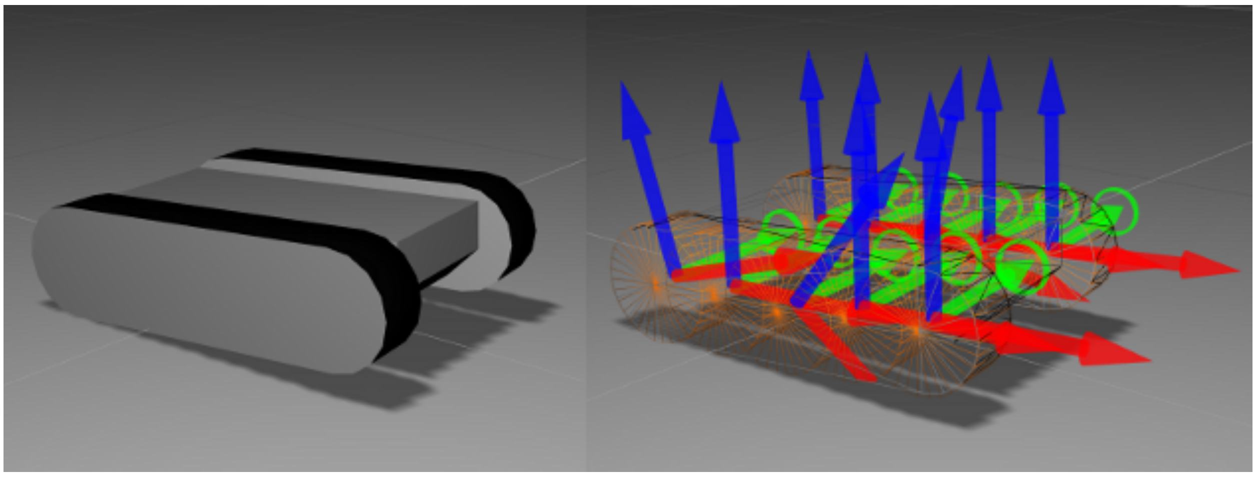
February 2014

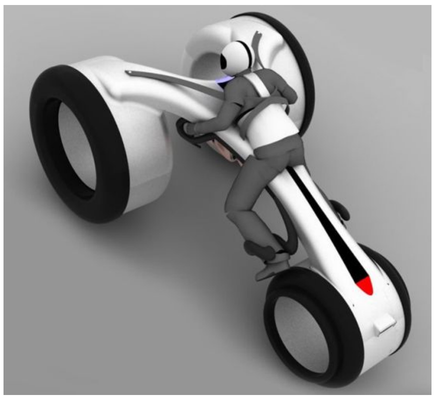


ALTREONIC "FROM DEEP SPACE TO DEEP SEA"

OPENCOMRTOS DESIGNER^{TM}: SIMPLIFYING DEVELOPMENT OF HETEROGENEOUS DISTRIBUTED REAL-TIME EMBEDDED SYSTEMS

Third publication in the Gödel Series:

Systems Engineering for Smarties



Published by:

Altreonic NV

Gemeentestraat 61A B1

B3210 Linden

Belgium

www.altreonic.com

info.request (@) altreonic.com

Feb- 2014

Author: Pieter van Schaik

Contact: goedelseries@altreonic.com

3rd publication in the Gödel Series:

Systems Engineering for Smarties©

OpenComRTOS Designer™: Simplifying Development of Heterogeneous Distributed Real-Time Embedded Systems

Table of Contents

Preface

This booklet is the third in the **Gödel* Series**, with the subtitle "Systems Engineering for Smarties". The aim of this series is to explain in an accessible way some important aspects of trustworthy systems engineering with each booklet covering a specific domain.

The first publication is entitled "**Trustworthy Systems Engineering with GoedelWorks**" and explains the high level framework Altreonic applies to the domain of systems engineering. It discusses a generic model that applies to any process and project development. It explains the 16 necessary but sufficient concepts. This model was applied to the import of the project flow of the ASIL (Automotive Safety Integrity Level) project of Flanders's Drive whereby a common process was developed based on the IEC-61508, IEC-62061, ISO-DIS-26262, ISO-13849, ISO-DIS-25119 and ISO-15998 safety standards covering the automotive on-highway, off-highway and machinery domain.

The second publication is entitled “**QoS and Real Time Requirements for Embedded Many- and Multicore Systems**”. It explains the principles behind real-time scheduling for embedded real-time systems whereby meeting the real-time constraints often is a top level safety requirement. What distinguishes this booklet is that it also deals with systems that have multiple processors (on-chip or connected over a network). The complexity and challenges on such targets mean that the system must now schedule all available resources, such as communication backbones, peripherals and energy. In combination with new functional needs this results in new approaches focusing on the Quality of Service and requiring specific support from the hardware.

This third publication was written as an application note and shows how OpenComRTOS Designer can be used as a simulation as well as a development environment while keeping the source code. This is achieved by way of the transparent programing model that allows considering a network of processors as a virtual single processor one. The application note applies it to the development of a skid steering controller.

The name of Gödel (as in GoedelWorks) was chosen because Kurt Gödel's theorems have fundamentally altered the way mathematics and logic was approached, now almost 80 years ago. The attentive reader will also recognise Heisenberg, Einstein and Wittgenstein on the front page. What all these great thinkers really did was to create clarity in something that looked very complex. And while it required a lot of hard thinking on their side, it resulted in a very concise and elegant theorem or formula. One can even say that any domain or subject that still looks complex is really a problem domain that is not yet fully understood. We hope to achieve something similar, be it less revolutionary, for the systems engineering domain and it's always good to have intellectual beacons to provide guidance.

The Gödel Series publications are freely downloadable from our web site. Further titles in the planning will cover topics of Real-Time programming, Formal Methods and Safety Analysis methods. Copying of content is freely permitted provided the source is referenced. As these booklets will be updated based on feedback from our readers, feel free to contact us at goedelseries @ altreonic.com.

Eric Verhulst,

CEO/CTO Altreonic NV

*: pronunciation [ˈkʊʁt ˈgøːdəl] (🔊 listen)

Abstract

This technical note serves to demonstrate how Altreonic's OpenComRTOS Designer allows for embedded software developers of heterogeneous distributed systems to cross develop and simulate their application on a PC environment and seamlessly transfer their code to the target hardware. To this end a Microsemi SmartFusion2 SOC Evaluation Kit is utilised as the target hardware for a hub motor skid steering based speed controller of an Electric Personal Mobility Device (EPMD).

1 INTRODUCTION

It is evident that the current trend in the real-time embedded systems domain is the proliferation of multiprocessor designs. Multiprocessor designs bring with it many advantages but also introduces several challenges. In many development projects the software and hardware design is done concurrently and therefore requires the embedded software engineers to commence with software design prior to the hardware being finalized. In theory such a concurrent approach reduces development time and is therefore a potential cost saver. However, if the software development environment does not facilitate for the software to be developed such that it can be functionally verified and at the same time be easily ported to the target hardware, the potential time savings can be easily lost during hardware-software integration.

Altreonic's OpenComRTOS Designer with its Visual Modeling and Simulation Environment, named Visual Designer, allows to cross develop OpenComRTOS applications on a PC, develop a simulator and generate the runtime code for the target hardware with minimal or no source code modifications. [6], [7]. This capability of Visual Designer does not only alleviate schedule related issues of typical concurrent engineering projects but also facilitates a systematic software development process which results in improved software quality.

This technical note demonstrates the software development approach for an application which entails a speed controller for a skid steering based Electric Personal Mobility Device. The approach commences with the architectural design and functional verification of the application from within Visual Designer on a PC environment. Once the functional verification yields acceptable results part of the application will be transferred to the target hardware for integration with the actual motor controller and motor.

2 SKID STEERING

Skid steering is a steering technique often used on differential drive systems such as wheeled or tracked robot platforms as well as vehicles such as excavators and bulldozers. This method generates differential velocity at opposite sides of the vehicle in order to turn a vehicle with non-steerable wheels or tracks. Figure 1 illustrates this concept for the case of a tracked robot platform [1].

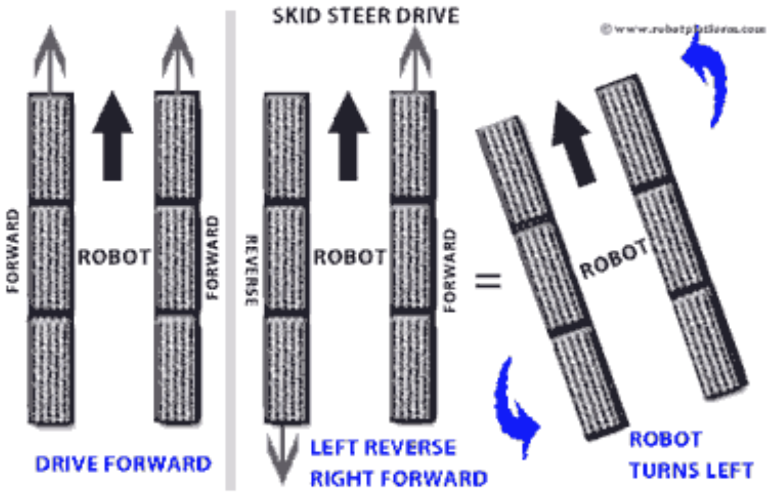


Figure 1: Tracked Robot Platform Skid Steering

In its simplest form the effective turning radius is therefore a function of the difference in velocity of the left vs right wheels. Figure 2 below illustrates the relationship of the difference in velocity to the turning radius [1]:

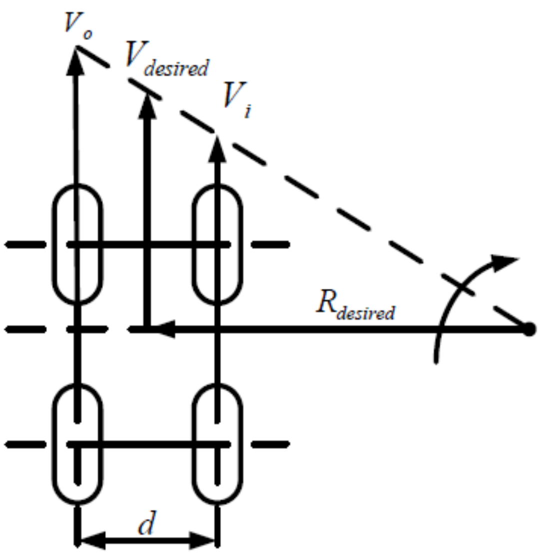


Figure 2: Relation between differential velocity and turning radius

The desired turning radius, $R_{desired}$, can therefore be expressed as follows:

$$R_{desired} = \frac{V_o + V_i}{V_o - V_i} \bullet \frac{d}{2}$$

However, it is evident that the more accurately the torque applied to each wheel can be controlled the better the steer mechanism will perform. Such accurate torque control is arguably best achieved through the use of motorized wheels, also known as hub-motors. The increased availability of hub-motors holds many advantages for the development and proliferation of electrical vehicles. Not only do they enable the design of higher performance skid steer systems but also permit packaging flexibility by eliminating the central drive motor and the associated drive line and transmission.

3 EPMD SPEED CONTROLLER HARDWARE

Figure 3 below depicts the functional block diagram of the steering controller of the EPMD.

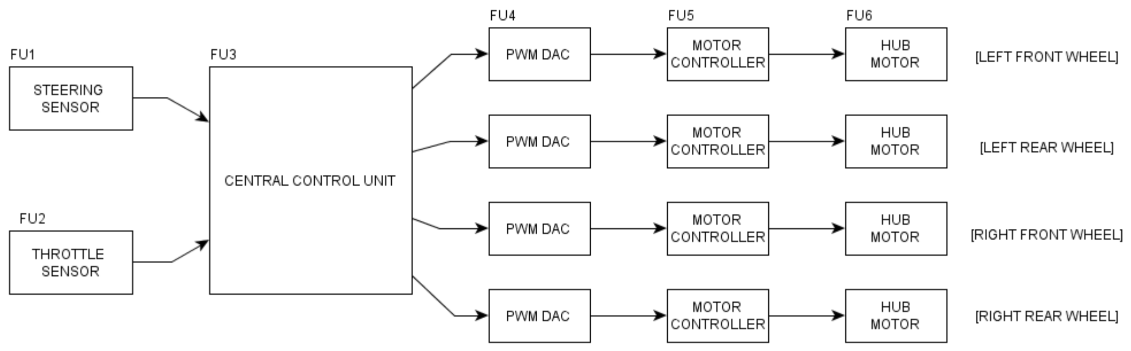


Figure 3: EPMD Steering Control Functional Block Diagram

The following is a brief description of the functional units indicated in Figure 3:

- FU1 - Steering Sensor: This unit is responsible for sensing the steering command issued by the operator.
- FU2 - Throttle Sensor: This unit is responsible for sensing the speed command issued by the operator.
- FU3 - Central Control Unit: This unit is responsible for processing the sensor inputs and calculating the required speed command for each of the Hub motors.
- FU4 - PWM DAC: This unit is responsible for converting the Pulse Width Modulated signal from the CCU to an analog voltage in accordance with the input requirements of the Motor Controller.

- FU5 - Motor Controller: This unit is a COTS motor controller component responsible for commutating the BLDC Hub motor.
- FU6 - Hub Motor: This unit is a COTS BLDC Hub Motor

The required functional behaviour of the steering control system can be described as follows: FU3 receives a throttle command from FU2 and a steering command from FU1. FU3 calculates a "delta" speed command that is to be added or substracted to the current nominal command of each Hub motor. The delta speed command is a function of the current system speed so as to prevent a turning radius that will cause the system to topple over. The updated speed command for each Hub motor is then issued to the motor controller. The speed command from FU3 is in the form of a Pulse Width Modulated signal which is converted to the appropriate analog voltage by FU4. FU5 receives the speed command voltage from FU4 and updates the commutation speed of the FU5 with Hall-Effect sensors providing the feedback of the actual motor speed. Figure 4 below depicts this behaviour in the form of flowchart.

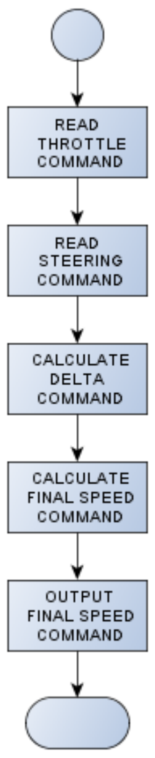


Figure 4: Functional Behaviour of the Steering Controller

For the purposes of this technical note FU1 and FU2 will be implemented in software (simulators) whereas FU3 - FU6 of the front-left wheel will be implemented in actual hardware.

3.1 Central Control Unit

The central control unit will be realised by utilising a SmartFusion2 Starter Kit from Emcraft Systems [4].

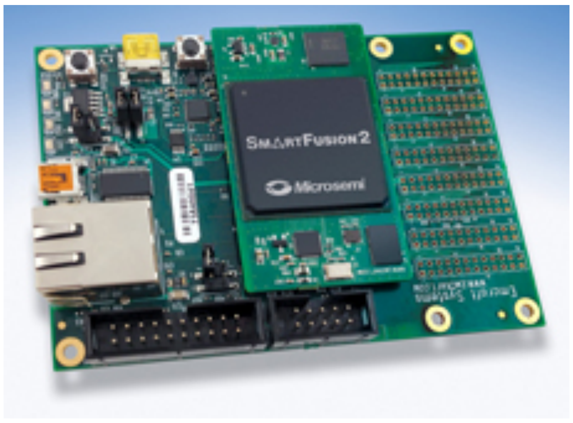


Figure 5: SmartFusion2 Starter Kit

The SmartFusion2 SOC contains an ARM Cortex-M3 Microcontroller Subsystem which is able to interface to peripherals implemented in the FPGA fabric. Appendix A depicts the SOC design implemented on the SmartFusion2 device. The design utilises a CorePWM IP [5] component from the Microsemi catalog which interfaces to the Microcontroller Subsystem through an ARM Peripheral Bus (APB) interface. The following parameters were used for the PWM peripheral implemented in the FPGA fabric:

Table 1 PWM Peripheral Configuration

Master Clock	83	MHz
Max PWM Count	255	
PWM Frequency	32.549	KHz
PWM Period	3.072	Micro-seconds

3.2 PWM DAC

In order for the CCU to interface with the COTS motor controller the PWM output is to be converted to an appropriate analog voltage in accordance with the interface specifications of the motor controller. As described in [3] a RLC circuit can be used as a 2nd order low pass filter to convert a PWM signal into an analog voltage. The following RLC values proved to provide the best performance based on SPICE simulations:

Table 2: PWM DAC Low Pass Filter Parameter

L	0.1	mH
C	0.022	microF
R	91	Ohm
Wn	674.2	Krad/s
Zeta	0.674874062	
BW	112.183	KHz

3.3 Motor Controller and Hub Motor

Figure 6 below illustrates the Hub Motor Kit that was utilised for the purpose of this technical note. The kit provides a matched 48V motor controller and 250W BLDC Hub motor wheel.

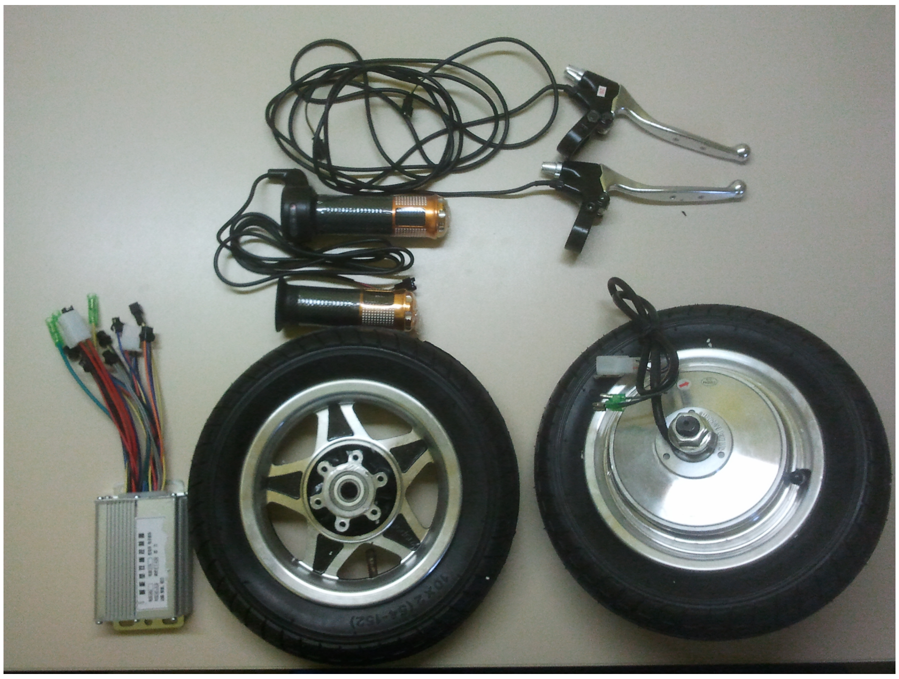


Figure 6: COTS Hub Motor Kit

4 EPMD SPEED CONTROLLER SOFTWARE

As discussed in Section 1, one of the primary goals of this software development approach is to firstly verify the functional behaviour of the application in a simulation environment with the additional requirement that the resulting software architecture can be seamlessly transferred to the target hardware. To this end Altreonic's "Interacting Entities" paradigm guides the architectural design so as to maximise the modularity and portability of the software architecture. According to this paradigm the design commences by identifying the entities involved and thereafter the interactions that are required between these entities in order to implement the required functionality.

4.1 Architecture and Domain Objects

In accordance with the conventional approach of software design one could start by creating a domain model for the EPMD Speed Control Application. The following domain model represents a common layered software architecture approach based on the indicated hardware architecture:

The function of the indicated domain objects can be described as follows:

- Steering Sensor Driver: This entity is responsible for obtaining the data from the steering sensor and abstracts the physical interface between the CCU and the actual sensor. Report the updated steering command to the maintenance manager.
- Throttle Sensor Driver: Responsible for obtaining the data from the throttle sensor and abstracts the physical interface between the CCU and the actual Sensor. Report the updated throttle command to the maintenance manager.
- Motor Controller Driver: Responsible for communicating speed commands to the motor controller and abstracts the physical interface to the actual motor Controller. Reports the actual speed command to the maintenance manager
- Speed Controller: Responsible for obtaining the sensor data, calculating the speed commands and issuing the commands to the motor controller driver. Report the calculated speed command to the maintenance manager
- Maintenance Manager: Report the received data on the maintenance interface for diagnostic purposes.

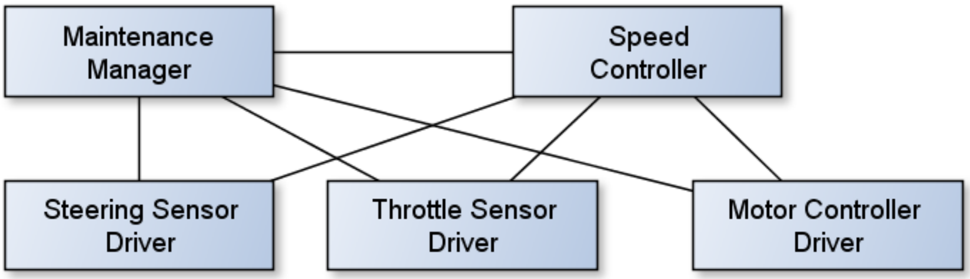


Figure 7 EPMD Speed Control Domain Model

In addition, as previously mentioned the steering and throttle sensors will be emulated in software. In order to easily exchange data with external applications, or even machines that may perform the sensor emulation, a standard UART interface will be utilised. Furthermore, as indicated by the hardware architecture it is already known that the motor controller driver will make use of a PWM peripheral to communicate the speed command to the actual motor controller. The maintenance manager will make use of a standard TCP/IP over ethernet socket interface in order to provide a maintenance interface. The maintenance manager will act as the server to which a client application can connect in order to obtain diagnostic data. The domain model can therefore be further refined as indicated in the Figure 8:

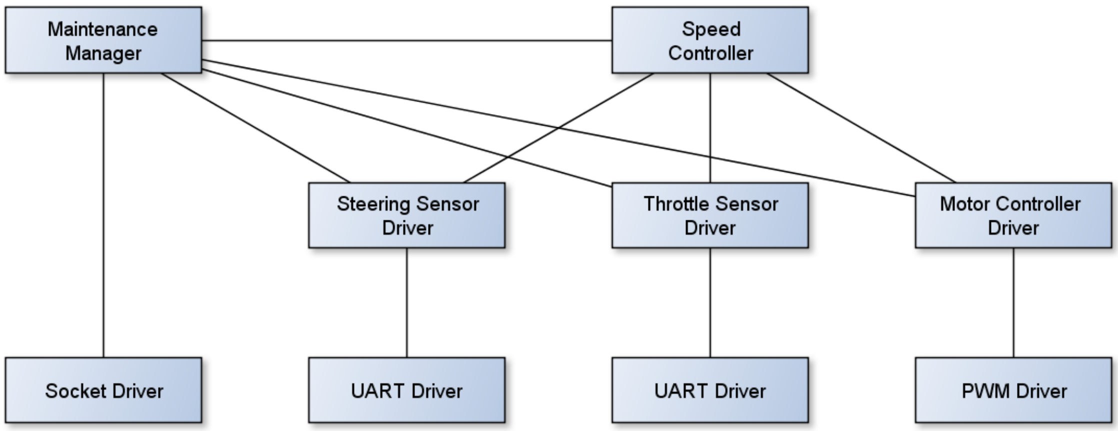


Figure 8: Domain Model indicating Hardware Driver Interfaces

The next step in the design phase is to determine the nature of the interactions (associations) between the domain objects. It is easily seen that the primary activity of the Speed Control Application can be defined as "response to action" and can therefore be considered as a reactive system. The action in this instance being either a throttle or steer command issued by the operator of the EPMD and the response being the updated speed command issued to the motor controller. It is therefore evident that an event driven architecture is best suited for the Speed Control

Application and consequently also determines the nature of the associations between the domain objects. Figure 9 illustrates the next step of refinement of the domain model whereby the nature of the associations between the domain objects are indicated.

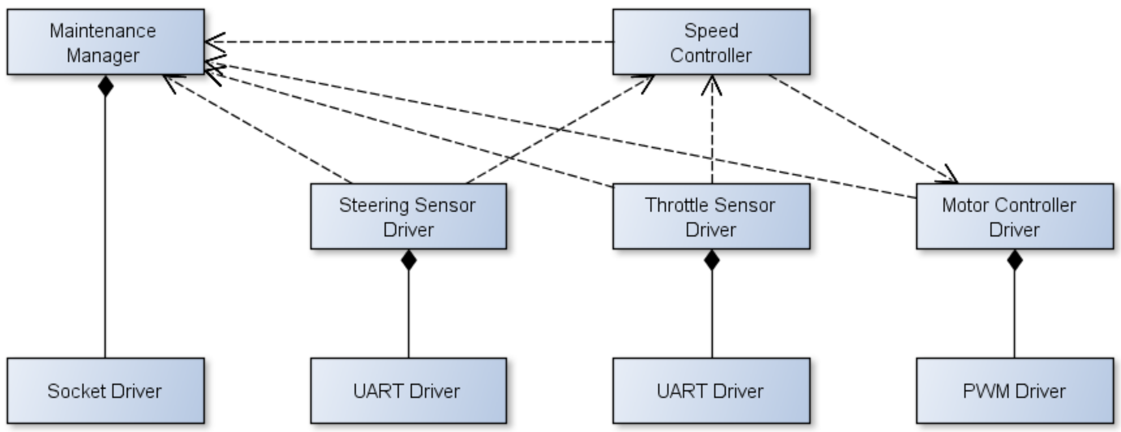


Figure 9: Domain Model with Resolved Entity Association

The decision to implement an event driven architecture implies that data in the system be "pushed" from the producers to the consumers instead of the consumers polling or "pulling" the data from the producers. This is indicated by the direction of the association arrows in Figure 9.

4.2 From Domain Objects to Interacting Entities

At this stage the design can be transferred to the Visual Designer environment where the translation from domain objects and their associations to interacting entities will take place. Initially the Visual Designer project will contain only one Windows node on which the initial development and testing will be done.

Figure 10 depicts the instantiation of the domain objects from the domain model since the design process has shown these to be the interacting entities. The domain objects are instantiated as tasks which will run in their own thread of execution and will interact amongst each other.

As a next step the implementation of the interactions between the sensor driver entities and the speed control entity will be considered. These interactions are of a sampling nature, meaning that the driver entities will sample the incoming sensor data at an appropriate frequency and once an updated sample becomes available it has to be distributed to the interested entities. Given that more entities may be added to the design in future ,which may also be interested in the sensor data, a BlackBoard (BB) Hub provides the most flexibility for distributing the sensor data. In essence the BB is a global space where one entity can publish data and other entities can access it. However, it is only logical that the BB cannot know a priori how many entities will read from it and therefore it remains the responsibility of the reading entities to determine whether the data on the BB has been updated. This is easily remedied by including a "Timer Event" according to which the Speed Control Task will query the BBs of the sensors to determine if the data has been updated. Once either of the sensors have updated their data the Speed Controller Task will calculate the updated speed commands according to a pre-defined algorithm.

The updated speed command will then be issued to the Motor Controller Driver. The interaction between the Speed Controller Task and the Motor Controller Driver is therefore such that the Motor Controller can afford to be idle until commanded by the Speed Controller so as to ensure that the Motor Controller Driver acts upon the speed command with minimum latency. For such interactions OpenComRTOS Designer offers a Port Hub. The Motor Controller reads from the Port and blocks until the Speed Controller has written to the Port. Figure 11 illustrates the updated Visual Designer design with the added interactions between the drivers and the controllers.

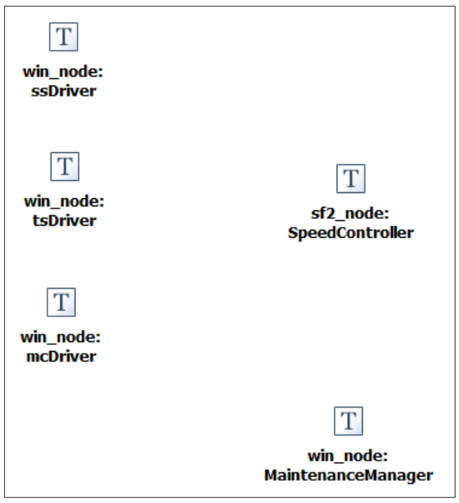


Figure 10: Visual Designer Task Entities

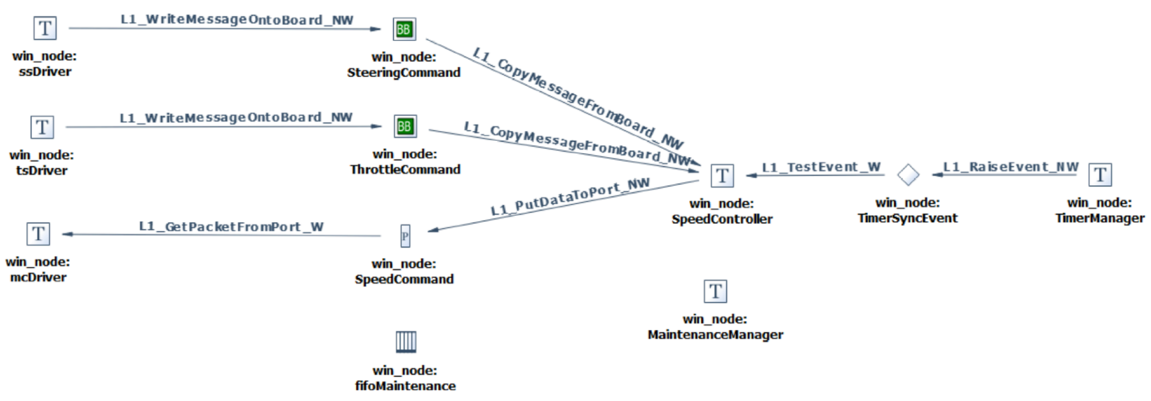


Figure 11: Visual Designer Driver and Controller Entity Interactions

The remaining interactions between the Maintenance Manager and the various entities can now be implemented. These particular interactions exhibit a many-to-one nature and therefore requires a non-blocking buffering mechanism. For such interactions OpenComRTOS Designer offers a FIFO hub which, as the name implies, allows for messages to be queued and subsequently de-queued on a first in - first out basis. Figure 12 presents the finalised Visual Designer design with all entities and interactions defined.

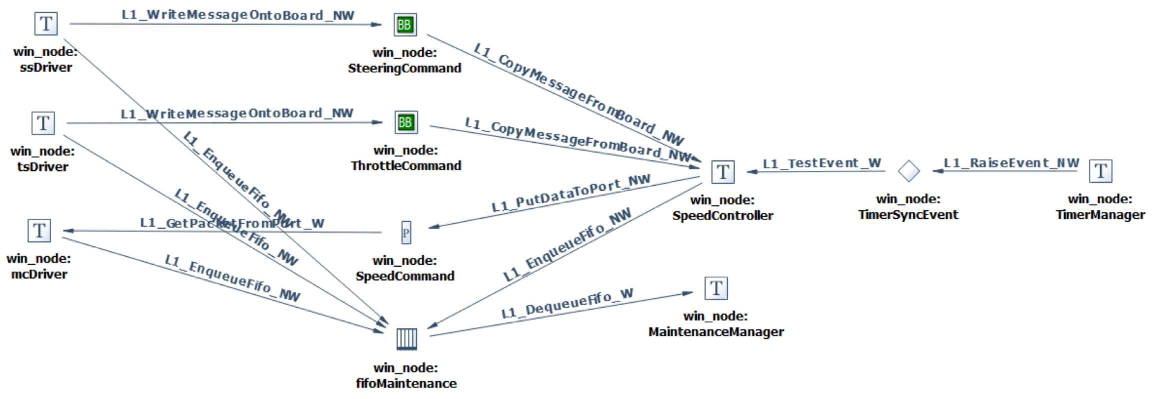


Figure 12: Final Speed Controller Visual Designer Simulation Project

The implementation of the design can now proceed by adding the envisaged functionality to each of the Task entities. Appendix B provides the flow diagrams for each of the Task entities.

4.3 Simulation and Functional Verification

Figure 13 serves as a deployment diagram for the simulation environment of the Speed Controller Application.

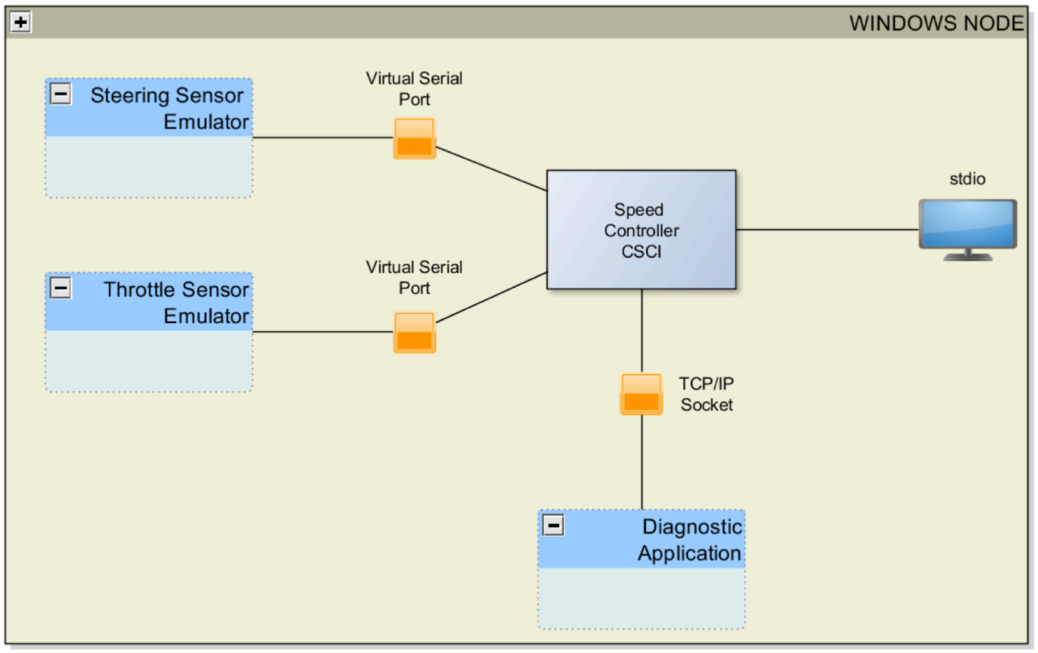


Figure 13: Speed Controller Simulation Deployment Diagram

During the initial simulation phase the Motor Controller Driver will only print the calculated speed command to the standard output (stdio). Several print statements are also added to the Maintenance Manager for debugging purposes. The Steering Sensor Emulator, Throttle Sensor Emulator as well as Diagnostic Application is written in PyQt and runs externally from the Speed Controller CSCI. The sensor emulators connect to the Speed Controller CSCI via virtual serial ports created under Windows. The Diagnostic Application makes use of a socket connection to connect to the maintenance interface provided by the Speed Controller CSCI. Appendix C depicts the simulation environment of the EPMD Speed Controller.

The functional verification of the EPMD Speed Controller application mainly entails verification of the differential drive speed control algorithm. In its simplest form the algorithm can be described as follows:

Two , two dimensional lookup tables are defined namely:

$$OuterWheelDiffDelta[11][11]$$
$$InnerWheelDiffDelta[11][11]$$

These lookup tables contain a ratio factor for the outer and inner wheels that is to be added and subtracted respectively to/from the nominal speed command. The row and column index of the lookup tables are calculated as a function of the current speed and steering commands as follows:

$$RowIndex = f(SteeringCommand)$$
$$ColIndex = g(ThrottleCommand)$$

Where $f()$ and $g()$ are simple scaling functions.

The updated speed command for the outer wheels is therefore calculated as follows:

$$Speed_{Outer-Wheels} = (1 + OuterWheelDiffDelta[RowIndex][ColIndex]) \times NomSpeed$$

The updated speed command for the inner wheels is therefore calculated as follows:

$$Speed_{Inner-Wheels} = (1 - InnerWheelDiffDelta[RowIndex][ColIndex]) \times NomSpeed$$

And

$$NomSpeed = h(ThrottleCommand)$$

where

$$h(x) = a\big|_{x=0}, h(x) = x\big|_{x\neq 0}$$

The function definition of $h()$ serves to cater for the scenario when a steering command is issued whilst the vehicle is stationary.

The definition of whether the left or right wheels are to be considered as inner or outer wheels is based on the direction of the steering command. Thus:

$$LeftSpeed = Speed_{Outer-Wheels}\big|_{SteerDir=Right}$$
$$LeftSpeed = Speed_{Inner-Wheels}\big|_{SteerDir=Left}$$
$$RightSpeed = Speed_{Inner-Wheels}\big|_{SteerDir=Right}$$
$$RightSpeed = Speed_{Outer-Wheels}\big|_{SteerDir=Left}$$

The final control command issued to the motor controllers are therefore:

$$LeftCommand = p(LeftSpeed)$$
$$RightCommand = p(RightSpeed)$$

where $p()$ is also a simple scaling and offset function in order to generate an output voltage in accordance with the motor controller input interface specification.

The envisaged functional behaviour of the algorithm is therefore to increase the outer wheel speed and decrease the inner wheel speed by a factor that is proportional to the current speed command and steering command. The exception being when the nominal speed command is zero in which case the speed of the wheels is a function of the steering command only.

It is readily seen that the simulation environment facilitates to easily verify the functional behaviour and algorithm correctness of the EPMD Speed Controller Application. The multiple output mechanisms provided by the simulation environment allows to quickly determine the integrity of the data paths as well as the algorithm outputs. In addition the use of debuggers is also simplified at this stage since the application is built into a single executable. The ease with which the application can be modified and rebuilt also allows to freely experiment with various implementation concepts as well as execute a multitude of test cases and record results without the need to reprogram target hardware.

4.4 From Simulation to Target Hardware

The next step in the development process entails transferring Task entities to the target hardware. Figure 14 serves as a deployment diagram for the case where the Speed Controller Application has been partly transferred to the target hardware.

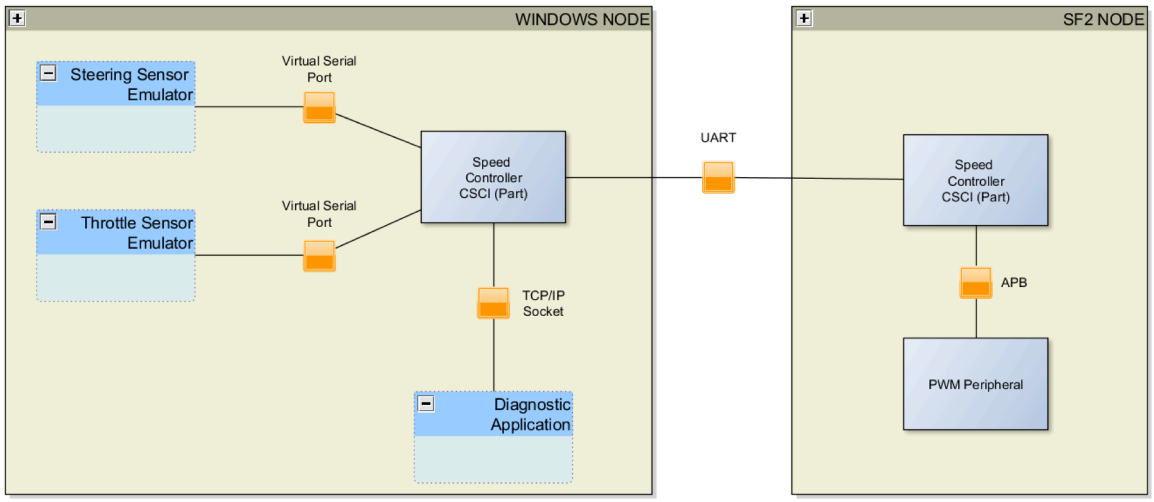


Figure 14: Speed Controller Application Partly Transferred to Target Hardware

In order to transfer a Task to the target hardware the topology definition in Visual Designer needs to updated. This process very simply entails adding a SmartFusion2 node to the topology and defining and connecting the Link Ports between the Nodes. Figure 15 illustrates the updated topology within Visual Designer with the Windows Node and the SmartFusion2 Node as well as the Link Port connection.

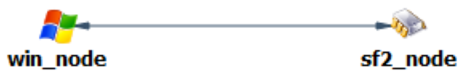
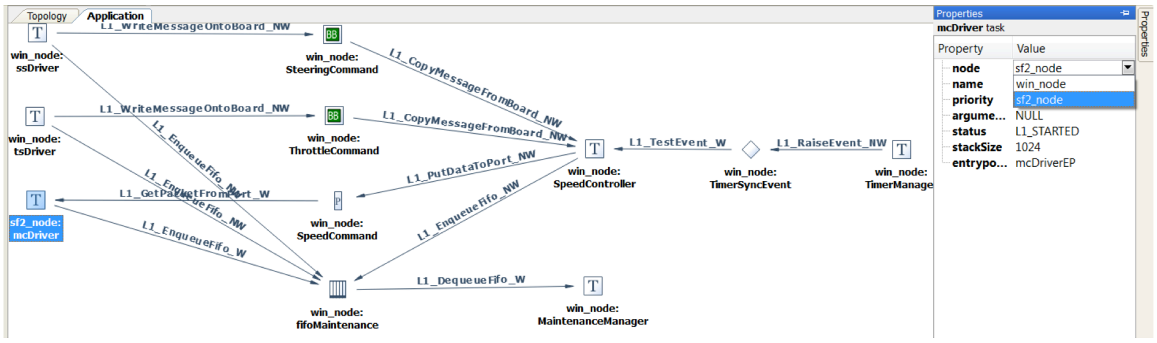


Figure 15: Speed Controller Topology

The next step entails transferring tasks to the target hardware, or to be more specific to the sf2_node in the Visual Designer project. Figure 16 illustrates the mechanism in Visual Designer to assign a Task to a hardware Node. For the purpose of this technical note the mcDriver Task as well as the SpeedController Task will be transferred to the SmartFusion2 Node. The only minor code modifications necessary for the mcDriver task is to remove the stdio output and include the actual PWM peripheral interface.



Properties

mcDriver task

Property	Value
node	sf2_node
name	win_node
priority	sf2_node
argume...	NULL
status	L1_STARTED
stackSize	1024
entrypo...	mcDriverEP

Figure 16: OpenVE Task to Node assignment

Figure 17 shows the resulting Visual Designer project with the mcDriver task and SpeedController task assigned to the SmartFusion2 node.

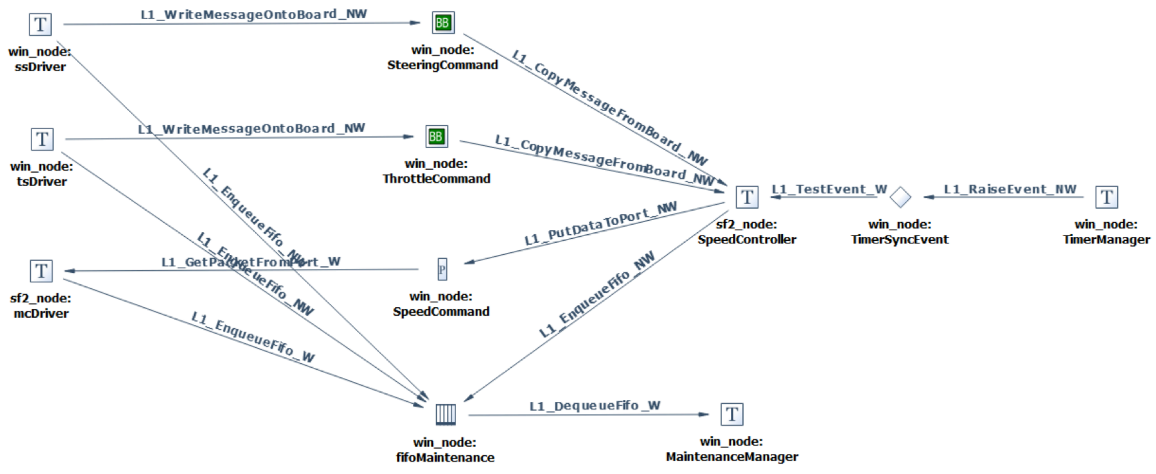


Figure 17: Speed Controller Application with Tasks transferred to Target Hardware

5 RESULTS

5.1 PWM DAC

Prior to being able to integrate the SmarFusion2 hardware with the motor controller it is necessary to verify that the effective resolution of the PWM DAC is sufficient to control the hub-motor. Figure 18 illustrates the results obtained from generating a 1 kHz sine wave from a test program running on the SmartFusion2 Microcontroller. From the Figure it is seen that the resolution of the analog signal is sufficient to serve as an input to the motor controller.

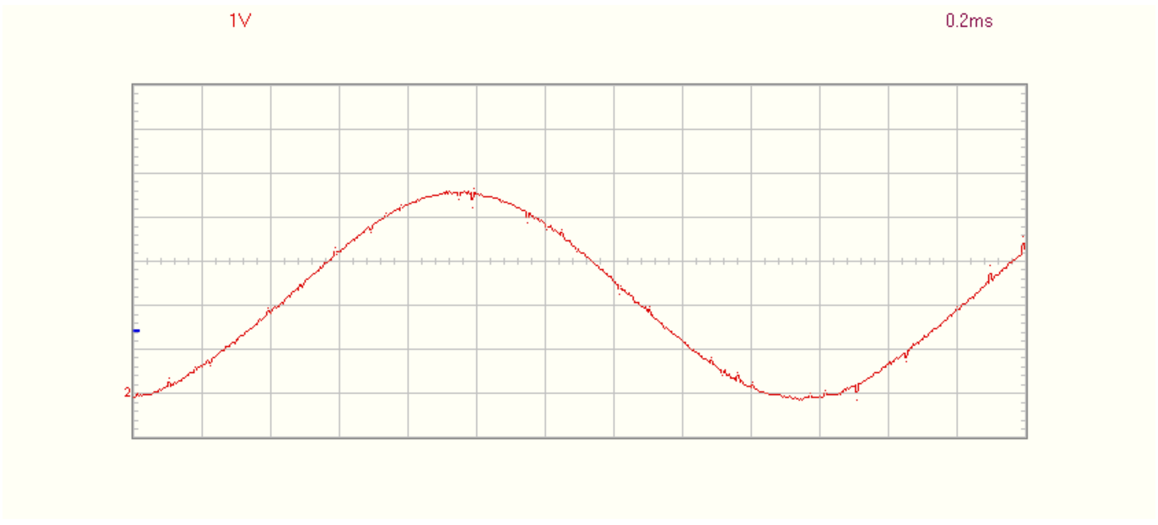


Figure 18: PWM DAC 1 kHz Sine Wave

5.2 Motor Speed Control

In Appendix C the simulation environment is depicted whereas Appendix D illustrates the final hardware experimental setup as well as the sensor emulators and software oscilloscope configuration. The oscilloscope channel connections are as follows:

Table 3: Oscilloscope Measurement Configuration

Channel No	Connection Node	Measured Parameter	Measurement Unit
1	PWM DAC Output	Motor Controller Speed Command	Volt
2	BLDC Hall Sensor Output	Motor Commutation Speed	Hertz

In order to compare the performance of the final implementation with the simulation results identical test cases were executed in the simulation environment as well as the experimental hardware setup. To be more specific a throttle command was set at different set points and for each set point the steer command was varied from -100% to 100% which corresponds to a hard left and hard right steer command respectively. The steer command was incremented by 10% increments whilst the output of the control algorithm was documented in the case of the simulation tests and the frequency of the motor hall sensor output was measured in the case of the hardware tests. Finally, a plot of the Steer Command vs Speed Control Algorithm Output as well as Steer Command vs. Motor Hall Sensor Output Frequency was created to compare the simulation results with the results obtained from the hardware measurements. Appendix E shows the simulation results and measured motor speed for the 0%, 50% and 100% throttle set points. From the results it can be seen that the measurements correlate quite well with the simulation results although the actual motor speed exhibits a steeper gradient which causes earlier saturation. This behaviour can be corrected by further optimising the scaling algorithm $p()$ implemented in the mcDriver Task.

6 CONCLUSION

This technical note has clearly demonstrated the ease with which a heterogeneous distributed real-time application can be developed, simulated and functionally verified on a PC platform and subsequently transferred to the target hardware with Altreonic's OpenComRTOS Designer. In addition it was also shown how Altreonic's "Interacting Entities" paradigm naturally ties in with the traditional software design methodology of utilising a domain model and how easily translation of domain objects and their associations into Tasks and interactions within the Visual Designer environment is accomplished. Furthermore, given that eventual integration with the target hardware is as simple as re-assigning Tasks to a different Node affords more time and effort to be spent refining the architecture and verifying the functional behaviour of the application which altogether results in better quality software.

7 APPENDIX A:

7.1 SmartFusion2 SOC Design

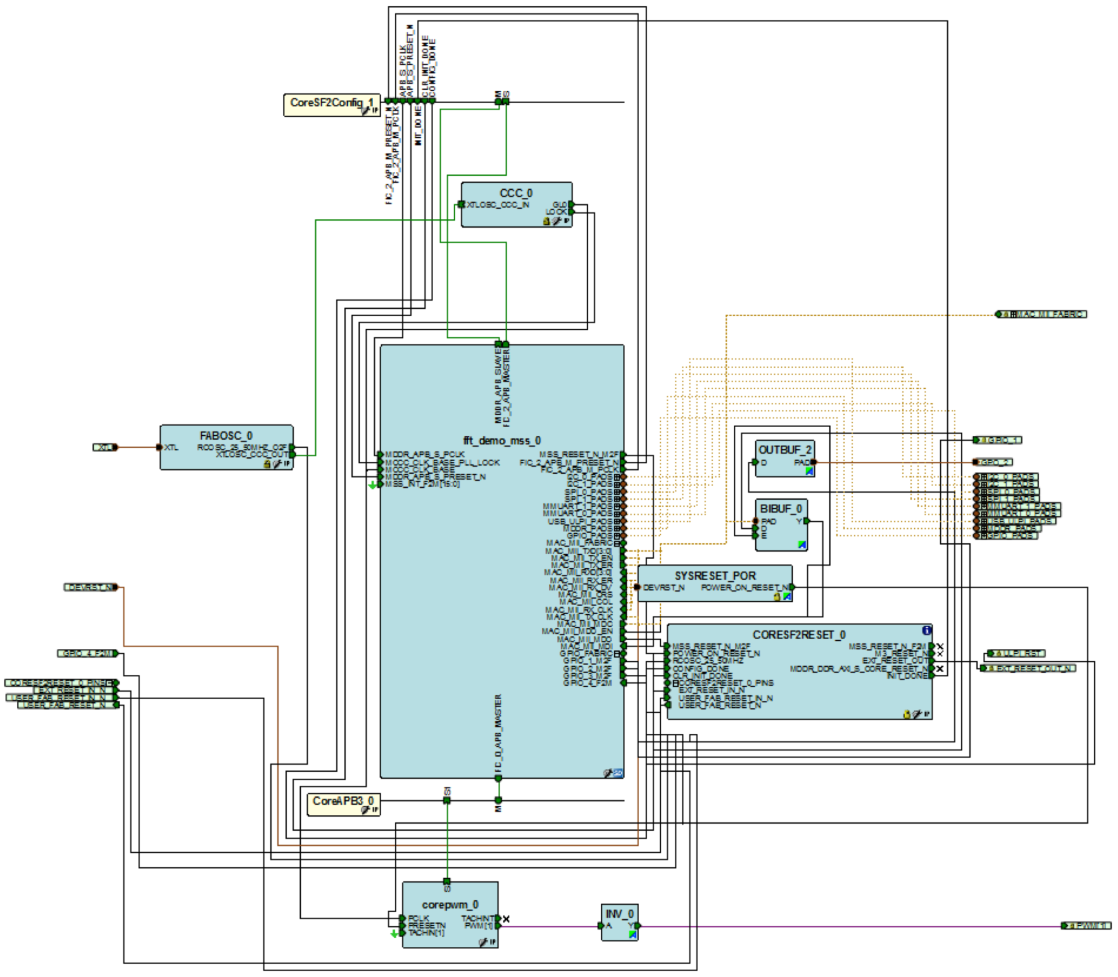


Figure 19: SmarfFusion2 SOC Design Block Diagram

7.2 APPENDIX B: Visual Designer Task Flow Diagrams

7.2.1 ssDriver

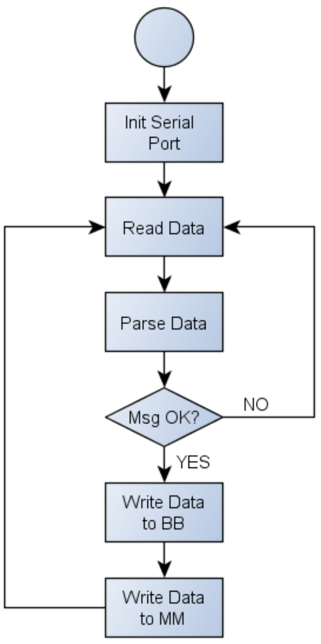
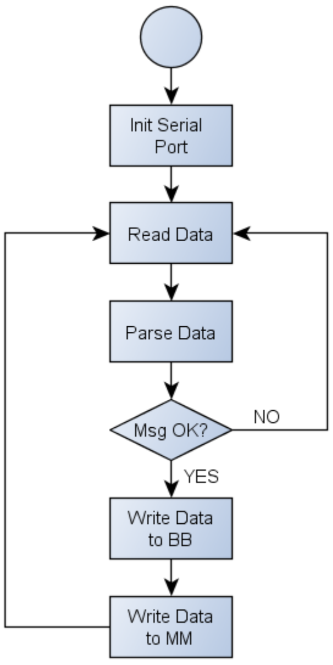


Figure 20: Steering Sensor Driver Flow Diagram



7.2.2 tsDriver

Figure 21: Throttle Sensor Driver Flow Diagram

7.2.3 mcDriver

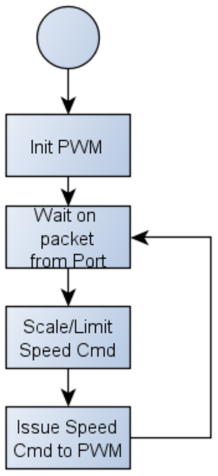


Figure 22: Motor Controller Flow Diagram

7.2.4 SpeedController

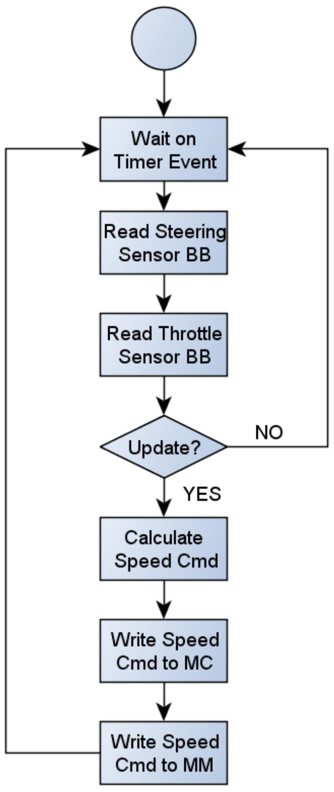


Figure 23: Speed Controller Flow Diagram

7.2.5 MaintenanceManager

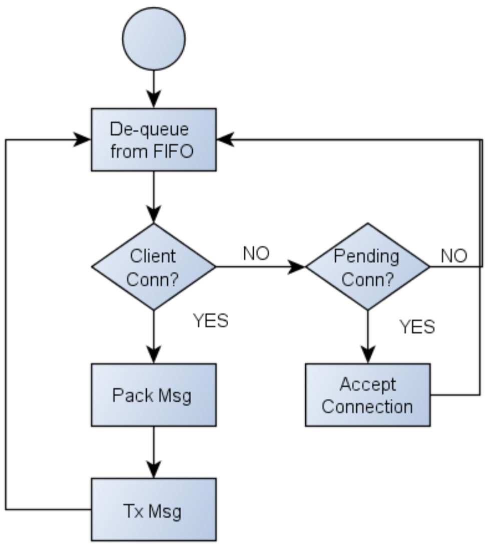
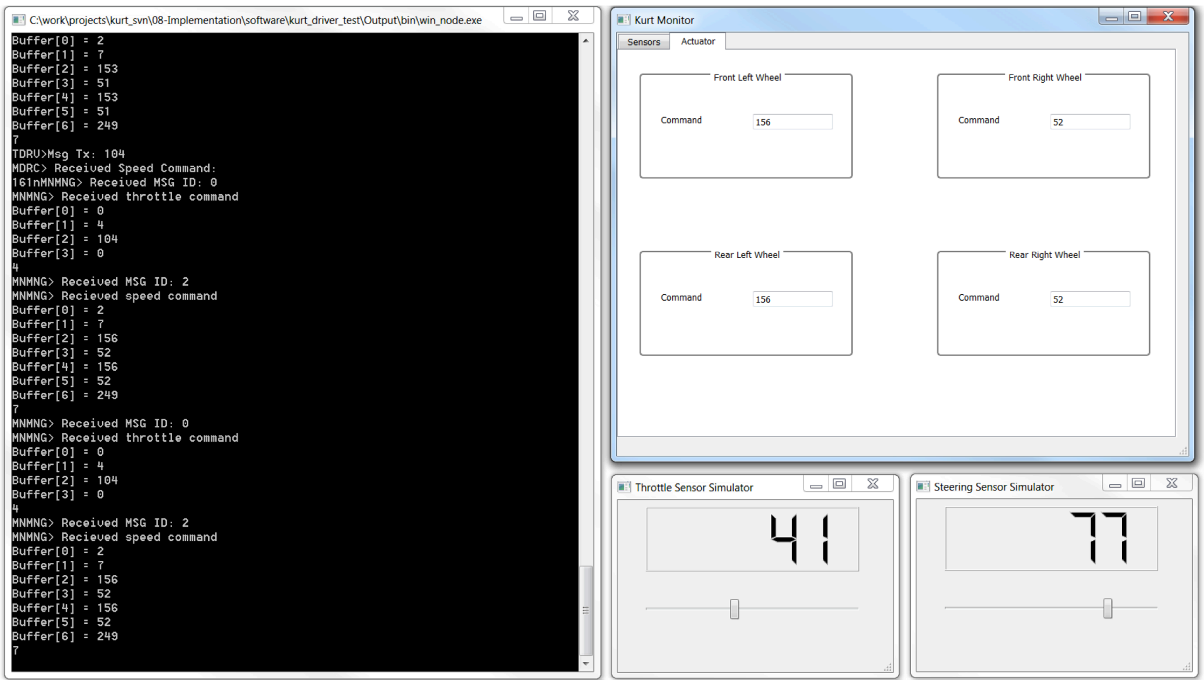


Figure 24: Maintenance Manager Flow Diagram

7.3 EPMD Speed Controller Simulation Environment


C:\work\projects\kurt_svn\08-Implementation\software\kurt_driver_test\Output\bin\win_node.exe

```
Buffer[0] = 2
Buffer[1] = 7
Buffer[2] = 153
Buffer[3] = 51
Buffer[4] = 153
Buffer[5] = 51
Buffer[6] = 249
7
TDRV>Msg Tx: 104
MDRC> Received Speed Command:
161nMNMNG> Received MSG ID: 0
MNMNG> Received throttle command
Buffer[0] = 0
Buffer[1] = 4
Buffer[2] = 104
Buffer[3] = 0
4
MNMNG> Received MSG ID: 2
MNMNG> Recieved speed command
Buffer[0] = 2
Buffer[1] = 7
Buffer[2] = 156
Buffer[3] = 52
Buffer[4] = 156
Buffer[5] = 52
Buffer[6] = 249
7
MNMNG> Received MSG ID: 0
MNMNG> Received throttle command
Buffer[0] = 0
Buffer[1] = 4
Buffer[2] = 104
Buffer[3] = 0
4
MNMNG> Received MSG ID: 2
MNMNG> Recieved speed command
Buffer[0] = 2
Buffer[1] = 7
Buffer[2] = 156
Buffer[3] = 52
Buffer[4] = 156
Buffer[5] = 52
Buffer[6] = 249
7
```

Kurt Monitor

Sensors | Actuator

Front Left Wheel — Command: 156

Front Right Wheel — Command: 52

Rear Left Wheel — Command: 156

Rear Right Wheel — Command: 52

Throttle Sensor Simulator: 41

Steering Sensor Simulator: 77

Figure 24 Maintenance EPMD Speed Controller Simulation Environment

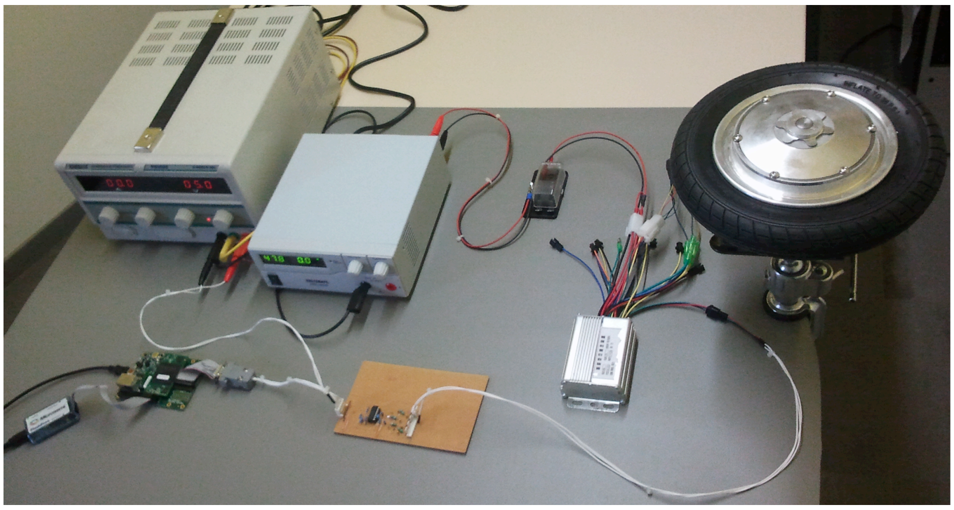


Figure 26 Speed Controller Hardware Configuration

7.4 Speed Control Experimental Setup

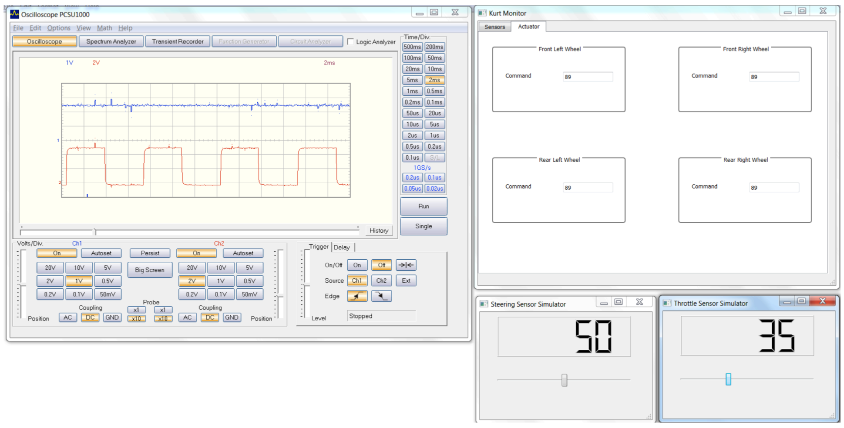


Figure 27: Speed Control Application Measurement Setup

7.5 Speed Control Algorithm Test Results

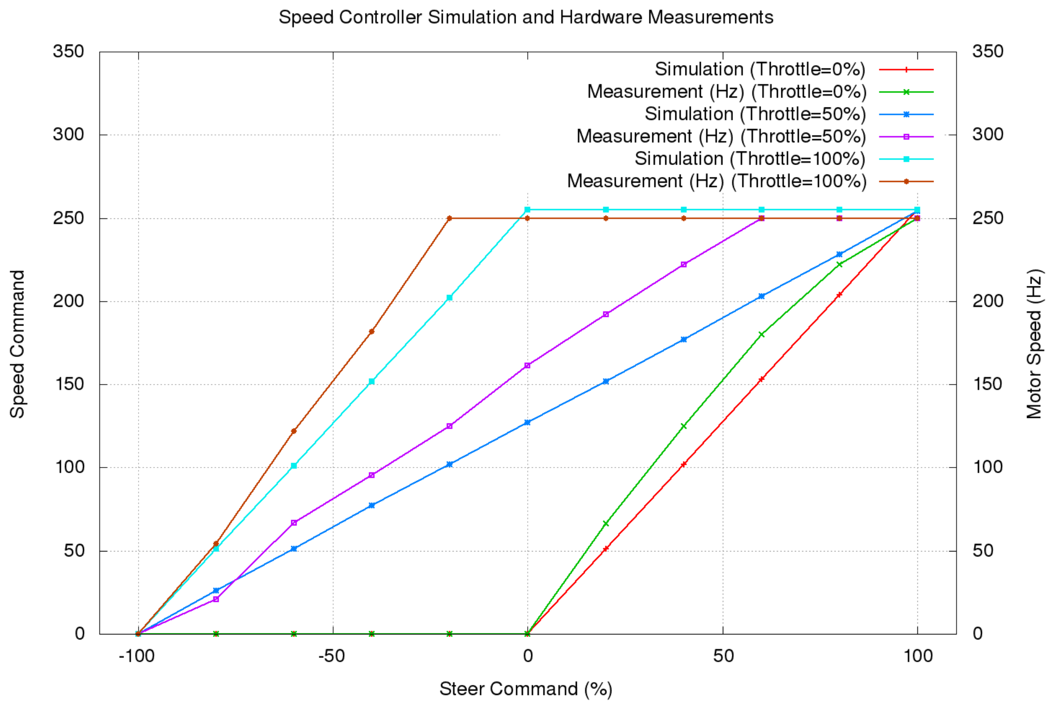


Figure 28: Comparison of Simulation results and Motor Speed Measurements

8 References

[1] "Wheel Control Theory",

www.robotplatform.com/Classification_of_Robots/wheel_control_theory.html

[2] *"Skid Steering in 4WD EV"*, Gao Shuang Norbert C. Cheung Eric K. W. Cheng Dong Lei Liao Xiaozhong.

[3] "Using PWM Output as a Digital-to-Analog Converted on a TMS320F280x Digital Signal Controller", Texas Instruments Application Report SPRAA88A, September 2008

[4] *"SF2-STARTER-KIT"*, EmCraft, http://www.emcraft.com/products/153

[5] *"CorePWM v4.1 Handbook"*, Actel Corporation, 2010

[6]. E. Verhulst, R.T. Boute, J.M.S. Faria, B.H.C. Sputh, and V. Mezhuyev. Formal Development of a Network-Centric RTOS. Software Engineering for Reliable Embedded Systems. Springer, Amsterdam Netherlands, 2011.

[7]. A Formalised Real-time Concurrent Programming Model for Scalable Parallel Programming" authors Eric Verhulst, Bernhard H.C. Sputh at the Workshop on High-performance and Real-time Embedded Systems(HiRES 2013) January 23, 2013, Berlin, Germany.

http://www.altreonic.com/content/altreonic-hires2013-workshop

A free downloadable version of OpenComRTOS Designer for MS-Windows and an MP simulator is available from the download section on www.altreonic.com

What this booklet is all about

This technical note serves to demonstrate how Altreonic's OpenComRTOS Designer allows for embedded software developers of heterogeneous distributed systems to cross develop and simulate their application on a PC environment and seamlessly transfer their code to the target hardware. To this end a Microsemi SmartFusion2 SOC Evaluation Kit is utilised as the target hardware for a hub motor skid steering based speed controller of an Electric Personal Mobility Device (EPMD).

Third publication in the Gödel Series:

Systems Engineering for Smarties©