML files.

```
#ifndef RunTimeEnvironment_H
#define RunTimeEnvironment_H

#include "MOKA_Type_Definitions.h"
namespace MOKAPckg
{
    class RunTimeEnvironment :
    {
    private:
        RunTimeEnvironment();

    public:
        virtual ~RunTimeEnvironment();

    private:
        static float_t x_mis_auto[3];
        static float_t v_mis_auto[3];
        static float_t a_comm[3];
        static float_t v_0_body[3];
        static float_t x_0_mis[3];
        static bool_t IMU_Tx_Flag;
        static bool_t IMU_Rx_Flag;
        static float_t a_raw[3];
        static float_t euler321[3];
        static bool_t CAS_Flag;
        static float_t delta_fin_comm[3];
    public:
        static inline void set_x_mis_auto(float_t& p_x_mis_auto, uint32_t arrayIndex)
        {
            x_mis_auto[arrayIndex] = p_x_mis_auto;
        }

        static inline float_t get_x_mis_auto(uint32_t arrayIndex)
        {
            return x_mis_auto[arrayIndex];
        }
```

Figure 30 Generated *RunTimeEnvironment* Header File

In the *Software Component Layer* abstract *Task* classes are generated automatically. The interface methods of the abstract class must be implemented and must be connected to *Run Time Environment Layer*. This is done by the python script named *ConcreteTaskGenerator*. The usage of script is shown in the Figure 31.

```
taskNameList = ["AutoPilot_Task"];

for taskName in taskNameList:
    concreteTaskGeneratorObj = ConcreteTaskGenerator(taskName);
    concreteTaskGeneratorObj.generate();
```

Figure 31 *ConcreteTaskGenerator* Usage Example

The output file of the script is shown in the Figure 32. As shown, the interface methods are implemented and connected to *Run Time Environment Layer*.

```
#ifndef Concrete_AutoPilot_Task_H
#define Concrete_AutoPilot_Task_H

#include "MOKA_Type_Definitions.h"
#include "RunTimeEnvironment.h"
#include "AutoPilot_Task.h"

namespace MOKAPckg
{
    class Concrete_AutoPilot_Task :
        public AutoPilot_Task
    {
    public:
        Concrete_AutoPilot_Task();

        virtual ~Concrete_AutoPilot_Task();

        virtual inline void set_x_mis_auto(float_t& p_x_mis_auto, uint32_t arrayIndex)
        {
            RunTimeEnvironment::set_x_mis_auto(p_x_mis_auto, arrayIndex);
        }

        virtual inline void set_v_mis_auto(float_t& p_v_mis_auto, uint32_t arrayIndex)
        {
            RunTimeEnvironment::set_v_mis_auto(p_v_mis_auto, arrayIndex);
        }

        virtual inline float_t get_a_comm(uint32_t arrayIndex)
        {
            return RunTimeEnvironment::get_a_comm(arrayIndex);
        }
```

Figure 32 Concrete Task Example

5.4 Data Abstraction Layer Code Generation

In this section, detail of the *Data Abstraction Layer* code generation has been given.

5.4.1 UART Data Abstraction Layer Code Generation

The incoming data and outgoing raw data over UART peripheral must be converted into meaningful data. The raw data is converted into meaningful data by using an

XML file definition. In the following figure, there is an XML file which has the definition of the UART protocol messages. A message has a unique number, input and output parts. Input and output data is indicated by a name, type, dimension and offset. To generate *Task* object from the XML file, it is needed to know the interface which is used to communicate with *Run Time Environment Layer*. For this purpose, the XML file contains interface section inside.

```
<?xml version="1.0"?>
<run_time_environment_uart_protocol_task_definition>
    <task name="RunTimeEnvironment_MissileComputer_IMU_Task">
        <interface_list>
            <interface name="IMU_Output_Interface"/>
        </interface_list>
        <avionic_message_definition>
            <avionic_message name="Exchange">
                <avionic_message_num>16</avionic_message_num>
                <rx_avionic_message_definition>
                    <rx_avionic_message_data_length>24</rx_avionic_message_data_length>
                    <rx_flag>IMU_Rx_Flag</rx_flag>
                    <attribute name="a_raw">
                        <type>float_t</type>
                        <dimension>3</dimension>
                        <offset>0</offset>
                    </attribute>
                    <attribute name="euler321">
                        <type>float_t</type>
                        <dimension>3</dimension>
                        <offset>12</offset>
                    </attribute>
                </rx_avionic_message_definition>
                <tx_avionic_message_definition>
                    <tx_avionic_message_data_length>0</tx_avionic_message_data_length>
                    <tx_flag>IMU_Tx_Flag</tx_flag>
                </tx_avionic_message_definition>
            </avionic_message>
        </avionic_message_definition>
    </task>
</run_time_environment_uart_protocol_task_definition>
```

Figure 33 UART Protocol Task Definition

The automatically generated *UART Protocol Abstraction Layer* class header file is shown in the Figure 34. It has methods which are defined in the XML file to communicate with the *Run Time Environment Layer*. The *setter* and *getter* methods are generated according to XML file.

```
namespace MOKAPckg
{
    class RunTimeEnvironment_MissileComputer_IMU_Task :
            public IMU_Output_Interface,
            public Avionic
    {
        explicit RunTimeEnvironment_MissileComputer_IMU_Task(const AvionicIdType& p_SubSystemId);

        ~RunTimeEnvironment_MissileComputer_IMU_Task();

    public:
        virtual inline bool_t get_IMU_Tx_Flag()
        {
            return RunTimeEnvironment::get_IMU_Tx_Flag();
        }

        virtual inline void set_IMU_Rx_Flag(bool_t& p_IMU_Rx_Flag)
        {
            RunTimeEnvironment::set_IMU_Rx_Flag(p_IMU_Rx_Flag);
        }

        virtual inline void set_a_raw(float_t& p_a_raw, uint32_t arrayIndex)
        {
            RunTimeEnvironment::set_a_raw(p_a_raw, arrayIndex);
        }

        virtual inline void set_euler321(float_t& p_euler321, uint32_t arrayIndex)
        {
            RunTimeEnvironment::set_euler321(p_euler321, arrayIndex);
        }

    protected:
        virtual void handleRxAvionicMessage(const AvionicMessage& rxAvionicMessage);
        virtual void handleTxAvionicMessage(AvionicMessage& txAvionicMessage);
```

Figure 34 Generated UART Protocol Abstraction Layer Header File

The generated glue code for receiving an avionic message is shown in the Figure 35. Messages are parsed according to their message identification number which is defined in the XML file. The raw data is converted into meaningful data according to offset definitions in the XML file.

```
    void RunTimeEnvironment_MissileComputer_IMU_Task::handleRxAvionicMessage(const AvionicMessage& rxAvionicMessage)
    {
        //Handle Rx Avionic Message here!!!
        {
            bool_t tmpValue = false;
            set_IMU_Rx_Flag(tmpValue);
        }
        if(getRxEnableFlag())
        {
            switch(rxAvionicMessage.getAvionicMessageId())
            {
                case(16):
                {
                    {
                        bool_t tmpValue = true;
                        set_IMU_Rx_Flag(tmpValue);
                    }
                    {
                        float_t tmpValue;
                        rxAvionicMessage.getValue(0 + 0 * sizeof(float_t), tmpValue);
                        set_a_raw(tmpValue, 0);
                    }
                    {
                        float_t tmpValue;
                        rxAvionicMessage.getValue(0 + 1 * sizeof(float_t), tmpValue);
                        set_a_raw(tmpValue, 1);
                    }
                    {
                        float_t tmpValue;
                        rxAvionicMessage.getValue(0 + 2 * sizeof(float_t), tmpValue);
                        set_a_raw(tmpValue, 2);
                    }
```

Figure 35 Generated Receive Handler Method

The avionic message which will be sent to UART is generated by the *handleTxAvionicMessage* method according to the XML definition of the messages. The generated method according to the XML file is shown in the Figure 36.

```
231     void RunTimeEnvironment_MissileComputer_IMU_Task::handleTxAvionicMessage(AvionicMessage& txAvionicMessage)
232     {
233         //Handle Tx Avionic Message here!!!
234         setTxEnableFlag(false);
235
236         if (get_IMU_Tx_Flag())
237         {
238             txAvionicMessage.setMessageId(16);
239             txAvionicMessage.setMessageDataLength(0U);
240             setTxEnableFlag(true);
241         }
242         setDataExchangeMode(RX_MODE);
243     }
```

Figure 36 Generated Transmit Handler Method

5.4.2 DIO Data Abstraction Layer Code Generation

In Figure 37, XML definition of a *DIO Abstraction Layer* has been given. In the XML structure, DIO driver and DIO definitions resides.

```
1   <?xml version="1.0"?>
2   <dio_data_abstraction_layer_definition>
3       <class name = "DIODataAbstractionLayer">
4
5           <dio_driver_definition_list>
6               <dio_driver_definition>
7                   <dio_driver_name>CPU_DIO_DRIVER</dio_driver_name>
8                   <dio_driver_object_name>pDIODriver</dio_driver_object_name>
9               </dio_driver_definition>
10          </dio_driver_definition_list>
11
12          <dio_definition_list>
13
14              <dio_definition>
15                  <attribute name = "mode_select_1"/>
16                  <dio_id>DIO_5200B_0</dio_id>
17                  <dio_direction>output</dio_direction>
18                  <dio_driver_object_name>pDIODriver</dio_driver_object_name>
19                  <active_logic_level>LOGIC_LOW</active_logic_level>
20              </dio_definition>
21
22              <dio_definition>
23                  <attribute name = "mode_select_2"/>
24                  <dio_id>DIO_5200B_1</dio_id>
25                  <dio_direction>output</dio_direction>
26                  <dio_driver_object_name>pDIODriver</dio_driver_object_name>
27                  <active_logic_level>LOGIC_HIGH</active_logic_level>
28              </dio_definition>
```

Figure 37 DIO Data Abstraction Layer XML

```
namespace MOKAPckg
{
    class DIODataAbstractionLayer:
    {
    public:
        DIODataAbstractionLayer();
        virtual ~DIODataAbstractionLayer()
    public:
        inline void set_mode_select_1(bool_t& p_mode_select_1)
        {
            if (p_mode_select_1)
            {
                pDIODriver->writeChannel(pDIODriver, DIO_5200B_0, LOGIC_LOW);
            }
            else
            {
                pDIODriver->writeChannel(pDIODriver, DIO_5200B_0, LOGIC_HIGH);
            }
        }
        inline bool_t get_motor_ignition()
        {
            DIOLevelType level;
            bool_t boolValue;
            pDIODriver->readChannel(pDIODriver, DIO_5200B_2, &level);
            if (LOGIC_HIGH == level)
            {
                boolValue = true;
            }
            else
            {
                boolValue = false;
            }
            return boolValue;
        }
```

Figure 38 DIO Data Abstraction Layer Header File

In Figure 38, the generated header file is given. From the XML definition, the methods are generated. These methods are used by *Run Time Environment Layer* to make connection with *Software Component Layer*.

```
#include "DIODataAbstractionLayer.h"

namespace MOKAPckg
{
    DIODataAbstractionLayer::DIODataAbstractionLayer()
    {
        pDIODriver = createDIODriver(CPU_DIO_DRIVER);
        pDIODriver->initialize(pDIODriver);
    }

    DIODataAbstractionLayer::~DIODataAbstractionLayer()
    {

    }
}
```

Figure 39 DIO Data Abstraction Layer CPP File

In the generated CPP file, the DIO drivers used in the software design are created and initialized as shown in the Figure 39.

5.5 System Builder Definition

To generate *Task* objects used in the software design, an XML definition has been introduced in RSAR. An example of the XML file is given in Figure 40. This file is used by the system builder script to generate objects used in the software design.

```xml
<?xml version="1.0"?>
<system_builder_definition>

    <scheduler_setup_definition>
        <object_name>pScheduler</object_name>
        <number_of_tasks_to_execute>5</number_of_tasks_to_execute>
        <loop_time_in_microseconds>10000</loop_time_in_microseconds>
    </scheduler_setup_definition>

    <run_time_environment_definition name = "RunTimeEnvironmentLayer">
        <generator_script_name>RunTimeEnvironmentGenerator</generator_script_name>
        <name_space name = "MOKAPckg"/>
        <type_include_file name = "MOKA_Type_Definitions"/>
    </run_time_environment_definition>

    <task_list_definition>
        <task_definition name = "AutoPilot_Task">
            <object_name>pAutoPilot_Task</object_name>
            <generator_script_name>Task_Generator</generator_script_name>
            <execution_step_list>
                <execution_step_definition>
                    <execution_step>0</execution_step>
                    <execution_time_in_microseconds>0</execution_time_in_microseconds>
                </execution_step_definition>
                <execution_step_definition>
                    <execution_step>1</execution_step>
                    <execution_time_in_microseconds>0</execution_time_in_microseconds>
                </execution_step_definition>
            </execution_step_list>
        </task_definition>

        <task_definition name = "CAS_Task">
            <object_name>pCAS_Task</object_name>
            <generator_script_name>Task_Generator</generator_script_name>
            <execution_step_list>
                <execution_step_definition>
                    <execution_step>2</execution_step>
                    <execution_time_in_microseconds>1000</execution_time_in_microseconds>
                </execution_step_definition>
            </execution_step_list>
        </task_definition>
```

Figure 40 System Builder XML File

In the XML file, *Scheduler* object configuration values are given. *Task* object configuration values are also given in the XML file. Inside the given example, there are five *Task* objects to run by the *Scheduler*. The loop time is defined as 10 milliseconds.

To generate *Run Time Environment Layer*, name of the layer and code generation script is defined inside the XML file.

As seen, *Tasks* are defined inside the XML file. The name of the *Task* and its code generation script is defined inside the file. Execution step and time of the *Task* is used to introduce the *Task* object to the *Scheduler*. By doing this definition, real time characteristic of the software system is defined by using the XML file.

CHAPTER 6

CASE STUDY AND EVALUATION

6.1 Case Study

The overview design of the implementation project named *RSAR Demonstrator* is shown in the Figure 41. Design includes three separated hardware. Using RS232 electrical interface, hardware connection was established. In scenario for the implementation, *Missile Computer* requests the IMU data and *Simulated IMU* sends the simulated IMU data back. Then *Missile Computer* calculates fin angles and sends them to *CAS*. *CAS* is used to log data sent by the *Missile Computer*. The *Missile Computer* software has been developed using the RSAR layered architecture and the RSAR tools. The design consists of the layers of the RSAR given in the Figure 3.

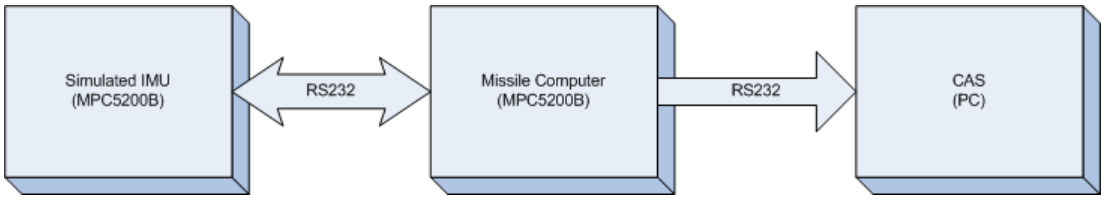

Figure 41 Implemented Design Overview

6.1.1 Algorithm Design

To demonstrate thesis work a guidance algorithm has been developed. The algorithm is an FMU which is used within a *Task* object in the design. The algorithm consists of two blocks which are shown in the Figure 42. The figure shows us the modeling of the FMU. As shown in Figure 3, the *Software Component Layer* can include *Tasks* with FMU. This FMU is used inside the *Task* object in the design.

The guidance algorithm has been developed in the environment of MATLAB Simulink.

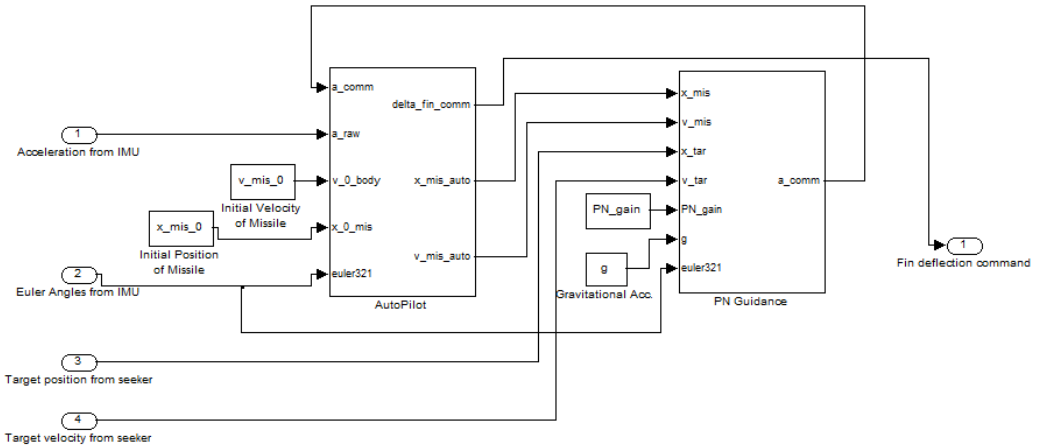

Figure 42 Algorithm Block Relations

Input and output signal definition of the algorithm design is described as shown in the Table 1.

Table 1 Algorithm Signals

Signal Name	From	To	Description
Acceleration from IMU	Simulated IMU	Autopilot	Raw three axial acceleration data
Euler Angles from IMU	Simulated IMU	Autopilot	Missile attitude angles
x_tar	Simulated Seeker	PN Guidance	Three axial target position data
v_tar	Simulated Seeker	PN Guidance	Three axial target velocity data
a_comm	PN Guidance	Autopilot	Accleration command generated by guidance
v_miss_0	Mission parameter	Autopilot	Missile initial velocity
x_miss_0	Mission parameter	Autopilot	Missile initial position
v_miss	Autopilot	PN Guidance	Calculated misille velocity state
x_miss	Autopilot	PN Guidance	Calculated misille position state
PN_gain	Missile parameter	PN Guidance	PN Guidance gain parameter
g	Mission parameter	PN Guidance	Gravitational acceleration
delta_fin_comm	Autopilot	CAS	Fin deflection command

The algorithm includes two blocks which are described as follows.

6.1.1.1 *AutoPilot* Block

This block estimates the position of the missile. Then for the given acceleration command from guidance block it calculates proper total control input for the two axes of the missile which are rudder and elevator. Rudder command is used to maneuver in yaw direction and elevator command is for pitch maneuver. Afterwards these total maneuver command are distributed to four different control surfaces.

6.1.1.2 *PN* Guidance Block

Purpose of this block is to calculate the acceleration command in order to maneuver the missile to intercept the target. To achieve this maneuver proportional guidance law algorithm is implemented. This algorithm calculates the commanded acceleration proportionally to the line of sight change between the missile and target. This block takes the position of missile and target from autopilot and seeker block and calculates the line of sight change of the target. Then, it compensates the gravitational acceleration.

6.1.2 Real-time Software Design

In the design, firstly interfaces have been defined according interaction of the software. Then *Tasks* have been defined. The *Task* classes used in the design are

- *RTE_MC_IMU_Task*
- *Seeker_Task*
- *PN_Task*
- *AutoPilot_Task*
- *RTE_MC_CAS_Task*

In the implementation project, *Missile Computer* software runs at 100 milliseconds interval. This information is used by the system builder script to configure *Scheduler*. The *Scheduler* object runs *Task* objects periodically according to their running time definition in the XML file.

RTE_MC_IMU_Task object is used to communicate with *Simulated IMU*. It sends request to *Simulated IMU* and parse the incoming data. The parsed data is set to the *Run Time Environment Layer*. The object is called two times in the main loop. First call is to request IMU data. The second call is to parse incoming IMU data. This class is automatically generated with using the XML definition. The class generation operation is described in 5.4.1.

The *Seeker_Task* object is used to simulate seeker data. The *Concrete_Seeker_Task* class is generated automatically using the XML definition. The steps to generate a concrete class are described in section 5.3. The seeker data generated by the object is set to the *Run Time Environment Layer*.

PN_Task and *AutoPilot_Task* objects include *FMU*s inside. *Concrete_PN_Task* and *Concrete_AutoPilot_Task* classes are generated automatically using the XML definitions. The class generation operation is described in sections 5.2 and 5.3. The input data is get from the *Run Time Environment Layer*. The FMU is executed. Then, the output data is set to *Run Time Environment Layer*.

RTE_MC_CAS_Task object is used to send the fin angle data to CAS which is a PC in the *RSAR Demonstrator* project. It gets the data which is generated by the *AutoPilot_Task* from *Run Time Environment Layer* and the data is formatted to send over UART. The formatted data is sent by using UART driver over RS323 electrical interface. This class is automatically generated with using the XML definition. The class generation operation is described in 5.4.1.

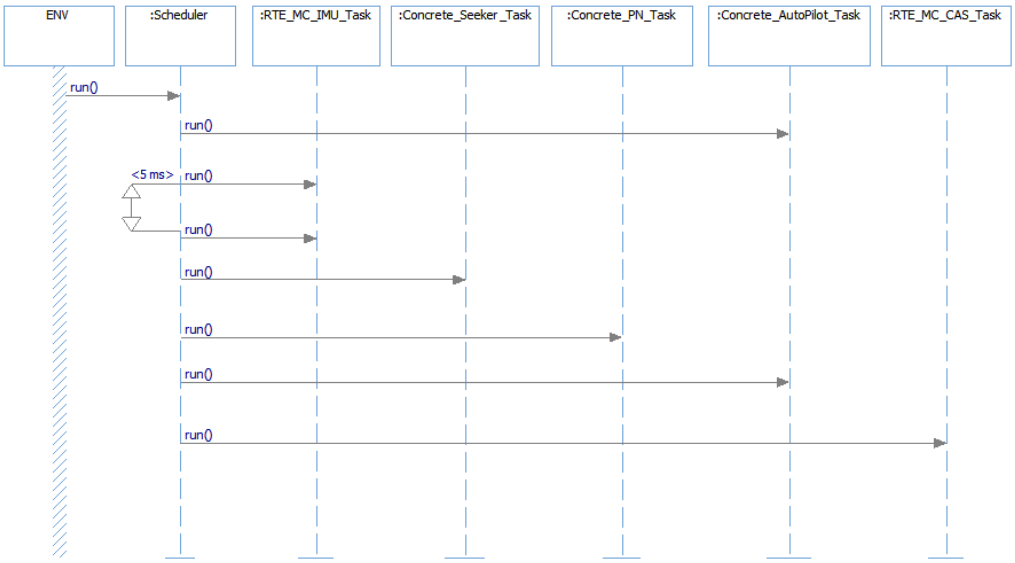

Figure 43 Missile Computer Sequence Diagram

The sequence diagram of the *Missile Computer* software is given in the Figure 43. In the scenario of *RSAR Demonstrator* project, the *Missile Computer* sends request to the *Simulated IMU* within 100 milliseconds period. Then, the *Missile Computer* waits for the response for five milliseconds. The *Simulated IMU* software sends the simulated IMU data as answer to request. In the Figure 44, the communication between the *Missile Computer* and the *Simulated IMU* is shown.

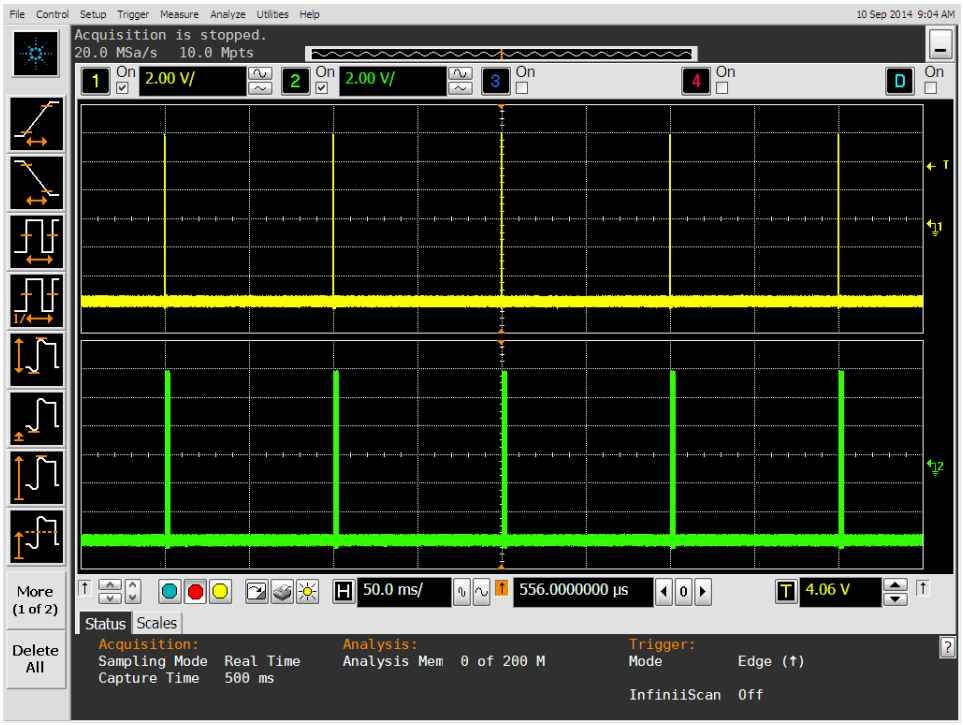

Figure 44 Communication Between *Missile Computer* and *Simulated IMU*

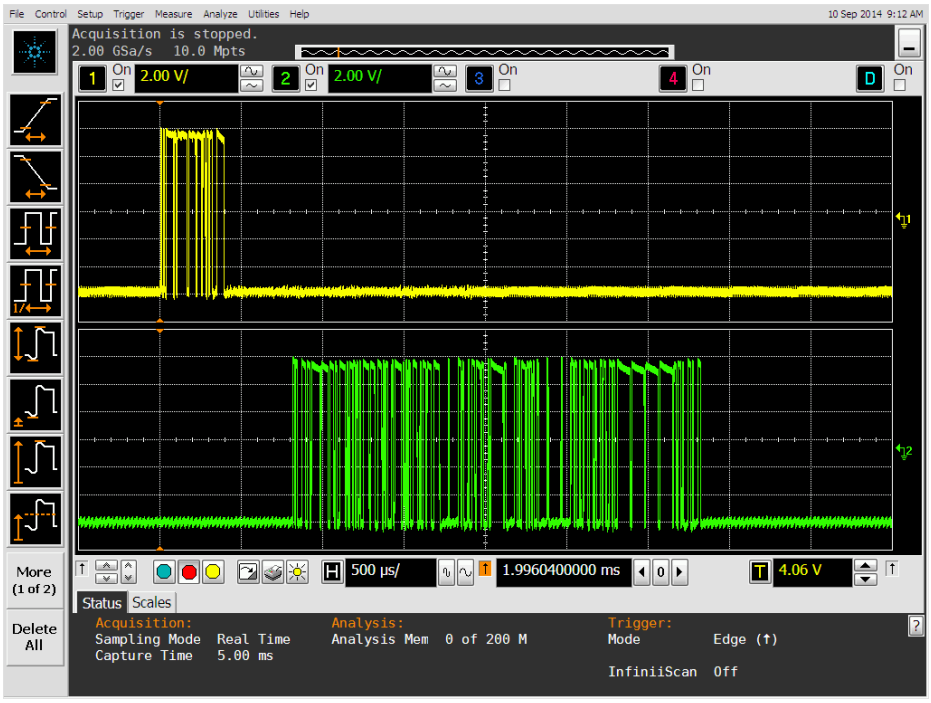

Figure 45 Detailed RS232 Communication

In the Figure 45, the detailed RS232 communication between *Missile Computer* and *Simulated IMU* is shown. Firstly, the *Missile Computer* sends the IMU data request

message and then the *Simulated IMU* sends the answer message which is shown in the oscilloscope screenshot.

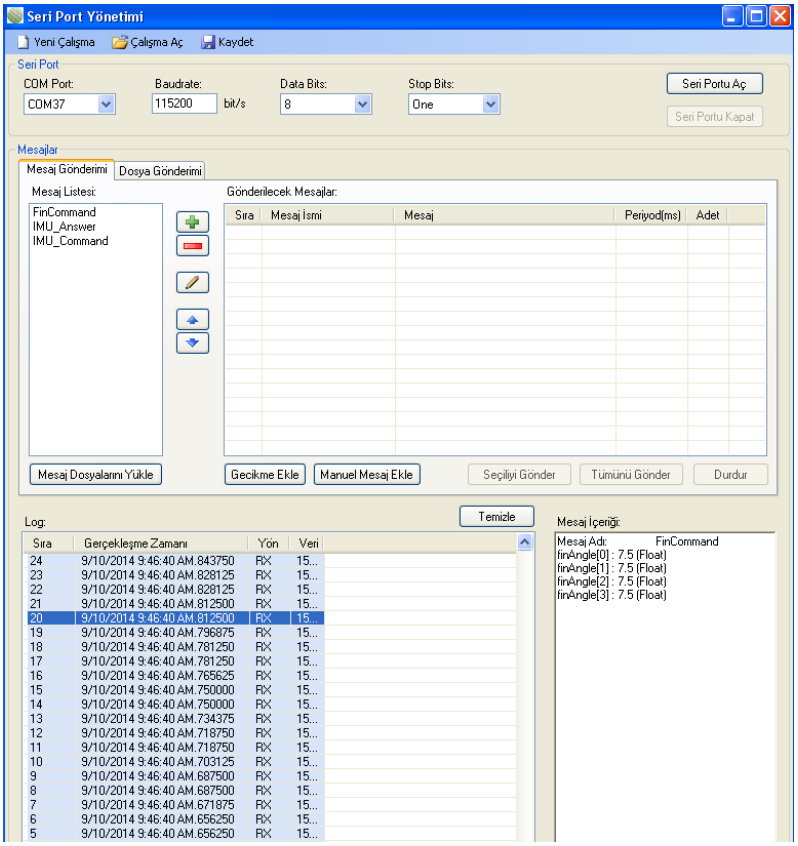

Figure 46 CAS Logs

The messages sent to CAS are logged on the PC. The angle messages sent to CAS is shown in the Figure 46. By following the messages received on the PC side, the correctness of the output values are checked against simulation results and time.

By implementing the case study, the real time operation of the RSAR has been shown in detail.

6.2 Evaluation

In this section, performance and development benefit evaluation of the RSAR layered software architecture has been explained.

6.2.1 Performance Evaluation

To evaluate the RSAR performance, some benchmark software has been developed. Here, the detail of the study has been given. To evaluate performance reduction

between layers of RSAR, three benchmark cases have been developed. First benchmark case is for to measure UART protocol message transmit time. Aim of the benchmark is to see how much more time is required to transmit a message over UART in the RSAR design than in a tightly coupled software-hardware design. To measure UART message transmit time, two test messages have been defined.

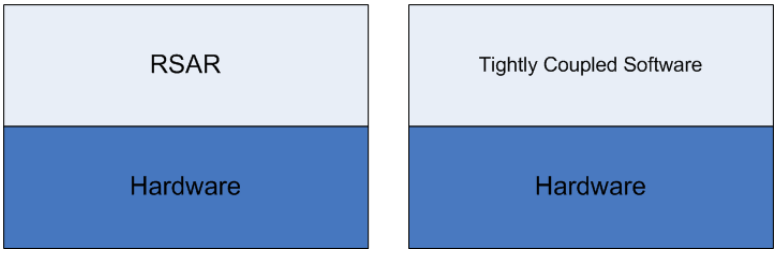

Figure 47 Benchmark Design

Table 2 Test Message 1 Attribute Content

Attribute No	**Type**	**Dimension**
1	uint8_t	1
2	uint16_t	2
3	uint32_t	4
4	uint64_t	8
5	float_t	4
6	double_t	8
7	uint32_t	16
8	int8_t	1
9	int16_t	4
10	int32_t	3

Table 3 Test Message 2 Attribute Content

Attribute No	Type	Dimension
1	uint8_t	64
2	uint32_t	12
3	float_t	8
4	double_t	5
5	int32_t	4
6	float_t	12
7	int8_t	2

The detail of the test messages are given in the Table 2 and Table 3.

The RSAR software uses layers defined in architecture to transmit test messages over UART peripheral. The tightly coupled software-hardware design accesses directly to hardware to send messages over UART peripheral. The tightly coupled software-hardware design converts message attributes into convenient format by doing memory copy operations. Then prepares message header, adds CRC of the message and sends it over UART peripheral. On the other side, RSAR design uses layers. The *Software Component Layer* entity sets the values of the attributes defined. The values are stored in the *Run Time Environment Layer.* The *Data Abstraction Layer* auto-generated *Task* generates the message and sends it using RSAR template classes to UART peripheral.

Table 4 Message Transmit Time Comparison

RSAR Average Performance	Tightly Coupled Software Average Performance
56 microseconds	48 microseconds

The result of the first benchmark case is given in the Table 4. The RSAR design layers are using just more 8 microseconds to transmit UART messages.

The second benchmark case is for to measure performance of DIO usage in RSAR. Aim of the benchmark is to see how much more time is required to control DIOs in the RSAR design than in a tightly coupled software-hardware design. Eight DIOs has been determined for the benchmark. Three of the DIOs are used as input and five of the DIOs are used as output. Ten times inputs are captured and outputs are set and time is measured. In the RSAR design these operations are performed by using defined RSAR layers. On the other hand, tightly-coupled software-hardware design uses hardware directly to control determined DIOs. The result of the benchmark case has been given in the Table 5.

Table 5 DIO Access Time Comparison

RSAR Average Performance	**Tightly Coupled Software Average Performance**
50 microseconds	46 microseconds

The third benchmark case is for to measure FMU execution performance in RSAR. Aim of the benchmark is to see how much more time is required to execute a FMU inside RSAR *Task*. For this purpose, the FMU which is given in section 6.1 has been used. The FMU has been executed inside the RSAR *Task* and directly by calling FMI methods. The execution repeated ten times and the median time measured. The result of the benchmark case is seen in the Table 6.

Table 6 FMU Execution Time Comparison

FMU Execution Performance Inside the RSAR *Task*	**FMU Execution Performance without the RSAR *Task***
41 microseconds	35 microseconds

6.2.2 Development Benefit Evaluation

To measure design benefits of RSAR, a missile guidance computer software project has been studied. The missile guidance computer has communication connection with various avionics seen in the Figure 48. There are four kinds of avionics, one external world line and two telemetries in the studied project. The guidance computer software communicates with UARTs and DIOs. There is a potential to design this software project using the layers of the RSAR given in the Figure 3.

The aim of the study is to evaluate how much percentage of source code of the software project could be automatically generated by using RSAR tools developed in the thesis work. The software project has logs which show development details in time. The logs kept using JTRAC application. The developer enters the development steps of the project into the application. By doing that, it is possible to observe how much time was consumed to develop a specific part of the software project. After finding the parts of the software project could be generated by the RSAR tools and the time consumed to develop this part actually, shortening of time in the project calendar can be found. This information shows us the benefit of the RSAR usage in a real software project.

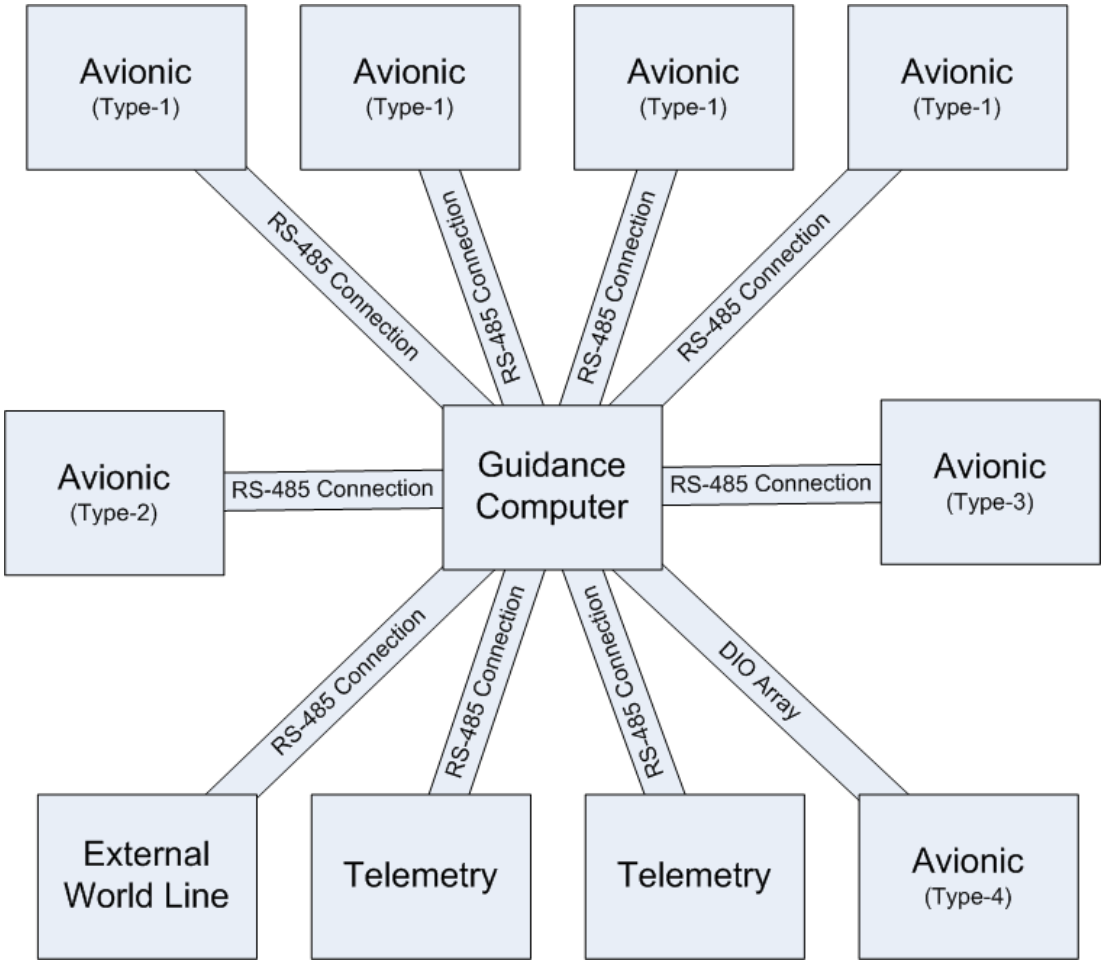

Figure 48 Guidance Computer Connections

Some of the properties of the studied software project are given

- includes 136 files, 68 classes
- have 520 methods
- contains 16104 lines of code,
- controls 7 DIOs including 5 outputs and 2 inputs
- have connections with four kind of avionics, one external world line and 2 telemetries
    - The total numbers of exchanged messages between avionics and external world line are 25. It can be seen in the Table 7.

Table 7 Missile Guidance Computer Exchanged Message Numbers

Connected System	**Number Of Messages Exchanged**
Avionic Type-1	1
Avionic Type-2	5
Avionic Type-3	1
External World Line	18

To determine the percentage of automatically generated code possible in the studied project, firstly classes have grouped according to their duties in the software system. The groups are determined according to the RSAR content. The determined groups are

- Scheduling and timing classes,
- Message processing classes,
- Message parsing classes,
- DIO control classes,
- UART control classes,
- Algorithm execution classes,
- Out of the RSAR scope classes.

After classes were grouped, source code of the classes has been examined. Groups are examined with respect to counterpart in the RSAR. Then classes have been grouped again according to possibility of automatically code generation. At the end of the study, it has been determined that

- 25 classes in 68 classes, 5391 lines in 16104 lines of code could be generated automatically by the RSAR tools.

The first fully functional version of the studied software project was developed in 5 months. When the development logs of the software project examined, it is seen that the parts could be generated automatically by the RSAR tools have been developed in 1.4 months.

As a result, by using the RSAR tools

- 36.8% of the classes and 33.48% of the total lines of code could be generated automatically,
- development time of the software project could be shortened by 28% at maximum. If automatic code generation process preparation is taken into account, development time of the software project could be shortened by up to 24 %.

The model usage in the RSAR could reduce the development time of the software system. Thanks to reducing development time, the costs of the system would reduce also.

6.2.3 Software Development on Different Hardware

In this section, the software development evaluation of the RSAR has been introduced for different scenarios from the hardware perspective.

To carry an existing RSAR software project on a new MPU/MCU platform, *Hardware Abstraction Layer* must be implemented for the new MPU/MCU. The RSAR defines *HAL* interfaces for some peripherals which are given in the 4.7. For the hardware peripherals used in the existing software project, hardware peripheral drivers for the new MPU/MCU platform must be implemented according to the defined *HAL* interfaces. By implementing the drivers for the new MCU/MPU platform, the existing RSAR software can be executed on the new MCU/MPU platform. As seen in the Figure 3, the layers above the *HAL* remain same without any change in this scenario. All the RSAR tools used to generate code automatically remain same also.

To use a new hardware peripheral in the RSAR, a *HAL* driver interface must be defined in a similar manner which has been illustrated for GPT, UART and DIO drivers in the 4.7. Then driver for the new peripheral must be implemented according to defined interface. The raw data manipulated by the driver must be interpreted into meaningful data in the *Data Abstraction Layer*. A data abstraction model is required for the new hardware peripheral. A new script must be coded to generate code automatically for the RSAR architecture. For this scenario, the *Run Time Environment Layer* and the *Software Component Layer* are designed without any change.

CHAPTER 7

CONCLUSION

This thesis work represents a study to develop a software infrastructure which can be used to develop cyber-physical systems. It has been shown how to develop layered software architecture for cyber-physical systems. It is offered to develop dynamic models as FMI to break tool dependency while developing dynamic models. Thanks to layered software architecture and the FMI standard, it is shown that hardware and software dependency can be broken.

Automatic code generation models have been given for the layers defined in the RSAR. By doing a case study, the steps required to design a real time software inside the RSAR has been shown. By doing performance evaluation, performance reduction between the RSAR layers has been evaluated. By doing development benefit evaluation, it has been shown that how the RSAR can shorten development time of a software project.

The aims of the RSAR have been achieved. By using the layered software architecture solution, hardware dependency has been minimized / eliminated. By using automatic code generation methods in the RSAR, design and development phases of the software project has been shortened. The RSAR design has provided code re-usability between various software projects. Modeling tool dependency has been broken in the RSAR by using FMI. By providing layered software models which contain C++ classes and automatic glue code generation to these template models, implementation errors have been reduced.

In future, offered layered software architecture, the RSAR, can be expanded with new hardware interfaces such as SPI and I2C. For these hardware interfaces *Data Abstraction Layer* models can be implemented to provide more automatic code generation coverage in the RSAR.

REFERENCES

[1] Modeling Cyber–Physical Systems, Derler, P.; Lee, E.A.; Vincentelli, A.S., Proceedings of the IEEE , vol.100, no.1, pp.13,28, Jan. 2012

[2] Systems Engineering for Cyber-Physical Products, Bernard Clark, Dassault Systems, Feb. 2012

[3] Applying AUTOSAR in Practice, Jesper Melin, Daniel Boström, 2011

[4] AUTOSAR Basics – autosar.org/about/basics/, Dec. 2014

[5] AUTOSAR Technical Overview – autosar.org/about/technical_overview/, Dec. 2014

[6] AUTOSAR Runtime Environment and Virtual Function Bus – Nico Nauman – Department for System Analysis and Modeling Hasso-Plattner Institute for IT-Systems Engineering, Prof. Dr. Helmert Str. 2-3, D-14482 Potsdam, 2009

[7] AUTOSAR Software Architecture – Robert Warschofsky – Hasso-Plattner-Institute für Softwaresystemtechnik, 2009

[8] FMI Basics – fmi-standard.org, Dec. 2014

[9] Using the Functional Mockup Interface as an Intermediate Format in AUTOSAR Software Component Development – Bernhard Thiele, Dan Henriksson, German Aerospace Centre (DLR), Institute for Robotics and Mechatronics, Germany, Mar. 2011

[10] Powertrain Co-Simulation using AUTOSAR and the Functional Mockup Interface Standard – Christoph Stoermer, Ghizlane Tibba, ETAS GmbH, Stuttgart, Germany, Jun. 2014

[11] MPC5200B User's Manual, Rev. 3, Freescale Semiconductor, May 2010

[12] PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors, 02/21/2000, IBM