CHALMERS

Use of Synchronized AUTOSAR for Adaptive Real Time Scheduling of Distributed Systems

Master of Science thesis in Computer Science and Engineering

MAYANK ARYA
Embedded Electronic System Design
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2014
Master's thesis 2014

The Authors grant to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Authors warrant that they are the authors to the Work, and warrant that the Work does not contain text, pictures or other material that violates copyright law.

The Authors shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Authors have signed a copyright agreement with a third party regarding the Work, the Authors warrant hereby that they have obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Use of Synchronized AUTOSAR for Adaptive Real Time Scheduling of Distributed Systems

MAYANK ARYA

© MAYANK ARYA, 2014

EXAMINER: JAN JONSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone +46(0)31 772 5220

Department of Computer Science and Engineering
Göteborg, Sweden, 2014

Acknowledgements

During the course of master thesis project I have received continuous help, support and supervision from many people who have been very crucial in the light of the project.
Professor Jan Jonsson who have been my supervisor and examiner at Chalmers. He helped me with the issues in the report and necessary inputs about structuring it. Besides the current context I would also like to thank him for providing us a splendid course work in parallel and distributed real time systems without which it would have been very hard to even understand the context of the project.
Johan Ekberg who have been my supervisor at ARCCORE AB. He helped me in nurturing the scope of the project and guiding me towards a concrete direction. Besides he provided me all the software and the hardware tools necessary for working on the project.
Adithya Manjunath who gave his precious time out of his very busy schedule in helping me debug the hardware and software issues throughout the course of the project.
Marten Hildell without whose guidance it would have been a daunting task to understand the operating system kernel architecture used in the project.
My father Dr. J Singh Arya and my mother who have been my constant source of inspiration since childhood.

Abstract

An automotive system may comprise of a distributed ECU system. The tasks can be distributed over different nodes in the network. The distributed tasks on different nodes may communicate with each other. Thus the synchronization of these distributed tasks is an important constraint for automotive software development. Moreover, the growing number of ECU's inside the vehicle has lead to an increase in demands for bandwidth and a need for high level of fault tolerance[1] in the communication network. In order to facilitate ECU software development, a group of automobile manufacturers, suppliers and tool developers have specified a standardized automobile software, AUTOSAR (Automotive Open System Architecture)[2]. Also noteworthy is the point that implementation of such a system adhering to an open standardized automotive software architecture, such as an AUTOSAR greatly increases the usage of the design. Besides the motivation of usability of the implemented system, the specification of AUTOSAR suggests several operating system level features which can be utilized to synchronize the task execution to a global time base of the network. In context to it, the choice of communication standard deployed for a distributed network is also an important factor of consideration. One such communication protocol is the Flexray protocol[3] that supports clock synchronization mechanism for the local clocks at the nodes in the network to a calculated global time base. Apart from it the Flexray communication architecture is time deterministic, fault tolerant and offers a higher bus bandwidth compared to the existing communication protocols such as a CAN[4] or a LIN[5]. The realisation of such a system that supports adaptive clock synchronization during run time is implemented in ARCTIC Studio[6] environment. The ARCTIC Studio environment supports a C development environment that supports module plug-ins to facilitate efficient development of AUTOSAR solutions. The layered software AUTOSAR architecture comprising of all the necessary modules can be integrated with a Flexray communication stack to implement the system. The testing of such a system is performed using simulators and debug analyzers, to ensure that all timing properties are fulfilled.

ABBREVIATIONS

API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basic SoftWare
CAN	Controller Area Network
CC	Communication Controller
COM	Communication Module
CRC	Cyclic Redundancy Check
ECU	Electronic Control Unit
ECUC	Electronic Control Unit Configuration
ECUM	Electronic Control Unit Module
FIFO	First In First Out
Fr	Flexray
FrIf	Flexray Interface
I-PDU	Interaction Layer Protocol Data Unit
IP	Intellectual Property
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
LAN	Local Area Network
LIN	Local Interconnect Network
L-PDU	Link layer Protocol Data Unit
MAC	Media Access Control
MCAL	MicroController Abstraction Layer
MCU	Module Configuration Unit
MCR	Module Configuration Register
MFD	Multiplication Factor Divider
N-PDU	Network layer Protocol Data Unit
NIT	Network Idle Time
ODEEP	Open Dependable Electrical Electronic Platform
OS	Operating System
OSEK	Open Systems and their Interfaces for the Electronics in Motor Vehicles
PCR	Protocol Configuration Register
PDU	Protocol Data Unit
PIER0	Protocol Interrupt Enable Register0
PIFR0	Protocol Interrupt Flag Register0
POC	Protocol Operation Control
POCR	Protocol Operation Control Register
PSR1	Protocol state register1
PREDIV	Predivider
RFD	Reduced Frequency Divider
ROM	Random Access Memory
RTE	Run Time Environment
SC	Scalability Class
SDU	Service Data Unit
SIMPPC	Simulator For Power PC
SWC	Software Component
TDMA	Time Division Multiple Access
UDE	Universal Debug Engine
USB	Universal Serial Bus

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Customer . . . 1
- 1.3 Purpose . . . 2
- 1.4 Project Materials . . . 2

2 Goal **3**
- 2.1 Compulsory goals . . . 3
- 2.2 Optional goals . . . 4

3 Theory **5**
- 3.1 AUTOSAR . . . 5
  - 3.1.1 Layered software architecture . . . 5
  - 3.1.2 AUTOSAR communication stack . . . 7
  - 3.1.3 Flexray communication architecture . . . 9

4 Development Environment **13**
- 4.1 Software environment . . . 13
- 4.2 Hardware environment . . . 13

5 Implementation **15**
- 5.1 OS Schedule Table . . . 15
- 5.2 Flexray communication stack . . . 19
- 5.3 Interaction with other modules . . . 22
  - 5.3.1 Port module . . . 23
  - 5.3.2 Module configuration unit . . . 23
  - 5.3.3 Flexray state manager . . . 23
  - 5.3.4 Interaction of OS schedule table with Flexray communication stack . . . 25
  - 5.3.5 Call back functions . . . 26

6 Configuring the development environment **27**
- 6.1 Arctic studio BSW builder . . . 27
  - 6.1.1 OS . . . 27
  - 6.1.2 FrIf . . . 28
  - 6.1.3 Fr . . . 29
  - 6.1.4 MCU . . . 29
  - 6.1.5 Port . . . 30
  - 6.1.6 ECUM . . . 31
  - 6.1.7 ECUC . . . 31

7 Test and verification **32**
- 7.1 Test and verification of OS schedule table . . . 32
  - 7.1.1 Feature testing . . . 32
  - 7.1.2 Functional testing . . . 33
- 7.2 Test and verification of Flexray communication stack . . . 34
- 7.3 Unit level testing . . . 35

8 Results **36**
- 8.1 Inter-ECU synchronization . . . 36
- 8.2 Intra-ECU synchronization . . . 36

9 Discussion **37**

10 Conclusion **38**

11 Appendix **41**
11.1 Flow charts 41

1 Introduction

The automotive industry has developed AUTOSAR standard, an open standardized automotive software architecture in order to pave the way for innovative electronic system that aids in improving performance, safety and friendliness in an automotive system. The project dealt with adaptive real time scheduling of a distributed network in an AUTOSAR system. This goal could be achieved by synchronizing the local clocks in the distributed ECU units to a global network time base. The IP core used in the project was provided by ARCCORE AB with MPC5567[7] processor provided by Freescale semiconductor as the central processing unit. The system was implemented on ODEEP[8] and MPC5567EVB[9] development boards provided by QRTECH AB and Freescale semiconductor respectively. An NXP Semiconductor designed Flexray transceiver TJA1080[10] was used within the project for meeting the bus voltage level requirements.

1.1 Background

A distributed system is often implemented for achieving a synchronized behaviour. The algorithms in real time vehicular processors are often executed in round robin manner. If the clocks at every node are synchronized , algorithms designed for synchronous system can be employed to execute in "round robin manner"[11]. For hard real time systems synchronization of individual clocks becomes even more important, where it is of utmost concern to preserve a logical or temporal ordering of tasks in the system. Each processor node in a distributed system has its own hardware clock, which can drift due to ageing or temperature variations. For distributed systems in vehicles the following two problems can arise if the clocks are not synchronized:

- In a distributed system, the sensor data acquisition may be time bound and needed to be carried out at fixed time points with respect to an algorithm that process this sensor data. Also these algorithms may be processed in a set of different processors. This fixed timing relationship cannot be maintained unless all clocks of the processors in the execution of algorithms are synchronized. Similar problem can also arise for jobs that are waiting for data to launch the vehicle control system actuators synchronously.

- There may exist a precedence relationship among tasks distributed on different processors. Since the pre-run time scheduling algorithms which are commonly used to schedule tasks on different processors run on individual clocks, the only way to guarantee the precedence constraint is to maintain a good clock synchronization among processors.

In order to synchronize the logical clocks in a distributed system a slower processor is always forced to jump forward which results in unfinished or unscheduled tasks with high utilization factor which makes the system unpredictable. Also noteworthy is the point that moving the clock back does not make any sense as a time stamp cannot occur before the cycle rounds of to an initial value. Large number of synchronization messages exchanged in the system also results in inefficient utilization of the system bus.

1.2 Customer

This project was part of master thesis in the master program of Embedded Electronic System Design, for ARCCORE AB as the customer. Professor Jan Jonsson at Chalmers University Of Technology acted as the examiner and supervised the project.

1.3 Purpose

It was an intended to introduce a time deterministic communication behaviour between distributed ECU units in the network. Also noteworthy was the point that execution of tasks/instructions on individual ECU units needed to be synchronous in order to reduce the jitter between the communicating tasks. The purpose of this project was to implement a time deterministic bus architecture which provided a global time base for a distributed ECU network unit, to which the execution of communicating tasks could synchronize .

1.4 Project Materials

This section described the hardware and software utilities on windows 7 OS that were required for the project:

- ODEEP development board provided by QRTECH. It supported two Flexray interfaces with an onboard processor named MPC5567 produced by Freescale.
- MPC5567EVB development board provided by Freescale for MPC5567 processor supporting two Flexray interfaces.
- TJA1080 Flexray Transceivers, produced by NXP semiconductor for data transmission and reception over the physical bus.
- An open source development platform called Arctic Studio with an Eclipse IDE for C/C++ code development. A source code comprising of the operating system kernel supporting several I/O modules for various microcontrollers acting as the backbone code for the development process.
- UDE[12], a platform to develop, test and maintain microcontroller software applications.
- WinIDEA iSYSTEM's[13] integrated development environment, a tool for embedded software development and testing .
- A power PC simulator tool[14] provided by Lauterbach inc.

2 Goal

The project had certain compulsory goals that needed to be achieved in order for the project to be considered as a success. There also existed another set of goals that were optional. They were not critical to fulfil but represented features that added value to the system.

2.1 Compulsory goals

The scope of the project targeted at mixed criticality systems where a strict timing constraint existed. Also the nature of task was assumed to be periodic in nature for the scope of the project. Thus for such a system, their could exist a deviation from the periodicity of such a periodic signal. The permissible limit for such deviation should be as low as possible within defined limits. Hence a system which could detect the deviation if any, and adjust the task accordingly was the primary goal of the project. In order to detect a deviation it was necessary to introduce a notion of time into the system. The system comprised of a several distributed ECU units connected over a bus subsystem. The project dealt with distributed systems, thus a global notion of time at the bus level could help in task synchronization at distributed system level. A jitter[15] was introduced in the system when the tasks at the local ECU units missed the time stamp at which they were supposed to be executed. The choice of the bus protocol was very crucial in the context of the project. The bus architecture was supposed to comprise of two most important properties:

- It should provide a global sense of time to every ECU unit in the distributed system.
- It should provide time division multiple access to the ECU units in order to remove the contention resolution[16] between the ECU's competing for the bus. This feature shall introduce a time deterministic behaviour in the system.

For a distributed system CAN has been the most widely used bus protocol for a distributed ECU system. But it suffered from some drawbacks for achieving the intended goal:

- Bus access was event-driven[17] and took place randomly. Only one node out of several nodes, was able to transmit based on bit-wise priority arbitration[18].
- Each node was driven by its own local clock and their was no notion of global time stamp for the nodes in the distributed network. Thus a synchronous system was difficult to achieve in the event of clock drift between different ECU units in the network.

In contrary to CAN bus protocol, Flexray or a Time triggered CAN protocol had following advantages:

- Each node accessed the bus by time division multiplexing which made the distributed system behaviour time deterministic.
- The protocol provided provision for clock synchronisation of local clocks over the distributed network based on a global time base[19]. Thus it allowed for clock synchronisation over a distributed network.

Of the existing proposed time triggered protocols, Flexray was preferred over time triggered CAN as the MPC5567 supported a Flexray peripheral unit. The system implementation was AUTOSAR compliant, which had the provision to schedule the tasks in a schedule table. The notion of schedule table introduced, time stamps at which the tasks could be launched. These time stamps could always be verified against a global time stamp derived from the bus architecture. In an event of drift between local ECU clocks and calculated global time, the schedule table could be adjusted according to the derived global time base. Thus based on requirement and motivation the compulsory goal was to :

- Implement and integrate a flexray bus architecture for the distributed network.
- Implement a AUTOSAR complaint OS schedule table, that could synchronize its execution to the global time base.

2.2 Optional goals

Besides the compulsory goals there are also certain optional goals to increase the usability of the system. These are:

- Generator code for generating Flexray interface configuration file
- Generator code for generating Flexray driver configuration
- Generator code for generating configuration files for explicitly synchronized OS schedule table

These optional goals existed because a generator code made it possible to generate configuration files required to initialize the data structures for OS schedule table and the Flexray communication stack. The generator code thus helped in making the system more generic as it could generate different values for data structures type based on the system requirement.

3 Theory

This chapter contains theory of AUTOSAR hierarchical stack architecture required to realise the system. It also discusses in detail the Flexray communication architecture implementation and integration to realise the time deterministic distributed system.

3.1 AUTOSAR

The AUTOSAR software architecture was developed by global companies in order to establish a de-facto open industry standard for automotive and electrical domain. It facilitated software development and its maintenance, independent of the existing hardware.

3.1.1 Layered software architecture

The AUTOSAR software stack had been designed such that the hardware independent software layer could utilise the services of any hardware via the hardware specific software drivers. This made the development of software stack independent of hardware specific configuration. The three highest level of abstraction layers were Application Layer, RTE (Run-Time Environment), and Basic Software.

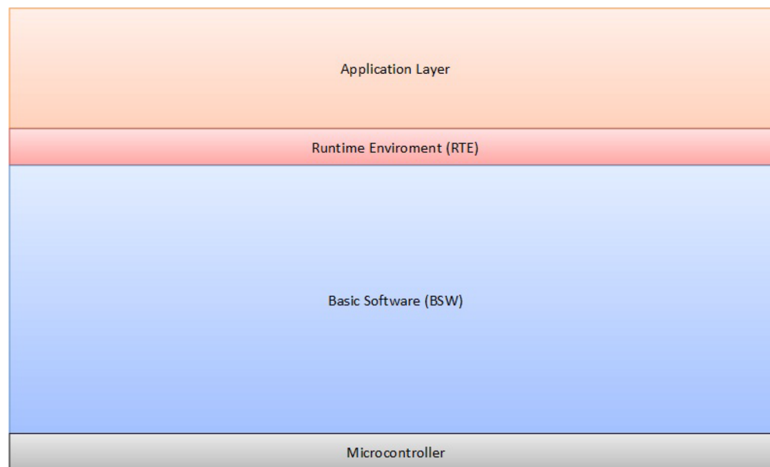

Figure 1: Software layers on highest abstraction layer:Application, Runtime and Basic software.

3.1.1.1 Application layer

In AUTOSAR, an application comprised of several connected SWC's. The SWC were constrained to be atomic in nature. This implied that only one instance of a software component existed in each ECU unit.

3.1.1.2 RTE

This layer provided functionalities of a real time operating system such as:

- Tasks
- OS layer Schedule Tables
- Interrupt service routines[20]
- Alarm

- Resources
- OS services

All communication that occured between SWC's and the layer below and/or between a SWC with another SWC at the Application Layer was routed through the RTE. This layer supported inter-ECU communication and intra- ECU communication either for communication between SWC's mapped on the different ECU or within same ECU(using e.g. FlexRay, CAN, LIN, etc.). An OS layer Schedule Tables was important to this project in the following context:

- It supported expiry points which are statically defined time stamps at which configured event or time driven tasks could be dispatched to the processor for execution.
- It supported explicit synchronization in which the schedule table was driven by an OS counter. This Operating System counter was synchronized to an external synchronization counter. The expiry points to be scheduled next could be delayed or released early in an event when the drive counter was early or later compared to the synchronization counter. In context to this project the external synchronization counter was the Flexray macrotick counter received from the Flexray time base. A Flexray macrotick has been discussed in detail in the timing hierarchy section of the chapter 3.1.2 Flexray communication structure.

The expiry points were released at statically configured offsets relative to the starting point of the OS Schedule Table. The task sets defined within an expiry point has relative priorities within themselves. Thus the highest priority task was always dispatched to the processor for execution and could not be preempted by any other task until its execution was complete. Thus worst case execution analysis[21] was a very important aspect because a lower priority task may never be executed in an event the that next scheduled expiry point got released before lower priority tasks could even get the processor for execution. This condition of starvation occured only when the execution time of a higher priority tasks within an expiry point was greater than or equal to the time interval between two consecutive expiry points. However this problem statement was removed from the scope of the project by allocating sufficient OS counter ticks for the execution of tasks defined in the expiry point. Noteworthy is the point that incrementation of OS counter tick was indirectly governed by Flexray macrotick which has been discussed in the later sections of implementation chapter. The generic structure of a schedule table is illustrated as below.

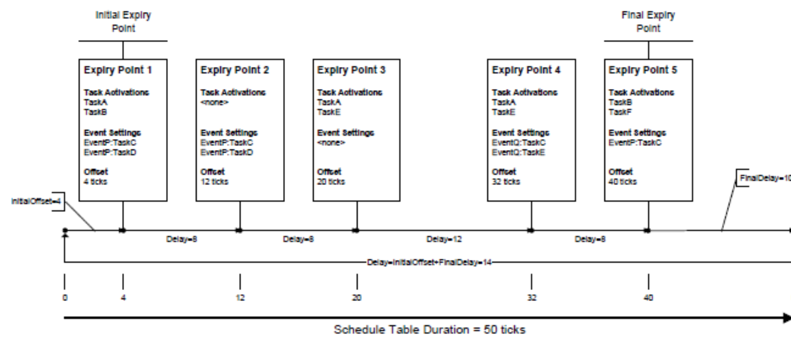

Figure 2: Structure of OS Schedule Table.

The OS Schedule Table comprised of following states:

- Stopped: The drive counter reached its maximum allowable value beyond which it rounded off to initial value.

- Waiting: The execution of a Schedule Table had been pre-empted by other higher priority Schedule Table and was thus in a suspended state waiting for being dispatched again in a queue.
- Running: The Schedule Table was running , but was not synchronized to the external synchronization counter.
- Running and synchronous: The Schedule Table was running, and was synchronized to the external synchronization counter.

Thus as discussed the above properties enabled the OS Schedule Table to synchronize the execution of tasks at OS level with respect to a global time base.

3.1.1.3 Basic software

The Basic Software was a standardized software layer and comprised of several layers. These layers were: Services, ECU Abstraction, and Microcontroller Abstraction.

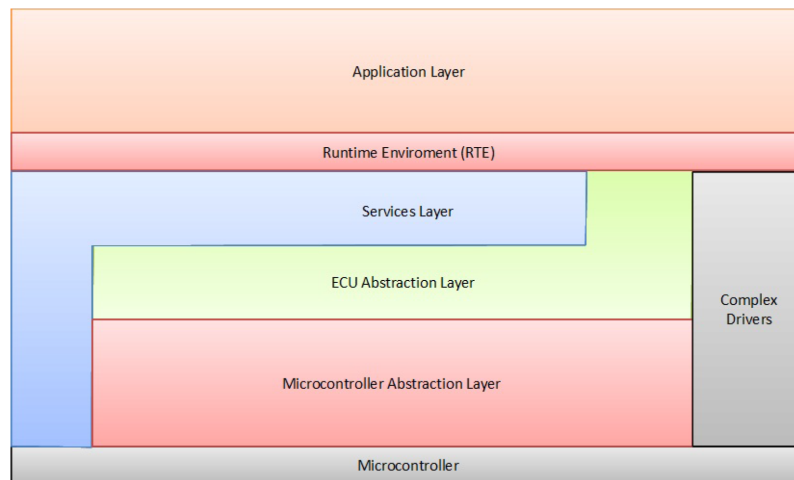

Figure 3: Layers of Basic Software: Services, ECU Abstraction, Microcontroller Abstraction and Complex Drivers.

The MCAL (MicroController Abstraction Layer) contained internal drivers and made the higher software layers independent of the microcontroller. The higher software layers were made independent of ECU hardware layout by the ECU Abstraction Layer. It provided API (Application Programming Interface) which acted as a wrapper function encapsulating the microcontroller architecture.

3.1.2 AUTOSAR communication stack

The Basic Software layer comprised of sub modules, forming various functional groups. One of these functional groups was the communication stack. The communication stack layer was responsible for establishing a connection between the microcontroller at the bottom to the RTE and also to the other SWC's at the top.

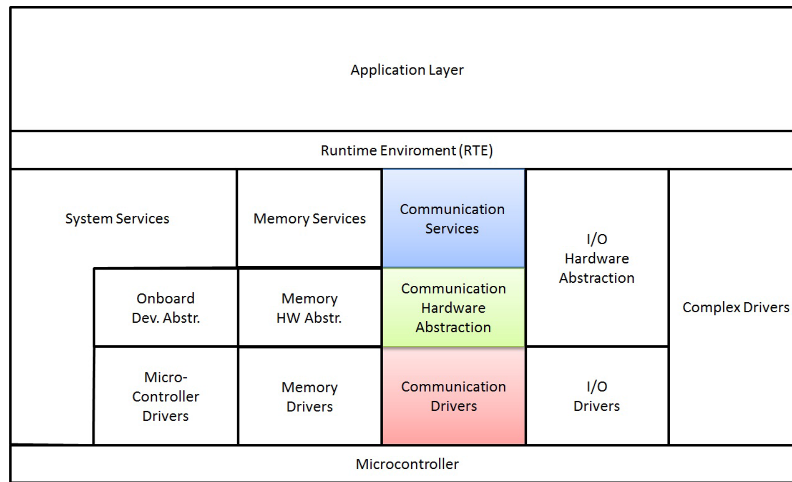

Figure 4: AUTOSAR communication stack; Communication Drivers, Communication Hardware Abstraction, and Communication Services.

The implementation of communication stack was protocol dependent, e.g. Flexray, CAN, LIN, etc. Our focus of implementation for this project was flexray communication stack whose architecture was as follows:

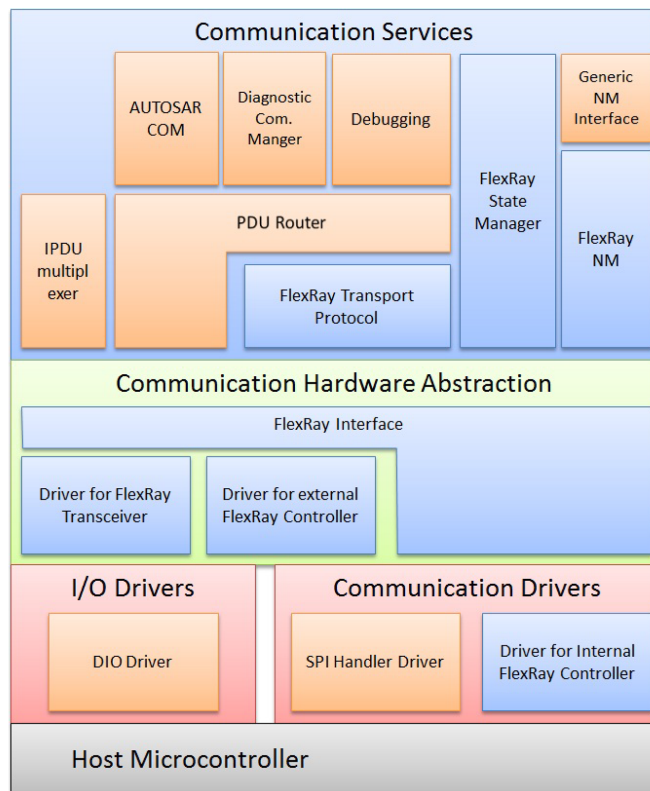

Figure 5: AUTOSAR communication stack for Flexray.

The communication services layer in context to Flexray communication stack comprised of Flexray State Manager, PDU router and the COM module. In order to simplify the project goal a part of Flexray State Manager had been implemented while the PDU router and COM module had been replaced by call back functions for multiplexing/demultiplexing the PDU data from the

Flexray bus. The communication hardware abstraction layer comprised of Flexray Interface and a Flexray Transceiver driver. The lower most layer comprised of a Flexray driver that initialized and provided an interface to the Flexray communication controller from the software point of view.

3.1.2.1 Flexray Interface

The Flexray Interface module provided interface as a part of communication system to the upper layer modules, such as a PDU Router or a COM. However, it was possible to implement call back functions which could be used for communicating with the tasks defined in upper OS layer. Thus it simplified the system implementation by bypassing the upper layer modules, such as a PDU Router or a COM. It is noteworthy to mention that the configuration of the Interface module depended on the communication bus. The Flexray Interface did not access the hardware directly, as the configuration relied on specific features of the communication system. Thus it used the Flexray Driver modules to access the Flexray CC(s). Similarly the Flexray Interface accessed the Flexray Transceiver(s), through specific Flexray Transceiver Driver module.

3.1.2.2 Flexray Driver

The Flexray Driver module initialized the Flexray CC and controlled its operation. The different Flexray CC's offered different hardware implementation features, thus a single Flexray Driver module supported only one specific type of a Flexray CC.

3.1.2.3 Flexray Transport Protocol

The Flexray Transport Protocol was not incorporated within the scope of this thesis as the message size had been constrained to be between 127 to 254 bytes. The Flexray Transport Protocol segmented and performed reassembly of PDU's only when the frame size was greater than 254 bytes.

3.1.2.4 PDU Router

Depending on layer the PDU have different definitions which are as follows:

Table 1: The definitions of the different PDU's

PDU Definition	Description
I-PDU	PDU of an upper layer module, e.g COM, DCM etc.
L-PDU	PDU of the FlexRay Interface module
N-PDU	PDU of the FlexRay Transport Layer.

The function of the PDU router was to statically route I-PDU's based on the I-PDU identifier thus eliminating the occurrence of dynamic routing during run-time. A PDU Router used the module COM to provide PDU data to Flexray interface. For the sake of simplicity of the project this module had been kept out of scope. The PDU data to and from the Flexray interface were multiplexed/demultiplexed using call back functions to provide the services for the OS layer tasks defined in the OS Schedule Table.

3.1.3 Flexray communication architecture

Flexray protocol was designed for vehicle network communication. The main motivations behind development of Flexray communication architecture were as follows:

- More calculations and communication required to achieve comfort, safety and fuel efficiency.

- Introduce coherency among the nodes of the network.
- A reliable fault tolerant protocol with higher bus bandwidth capability.

Flexray communication was capable of supporting both time and event triggered systems. A time slot was assigned to every message according to TDMA (Time Division Mutiple Access method)[22]. A communication cycle was divided into a static segment, a dynamic segment, a symbol window, and a NIT (Network Idle Time) as below.

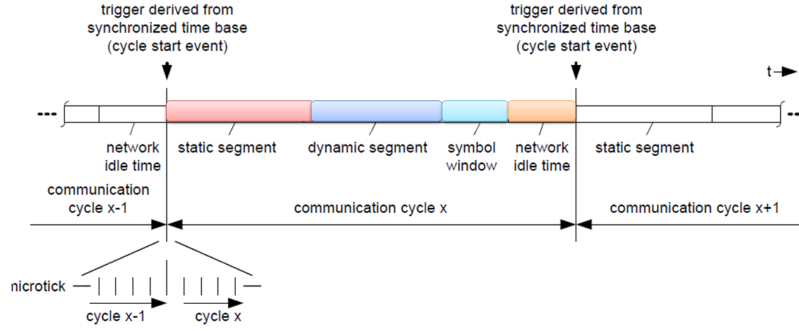

Figure 6: Division of flexray communication cycle.

3.1.3.1 Static segment

The static segment consisted of time slots of equal length, having same number of macroticks. A macrotick is a Flexray timing unit, discussed in detail in Timing Hierarchy section. The number of macroticks per time slot is defined by a global variable for the network. A Time slots could be either a key slot or a non key slot. A time slot of the type key slot was used by a node to transmit sync and startup frames, while a non-key slots were used for transmission of frames on either one channel or both. The number of static time slots were the same for all nodes in the network. As the timing characteristics of the static segment were precisely defined it was possible to have a time-triggered communication between the nodes. The frame transmission occured during a specific point of time, thus making it possible for all the nodes in the network to have a knowledge about when the transmission and reception of a frame was supposed to occur.

3.1.3.2 Dynamic segment

All event-triggered communication between Flexray nodes occurred in the dynamic segment. A communication slot in the dynamic segment is called a minislot. It is possible for a transmitted frame in the dynamic segment to be of different length compared to the equal length of transmitted frames in the static segment. A minislot in the dynamic segment contained an identical number of macrotick defined by a global variable for the network. There existed no sync or startup frames in the dynamic segment. The length of a dynamic slot was dependant on the transmitted frame size. If no transmission occurred, the dynamic slot consisted of one minislot. However during an ongoing transmission several minislots comprised of a dynamic slot. The communication behavior in the dynamic segment is asynchronous such that some frames could be prioritized over others. This meant that a node could use full bandwidth for the transmission of higher priority frames , while a frame with lower priority was prohibited for transmission in the current communication cycle.

3.1.3.3 Symbol window

The symbol window could be used for transmission of a start up or a wake up symbol. A wake up symbol is used as a power management tool to wake up a node, while in sleep mode. The symbol window was optional for a communication cycle.

3.1.3.4 Network idle time

The NIT is a phase used for calculating clock divergence and clock correction between the nodes in the network. Tasks such as error handling and updating counters could be performed if required during this period.

3.1.3.5 Clock synchronization

Flexray is a time-triggered communication protocol, thus it required all nodes in the network to have the same view of the time. Thus by having a global view of time all nodes can send and receive data on the bus at correct global time stamps. However different nodes in the network, have their own clock with different clock skew and offset, leading to their own interpretation of global time. Thus FlexRay needs a clock synchronization algorithm. The following text discusses the Flexray timing hierarchy and clock synchronization.

3.1.3.6 Timing Hierarchy

Flexray handles time in three different levels; the communication level, the macrotick level and the microtick level. Microtick is the smallest unit of time, and the length of a microtick is given as 'a' number of clock ticks on the CC's oscillator. The value of 'a' is different for each node as the oscillator rates is node dependant. The 'b' number of microticks constituted a macrotick. Within a cluster if all the nodes are correctly synchronized, then the number of macroticks within a communication cycle for each node is the same. A 'c' number of macroticks constituted a communication cycle, such that if the clock synchronization is correct for all the nodes in the cluster, then all the nodes in the cluster corresponded to same cycle number at any given time.

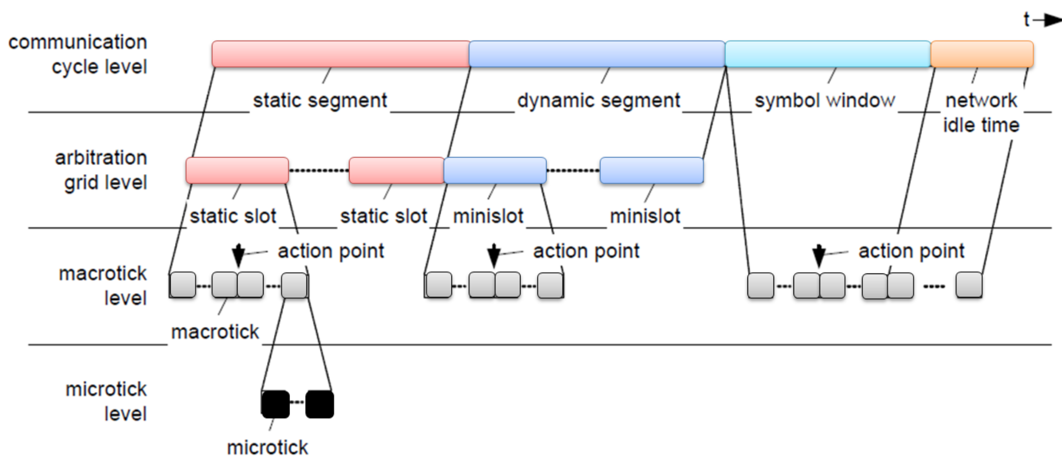

Figure 7: Flexray timing hierarchy.

3.1.3.7 Clock Calculation and correction

The global time is a time derived from events on the flexray bus. These events on the bus helps the nodes to virtually interpret the sense of global time. However, some constraints needed to be fulfilled in order to achieve it. The majority of all local clocks need to act correctly such that

they do not deviate too much from all other clocks in the network. The clock of one node should not drift more than 0.15% macroticks from the global time, thus implying that the difference between the slowest and the fastest clock can not be greater than 0.3% macroticks. The clock correction is done in two ways, rate and offset correction. The nodes which has been configured as sync nodes, send the sync frames over the network. These sync frames are used for time measurement. When a node receive a sync frame it compared the time of the arrival with the time of the expected arrival. An offset correction is performed by scheduling the next execution earlier or later compared to what otherwise would have been the case. This was achieved by adding or removing macroticks from the network idle time. In order that local clocks on the nodes "ticked" at the same rate, microticks were added or removed from a macrotick. Thus a node corrected its local clock without affecting the number of macroticks within a cycle, by making the macroticks either longer or shorter according to their local time.

4 Development Environment

This section discusses some design decisions taken at software and hardware point of view to assist the development involved in the project.

4.1 Software environment

The implementation of all the modules to be integrated into the system were developed according to AUTOSAR 4.0 specification. The AUTOSAR OS architecture described different scalability classes. The purpose of scalability classes was to help customize OS according to the requirements of the user to maximize the processor usage. The scalability classes were numbered from SC1-SC4. In order to support scalability class concept, from the software point of view several conditional checks were implemented. These conditional checks validated the features to be used, described within a scalability class. The version of operating system used in the project did not support all the scalability classes. The Schedule table functionality supported scalability class 3 and 4 which was not supported by the version of OS used in the project. The implementation of schedule table thus skipped the conditional checks for scalability classes and thus did not strictly follow the AUTOSAR architecture guidelines.
The implementation of generator code in eclipse xpand editor, to generate the data structures required for the initialization of Schedule table, Flexray Driver and the Flexray Interface was an important aspect. These data structures were used when the code was compiled to give the nodes the configured parameters. Thus besides the implementation of the required AUTOSAR modules, the generator code for generating the C configuration file was equally important.

4.2 Hardware environment

The hardware setup was established using the following components:

- ODEEP QR5567 (Hardware platform)
- MPC5567EVB (Hardware platform)
- Flexray Channels (Two twisted pair cables)
- UDE (Debugger and Flasher)
- WinIDEA iSYSTEM's (Debugger and Flasher)
- USB cables
- Ethernet cable
- PC

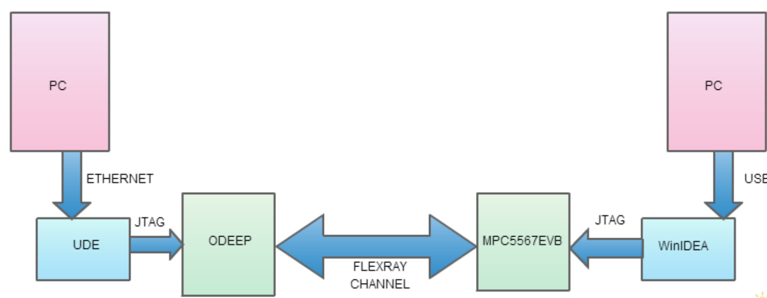

Figure 8: Hardware environment setup.

As discussed in the earlier chapters, the hardware setup comprised of ODEEP and MPC5567EVB development platform. The two boards were connected through two twisted pair cables acting as two Flexray channels. Although the bus could be extended to add more nodes, but the scope of the project had been restricted to only two nodes. The existing setup was sufficient enough for two or more nodes to realise a distributed system communicating to each other in time deterministic manner. In order to load new software on the two nodes two different Debugger/flash programmers were used namely UDE and WinIDEA iSYSTEM's. Both of them were connected to the nodes with a JTAG interface, while connected to the PC through ethernet and USB cable respectively. Each debugger was connected to two different PCs due to the constraint that it was hard to debug the software for two different nodes at the same time on a single PC. The PC accessed the UDE debugger/flash programmers through the ethernet cable via the LAN network, while the WinIDEA iSYSTEM's was accessed via the USB cable using a peer to peer network connection.

5 Implementation

This chapter describes the implementation of the different AUTOSAR modules. It covers relevant data structures, functions, and modules of the Arctic Studio involved in the project.

5.1 OS Schedule Table

The implementation of this module was based on AUTOSAR operating system specification 4.1. An OS schedule table comprised of several taskset. A taskset in an automotive real time system could either be time or event triggered[23]. A time triggered taskset is one which is dispatched to the processor at specific time stamps for execution, while an event triggered taskset is executed only when a condition related to an occurrence of an event is set to true. The taskset could have priorities within themselves such that a higher priority task may pre-empt an ongoing lower priority task. In a safety critical application such as in automotive domain task pre-emption was not a good solution. The worst condition being when a lower priority task always got starved and never got enough processor time for execution in a uniprocessor system. Thus for such a system where it was of utmost concern to assign a fair share of processor time for each configured task, a schedule table provided a good solution. The task preemption was removed by explicitly time multiplexing the task dispatching to the processor at specific time stamps called expiry points. It had following features:

- A drive counter to drive the schedule table.
- A synchronization counter to which the drive counter explicitly synchronized.

The OS architecture was based on OSEK [24] real time systems. According to it the tasks were assigned predefined priorities and made preemptive in nature. Further based on the task states they were of two kinds:

- Basic task

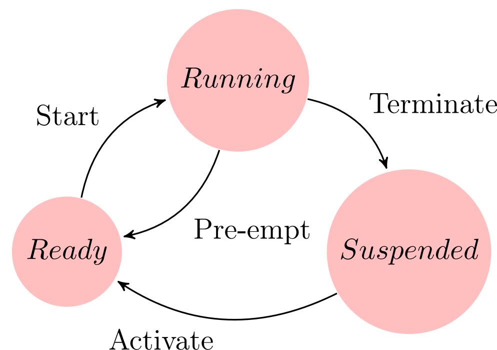

Figure 9: Basic task state diagram.

The time triggered tasks were implemented as basic tasks. The task after execution were terminated as they transitioned into suspended state. A task from suspended state if activated made a transition to ready state. A task was considered for execution by the processor only in ready state. The task was transitioned into running state from ready state by issuing the "start" API. It was during the running state, a task was dispatched for execution by the processor. A task in the running state could be terminated if a higher priority task or a higher priority ISR got triggered. The resources if any associated with the task were relinquished as soon as they were terminated.

- Extended task

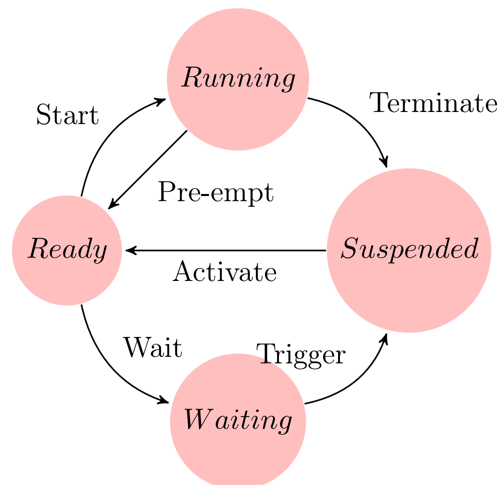

Figure 10: Extended task state diagram.

The event triggered tasks were implemented as extended tasks. The extended tasks had an additional state called waiting . They remained in the waiting state for an event bit to be set to true. In the event it was evaluated to true it transitioned to ready state. An extended task was then executed by the processor by using "start" API that caused a state transition from ready to running state.
A schedule table was a set of such basic and extended tasks. Several subsets of these tasks were made, that were configured to be triggered at specific time stamps called expiry points. An important constraint according to the AUTOSAR architecture was that within such a subset, it was the basic task that was given higher priority over extended tasks. Thus time triggered tasks were given higher priority over event triggered tasks. Such an implementation constraint had clear implications on reducing the response time[25] of a time triggered tasks, while trying to accomodate event triggered system in the system whose probability was less likely to occur. In order to ensure some degree of predictability in the system for a distributed automotive architecture, a schedule table was calculated for all the allocated task prior to the system implementation. Thus keeping in mind the above discussed AUTOSAR constraint of allocating higher priority to time triggered over event triggered tasks, a smart implementation at system level would be to allocate highly probable event triggered tasks as standalone tasks to the expiry points to reduce their response time.
The scope of the project was not in the context of choosing tasksets for the expiry points as it was already taken care by the system designer based on worst case execution of the individual task and factors such as cost of task preemption. An important issue addressed in this project was the clock skew within the ECU and an event of clock drift in a distributed system. These two factors contribute to timing related issues that degraded the performance of the distributed system, as the system suffered from factor such as clock aging. In reference to the context a possible solution would be a system that could detect a clock drift and adjust the task dispatching according to whether the drift was on negative or positive side. The AUTOSAR 4.1 specifications provided an insight on explicit synchronization of the schedule tables.
The drive counter for schedule table could be repeatedly synchronized to an external counter. Based on the calculation the schedule table drive counter could be adjusted to the synchronization counter. For such a system three factors decided the stability and effectability of the system:

- How stable was the external synchronization counter.

- How often the drive counter was synchronized to the external synchronization counter.
- With how much time unit the drive counter could be moved forward or backward in an event of drift between the two counters.

An important constraint was that the resolution of drive counter ticks should be either equal to or a multiple of synchronization counter for the predictability of the system. A schedule according to the AUTOSAR specifications had following state transistions:

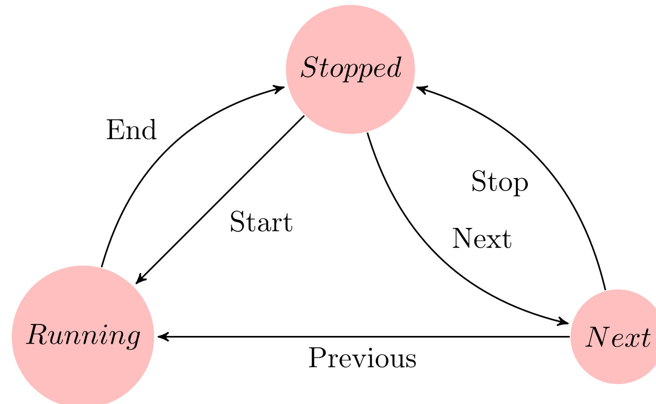

Figure 11: Schedule table state transistion.

Thus based on above discussions and AUTOSAR specifications, following API's had been implemented:

- StartScheduleTableSynchron(): This API checked the schedule table configurations and asserted the code to abort the system to make it stop in a hook routine , whenever an errorenous condition was encountered. If the assertion were not met the schedule table was updated to contain the first expiry point from the expiry point list.
- SyncScheduleTable(): This API read the time unit of the external synchronization counter and based on the drift calculated between the drive and synchronization counter it shifted the next expiry point either forward or backward based on whether the drive counter was faster or slower compared to the synchronization counter.
- GetScheduleTableStatus(): This API provided the current state of the schedule table.
- ScheduleTableFunction(): This API based on the current index of the expiry point counter sequentially dispatched the configured time and event triggered task for execution. After the task had been executed it updated the expiry point counter index to the next available expiry point.

A schedule table comprised of a data structure that needed to be initialized for compilation of C code into binary file. An example of data structure that needed to be initialized for the schedule table is given below:

```
typedef struct osSchTbl {
char* name;
TickType duration;
Bool repeating;
ApplicationType applownerld;
uint32 accessingApplMask;
struct Oscounter * counter;
```

```
const struct osSchTblAutostart * autostartPtr;
struct osScheduleTableSync * sync;
struct osSchTblAdjExpPoint * adjExpPoint;
uint32 id;
int expire_curr_index;
TickType expire_val;
ScheduleTableStatusType state;
struct osSchTbl *next;
SA_LIST_HEAD(alist,OsscheduleTableExpiryPoint) expirePointList
SLIST_ENTRY(osSchTbl) sched_list;
}
osSchTblType;
```

The initialization of data structures was done using C configuration files. These C files were generated by xpand editor generator code written in the Arctic studio eclipse tool. The next chapter gives a close insight in the configuration of development environment. The generator code takes values configured in the xml setting and does loop unrolling[26] to generate data structures based on the number of data structure required. Below is an example of generator code developed for schedule table in the eclipse development environment:

```
GEN_SCHTBL_EXPIRY_POINT_HEAD( « tableIt.Countero» ) {
«FOREACH table.OsscheduleTableExpiryPoints AS expiryPoint ITERATOR expirylt
«IF
expiryPoint.osScheduleTableTaskActivations.OsscheduleTableActivateTaskRef.size >0 -»
«IF expiryPoint.OsscheduleTableEventSettings.size > 0-»
GEN_SCHTBL_EXPIRY_POINT_w_TASK_EVENT(« tableIt.Countero
expiryPoint.osScheduleTblExpPointoffset.value
«ELSE-»
GEN_SCHTBL_EXPIRY_POINT_W_TASK(« tableIt.Countero
expiryPoint.osScheduleTblExpPointoffset.value-»)
«ENDIF-»
«ELSE-»
GEN_SCHTBL_EXPIRY_POINT_W_EVENT(« tableIt.Countero
expiryPoint.osScheduleTblExpPointoffset.value-»)
«ENDIF-»
«ENDFOREACH-»
«IF table.osScheduleTableAutostart.shortName.length > 0-»
GEN_SCHTBL_AUTOSTART(
tableIt.countero
«IF table.osScheduleTableAutostart.OsscheduleTableAutostartType.isSYNCHRON()-»
«table.osScheduleTableAutostart.OsscheduleTableStartValue.value.orError("O
sScheduleTableStartValue is unset")»### THESIS_2014
«ELSE-»
SCHTBL_AUTOSTART_«table.osScheduleTableAutostart.OsscheduleTableAutostartType.va
lue.orError()-»
«table.osScheduleTableAutostart.OsscheduleTableStartValue.value.orError("O
sScheduleTableStartValue is unset")»,
«ENDIF-»
OSDEFAULTAPPMODE
)
«ENDIF
«ENDFOREACH /* END TABLE DATA */-»
```

The drive counters for schedule table could be of three types:

- Hardware: An external crystal oscillator circuit drove the schedule table. the resolution achieved is as high as few nano or pico seconds.
- Software: An autonomous software counter having a resolution of a few micro seconds.
- OS Tick: The schedule table was driven by periodic OS ticks with a resolution obtained in few micro seconds.

The project used software based drive counter to drive the schedule table. Noteworthy is the point that this counter was not autonomous and was incremented only when a certain specific condition was met. The external synchronization counter was derived from Flexray time base having resolution as high as few nano seconds. Thus clearly emphasizing that an autonomous software counter incrementing at a rate of few microseconds would never be able to catch up a counter running at nano second scale under the notion that every tick for both the counters corresponded to the same time step. The schedule table was implemented as non repetative thus making it necessary to invoke the API to start the schedule table whenever required.

5.2 Flexray communication stack

The OS schedule Table synchronized to the macroticks of the Flexray module. The key idea of the schedule Table implementation was to write the data generated by the tasks defined in the expiry points at the configured physical buffer in the Flexray memory partition. As discussed in the previous chapter, the choice of Flexray was due to its time deterministic nature and clock synchronization algorithm. This algorithm ensured all the ECU's in the network to achieve clock synchronocity even in the event of clock drift by adding additional microtick as discussed in previous chapters. The MPC5567 processor used in the project comprised of a Flexray peripheral unit. In order to implement the Flexray communication setup it was necessary to configure the hardware unit. A Flexray version 2.1[27] was used for the project. The Flexray protocol comprised of a protocol engine. It was this protocol engine that maintained the operation and behaviour of a Flexray controller. The operation of a protocol engine was based on large number protocol variables. These variables were configured according to the specifications of Flexray version 2.1. In this context a Flexray protocol calculator had been written in C language that calculates the protocol variables and configures the hardware protocol registers. An example of such a structure contained in Flexray driver[28]comprising of protocol variables is given below:

```
typedef struct {
uint32 Fr_CtrlIdx;
Fr_AbsoluteTimersConfigType *Fr_CCAbsoluteTimersConfigPtr;
Fr_Fifo Fr_FifoPtr;
boolean Fr_IsLeadingColdstarter;
uint32 Fr_PKeySlotId;
boolean Fr_PKeySlotonlyEnabled;
boolean Fr_PKeySlotUsedForstartup;
boolean Fr_PKeySlotUsedForSync;
uint32 Fr_PSecondKeySlotId;
boolean Fr_PTwoKeySlotMode;
Fr_ChannelType Fr_PwakeupChannel;
uint32 Fr_PwakeupPattern;
Fr_ChannelType Fr_PChannels;
Fr_AbsoluteTimerType *Fr_AbsoluteTimerPtr;
boolean Fr_PAllowHaltDueToClock;
```

```
uint32 Fr_PAllowPassiveToActive;
uint32 Fr_PClusterDriftDamping;
uint32 Fr_PDecodingCorrection;
uint32 Fr_PDelayCompensationA;
uint32 Fr_PDelayCompensationB;
boolean Fr_PExternalSync;
boolean Fr_PFallBackInternal;
uint32 Fr_PLatestTx;
uint32 Fr_PMacroInitialoffsetA;
.
.
.
}Fr_ControllerType;
```

An important factor to be considered while writing the protocol calculator was the condition of overflow that occurred due to shift operation on the operand. This led to a value that was not large enough to hold the variable value leading to an overflow and hence a computation of wrong PCR[7] values. Thus such a condition was removed by using a 64 bit integer value safe enough to hold any computed value.

```
void protocol_operationcalculater(void){
uint64_t temp;
int i;
for( i=0;i<31;i++){
switch ( i ) {
case 0:
protocol_buff[0]=((FrIf_ClusterConfig.FrIf_GdActionPointOffset-1)<<10)|FrIf_ClusterConfig.FrI
  // Code
  break;
case 1:
protocol_buff[1]=(FrIf_ClusterConfig.FrIf_GMacroPerCycle -FrIf_ClusterConfig.FrIf_GdStaticSlo
  // Code
  break;
case 2:
protocol_buff[2]=(FrIf_ClusterConfig.FrIf_GdMinislot -FrIf_ClusterConfig.FrIf_GdMiniSlotActio
  // Code
 break;
case 3:
temp=FrIf_ClusterConfig.FrIf_GdMiniSlotActionPointOffset-1;
protocol_buff[3]=(FrIf_ClusterConfig.FrIf_GdWakeupRxLow)<<10|((temp<<5)&0x03E0)|(FrIf_Cluster
  // Code
 break;
case 4:
protocol_buff[4]=(FrIf_ClusterConfig.FrIf_GdCasRxLowMax - 1)<<9|(FrIf_ClusterConfig.FrIf_GdWa
  // Code
break;
case 5:
protocol_buff[5]=(FrIf_ClusterConfig.FrIf_GdTssTransmitter)<<12|(FrIf_ClusterConfig.FrIf_GdWa
  // Code
  break;
 case 6:
 protocol_buff[6]=((((FrIf_ClusterConfig.FrIf_GdSymbolWindow-FrIf_ClusterConfig.FrIf_GdAction
```

The AUTOSAR specified that it was more preferable to implement a wrapper function that encapsulated the driver function for the concerned hardware. This kind of implementation strategy generalised the upper layer to call the lower layer function without taking into account the hardware specific constraints imposed by the hardware architecture. This kind of design decision with the above strategy had an advantage at the user application level, but it also implied a constraint on implementation with additional lines of code in terms of development strategy. Thus in reference to the project it was necessary to implement data structures and their corresponding C configuration files to initialize them. It was these data structures, which were utilized by the wrapper functions and the driver function for configuring the Flexray controllers and making them to communicate with each other at the network level. An example of data structure for Flexray interface[29] is as follows:

```
typedef struct
FrIf_ControllerType const* FrIf_ControllerPtr;
FrIf_JobListType const* FrIf_JobListPtr;
float32 FrIf_GdCycle;
float32 FrIf_GdMacrotick;
float32 FrIf_MainFunctionPeriod;
uint32 FrIf_MaxIsrDelay;
uint32 FrIf_SafetyMargin;
uint16 FrIf_GMacroPerCycle;
uint16 FrIf_GNumberofMinislots;
uint16 FrIf_GNumberofstaticslots;
uint16 FrIf_GdNit;
uint8 FrIf_ClstIdx;
uint8 FrIf_GColdStartAttempts;
.
.
.
}FrIf_ClusterType;
```

An important point to be mentioned in the context of Flexray interface was that, it comprised of data structures which were configured as global parameters, implying that they were the same for every node participating in the Flexary cluster. An example of xpand generator code developed for Flexray interface according to the AUTOSAR specification 4.0 is as follows:

```
«LET FrIfConfig.FrIfClusters.first() AS cluster »
const FrIf_JobListType FrIf_JobList_«cluster.shortName»_Ptrconfig =
{
.FrIf_JobPtr = &FrIf_JobPtr_«cluster.shortName»_config[0],
    .FrIf_NbrOfJobs = «cluster.FrIfJobList.FrIfJobs.size»,
    .FrIf_AbsTimerRef = &FrIf_AbsTimerConfig[0]
};
/** Cluster configuration Only one cluster supported **/
const FrIf_ClusterType FrIf_ClusterConfig =
{
.FrIf_ControllerPtr  = &FrIf_«cluster.shortName»_ControllerConfig[0],
.FrIf_JobListPtr = &FrIf_JobList_«cluster.shortName»_Ptrconfig,
.FrIf_GdCycle   = «cluster.FrIfGdCycle.value.or( 0.000024)»,
.FrIf_GdMacrotick = «cluster.FrIfGdMacrotick.value.or(0.000006)»,
.FrIf_MainFunctionPeriod = «cluster.FrIfMainFunctionPeriod.value.or(0)»,
.FrIf_MaxIsrDelay = «cluster.FrIfMaxIsrDelay.value.or(10240000)»,
.FrIf_SafetyMargin = «cluster.FrIfSafetyMargin.value.or(10240000)»,
```

```
.FrIf_GMacroPerCycle = «cluster.FrIfGMacroPerCycle.value.or(8)»,
.FrIf_GNumberOfMinislots = «cluster.FrIfGNumberOfMinislots.value.or(0)»,
.FrIf_GNumberOfStaticSlots = «cluster.FrIfGNumberOfStaticSlots.value.or(2)»,
.FrIf_GdNit = «cluster.FrIfGdNit.value.or(2)»,
.FrIf_ClstIdx = «cluster.FrIfClstIdx.value»,
.FrIf_GColdStartAttempts = «cluster.FrIfGColdStartAttempts.value.or(2)»,
.FrIf_GCycleCountMax = «cluster.FrIfGCycleCountMax.value.or(63)»,
.FrIf_GListenNoise = «cluster.FrIfGListenNoise.value.or(2)»,
.FrIf_GMaxWithoutClockCorrectFatal = «cluster.FrIfGMaxWithoutClockCorrectFatal.value.or(15)»,
.FrIf_GMaxWithoutClockCorrectPassive = «cluster.FrIfGMaxWithoutClockCorrectPassive.value.or(1
.FrIf_GNetworkManagementVectorLength = «cluster.FrIfGNetworkManagementVectorLength.value.or(0
.FrIf_GPayloadLengthStatic = «cluster.FrIfGPayloadLengthStatic.value.or(127)»,
.FrIf_GSyncFrameIDCountMax = «cluster.FrIfGSyncFrameIDCountMax.value.or(2)»,
.FrIf_GdDynamicSlotIdlePhase = «cluster.FrIfGdDynamicSlotIdlePhase.value.or(0)»,
.FrIf_GdIgnoreAfterTx = «cluster.FrIfGdIgnoreAfterTx.value.or(0)»,
.FrIf_GdMiniSlotActionPointOffset = «cluster.FrIfGdMiniSlotActionPointOffset.value.or(1)»,
.FrIf_GdMinislot = «cluster.FrIfGdMinislot.value.or(2)»,
.FrIf_GdStaticSlots = «cluster.FrIfGdStaticSlot.value.or(4)»,
.FrIf_GdSymbolWindow = «cluster.FrIfGdSymbolWindow.value.or(0)»,
.FrIf_GdSymbolWindowActionPointOffset = «cluster.FrIfGdActionPointOffset.value.or(1)»,
.
.
.
```

For the Flexray protocol several physical buffers were configured that were associated to specific static slots. These physical buffers were configured in the Flexray memory partition, which was a set of contiguous memory location. The size of each physical buffer was on the choice of the user application depending on the frame size that needed to be send over the network. Each buffer size was allocated a 32 byte size in this implementation. It was of utmost importance to choose the base address such that the allocated buffer space did not fall into the ROM memory address, which if occurred caused the processor to throw an exception and abort execution. The Flexray protocol also supported FIFO implementation for dealing complex scenarios, which was evaded in the project for the sake of simplicity of the project.

5.3 Interaction with other modules

The ODEEP and the MPC5567EVB development board comprised of two TJA1080 Flexray transceiver. The data frames in the configured buffers were multiplexed to and from the physical bus through the Tx and Rx pins of the transceiver respectively. The basic purpose of the transceiver was to provide voltage transition to Flexray physical bus levels. The bus was a differential bus with voltage ranging by 0.5 volts above and below a mean voltage of 2.5V. Also noteworthy was the point that the transceiver could operate in three different modes:

- Transmission mode
- Receive mode
- Normal mode(Tx and Rx mode)

In order to achieve a state transition of the transceiver into normal mode it was required to set several of its pins either at high or low voltage according to the specification.

5.3.1 Port module

The MPC5567 processor comprised of several collection of configurable physical electrical connection called pads which formed a port. Based on the physical layout of the ODEEP and MPC5567EVB board these ports were configured for the required voltage . It was during the bootup[30] state of the operating system, these ports were clamped to the configured voltage which set the transceiver into the normal mode. The following pins of the transceiver were set by configuring the corresponding ports:

- Flexray Bus Guard Enable: This pin was configured low.
- Flexray Tx: This pin was configured high.
- Flexray Rx: This pin was configured low.
- Flexray Tx Enable: This pin was configured low.

5.3.2 Module configuration unit

The ODEEP and MPC5567EVB board comprised of a crystal oscillator with 16 and 40 MHz frequency respectively that clocked the board. The Flexray protocol engine however needed to be clocked at 120 MHz. Thus it clearly implies that the clock needed to be translated to a higher frequency. The frequency translation was governed according to the following equations:
$F_{sys}=F_{ref}*\frac{(MFD+4)}{(PREDIV+1)*2^{RFD}}$ [7], where F_{ref} was the oscillator frequency, 16 and 40 MHz for ODEEP and MPC5567EVB board respectively.

Table 2: Equation variables for different boards

Equation variables	ODEEP	MPC5567EVB
MFD	11	2
PREDIV	1	1
RFD	0	0.

Thus using the following parameters the protocol engine was made to clock at 120 MHz according to the Flexray 2.1 specification.

5.3.3 Flexray state manager

According to the Flexray protocol a Flexray controller had following state transitions:

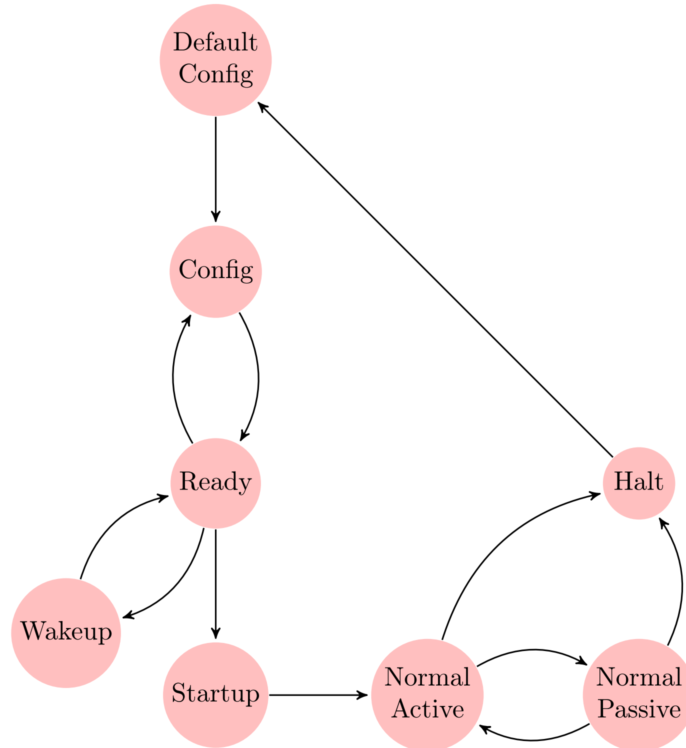

Figure 12: Flexray controller state transition.

As the board was provided the supply voltage the Flexray controller went into default config state. If for the controller its MCR[7], PCR's and the individual message buffers in the Flexray memory partition were initialized without any discrepancy, it could be made to transistion into ready state by writing the POCR[7]. The ECU's for this project were configured to never go into sleep mode. A sleep mode was referred to an ECU state in where it went into power saving mode. Thus for any further operation the ECU needed to be woken up by sending wakeup pattern bit. Since the ECU never went into wakeup mode thus it was unnecessary to send a wakeup pattern so as to switch it over into ready state from sleep mode. The startup of the Flexray cluster was a complex process. The necessary condition to start a Flexray cluster with n nodes was to have atleast two working nodes called coldstart nodes. It was during this startup phase atleast two coldstart nodes exchanged sync and startup frames so as to synchronize their respective clocks and have a global sense of time for the entire cluster. These sync and startup frames were exchanged during predefined static slots configured in the PCR. A provision of fault tolerance from babbling idiot failure[31] of a Flexray controller was provided by setting an inhibitor bit high. It was this inhibitor bit that prevented the Flexray controller from sending unwanted frames on the bus and disturbing an ongoing communication. Thus a leading coldstart node first set this inhibitor bit low by writing "startup" command to the POCR and then started sending the sync and startup frames . After receiving the frames for few cycles the nonleading coldstart nodes sent its startupframes leading to a transition from startup state to normal active state for both the communicating controllers. After achieving the normal active state the controllers could send and receive the data frames between each other. The Flexray state manager was[32] the module that called all the API's in sequential manner to achieve the required state transition of the Flexray controller to normal active state. The implementation of Flexray state manager was tricky in the sense that not every coldstart trial caused a successful "startup". Thus it was necessary to read the protocol state of the controller after every trial and if not successful to reset the controller to start the process again till a successful "startup" of the cluster was achieved. An example of Flexray state manager developed is as follows:

```
void Fr_SmStateInit(void){
int cnt=0;
Fr_CCRegisterPtr = (volatile uint16*) Fr_DriverSpecific[0].Fr_BaseAddress;
ISR_INSTALL_ISR2("Flexray",FR_ISR,FLEXRAY_PRIF,2,0);
/*
 * Initializing the Flexray Interface and Flexray Driver
 */
 Fr_Init(&Fr_ConfigPtr);
 FrIf_Init(&FrIf_Config);
 while(cnt<1000000){
 cnt++;
 }
 cnt=0;
 if(!FrIf_ControllerInit(Idx.FrCtrlIdx)){
 //Fr_CCRegisterPtr[FR_GIFER]=0x00C0;
 cnt=0;
 while(cnt<1000000){
 cnt++;
 }
 cnt=0;
Fr_SetWakeupChannel(Idx.FrCtrlIdx,0);
while(cnt<1000){
cnt++;
}
//Fr_SendWUP(0);

if(Fr_PbControllerPtr.Fr_IsLeadingColdstarter==TRUE){
cnt=0;
while(cnt<10000){
cnt++;
}
cnt=0;
FrIf_AllowColdstart(Idx.FrCtrlIdx);
.
.
.
}
```

5.3.4 Interaction of OS schedule table with Flexray communication stack

An ISR was installed in the system to facilitate the implementation of this project. It was installed to asynchronously trigger a specific function whenever certain flags of PIFR0[7] were set. As discussed in the previous sections that the drive counter of the schedule table rather being autonomously driven by the OS tick was incremented only when a condition evaluated to be true. It was within this function attached to the ISR, that the schedule table was started at the beginning of each Flexray communication cycle. The tasks associated with the first expiry point generated and wrote the data into the buffer from which it was scheduled to be transmitted over the bus according to the preconfigured Flexray macrotick value. It was during this ongoing transmission that the timer configuration was set for the next scheduled expiry point in the timer registers of the Flexray controller. Thus after timer being set for the next expiry point it was necessary to trigger the expiry point by incrementing the schedule table drive counter, so that the data was written to the buffer locations before it was scheduled to be transmitted over the

bus. After the transmission of current generated data and expiration of the timer , the process was repeated till all the expiry points were triggered and the schedule table came to a halt.
An important consideration while performing synchronization to the Flexray macrotick counter was to ensure the length of the schedule table to be either equal to or a multiple of Flexray macrotick cycle counter. This constraint was indirectly achieved by incrementing the drive counter regulating the dispatching of the expiry points just before the the scheduled Flexray macrotick. The next adjacent expiry point was similarly triggered just prior to the next scheduled macrotick. Thus such an implementation enabled the OS Schedule Table length equal to the Flexray macrotick cycle length in an event when the the number of scheduled transmission over the bus were equal to the number of expiry points in a Flexray communication cycle. Also noteworthy was the point that the adjacent expiry point triggering could be synchronized to the Flexray macrotick such that in an event that the schedule table was behind the macrotick count, the release of the expiry point could be scheduled earlier by some finite macrotick. Thus reducing the release jitter induced in the system by the task, which occured if released at a time stamp beyond the offset time at which it was originally expected to be earlier released.

5.3.5 Call back functions

A Flexray frame comprised of several PDU's. These PDU's comprised of payload data. The payload could be either meant for an application in the application layer or an OS layer task of the BSW layer. These PDU's were mapped to upper layer tasks using their PDU Id. The mapping of PDU's to the corresponding application layer could be performed using additional AUTOSAR layer called PDU router[33]. However adding an additional layer caused system implementation to become more complex. Also noteworthy is the point that in an event of implementing a time deterministic system such as intended in this project, it should be of utmost concern to reduce the code structure to as minimal as possible within the critical section. The critical section for this implementation corresponded to the ISR function implemented. The processing of PDU's through PDU router caused an additional latency in processing the PDU's fetched to and from the bus leading to an increase in their processing time. Due to the lack of any worst case execution time measurement tool during the course of the project, it was of utmost concern to remove additional layers that adds to the complexity of the implemented code and hence an increase in their processing time. A solution to remove PDU router was to implement a call back function. These call back functions retrieved the PDU's from the buffer location indicated by transmit and receive API. The retrieved PDU data could then be accessed by the application layer or the OS tasks directly without the need to call the PDU router layer.

6 Configuring the development environment

This section deals with how the Flexray configuration, and the contributory AUTOSAR modules were setup in the ARCTIC Studio tool.

6.1 Arctic studio BSW builder

The ARCTIC studio tool contained a plugin called BSW builder. An arctic core repository was provided that contained the C source code containing the API's for the modules provided by the BSW builder. The BSW builder provided containers through which parameters of the xml files could be configured. These container values were then used by the xpand editor code in the eclipse development environment to generate C configuration files. It was always possible to generate the files without using BSW builder but at the cost of very long development time. An example of BSW plugin is provided as below:

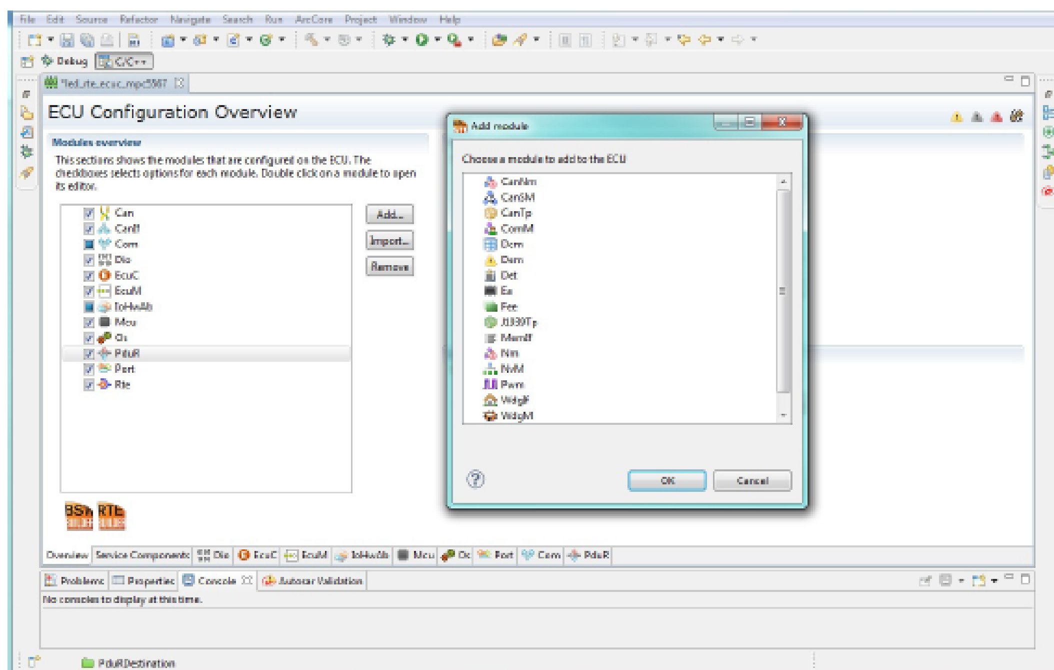

Figure 13: The Arctic studio environment with BSW builder plugin.

All the necessary plugins for the required modules were selected from the BSW plugin and configured as described below:

6.1.1 OS

This plugin was used to configure the OS level features. In the scope of this project it was necessary to generate data structures for following submodules:

- OS schedule table
- Drive counter for OS schedule table
- Event for event triggered tasks
- OS level taskset comprising of both basic and extended tasks

The OS schedule table comprised of five expiry points and seven OS level tasks. All the event triggered tasks were configured as extended tasks while time triggered tasks were configured as basic tasks. One of the task was configured as "AUTOSTART" task. An "AUTOSTART" task is one which is triggered automatically by the OS . This task was used for calling all the API's that initialized the schedule table and the Flexray communication stack. An example of configured OS plugin is provided as below:

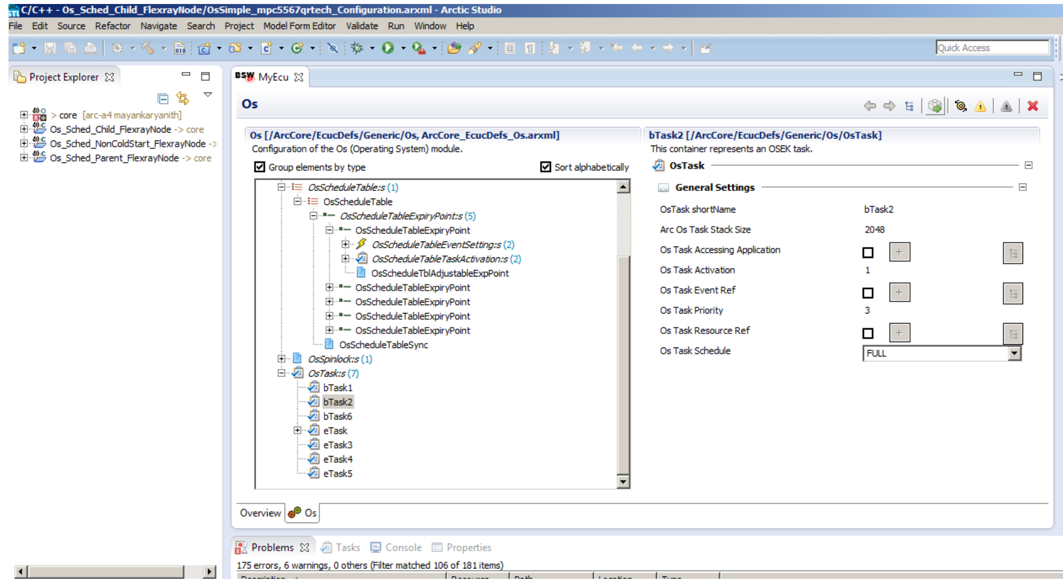

Figure 14: The Arctic studio environment for OS BSW plugin.

6.1.2 FrIf

This plugin was used to configure the Flexray interface. The corresponding plugin comprised of containers to configure the variables at Flexray cluster level and parameters to to configure Flexray frame structure. An important property about Flexray frame structure important to be mentioned is the communication operation type parameter. The communication operation type parameter were of the following types:

- Decoupled transmission: This operation implied that data needed to be written to the physical buffer for transmission.
- Prepare LPDU: This implied that an LPDU needed to be prepared.
- Receive and indicate: This implied that the upper layer was needed to be informed of the received data.
- Receive and store: This implied that the received data was needed to be stored at the configured memory location.
- Tx confirmation : This implied that it was necessary to notify that an event of successful transmission was always needed to be notified.

For the scope of the project, eleven data frames were configured to be transmitted over physical bus. An example of FrIf BSW plugin is as follows:

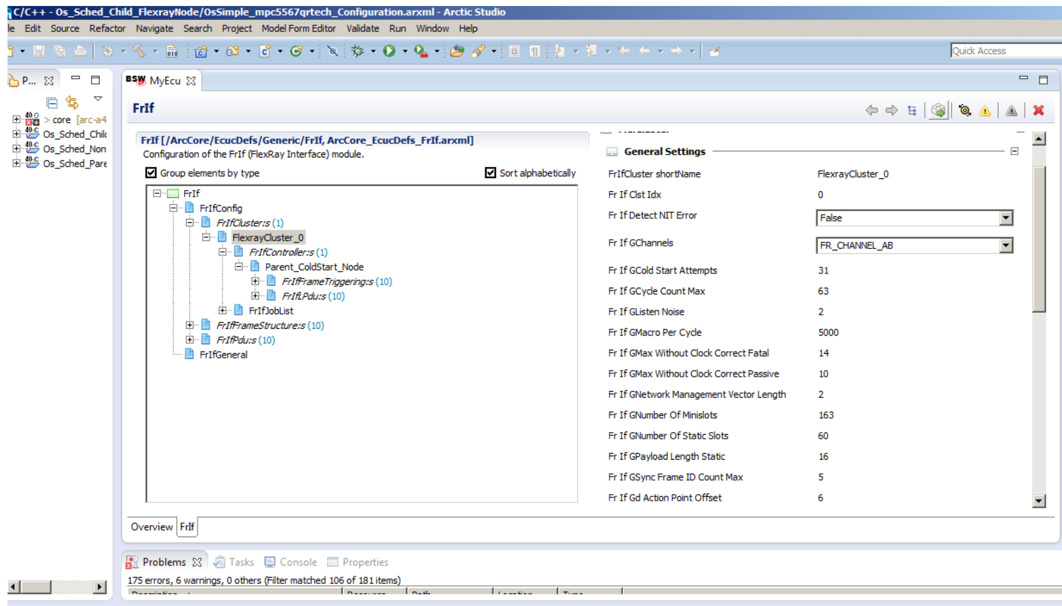

Figure 15: The Arctic studio environment for FrIf BSW plugin.

6.1.3 Fr

This plugin comprised of containers that were used to configure the hardware specific values to set up a working Flexray communication stack. A Flexray plugin is as follows:

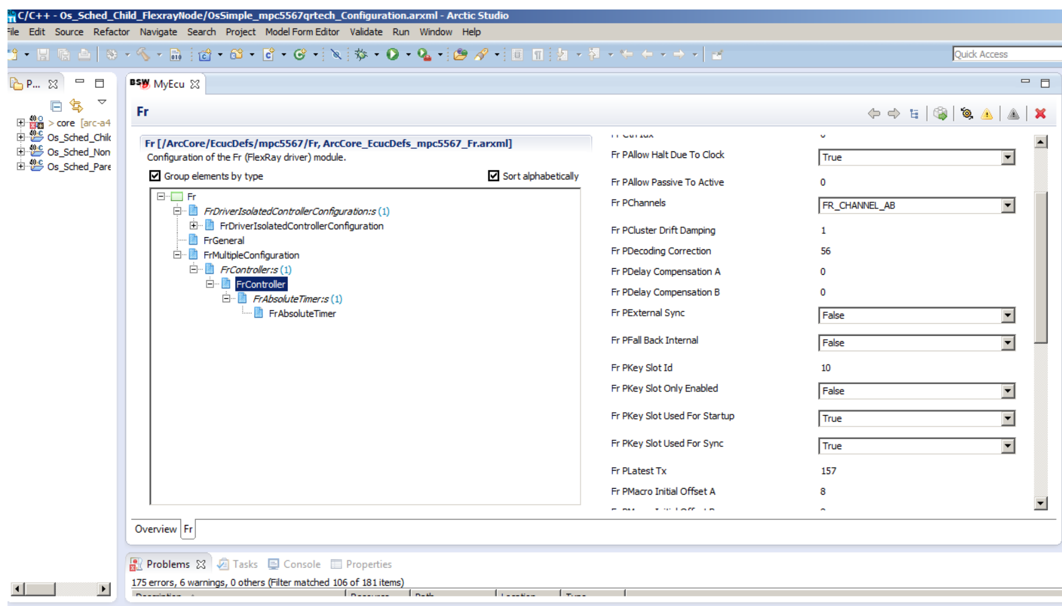

Figure 16: The Arctic studio environment with Fr BSW plugin.

6.1.4 MCU

This plugin comprised of containers that were used to configure the clock related parameter for the ECU. It was in this plugin the parameters MFD, PREDIV, RFD were configured according to the requirement.

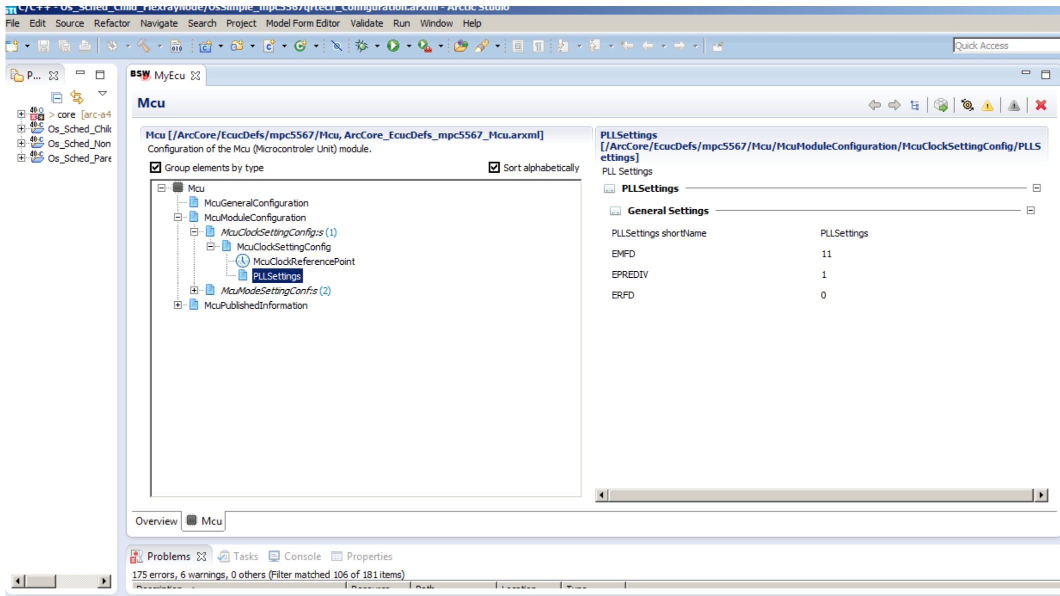

Figure 17: The Arctic studio environment with MCU BSW plugin.

6.1.5 Port

This plugin was used in the project to initialize the Flexray transceiver with correct pin voltages according to the specification. Also noteworthy was the point that Flexray Tx pin was set to Flexray mode. Thus by setting the Tx pin to Flexray mode, the Flexray controller could enable the Tx pin when required, while disabling the pin when not required during ideal transmission time. An example of Port plugin is as follows:

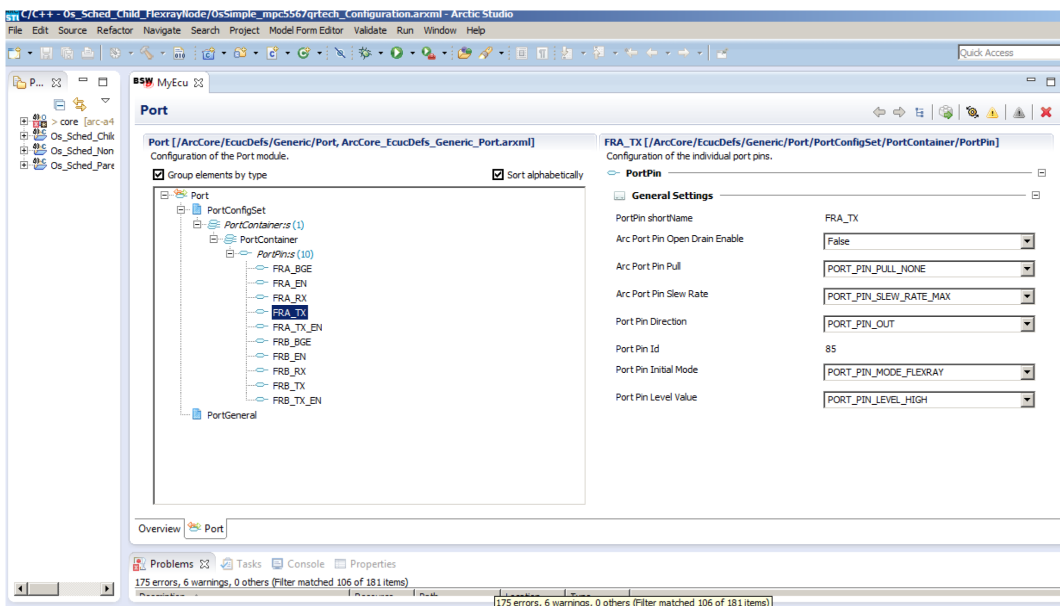

Figure 18: The Arctic studio environment with Port BSW plugin.

6.1.6 ECUM

This plugin was used to configure the ECU in always "active mode" by setting the boolean for sleep mode to be false. An "active mode" is one in which the ECU never goes into power saving mode and keeps the processor busy in some dummy loop which does not have any instruction to be executed. While in the sleep mode the processor though being clocked disables the power consuming computing unit by disabling the input for a gated clocked architecture.

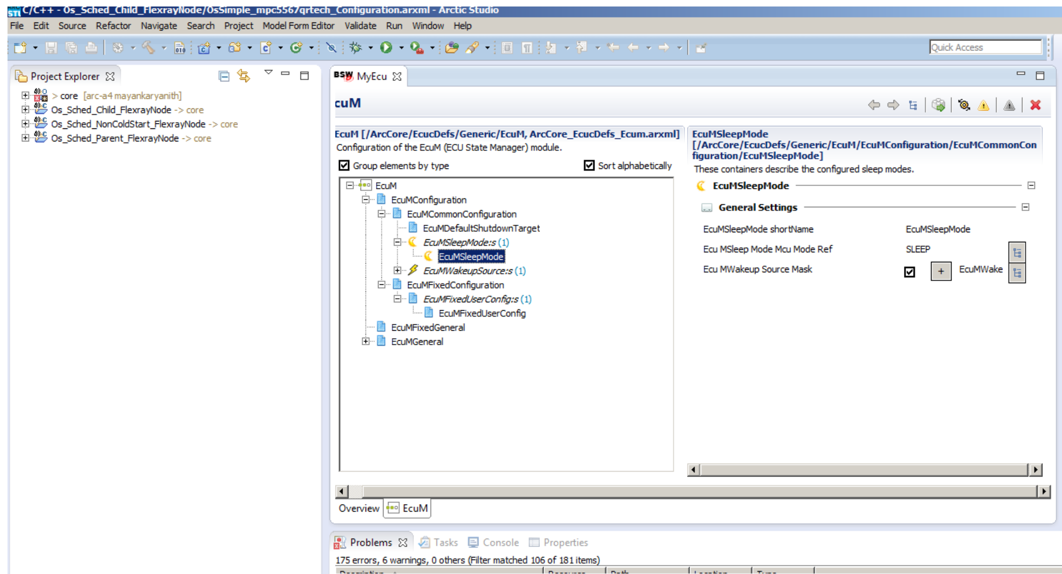

Figure 19: The Arctic studio environment with BSW builder plugin.

6.1.7 ECUC

This plugin was used to define the PDU's in the system. Considering the scope of the project ten PDU's had been configured with five PDU's each for Tx and Rx respectively.

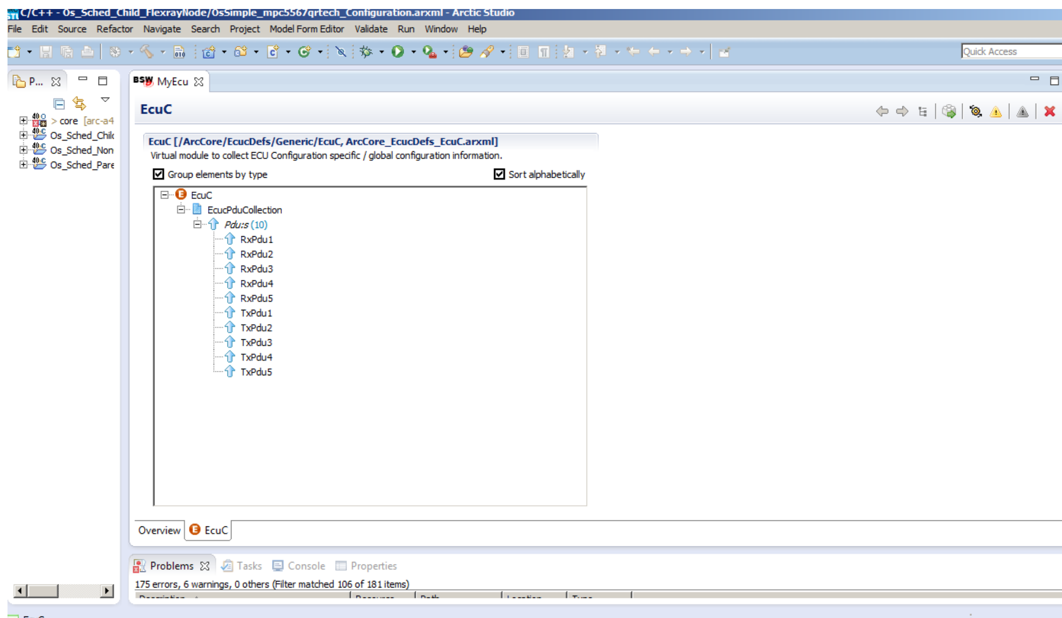

Figure 20: The Arctic studio environment with ECUC BSW plugin.

7 Test and verification

This section deals with test and verification strategy employed in the context of the project. The implementation was broken down into two different modules. :

- OS schedule table
- Flexray communication stack

Also noteworthy is the point that the scope of implementation of these two modules were at software and hardware levels respectively. Owing to this property, the test and verification for software module comprising of OS schedule table could be easily performed on a simulator. The choice of a software simulator was motivated by the fact that a software simulator provided hardware platform independance. Also an OS schedule table was located in the BSW layer of AUTOSAR layered architecture, beneath the application layer. Its implementation thus should be generic, independent of hardware architecture of the processor. Power PC simulator tool provided by lauterbach inc. was used as simulator for validating the schedule table.

7.1 Test and verification of OS schedule table

Feature and functional testing was performed for OS schedule table which is described in detail below.

7.1.1 Feature testing

Following schedule table features were validated:

- OS Schedule table Id.
- OS Schedule table length $\leq$ Drive counter length.
- Schedule table synchronization strategy.
- OS schedule table minimum shift $\leq$ Delta
- OS schedule table miximum shift $\leq$ Delta

7.1.1.1 Schedule table Id

The C configuration file for OS schedule table comprised of a parameter "OS_SCHED_COUNT_MAX" which gave the number of OS schedule tables configured. The schedule table Id was compared to check that a valid schedule table was passed to a schedule table API. If the schedule table Id was invalid the program execution went into an error hook comprising of an infinite loop.

7.1.1.2 OS schedule table length

A driver counter rounded off to an initial starting value. Thus it was necessary to have a schedule table length to be always less than or equal to the drive counter. If this condition was encountered to be false the program went into error hook and the processor execution was halted.

7.1.1.3 OS schedule table min max shift

As discussed in the previous chapter that based on the state of the synchronization counter the drive counter adjusts itself. Every expiry point is seperated by finite length called delta. Based on whether the schedule table is late or early compared to the synchronization counter the next expiry point is scheduled early or delayed accordingly. An important point to note was that the

next expiry point could not be moved forward to previous elapsed time until the next execution cycle occurred. While it could not also be delayed to a point beyond the adjacent neighbour of the next expiry point. Hence it was necessary to keep the schedule table min or max shift to be always less than delta between the expiry points. Thus if the computed delta was found greater than OS schedule table min max shift at any time instant an error hook was called to halt the processor execution.

7.1.2 Functional testing

The functional testing comprised of implementing a test C file for schedule table that triggered the schedule table and executed all the expiry points till it came to a halt. The preliminary aspect of testing involved validating the release of trigger points at correct time stamps. Both the drive and synchronization counter were incremented using software counter. The dummy code comprised of creating asynchronocity between the two counter to validate that the schedule table could adjust its expiry points at run time by calculating the jitter between the two counters. Also noteworthy is the point that the relative priority between the tasks in the expiry point was carefully assigned as the event triggered tasks were never terminated always waiting for an event thus in such a scenario if a time triggered task was assigned a lower priority it could have never prempted the task and would have suffered from starvation. An example of test code ment for functional testing is as follows:

```
for(;;){
        LDEBUG_FPUTS("Main Task starts\n");
/***List of Schedule table connected to the counter ***/
SLIST_FOREACH(sched_obj,&c_p->sched_head,sched_list){
sched_obj->expire_val=Os_SchTblGetInitialOffset(sched_obj);
if( !StartScheduleTableSynchron(sched_obj->id) && !SyncScheduleTable(sched_obj->id,c_p_sync->
/**Bare minimum number of lines of code should exist to reduce the overhead on schedule-table
while(sched_obj->state!=SCHEDULETABLE_STOPPED){
    exp_id=sched_obj->expire_curr_index;
IncrementCounter(COUNTER_ID_DriveCounter);
/**This code lines has been added to create deviation**/
#if 0
if(Os_CounterGetValue(c_p)%2==0){
IncrementCounter(COUNTER_ID_SynchroCounter);
}
#endif
/**In order to compute the deviation and state of the table Running or Running & Synchronous*
if(c_p->val==1)
SyncScheduleTable(sched_obj->id,0);
else
SyncScheduleTable(sched_obj->id,c_p->val);
/** Computes the expire value, state and executes the task and events**/
/** Also the updation of value , state and only after the occurance of an expiry point **/
if(Os_CounterGetValue(c_p)==sched_obj->duration){
sched_obj->state=SCHEDULETABLE_STOPPED;
}
Os_SchTblfunc(sched_obj,c_p);
/*
```

7.2 Test and verification of Flexray communication stack

As previously discussed, a Flexray communication stack comprised of a Flexray interface and a Flexray driver . It was the Flexray driver that lay beneath the microcontroller abstraction layer in an AUTOSAR architecture that directly communicated with the hardware of the Flexray peripheral unit. The test and verification focused on correctly initializing the Flexray hardware controller. The verification of Flexray peripheral registers was done with the assistance of a hardware debugger. The hardware debugger comprised of either a UDE or WinIDEA iSYSTEM for debugging the values allocated to the special function registers of the Flexray peripheral unit. The Flexray controller had several transitional states , before eventually reaching the normal active state. The following registers were of utmost importance to be debugged for a successful state transition:

- MCR
  The MCR was a 16 bit register in which the clock select and the prescaler bit fields decided the clock rate of protocol engine and the bus bandwidth respectively. This two bit fields constrained all the participating nodes of the cluster to have similar clock rate of either 40 MHz or 120 MHz . The bus bandwidth could vary from 0.71 to 10 Mbps. However it was necessary to configure all the nodes to have similar prescaler bit value. A discrepancy in which it could lead to the communication failure of the cluster.

- PCR
  The PCR registers comprised of several 16 bit registers. Each register comprised of bit fields that were used by the protocol engine for its correct functioning according to Flexray specifications 2.1. As discussed in the previous sections the cluster comprised of atleast two coldstart nodes that exchanged syncronization and startup frames to successfully coldstart cluster. In the light of this context few points were important with respect to proper functioning of the Flexray MAC layer. The synchronization and startup frames comprised of CRC header field which if found incorrect was discarded by the MAC layer. The CRC header field to be concatenated to the startup frame was taken from protocol configuration register 18. Thus it was necessary to compute the header correctly. Also noteworthy is the point that a leading coldstart node was configured to send the frames in the static slot prior to the non coldstart node. In an event of discrepancy to meet these two conditions a communication failure was encountered for the Flexray cluster. A CRC header calculater for the verification of CRC header is as follows:

```
#define CRC_HEADER_POLY 0xb85
#define CRC_HEADER_IV 0x1a
#define CRC_HEADER_LENGTH 11
#define CRC_HEADER_DATA_LENGTH 20
unsigned int crc_header_ref(unsigned int data) {
unsigned int shiftreg = CRC_HEADER_IV;
int i;
int bit;
for(i = CRC_HEADER_DATA_LENGTH-1; i >= 0; i--) {
bit = ((shiftreg >> (CRC_HEADER_LENGTH-1)) & 0x1) ^((data >> i) & 0x1);
shiftreg <<= 1;
if(bit)
shiftreg ^= CRC_HEADER_POLY;
shiftreg &= (1 << CRC_HEADER_LENGTH)-1;
}
return shiftreg;
}
```

```
unsigned int create_header(unsigned int frame_id,unsigned char payload_length,unsigned c
return (sync_bit << 19) |(startup_bit << 18) |payload_length |(frame_id << 7);
}
unsigned int crc_header(unsigned int frame_id,unsigned char payload_length,unsigned char
return crc_header_ref(create_header(frame_id, payload_length,sync_bit, startup_bit));
}
```

- POCR
  The state transitions for the controller were made by writing the protocol command bit field. However for a successful transition it was necessary to monitor the busy bit flag in the register. A write operation was unsuccessful if the busy bit flag was raised. Thus the busy bit flag needed to be monitored before a write operation. However while testing the Flexray module initialization this check point was evaded by implementing the dummy waiting loops for sufficiently large time within which the busy flag was cleared.

- PIFR1
  A illegal state transition if any was indicated by the illegal protocol command flag in the register. Thus it was necessary to monitor this bit flag after performing every write operation in protocol operation command bit field. Also noteworthy is the point that their were timer related flags in this register to indicate the expiry of a configured timer. It is these flags that assisted in the logic implementation in the ISR as discussed in the previous chapter.

- PSR1
  This register was used to debug the startup process of the Flexray cluster. This register comprised of bit fields that comprised of bit flag that indicated a successful startup of the flexray cluster. In the event of a startup failure the controller performed a retry based on the number of coldstarts attempts indicated by the certain bit fields. Thus based on the number of startup attempts and the flag fields a controller was initiated to be reset when the startup failed.

7.3 Unit level testing

The individual buffers configured for data frames were implemented as non queued buffers. A non queued buffer is one such that in an event of write operation to the buffer the old data gets overwritten. Thus to avoid any overwriting of the buffer and loss of data frame due to it, the static slots for individual buffers were configured for every third static slot. Under the assumption that the latency of transmitting the data buffer on the bus being atmost a slot period, it was certain to reach in the time period of at the most two consecutive static slot period after the data was latched on to the bus. The receiving node was configured to read the buffer at every third cycle. Thus by transmitting and receiving the data every third cycle a data was assured to be never overwritten before being read by the receiving node. Also noteworthy is the point that if the transmitting node was unable to write a data to the individual buffer, a null frame was transmitted. Thus a jitter induced, when the expiry point lagged behind in writing the buffer before the scheduled transmission time was, estimated by incrementing a global variable called jitter. Every null frame transmission lead to an increment of this jitter value. Thus giving a proportionate measure such that higher the jitter in the system higher was the jitter valued achieved.

8 Results

The integration testing approach went along in two different directions. One of them was towards Inter-ECU synchronization while the other was in the direction of Intra-ECU synchronization. In the Inter-ECU an attempt was made to trigger the schedule table, compute data by the tasks and then multiplex it over the bus to be transmitted to the destination node. While in the Intra-ECU synchronization and attempt was made to read the Flexray macrotick and synchronize the schedule table accordingly before triggering an expiry point.

8.1 Inter-ECU synchronization

An ISR was used to trigger the schedule table which itself could suffer from run time latency. In order to accomodate this latency and add jitter tolerance, a maximum ISR delay of 500 Flexray macrotick was allocated. Also similar to the OS schedule table their existed time stamps in the Flexray peripheral at which the configured buffers were supposed to multiplex the data on to the physical bus. However if the buffer were not written in time with the data and the time stamp expired a null frame was transmitted. In order to accomodate latency for schedule table processing and writing the data into the buffer an additional 500 Flexray macrotick were added to accomodate the latency in processing the OS layer tasks. However the additional macrotick added were not sufficient enough for the data to be written to the buffers prior to the expiry of the time-stamp due to following overhead:

- Schedule table processing
- Context switching of tasks
- Task processing
- Background OS ISR's

8.2 Intra-ECU synchronization

The OS schedule table was driven by an OS counter, whose tick frequency was of 1 ms. The Flexray macrotick frequency derived from the oscillator frequency calculated to be of 25 ns. A Flexray cycle was of 5000 macrotick length corresponding to a duration of 0.125 ms. Clearly eight Flexray communication cycle corresponds to one OS tick. The schedule table has been choosen to be of 50 ms/400 cycle length respectively. Also an important feature to be noted was that, the schedule table were equidistant. Each expiry point within the schedule table could have a min/max shift of 5 ms about its mean position. The API to read the Flexray macrotick was called after the triggering of the expiry point, such that in an event of deviation between the OS tick and the macrotick count, the next scheduled expiry point could be adjusted. Under the worst case assumption that every expiry point suffered from a maximum deviation, that is each expiry point needed to be adjusted by 5 ms, it could be easily deduced that the jitter reduction in the system could be achieved to be ≤ 0.05.

9 Discussion

The software computational speed always lost to the hardware speed. The reason being software counters at the higher layers were a derivative of hardware clocks. Several hardware clock ticks constituted the software counter tick. Thus by the time stamp a software tick clocked to a specific value, the hardware clock used to advance to much higher values compared to the software clock. The Flexray peripheral clock turned out to be too fast compared to OS counters. This difference in the clock rates severely deteriorated the performance in reference to the Inter-ECU synchronization. While the difference in clock rates in the Intra-ECU communication was compensated by deriving the time scale concluded from several hardware clock cycles. The OS environment imposed a strict constraint of implementing the OS counter tick as the drive counter for schedule table. A alternative solution of driving the schedule table with a hardware counter would have been a possible solution. However such a solution would have forced the schedule table to be very close to the hardware layer instead of being a part of OS layer. In the context of Flexray peripheral, the start up state was very important. The Flexray protocol however does not discuss in detail about the API calls to be taken care of in the context of parent and child coldstart nodes. In the project the flaw in understanding the API calls in the startup state led to an unnecessary delay in the project timeline. The Flexray startup state was inferred as behaving like an Ethernet protocol performing contention resolution, which was not the case in reality. In the context of physical layer it was very important to understand the transceiver operation being explicitly controlled by the Flexray controller rather than being statically controlled by the pin voltages, which is the normal case in CAN based bus architecture.

10 Conclusion

As discussed in the previous section the difference in the counter speed for the clocks caused a race around condition between the software and the hardware. Also to be noted that the processor MPC5567 used during the project comprised of a single core. A single core ECU was not sufficient for Inter-ECU communication. A multi-core processor would have been a possible solution in the context of current anomaly. However it was also necessary to keep in mind a complete knowledge of worst case execution time of individual tasks in the schedule table along with the overhead cost caused due to context switching and task pre-emption. The future scope of the project can be extended in the context of deteriorated performance of the Inter-ECU communication. A p-thread processing of schedule table, running on several cores could be a possible solution. However writing to the memory if shared would also impose a strict usage of semaphores to ensure mutual exclusion in order to remove chances of memory corruption. The implementation can be made even more predicatable by turning off all the OS ISR's which caused a lot of timing unpredictability in the system. In reference to the hardware another interesting feature would be look into processor architecture that supports configuring and installing user defined hardware interrupts. Such hardware interrupts could be set at specific Flexray macrotick check points for synchronization of schedule table in order to achieve better predictability of the system. Also hardware fault tolerance can be added into the system by enabling both the Flexray channels. Software redundancy can also be implemented by enabling several redundant copies of static slots in the static segment of Flexray communication cycle.

References

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, pp. 11–33, Jan 2004.

[2] S. Bunzel, "Autosar â" the standardized software architecture," *Informatik-Spektrum*, vol. 34, no. 1, pp. 79–83, 2011. [Online]. Available: http://dx.doi.org/10.1007/s00287-010-0506-7

[3] D. Paret, *FlexRay and its Applications*. US: John Wiley Sons Inc, 2012. [Online]. Available: www.summon.com

[4] V. Pallavi and N. Raju, "Can protocol implementation for industrial process," *International Journal of Computer Science and Mobile Computing*, vol. 2, no. 11, pp. 106–112, 2013. [Online]. Available: www.summon.com

[5] W. Elmenreich and S. Krywult, *A comparison of fieldbus protocols: LIN 1.3, LIN 2.0, and TTP/A*, 2005, vol. 1, pp. 747–753. [Online]. Available: www.summon.com

[6] ARCCORE, *Arctic Studio*, 2014 (accessed August 01, 2014). [Online]. Available: http://www.arccore.com/products/arctic-studio/for-application-developers/

[7] FreescaleSemiconductor, *MPC5567 Microcontroller Reference Manual*. [Online]. Available: http://cache.freescale.com/files/32bit/doc/ref_manual/MPC5567RM.pdf

[8] QRTECH, *ODEEP*, 2011 (accessed August 01, 2014). [Online]. Available: http://www.qrtech.se/produkter/odeep

[9] FreescaleSemiconductor, *MPC5567EVB Development Board for the Freescale MPC5567*, 2010 (accessed August 01, 2014). [Online]. Available: https://www.axman.com/content/mpc5567evb

[10] NXPSemiconductor, *TJA1080 FlexRay transceiver*, 2007 (accessed August 01, 2014). [Online]. Available: http://www.nxp.com/documents/data_sheet/TJA1080.pdf

[11] M. K. Mishra, "An improved round robin cpu scheduling algorithm," *Journal of Global Research in Computer Science*, vol. 3, no. 6, pp. 64–69, 2012. [Online]. Available: www.summon.com

[12] pls Development Tools, *UDE*, 1990 (accessed August 01, 2014). [Online]. Available: http://www.pls-mc.com/content/view/12/124/

[13] iSYSTEM, *WinIDEA iSYSTEM's*, 1986 (accessed August 01, 2014). [Online]. Available: http://www.isystem.com/products/software/winidea

[14] lauterbach inc, *TRACE32*, 1979 (accessed August 01, 2014). [Online]. Available: http://www.lauterbach.com/frames.html?home.html

[15] M. Lluesma, M. Lluesma, A. Cervin, P. Balbastre, I. Ripoll, and A. Crespo, "Jitter evaluation of real-time control systems." IEEE, 2006, pp. 257–260. [Online]. Available: www.summon.com

[16] S. P. P. Pronk, L. M. G. M. Tolhuizen, and K. P. E. N.V, "Contention resolution protocol," 2006. [Online]. Available: www.summon.com

[17] D. P. Borgers and W. P. M. H. Heemels, "Event-separation properties of event-triggered control systems," *IEEE Transactions on Automatic Control*, pp. 1–1, 2014. [Online]. Available: www.summon.com

[18] Z. Wang, H. Lu, and M. Stone, *Message priority assignment algorithm for CAN based networks*, 1992, pp. 25–32. [Online]. Available: www.summon.com

[19] J. Ho Kim, J. H. Kim, S. H. Seo, S. Hyung Seo, J. Hoon Chun, J. H. Chun, J. W. Jeon, J. Wook Jeon, Y. Youl Ha, Y. Y. Ha, T. J. Park, and T. Jin Park, "Distributed clock synchronization algorithm for industrial networks." IEEE, 2010, pp. 211–214. [Online]. Available: www.summon.com

[20] M. E. Birenbach, G. L. Brookshire, J. N. Dieffenderfer, S. G. Geist, R. A. Moore, T. A. Sartorius, R. W. Smith, and Q. Incorporated, "Two-level interrupt service routine," 2008. [Online]. Available: www.summon.com

[21] T. Harmon, M. Schoeberl, R. Kirner, R. Klefstad, K. H. K. Kim, and M. R. Lowry, "Fast, interactive worst-case execution time analysis with back-annotation," *IEEE Transactions on Industrial Informatics*, vol. 8, no. 2, pp. 366–377, 2012. [Online]. Available: www.summon.com

[22] "time-division multiple access," *McGraw-Hill Dictionary of Scientific and Technical Terms*, 2003. [Online]. Available: www.summon.com

[23] R. Obermaisser and S. (e-book collection), *Event-triggered and time-triggered control paradigms.* New York: Kluwer Academic Publishers, 2005, vol. 22; RTSS 22. [Online]. Available: www.summon.com

[24] OSEKSteeringcommittee, *OSEK*, 2004 (accessed August 01, 2014). [Online]. Available: http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf

[25] P. Richard, "A tool for controlling response time in real-time systems," vol. 2324, 2002, pp. 339–348. [Online]. Available: www.summon.com

[26] M. Booshehri, A. Malekpour, and P. Luksch, "An improving method for loop unrolling," *International Journal of Computer Science and Information Security*, vol. 11, no. 5, p. 73, 2013. [Online]. Available: www.summon.com

[27] FlexrayConsortium, *Flexray Communications System Protocol Specification Version 2.1*, 2004 (accessed August 01, 2014). [Online]. Available: http://www.softwareresearch.net/fileadmin/src/docs/teaching/SS08/PS_VS/FlexRayCommunicationSystem.pdf

[28] *AUTOSAR Specification of FlexRay Driver*, Version 2.5.0, Release 4.0 [pdf]. Munich: AUTOSAR, 2011.

[29] *AUTOSAR Specification of FlexRay Interface*, Version 3.1.0, Release 4.0 [pdf]. Munich: AUTOSAR, 2011.

[30] R. W. Cheston, D. C. Cromer, H. J. Locker, D. B. Rhoades, R. S. Springfield, J. P. Ward, and L. S. P. Ltd, "Method and system for configuring an operating system in a computer system," 2007. [Online]. Available: www.summon.com

[31] G. Buja, A. Zuccollo, and J. Pimentel, *Overcoming babbling-idiot failures in the FlexCAN architecture: A simple bus-guardian*, 2005, vol. 2, pp. 461–468. [Online]. Available: www.summon.com

[32] *AUTOSAR Specification of FlexRay State Manager*, Version 2.2.0, Release 4.0 [pdf]. Munich: AUTOSAR, 2011.

[33] *AUTOSAR Specification of PDU Router*, version 3.1.0, release 4.0 [pdf]. Munich: AUTOSAR, 2011.

11 Appendix

11.1 Flow charts

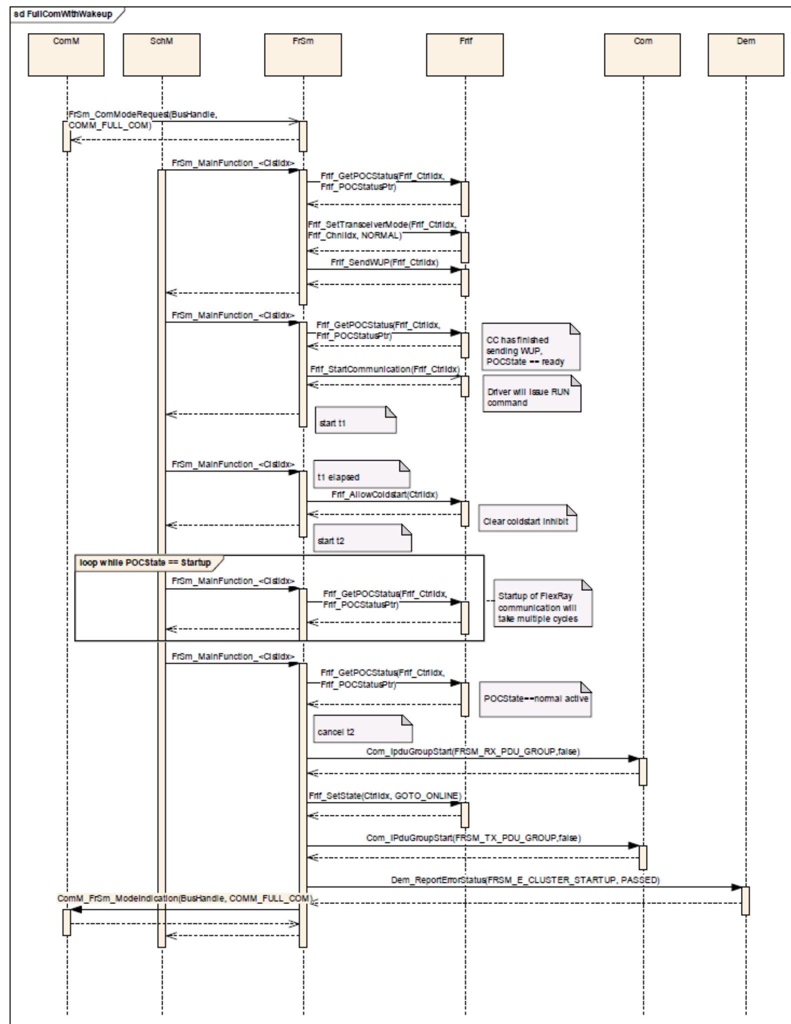

Figure 21: Flexray startup sequence.[32]

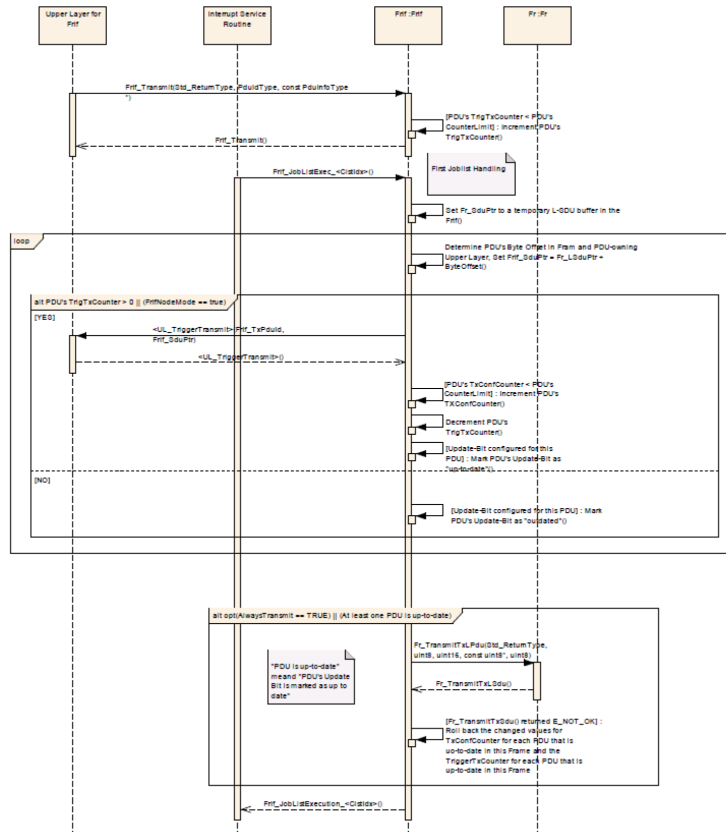

Figure 22: Flexray frame transmission sequence.[29]

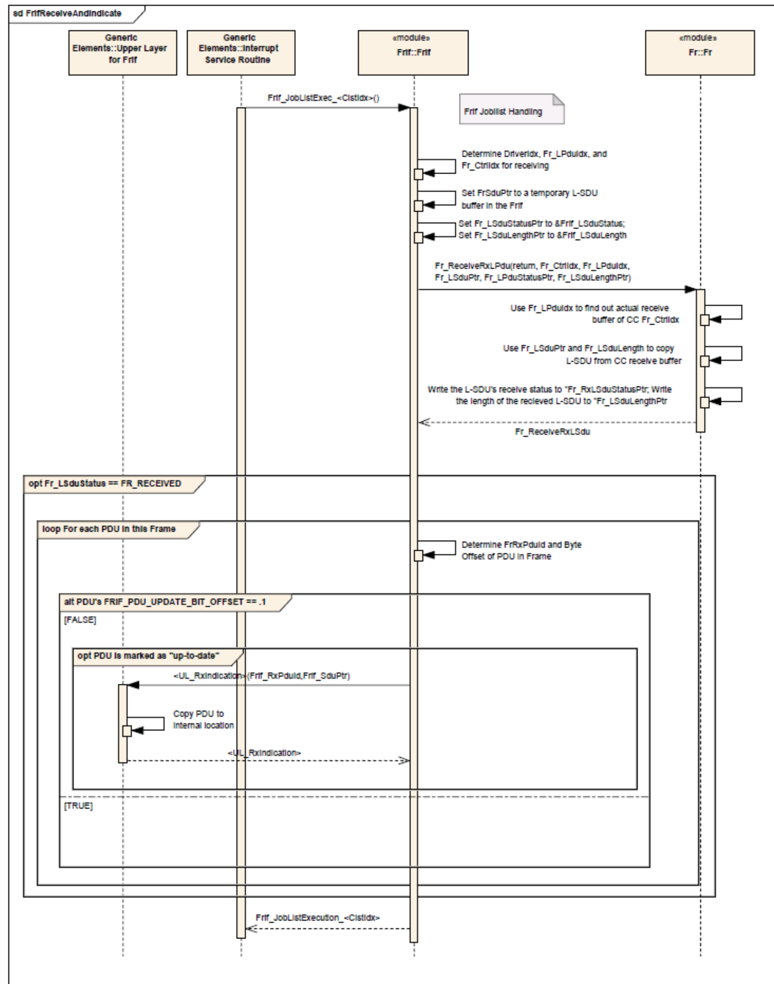

Figure 23: Flexray frame receive and indicate sequence.[29]