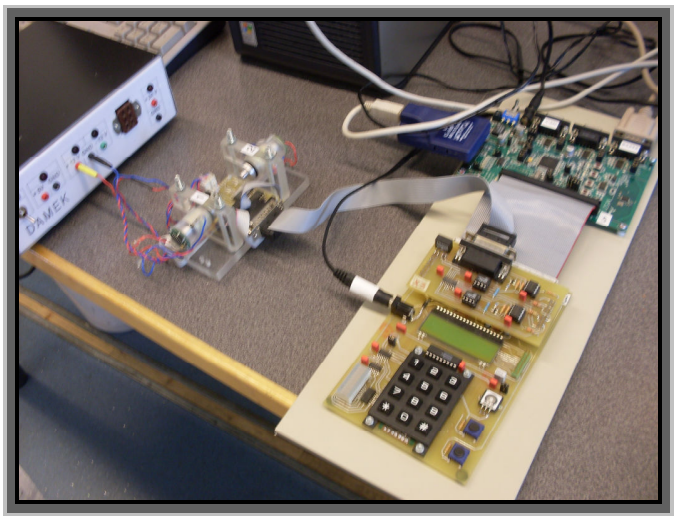
Simulation, code generation and implementation of a real-time system

Jenny Börlin



Master of Science Thesis MMK 2012:15 MDA 415
KTH Industrial Engineering and Management
Machine Design
SE-100 44 STOCKHOLM

KTH Industriell teknik och management

Examensarbete MMK 2012:15 MDA 415

Simulering, kodgenerering och implementation av ett realtidssystem

Jenny Börlin

Godkänt 2012-06-13	Examinator Jan Wikander	Handledare Martin Edin Grimheden
	Uppdragsgivare Skolan för industriell teknik och management	Kontaktperson De-Jiu Chen Tahir Naseer Magnus Persson

Sammanfattning

På institutionen för maskinkonstruktion utformas kurser i inbyggda system, som studenter ska läsa som går inriktningen i mekatronik. I kurserna ingår att praktiskt tillämpa realtidsreglering. Till detta använder man sig av dc-motorer, som styrs via en mikrokontroller och ett tillhörande motorkort som utvecklats på institutionen. För att implementera ett system modelleras systemet i Matlab/Simulink för att sedan implementera realtidsstyrningen i Rubus och CodeWarrior. Vidareutveckling av de tutorials man använder sig av sker ständigt, och tanken är att ta fram en sammanfattande laboration där studenterna får gå från modellering till realtidsimplementering.

Examensarbetet har gått ut på att undersöka möjligheten att använda genererad kod från Simulink tillsammans med en realtidsimplementering och demonstrera detta i en applikation. Vidare har syftet varit att så långt som det hinns med, modellera och implementera applikationen i realtid. En modell av en hiss har tagits fram, som har byggts upp i Simulink. Systemet har reglerats med en PID-regulator och för att styra hissens beteende har Stateflow använts. Med hjälp av Real-Time Workshop har kod för delsystemet Elevator controller sedan genererats och inkluderats i CodeWarrior. Examensarbetet visar att det är möjligt att implementera den genererade koden i samma projekt i CodeWarrior med en realtidsdesign från Rubus.

KTH Industrial Engineering and Management

Master of Science Thesis MMK 2012:15 MDA 415

Simulation, code generation and implementation of a real-time system

Jenny Börlin

Approved 2012-month-day	Examiner Jan Wikander	Supervisor Martin Edin Grimheden
	Commissioner School of Industrial Engineering and Management	Contact person De-Jiu Chen Tahir Naseer Magnus Persson

Abstract

At the School of Industrial Engineering and Management, KTH Stockholm, courses in embedded systems are developed which the students attending the mechatronics aim attend. In the courses, the students get to practice control design in real-time. For this purpose there are DC motors that are controlled by a microcontroller and an interface card that has been developed at the department. When realizing a system, it is modeled in Matlab/Simulink, in order to implement the real-time design in Rubus and CodeWarrior. The tutorials/lab exercises used in the courses are under constant development, and the thought is to develop a summoning tutorial/lab exercise where the students are supposed to realize a system from modeling to real-time implementation.

The aim of the thesis work has been to examine the possibility to use generated code from Simulink together with a real-time implementation, and demonstrating this with an application. Furthermore, the aim has been to model and implement the application as far as time allows. A model of an elevator has been developed and implemented in Simulink. A PID-regulator controls the system, and for the behavior of the elevator Stateflow has been used. Code generation with Real-Time Workshop has been accomplished for the subsystem Elevator controller and included

in CodeWarrior. A real-time design has been implemented in Rubus and tested in CodeWarrior along the code from Simulink. The thesis shows that it is possible to implement the generated code in the same project in CodeWarrior, along with a real-time design made in Rubus.

Table of Contents

1 Introduction

At the School of Industrial Engineering and Management, KTH Stockholm, students attend the course MF2044 Embedded Systems for Mechatronics, II [1]. This course aims (appendix 2) to provide an understanding of the design and implementation of embedded systems. The tutorials/lab exercises used in the course include introduction to the Freescale microcontroller evaluation board, the interface card [2] and software that is used for implementation. One idea by the course administrators is to develop a tutorial that sums up the previous tutorials in that the students are given the opportunity to use all the gained knowledge. The aim of the thesis is to investigate the possibility to use generated code from Simulink and the code from the real-time design in the same project. Furthermore, a real-time design will be suggested and the application that is developed is to be implemented as far as time allows. The challenge is to come up with a real-time suggestion for the application that sufficiently illustrates the concept of real-time design.

Most of the tutorials/lab exercises that follow with the course that preceded the current course, “Embedded Systems for Mechatronics”, have been gone through in order to accomplish the task. The code developed from these has been used as the base from which to work from. Both the application and the real-time design have been developed in a manner so that it has potential for further development.

2 A theoretical framework on learning

Learning how to simulate and implement a real-time system requires that students take in a lot of information about the concept and the tools involved. This is not only dependent on an individual's cognition and approach to learning, but also the way the information is given. This is why the course M2044, Embedded systems for mechatronics II, not only supplies lectures and classroom exercises, but several tutorials/ lab exercises to get the students acquainted with the software and hardware involved.

2.1 Cognition and theory

In the theory of learning, there are three terms defined with regard to the learning process. These are the individuals learning style, learning strategy and study strategy. An individual's learning style is the way he or she prefers to absorb, elaborate and store knowledge. Learning strategy refers to the approach to learning, the intention and motives behind learning the specific subject. Study strategy is a person's choice of method to improve his or her learning [3].

The student's success depends on his or her approach to these concepts. Motivation, intellectual capacity, and in what degree he or she is aware of his/her learning style and study strategy. A lack of this meta-cognition will result in a study strategy which may not be the most effective method for a specific subject, since he or she automatically will make use of his/her favorite strategy.

When talking about learning style, one distinguishes between surface learning and deep learning. The learning style of a person that lack motivation, or doesn't understand the relevance of the information, tend to be shallow. The student learns what is necessary to pass the exams. An inspired person on the other hand, searches for an understanding of the subject which leads to a deeper learning process. The knowledge retention in this case, i.e., the person's knowledge that remains, will be larger than in the case of surface learning [3].

2.2 Problem-based learning

Problem based learning is a way of designing a course and the way it is held. It emerged during the late 1960 and 1980 among health-related educations [4], even if one might ask oneself if not this method has been in use before in one way or the other. The basis of this method is that it starts out with a specific problem that the students are to tackle. There are procedures in how this

is accomplished that may differ in details depending on the teachers view of how the method should be approached. The most common procedure is “The seven jumps” [5].

1. Understand all terms
2. Define the problem
3. Analyze the problem (Brainstorm: activate prior knowledge, discuss)
4. Synthesize (Arrange ideas)
5. Define learning objectives
6. Self-study
7. Report back

Going through the course “Embedded Systems for Mechatronics II”, one can more or less recognize these steps. The difference compared to e.g. education in history, is that the problem that is to be solved is more or less bound to its solution. Meaning that, there are limited ways of accomplishing the goal since the tools and other means in a technological context are fixed. For example, there is one way to mathematically model a system, and a limited way in how the implementing of the system can be done, whereas in history education, one can come up with different scenarios based on the same facts and findings.

Understanding the concepts of real-time design is emphasized mainly during the lectures. Based on personal experience, it may be advantageous to repeat certain concepts when the students are to tackle problems in practice. This results in a kind of recursive method with regard to the seven jumps.

It is often desirable in education to make use of the computer to the greatest extent, one reason being economics. For that reason there have been several software popping up to represent hardware, thus these are not needed in real life. When it comes to the real-time design, it is necessary to present the design in some way. This is, with regard to the learning situation, preferably done in a physical manner that gives efficient feedback for cognition. This is done effectively with the motor and interface-card and the student can easily “connect the dots”.

A future workshop

The course tutorials/lab exercises are to a high degree instructional. Though using “the seven jumps” is in many cases a good choice, some aspects of the learning are preferably done in a step-by-step manner. These tutorials also use this method, for example in getting acquainted to the software involved. When there is opportunity to use a more problem-based learning, this is done so, as in the case of modeling and implementing a system.

In a final tutorial, the students are supposed to draw closer to a real practice where they identify and solve the problem at hand. This will steer the tutorial to be more like a workshop that allows

for a more free way of learning. A set-up like this proposes a higher degree of problem-based learning; hence it is possible to make good use of this strategy. Not only for the overall assignment, but also for parts of the problems that will be tackled. With this approach, the work will be more structured and thereby easier to grasp.

In courses with problem-based learning, the teacher formulates an assignment with a problem for the students to solve. The assignment can be formulated very loosely or with more specific requirements. If the problem is written without defined limits etc. it is up to the students to do this in order to set a boundary for the problem. This calls for more discussion among the participants in a group, which is beneficial when processing the problem.

3 Development environment

For the tutorial an example is needed that illustrates the implementation in the various tools. The environment for the development has previously been used for tutorials/lab exercises, and is preferably used for Freescale microcontrollers, which is why it is chosen in this case. Matlab/Simulink is a well known tool in the academic world and for similar tasks in the commercial world, and CodeWarrior has been around on the institutions laboratories for several years. For the real-time aspect, Rubus RTE version 2011a is used.

3.1 Hardware

To represent the application model, a motor assembly and an interface-card are used (fig. 1). In order to use this hardware, a microprocessor where the implementation can be stored is needed. The power supply and computer is common tools beside the above said hardware and will not be mentioned further.

Microcontroller MCF5213

The evaluation board MCF5213 from Freescale semiconductor [6] is used as the link between application and software. The communication with the computer is done via the BDM (Background Debugging Mode) Interface.

Interface-card and motor assembly

An interface card has been developed at the department [6] that is being used in the courses. This is connected to the microcontroller on one end and to the motor assembly on the other. The motor assembly consists of two motors that have built in encoders. It is fed with 12V from a power supply. The interface card has a keyboard, display, diode array and two pushbuttons. In order to use the encoders and the keyboard on the interface-card it is necessary to remove the jumper on the evalutation board that is associated to the led's, since these uses the same pins as the LED:s and the switches. The doctorands at the department that develop the course tutorials have developed drivers for the motors and Interface-card with the purpose to run on the microcontroller.

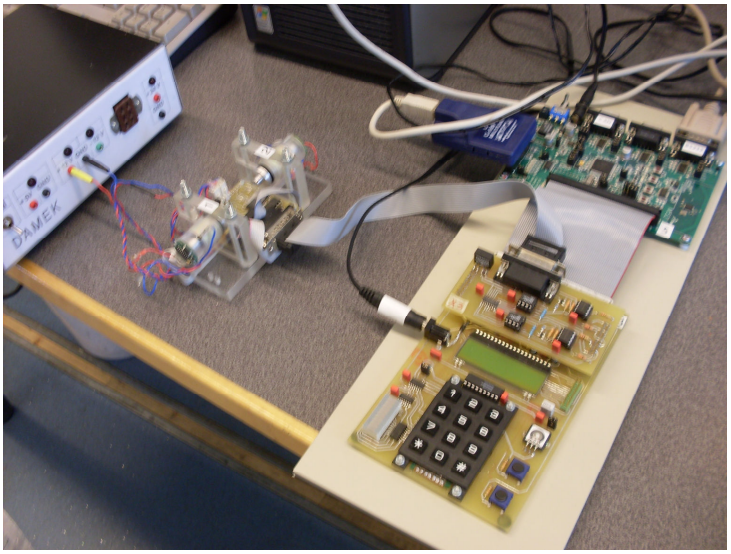


Figure 1: Microcontroller MCF5213, interface card and motor assembly.

3.2 Software

There are mainly three different software systems involved in the work. One software for the simulation of the system; Matlab/Simulink, another for the real-time implementation; Rubus, and the third software is the CodeWarrior development environment, version 5.9.0, for implementation of the programs involved. In the beginning while getting accustomed to the microcontroller, Hyperterminal version 5.1 was used.

The system is modeled in Matlab R2010a, one of the most common tools for mathematical and technical calculations, and Simulink is used for the simulation. Stateflow is a part of Simulink, where event driven logic is built with state charts.

CodeWarrior is a programming development tool where the languages that can be used are C, C++ or Java. The programming is done in C. CodeWarrior comes with an editor and project manager, build-target and debugger. Necessary flash-files are developed in order to make it possible to use larger memory storage.
The real-time environment in use is the real-time operating system that follows with Rubus ICE (Integrated Component Environment) by Arcticus. The environment has three parts: Rubus Designer, Rubus Analyzer and Rubus Inspector. For the purpose in question, the Rubus Designer has been used [7]. In order for the microprocessor to connect to Rubus, the program comes with HAL, a Hardware Adaption Layer [7, page 4-1].

Rubus RTOS

The Rubus real-time operating system is based on the Basic Kernel, Red Kernel, Blue Kernel and Green Kernel (fig. 2). Features that the red, blue and green kernels have in common, i.e. clock, timer, communication and analysis, are handled by Basic Services. The Hardware Adaption Layer (HAL) is provided in order to adapt to different hardware platforms. It is possible to perform run time analysis since Rubus supports this.

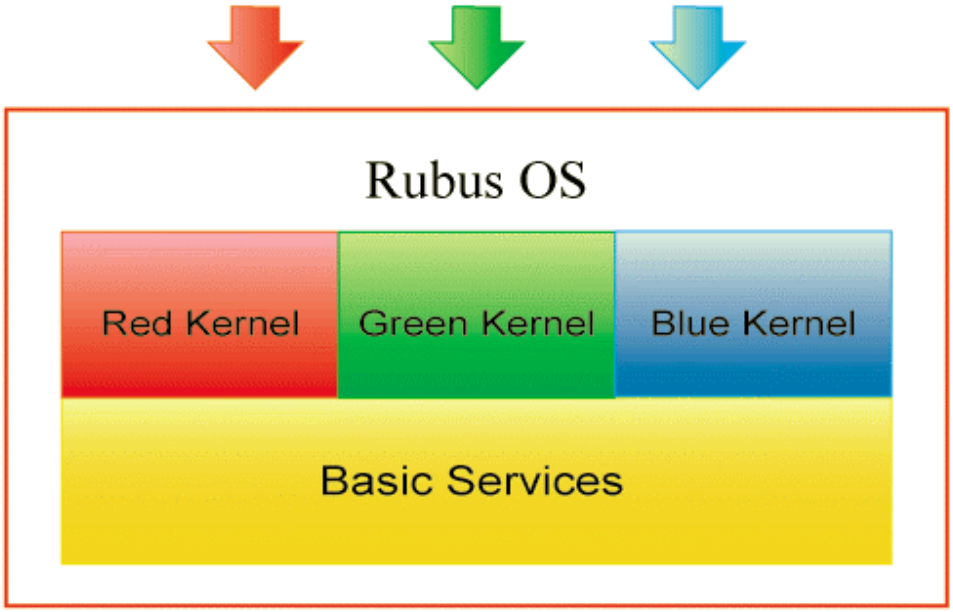


Figure 2: Rubus kernels [7].

The red threads are time-triggered, the blue threads are triggered by interrupts and the green threads are triggered by events. Static (pre-run-time) scheduling is used by the red threads, while the blue and green threads are dynamically scheduled. This means that the red schedules are implemented manually with regard to release time and deadline (fig. 28, 30, 31), while the blue and green schedules take use of the time that is not in use by the red kernel. In turn, the priority that is set for the blue threads decide when to release these.

Passing messages between threads of different kernels are done by Basic Queue, Basic Memory Pool or Basic Mailbox. If the information to be sent isn't large, this is preferably done in a global variable. For communication between blue threads one uses a Blue Message Queue with the help of blueMsg-functions [7, page 10-8].

In order to support dependability, the projects different parts are stored in memory pre-run-time by Rubus ICE, with static allocation. A feature's information is stored as "attribute" and "control block". The static information, such as thread objects, is attributes, while dynamic information refers to control block, such as run-time information associated to the blue thread.

4 Model and real-time design

A system is selected and represented as a simple model, with the thought of possible further development, both mechanically and with regard to functionality when implementing the behavior. In the same manner, the real-time design for the system is expandable in that it is possible to implement future features. In order to demonstrate a real-time behavior, the design contains time-dependent properties which are likely for the system in question.

4.1 Elevator model

The application that is to function as the modeled system in the workshop is an elevator. The elevator's velocity is determined by the user, pushing the keyboard on the interface-card. The elevator is a simple continuous model with a thought in mind that it can be developed further. Top floor is reached when the position value is at 1000 rad.

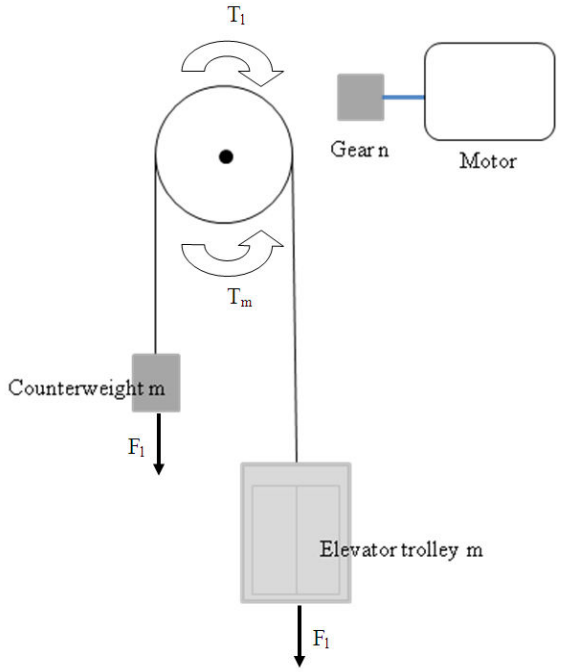


Figure 3: An elevator model.

Mechanically, the elevator consists of a trolley with an opposing weight on the other side of a pulley (fig. 3). When modeling the system, the weight of the load is assumed to be of no significance compared to the trolley and hence is not accounted for. The elevator trolley and the counterweight are equal in mass and are designated with *m*. The motor shaft is considered stiff. A motor drives the system via a gear "n".

Motor equations give the mathematical model of the system, with the designation of the constants. The inertial load times the angular acceleration equals the torques affecting the motor shaft. This term is balanced by the torque and friction (eqv. 1).

J	rotor inertia [kg◦m^2/s^2]
F_l	force on pulley [N]
T_m	motor torque [Nm/s^2]
T_l	load torque [Nm/s^2]
T	total torque [Nm/s^2]
L	induction [H]
K_m	motor constant [Nm/A]
d	viscous friction [Nm◦s/rad]
m	mass of weight [kg]
R_t	pulley radius [m]
R	rotor resistance [Ohm]
v	velocity [m/sec]
ω	angular velocity [rad/sec]
i	current [A]
u	voltage [V]

Table 1: Model constants and variables

$$J \cdot \omega' = T - d \cdot \omega \qquad \text{(eqv. 1)}$$

The torque T is a total of the load torque and motor torque. The pulley is assumed to have no significant weight, thus the inertia of this is assumed to be zero.

$$T_m = K_m \cdot i \qquad \text{(eqv. 2)}$$

$$T_l = R_t \cdot F_l \qquad \text{(eqv. 3)}$$

$$T = T_m + T_l \qquad \text{(eqv. 4)}$$

The force exerted on the pulley when in motion at speed v is F_l. Since the masses are the same on each side, the term with the gravitation involved is eliminated. With the velocity expressed as angular velocity, the force on the pulley is expressed as the product of the mass, pulley radius and angular velocity.

$$F_l = m \cdot g - m \cdot g + m \cdot \frac{\partial v}{\partial t} \qquad \text{(eqv. 5)}$$

$$v = \omega \cdot R_t \qquad \text{(eqv. 6)}$$

This gives the load torque

$$T_l = R_t^{\,2} \cdot m \cdot \frac{\partial \omega}{\partial t} \qquad \text{(eqv. 7)}$$

The product of inertial load and angular velocity give

$$J \cdot \omega' = K_m \cdot i + R_t^{\,2} \cdot m \cdot \omega' - d \cdot \omega \qquad \text{(eqv. 8)}$$

The resulting system of equations is

$$\frac{\partial i}{\partial t} = -\frac{R}{L} \cdot i - \frac{1}{L} \cdot K_e \cdot \omega + \frac{1}{L} \cdot u \qquad \text{(eqv. 9)}$$

$$\frac{\partial \omega}{\partial t} = \frac{1}{(J - R_t^{\,2} \cdot m)} \cdot K_m \cdot i - \frac{1}{(J - R_t^{\,2} \cdot m)} \cdot d \cdot \omega \qquad \text{(eqv. 10)}$$

Representation in state space

$$A = \begin{pmatrix} -\frac{R}{L} & -\frac{K_e}{L} \\ \frac{K_m}{(J - R_t^2 \cdot m)} & -\frac{d}{(J - R_t^2 \cdot m)} \end{pmatrix} \quad B = \begin{pmatrix} 0 & \frac{1}{L} \end{pmatrix}' \quad C = \begin{pmatrix} 1 & 0 \end{pmatrix} \quad D = 0$$

This system is modeled and simulated, including a behavioral control of the elevator, see chapter 5. The state diagram in figure 4 shows the desired behavior. The elevator will of course go up and down, and the function of the behavioral control will be to manage this through a state called runMotor, which is a function used in previous tutorials. When the elevator has reached ground or top floor, the elevator will stop. In order to know when this occurs, the position needs to be monitored, and for this the time is needed to calculate the speed of the trolley. On the whole, a set-up for the behavior looks like table 2.

Function	Description	Dependencies
Start	Enables elevator	Read button pushes
runMotor	Run the elevator up or down	Stop running up if position < top floor Stop running down if position > ground floor
Position	Updates the position	-
Stop	Stop elevator	-

Table 2: Functions for the elevator application.

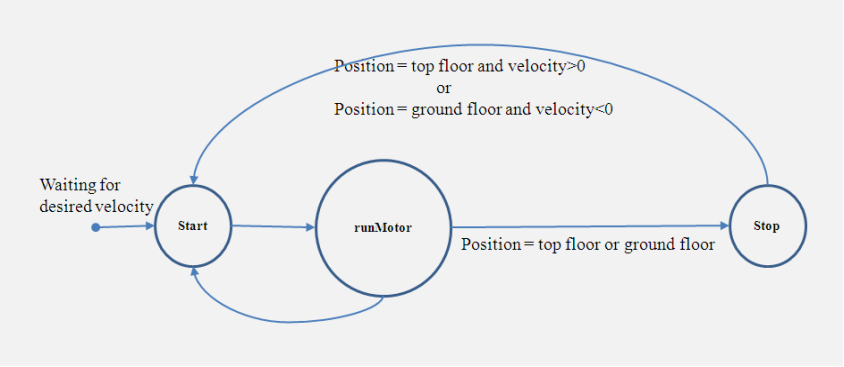


Figure 4: Behavior of the elevator.

Input to the behavioral control is the speed that the user chooses, and is called “button”, and the time, in order to update the position. The output, “velocity”, is the reference signal to the motor and its regulator.

Input	Output
button	position
time	velocity

Table 3: Input and output for the elevator application.

The user can choose between ten different speeds as stated in the switch-case code below. The buttons on the interface card correspond to the parameter “ch” in the switch parenthesis.

```
switch(ch) {
                    case '0': speed = -124; break;
                    case '1': speed = -100; break;
                    case '2': speed = -75; break;
                    case '3': speed = -50; break;
                    case '4': speed = -25; break;
                    case '5': speed = 25; break;
                    case '6': speed = 50; break;
                    case '7': speed = 75; break;
                    case '8': speed = 100; break;
                    case '9': speed = 124; break;
                    case '*': redSetSchedule(&redScheduleA);
                    break;
                    case '#': redSetSchedule(&redScheduleB);
                    break;
                }
```

4.2 Real-time design

This section describes a suggestion for a real-time design, where the schedules indicate that they are active by lightning different LED:s sending the elevator up and down and stop when reaching ground floor again. Three schedules are thought of to be realized in the real-time environment as RedSchedules. The functions readKeyboard and runMotor are interrupt triggered blue threads.

Schedule A

This schedule is the default and is active when a session starts. The user can run the elevator at ten different velocities by pushing the buttons, while the real-time design keeps account on the position. Diode number 1 indicates that A is active. The period time is 100 ms.

Schedule B

Schedule B is thought of as a demo, showing the elevator preprogrammed to run up and down. When pushing button "#" on the interface-cards keyboard, schedule B is activated.
This schedule runs the elevator upwards at speed 50, and when reaching the top floor the elevator goes down to ground floor at speed -50. The position is written on the display of the interface-card when the trolley has reached top or ground floor. Diode number 2 indicates that B is active. Since the top floor is at height 1000, schedule B is set for a period time of 2 ⊙ 20 000 ms, i.e. 40 000 ms, in order to let schedule B execute all the way up and back down.

Schedule C

When the elevator has reached top or ground floor, Schedule C takes over and indicates that the elevator has reached a floor. If the top floor has been reached, diode number 8 is lit, while when reached ground floor, diode number 3 is lit. The period time is 100 ms.

A scenario

Suppose a scenario where the session starts with the trolley at bottom (fig. 5). Initially, the elevator is at the ground floor. The session starts, and diode number 1 is lit (*i*).
The button "#" is pushed shortly after start and diode number 2 is lit. The speed is set to 50 (*ii*).
When the elevator has reached top floor, the speed will change to -50 (*iii.*).
Finally, the elevator has reached ground floor and the speed is set to zero. Diode number 3 is lit (*iv.*).

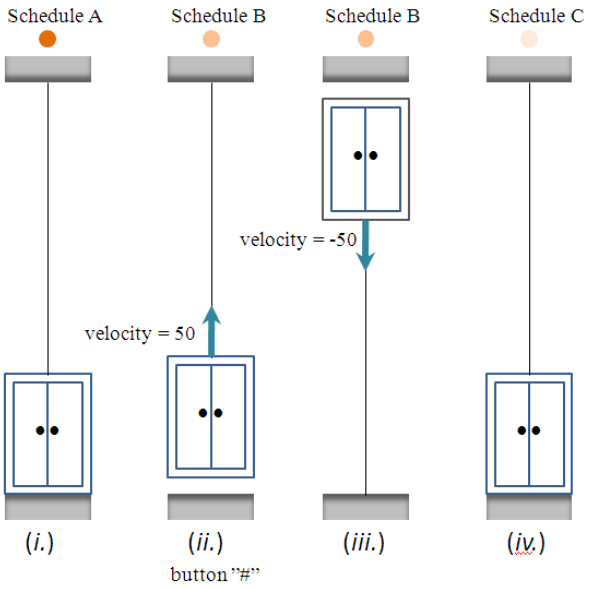


Figure 5: Elevator scenario.

i. diode number 1
ii. button "#", diode number 2, speed = 50
iii. speed = -50
iv. speed = 0, diode number 3

For the timing aspect of the scenario, see figure 6. At start, schedule A is active with the period time 100 ms. Diode number 1 is lit and the thread released. At time 80 ms the button "#" is pushed. Schedule B, which has the period time 40 000 ms, starts to run and diode number 2 is lit (*ii*).

When time has reached 20 070 ms, the speed is set to -50. Schedule B has run to the end of the period and has reached ground floor when the time is 40 080. Then the speed is set to zero, schedule C takes over and diode number 3 is lit (*iv*.).

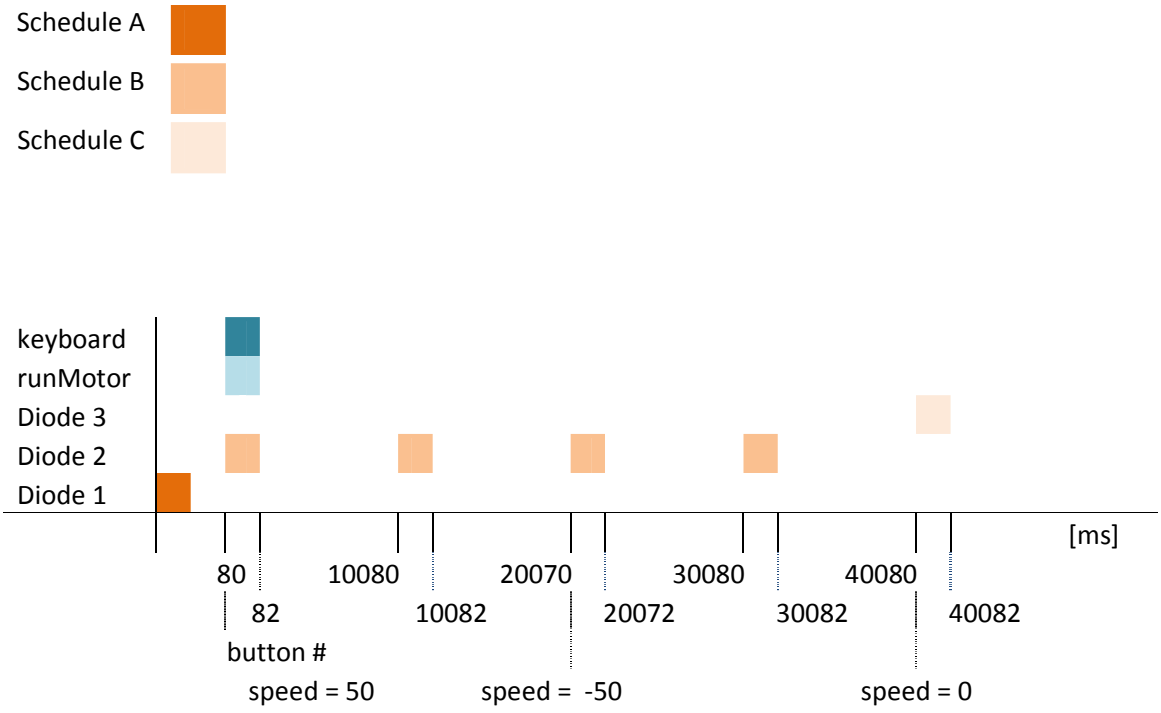


Figure 6: Time schedule, roughly presenting the scenario.

5 Simulink model and code generation

The Matlab code for the elevator can be seen in Appendix A.Based on the previously shown mathematical model, the elevator is built in Simulink. This is then regulated via a PID-block and a Stateflow chart is added for the behavior control. The code generation is done with Real-Time Workshop, which generates the files that are to be used in the CodeWarrior project.

5.1 Simulink model

Implementing the system (eqv 9 and 10, page 20) in Simulink with the voltage and current as input, and the angular velocity as output, gives the model shown in the fig below (fig. 7):

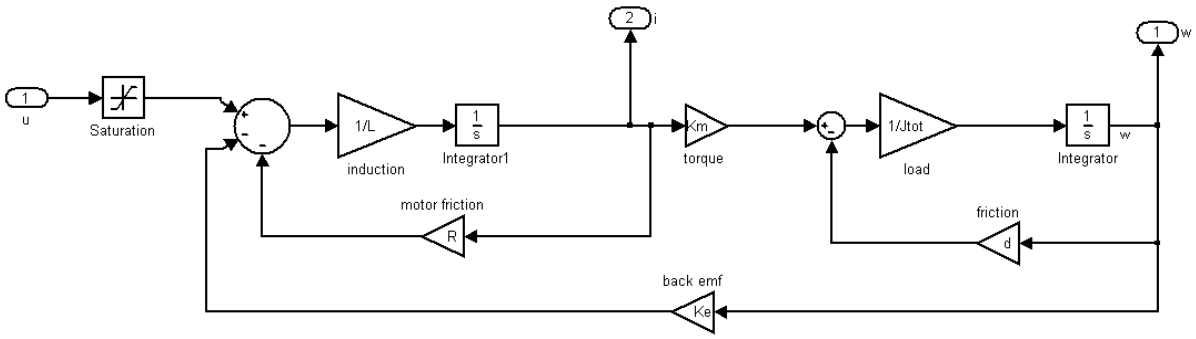


Figure 7: Model from equation 9 and 10.

Reducing the system by ignoring the current *i*, results in the representation in figure 8. The desired output is the angular velocity ω, while the input is the voltage *u.*

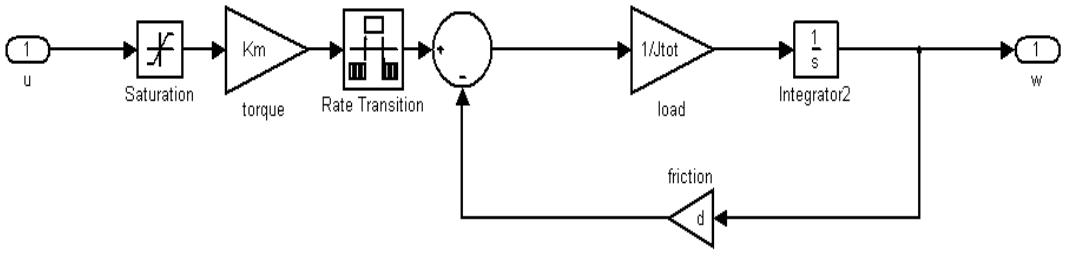


Figure 8: Reduced model.

The model is then represented as the subsystem "Motor" (fig. 9), and a PID-controller is added.

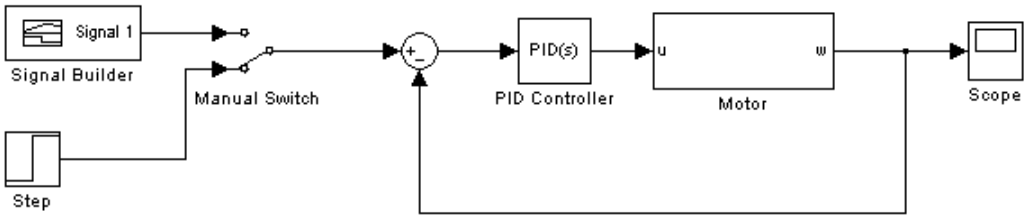


Figure 9: Closed system, with Signal Builder representing button pushes.

The values in the PID-controller are produced experimentally with optimization based tuning (fig. 10). A control design is made, setting the maximum overshoot to 5% and dragging the Bode-curve (fig. 11) manually downwards to lessen the amplification. Updating the block parameters in the PID- controller gives the desired regulation of the system, with a 0.3 s settling time.

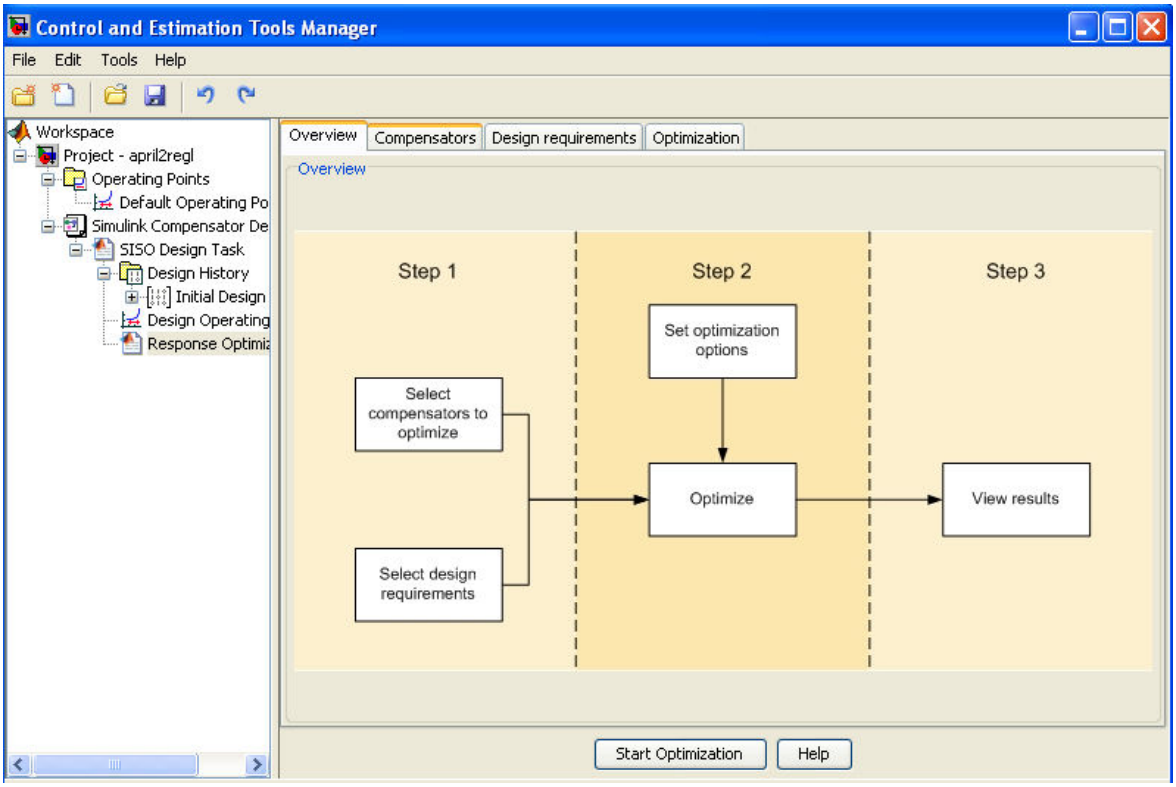


Figure 10: The Control and Estimation Tools Manager window.

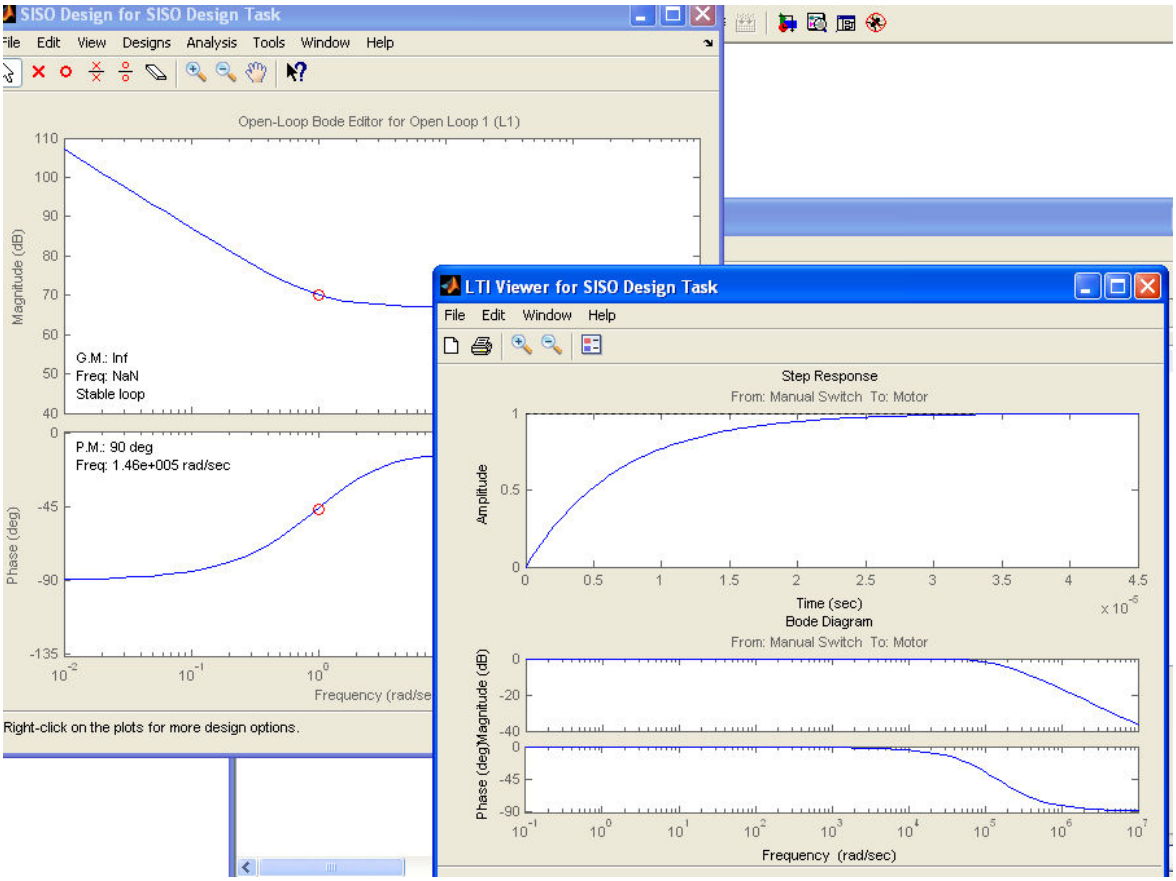


Figure 11: Bode graph and step response.

Next, the motor subsystem is connected to the subsystem “Elevator controller”, which contains a stateflow chart and the PID-controller. The Elevator controller has the input “button” and “utsgn”, where “button” represents the velocity and “utsgn” gives the feedback and closes the system (fig 12). The outputs from the elevator controller is “position”, ControlOut” and “time” and can be viewed with the scope. Position is, naturally, the position of the elevator at any time.

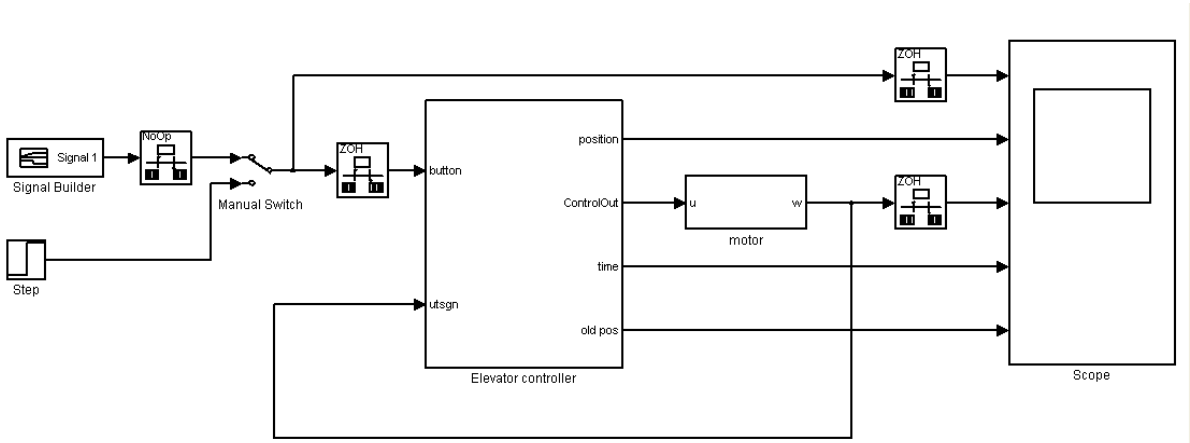


Figure 12: Simulink system.

The PID-controller gets the input from the stateflow chart. This input is the velocity, corresponding to the input voltage to the motor (fig. 13).

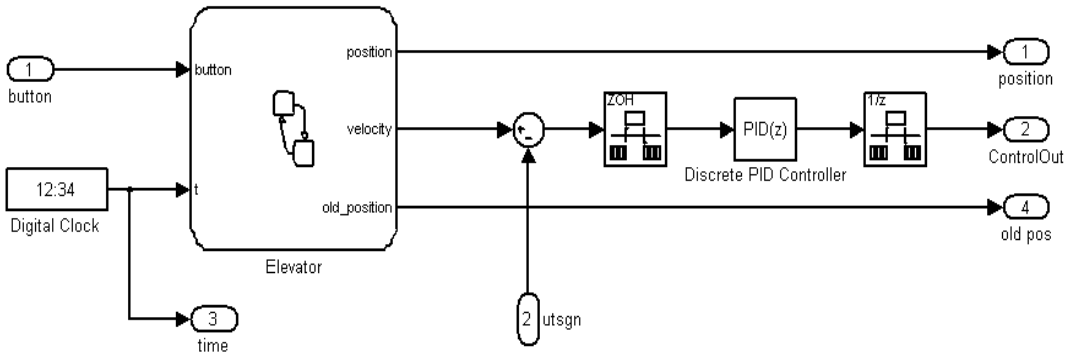


Figure 13: Subsystem Elevator controller.

A signal builder (fig. 14) is added to represent different speeds as if pushed by buttons. The stateflow chart calculates the position, and manages the elevator and stops it if the elevator has reached ground or top floor.

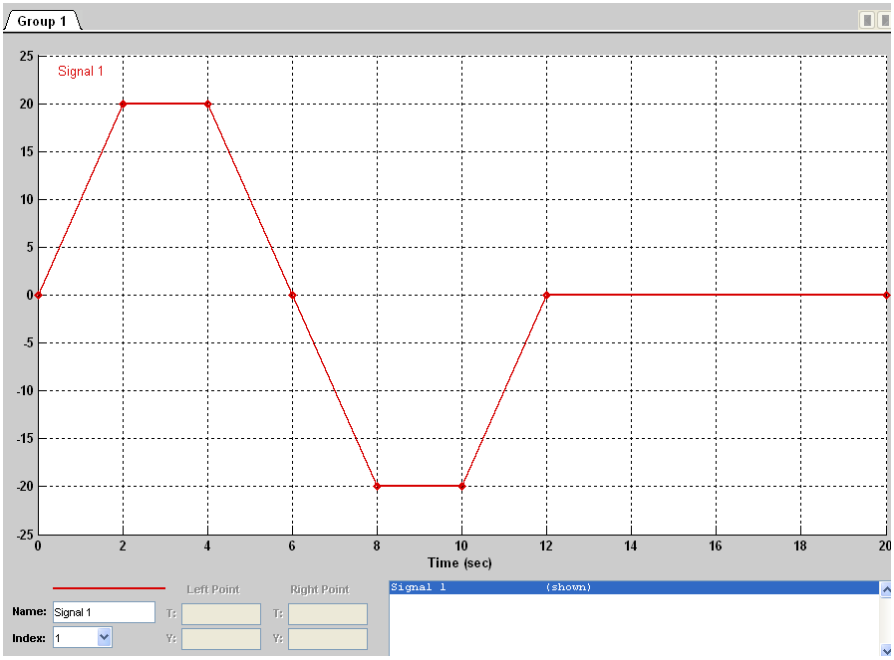


Figure 14: Signal builder.

5.2 Stateflow

The behavior of the elevator is controlled by a stateflow chart named Elevator (fig. 15). Input to the chart comes from a digital clock, and the Signal builder block corresponding to button pushes. These inputs are named “button” and “t”. Output is “position” and “velocity”. Four states are used to make sure that the elevator does not violate the limits of ground and top floor.

The stateflow chart state “Elevator” contains the states ReadKeyboard, ReadKeyboard2, RunMotor and Stop. When starting a session, the default state Stop is entered first, passing over to either ReadKeyboard or Keyboard2. The output variable “velocity” gets the current value on the desired speed (button), which comes from a Signal Builder block.

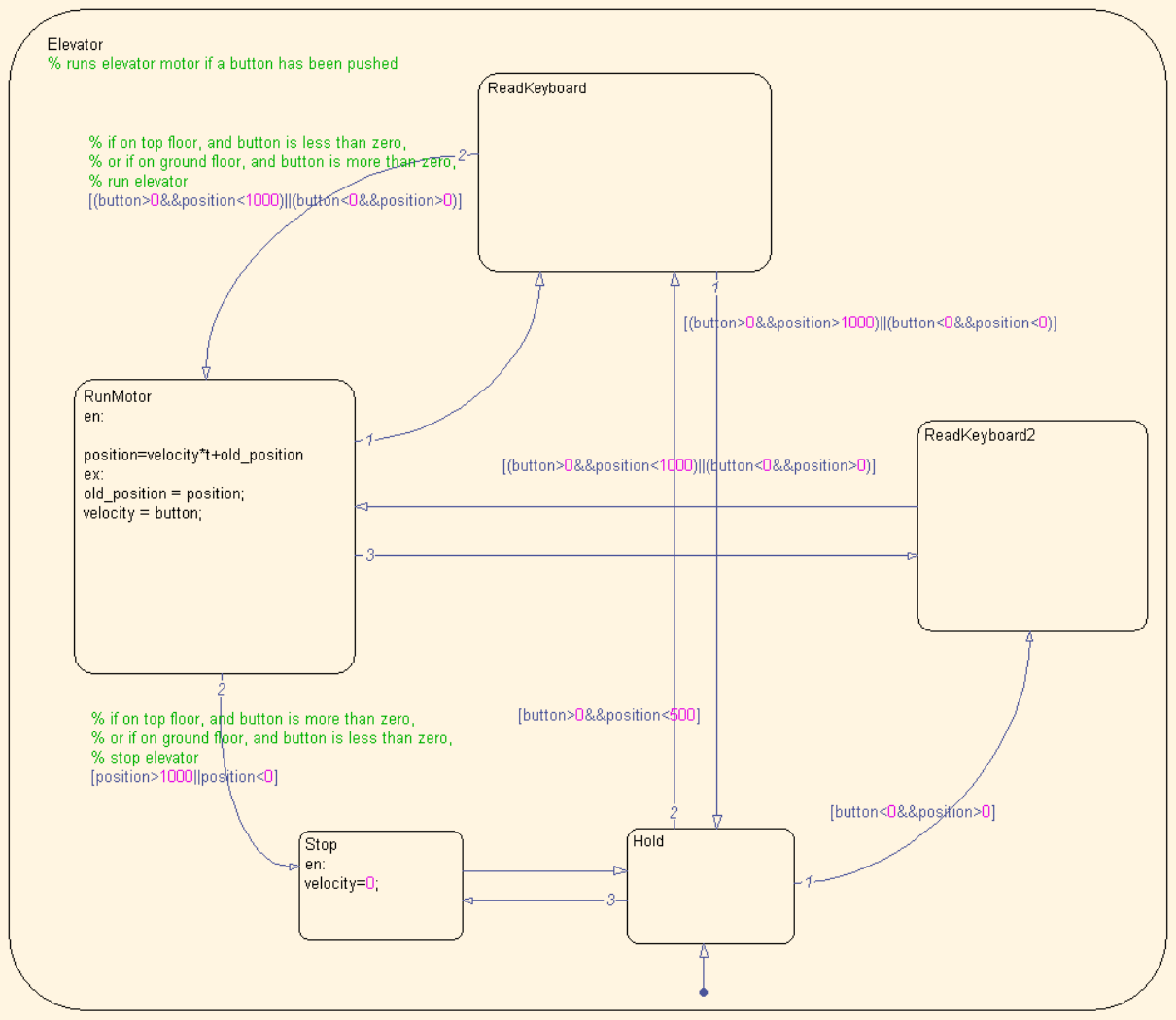


Figure 15: Stateflow chart Elevator.

The ReadKeyboard state passes over to RunMotor, on condition that the velocity will not cause the position to be more than one thousand or less than zero. When entering RunMotor, the position is updated, calculating this using the simulated time, velocity and the previous position. Upon exiting the RunMotor state, the variable “old_position” is set in order to be used when calculating the position the next time.

The passing between RunMotor and ReadKeyboard/ReadKeyboard2 continues as long as the position has not reached ground or top floor. The states ReadKeyboard and ReadKeyboard2 are different in that RunMotor passes over to ReadKeyboard if the desired speed is positive and to ReadKeyboard2 if the speed is negative.

Whenever the position has reached zero or one thousand, the RunMotor state will pass over the execution to the state Stop. This will set the velocity to zero, and the chart will not reach RunMotor until the direction of the speed changes.

Executing the Simulink model with the stateflow chart gives the output shown below in fig 16. The top graph is the trajectory from Signal Builder. The second graph shows the position, which rises as the velocity increases and reaches it's top position. When the speed is negative, the position is updated accordingly, and stays on the zero level since the speed will not be positive again.

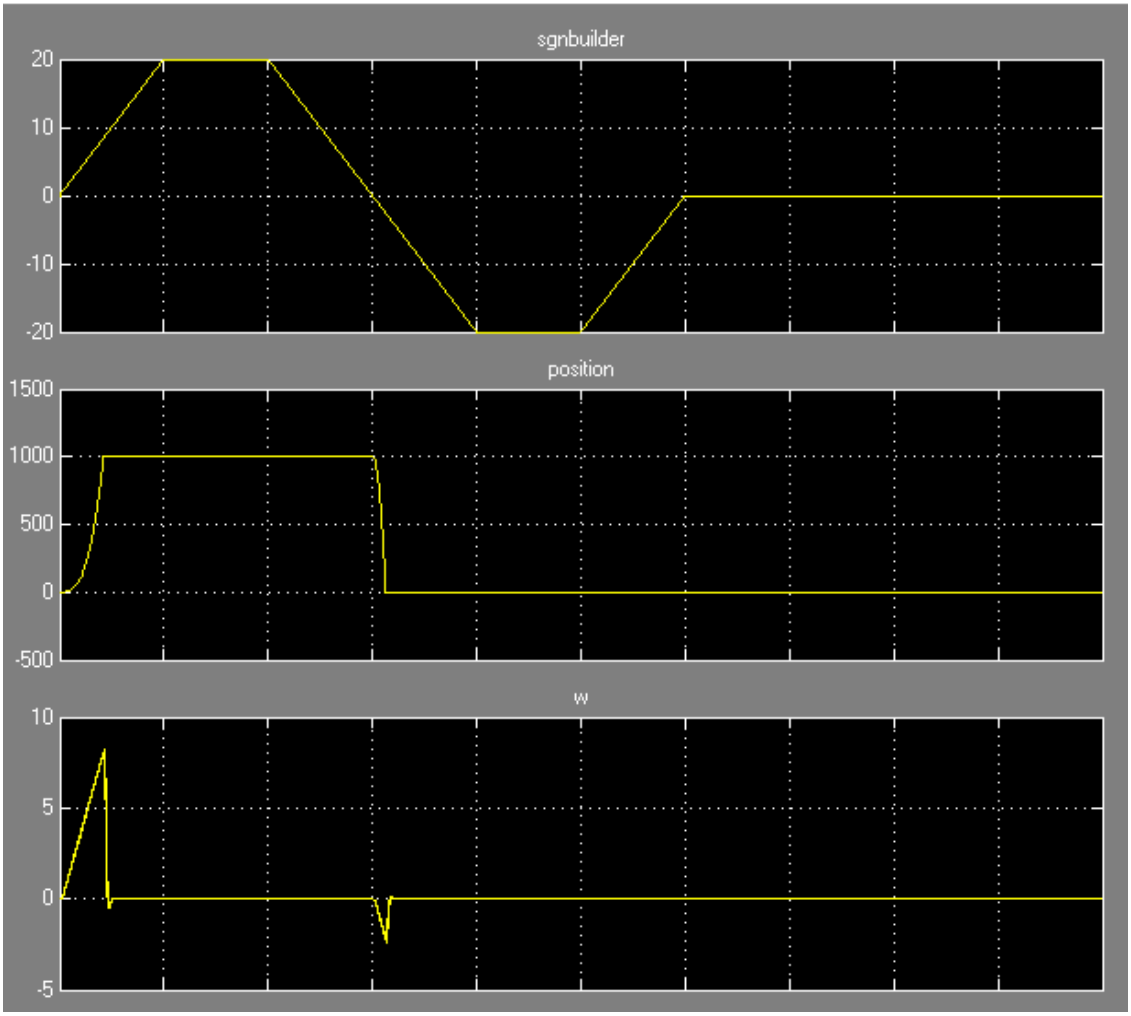


Figure 16: Scope output. Top graph is Signal Builder, the next graph position, and the third shows the angular velocity ω.

5.3 Code generation with Real-Time Workshop

A Simulink model has been implemented, and the next step is to generate the code for the Elevator controller. Code generation ensures that the code is written as effectively as possible, in order to cope with limited memory space and speed of the hardware. The following pages will demonstrate the code generation of model Elevator.mdl and show this included in CodeWarrior.

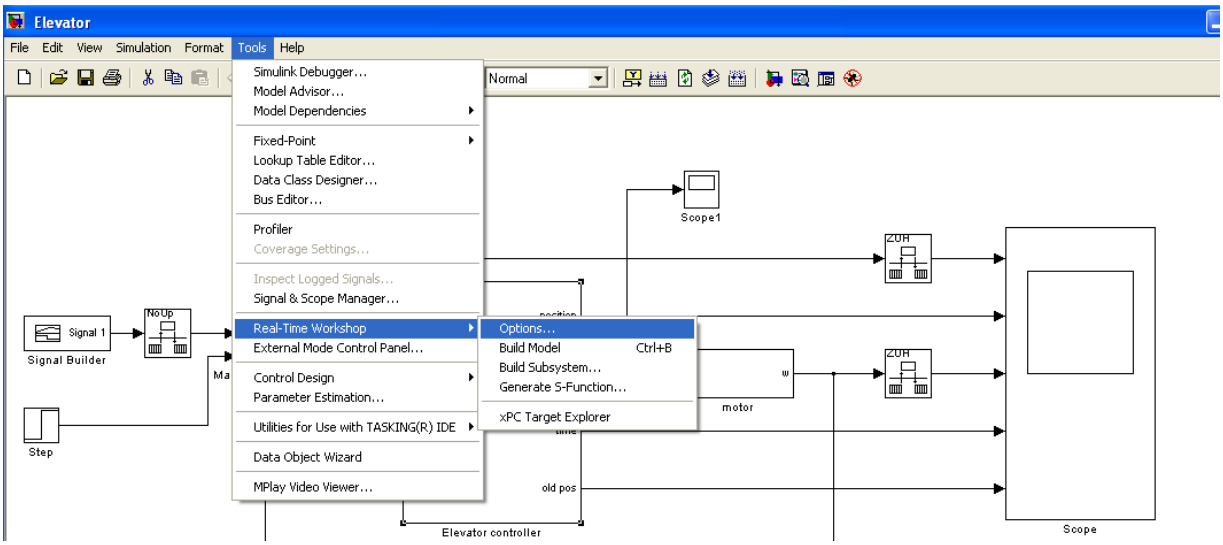


Figure 17: Way to Real-Time Workshop/Options.

The Real-Time Workshop is found under Tools in the file menu (fig. 17). Choosing Options, will open the Configurations Manager window. A system file is browsed, ert.tcl, with the target set for Real-Time Workshop Embedded Coder. The language is set to C and selecting Optimizations off will set the configuration for faster builds (fig. 18).

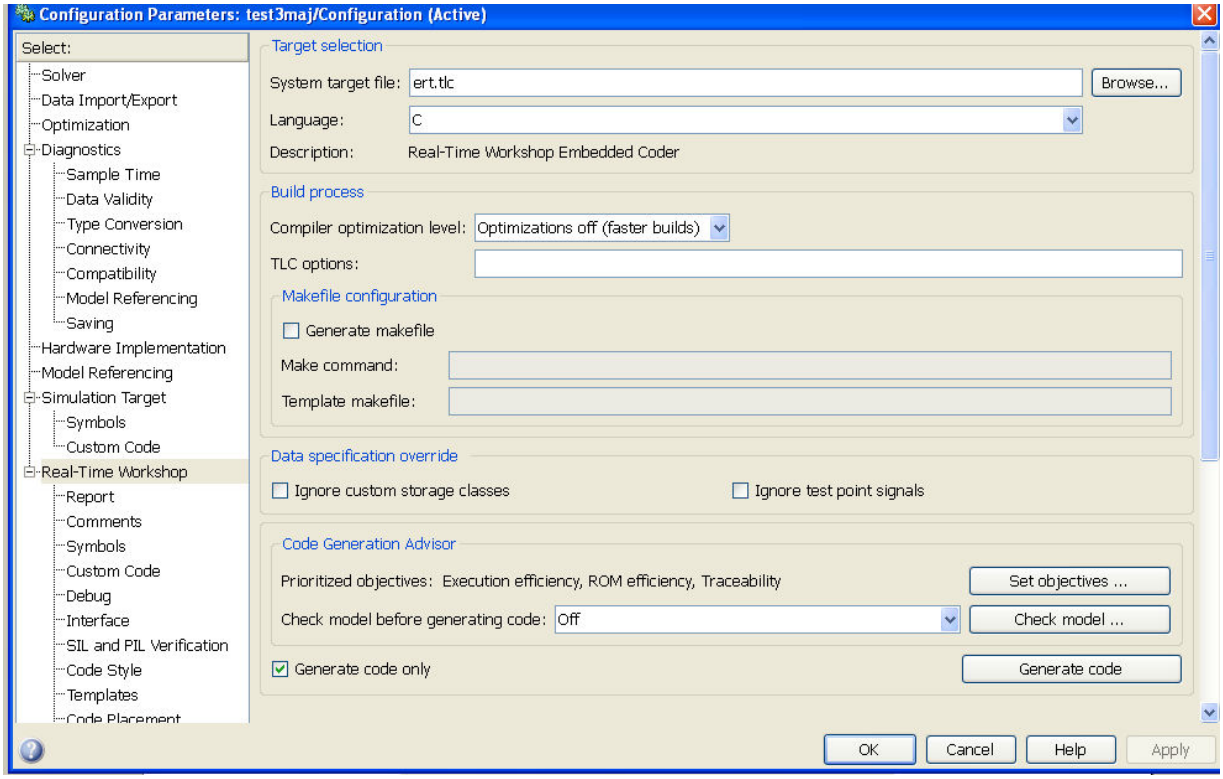


Figure 18: Configurations Manager window.

The option for Generate makefile is unselected and Generate code only is set. When clicking Set objectives, a Code generation Advisor is found and the Execution efficiency, ROM efficiency and Traceability is selected.

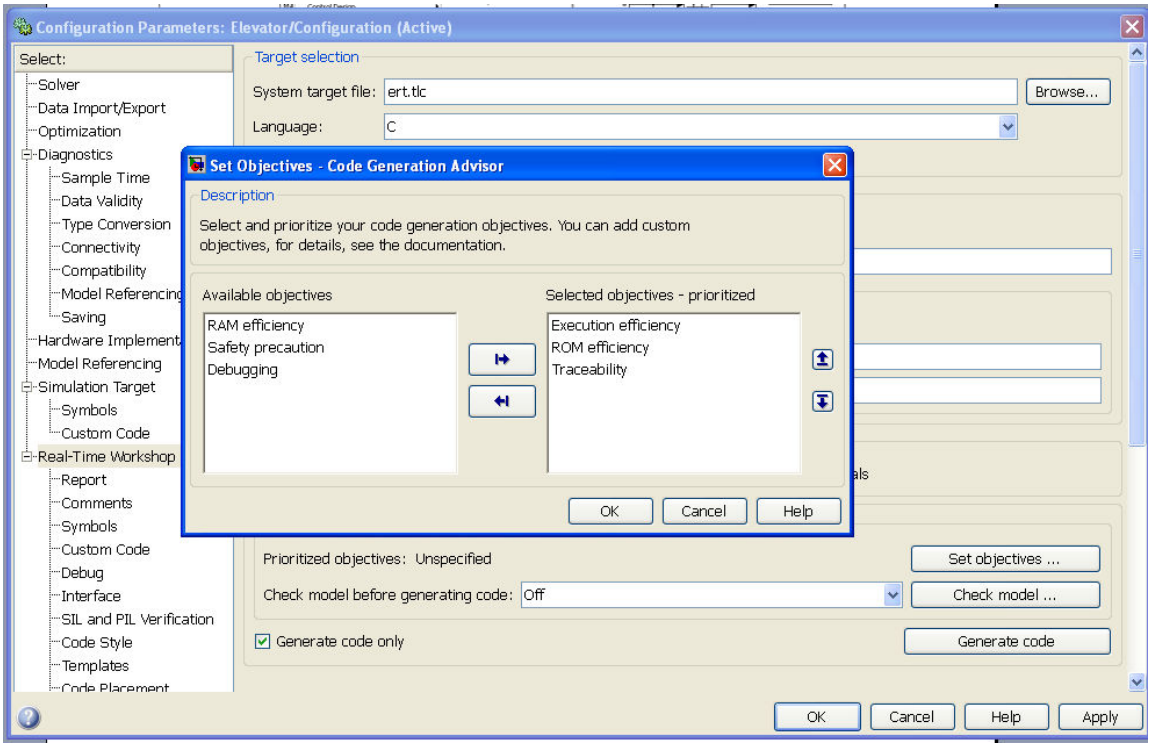


Figure 19: Set objectives window, choosing Execution efficiency, ROM efficiency and Traceability.

Furthermore, the following settings are chosen:

Report	All options are selected
Comments /Overall control	Include comments
/Auto generated comments	Simulink block / Stateflow object comments
Debug /Build process	Verbose build

Table 4: Settings in Configuration Manager.

Interface	
/Software environments	- Target function library: C89/C90 (ANSI) - Utility function generation: Auto - Support: floating point numbers, non-finite numbers, complex numbers, absolute time - Multiword type definitions: System defined
/Code interface	- single output/Update function, Terminate function required - Generate preprocessor conditionals: Use local settings
/Data exchange	Interface: none
Templates	
/Code and /Data	ert_code_template.cgt for both header and source template
/Custom	- File customization template: example_file_process.tlc - Generate an example main program - Target operating system: BareBoardExample
Hardware implementation	
/Embedded hardware	- Device vendor: Generic - Device type: 32-bit Embedded Processor - Byte ordering: Unspecified - Signed integer division rounds to: zero
Optimization	
/Simulation and code generation	- Block reduction - Conditional input branch execution - Implement logic signals as Boolean data - Signal storage reuse - Inline parameters - Application lifespan: Inf

Table 4, continuing: Settings in Configuration Manager

/Code generation	Parameter structure: Nonhierarchical
/Signals	- Enable local block outputs - Reuse block outputs - Inline invariant signals - Eliminate superfluous local variables - Minimize data copies between local and global variables - Loop unrolling threshold: 5 - Maximum stack size: Inherit from target - Use memcpy for vector assignment - Memspy threshold: 64 - Pass reusable subsystem output as: Structure reference
/Data initialization	- Use memset to initialize floats and doubles to 0.0 - Optimize initialization code for model reference
/Integer and fixed-point	- Remove code from floating-point to integer conversions that wraps out-of-range values - Remove code from floating-point to integer conversions with saturation that maps NaN to zero
/Acceleration simulations	Optimizations off

Table 4, continuing: Settings in Configuration Manager.

Now the build is ready to take place that generates the Real-Time Workshop Report. This is done by right-clicking the Elevator controller subsystem, choosing Build Subsystem under Real-Time Workshop.

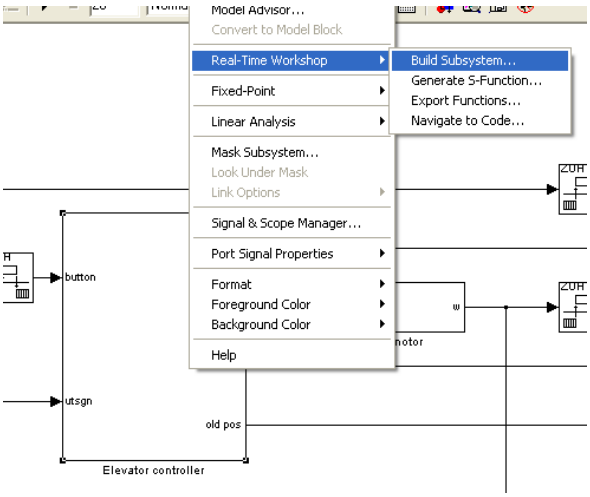


Figure 20: Building the subsystem Elevator controller with Real-Time Workshop.

The report (fig. 21) contains, except for traceability and interface report, header and source-file for the Elevator Controller, definition files and the file ert_main.c which is an example-file to draw from during the integration into CodeWarrior. A folder, Elevator_ert_rtw, containing the files has been generated and can be found in the directory where the Simulink model is placed.

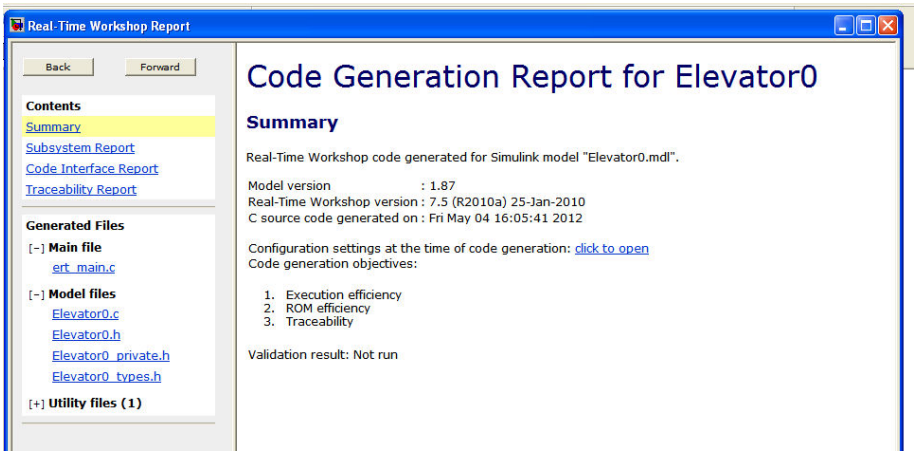


Figure 21: Real-Time Workshop Report.

6 Rubus Designer

With the functions defined and the system modeled in Simulink , the real-time design is to be implemented. In the Rubus Designer a new project is created, clicking File/New – Project (fig. 22 a). A New project window is opened showing the Categories area and the Templates area. A template Project_RubusOS4.1.rubusDesigner is chosen and the project name and the path are set.

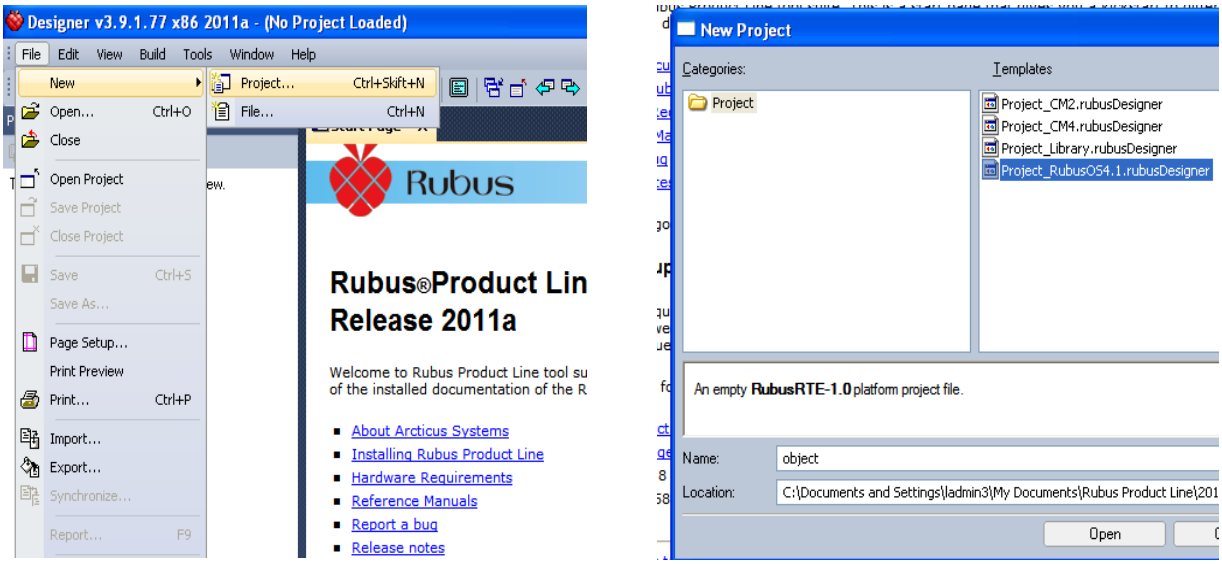


Figure 22 a,b : (a) New Project (b) Template and project name and location setting.

As seen below in figure 23, the project folders are located to the left and the properties window of the project is located to the right.

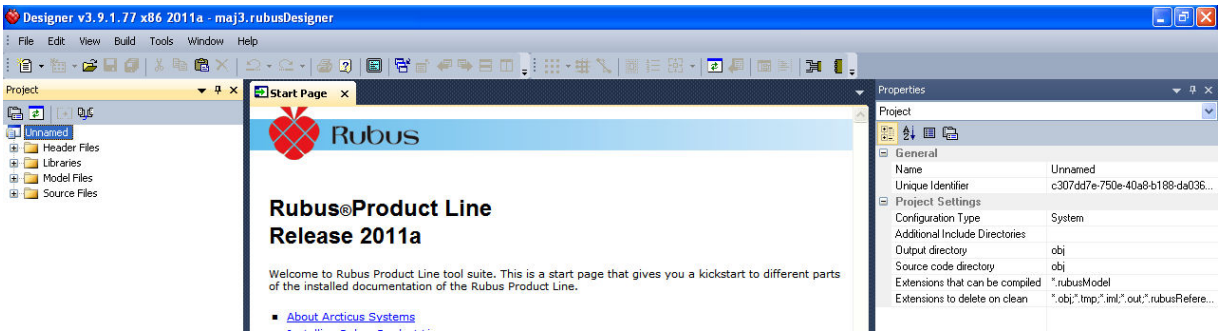


Figure 23: The newly created project.

The folder Libraries contains a reference to a library, rubusOSResources.rubusLibrary, whose path needs to be set in the Properties window to the right (fig 24 b).

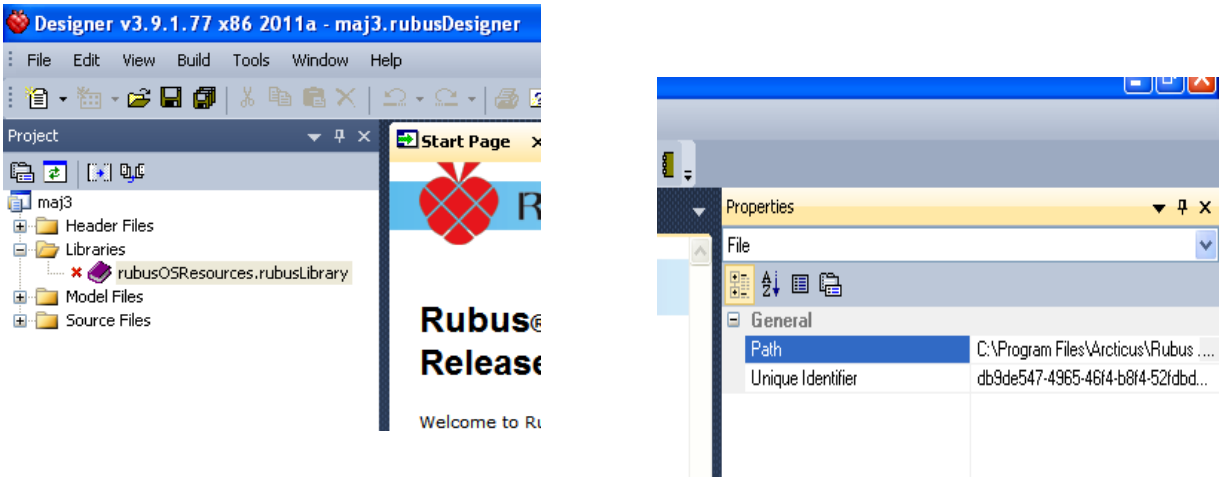


Figure 24 a, b: (a) Rubus library reference (b) Path for Rubus library.

When right-clicking the project name, the Property Manager window appears. Under Tools-XMLLinker, the Linker base object is set to “ex”. Further under XMLLauncher the Input objects is set to ex\ex. This will make the program find the file when the project is built.

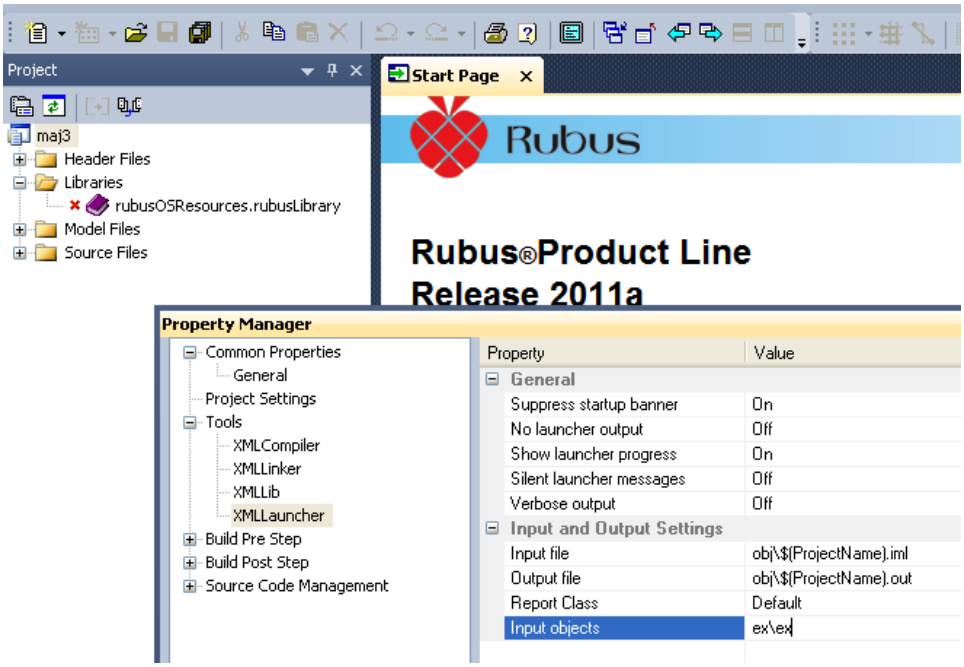


Figure 25: Property Manager window.

Creating a new file for the project will open up an empty workspace where the implementation takes place. In the Categories and Templates area respectively, the category Model Files and the template Model_RubusOS41.rubusModel are chosen. After creating, naming and setting the path of the file, it is included under the projects folder Model Files.

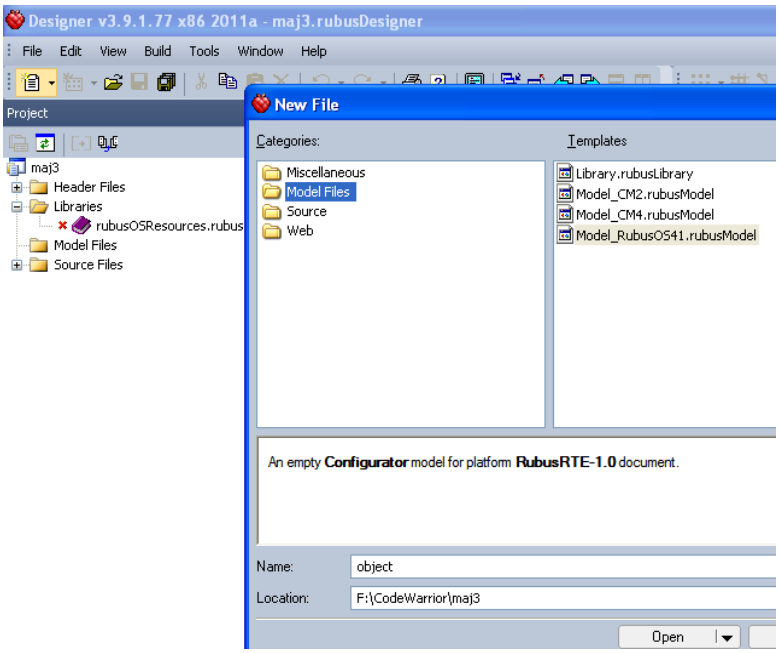


Figure 26: Creating a new file.

Thread implementation

Implementation of systems in Rubus is made with so called Component Models [4]. When working with a project, a file is created (file.RubusModel) that function as a worksheet. The components are found in the Library, whose path is set to the desired one. The components used in this case are the “node” and a “target”. The node represents where in a system that a microprocessor is placed, and the target corresponds to the microprocessor used for the project.

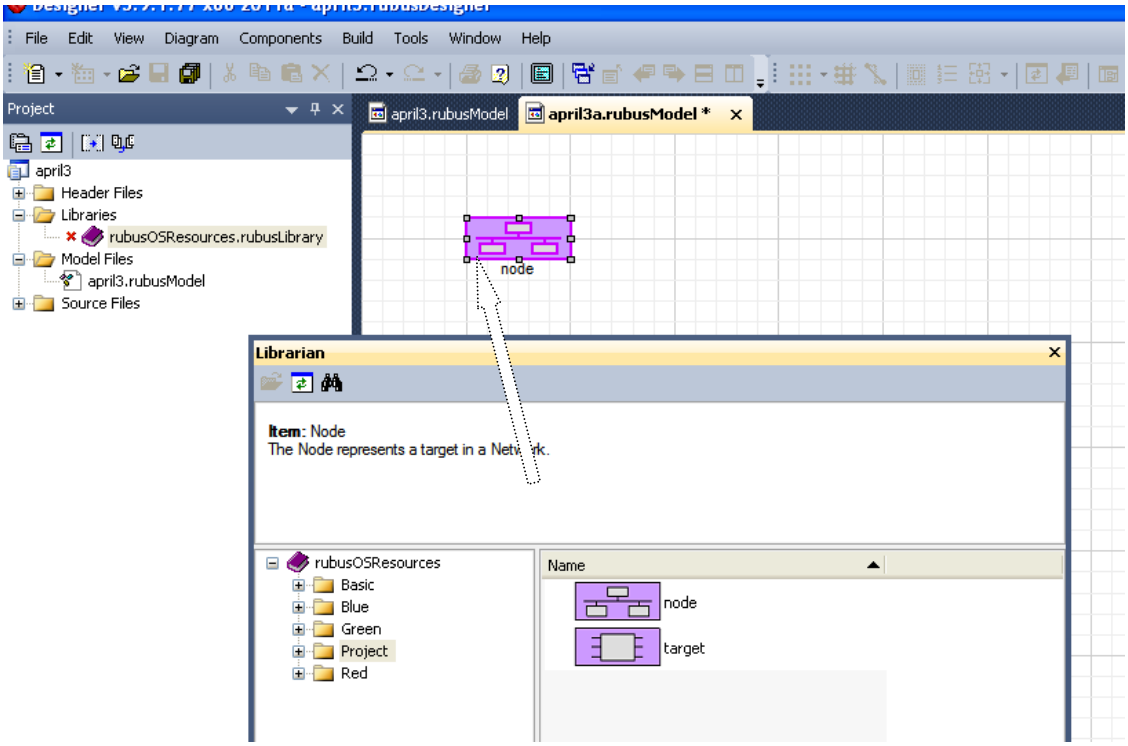


Figure 27: Librarian window. Dragging a node to the worksheet.

Double-clicking the node opens up a worksheet, where the target is placed. This in turn opens up a worksheet when double-clicked, where the threads are implemented. Implementing the system functions in real-time can be done in various ways, by choosing the threads differently. However, to correspond the threads with the simulation, the threads implemented in this case are the red threads “runMotor” and “write”, and the blue threads and the blue message queue “desiredSpeed”.

Firstly, red schedules are drawn into the worksheet. Double–clicking the schedules show the RedRelease Time block, which when double-clicked, shows a window with the threads that are released in the schedule (fig. 29). In the figure the thread “redstart” is added to the schedule with the release time 10 ms. The schedule “redScheduleStart” needs to be implemented for the program to know where to start, otherwise the error “No Red default Schedule declared” will show when trying to build the project.

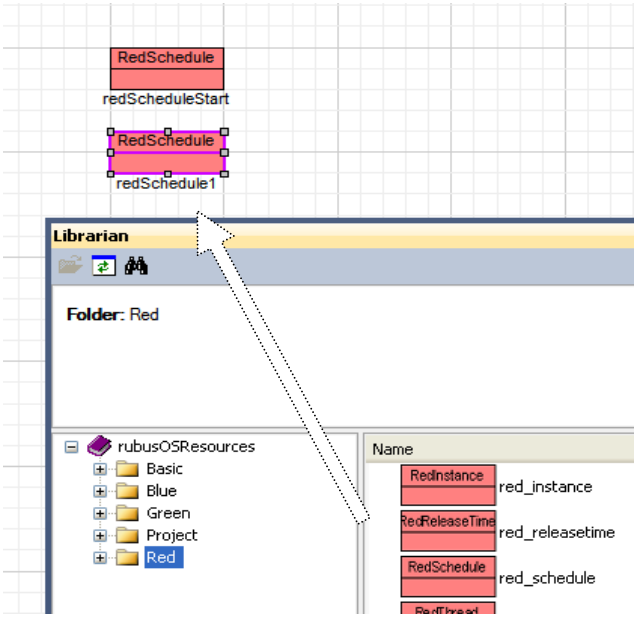


Figure 28: Librarian window. Dragging a RedScedule to the worksheet.

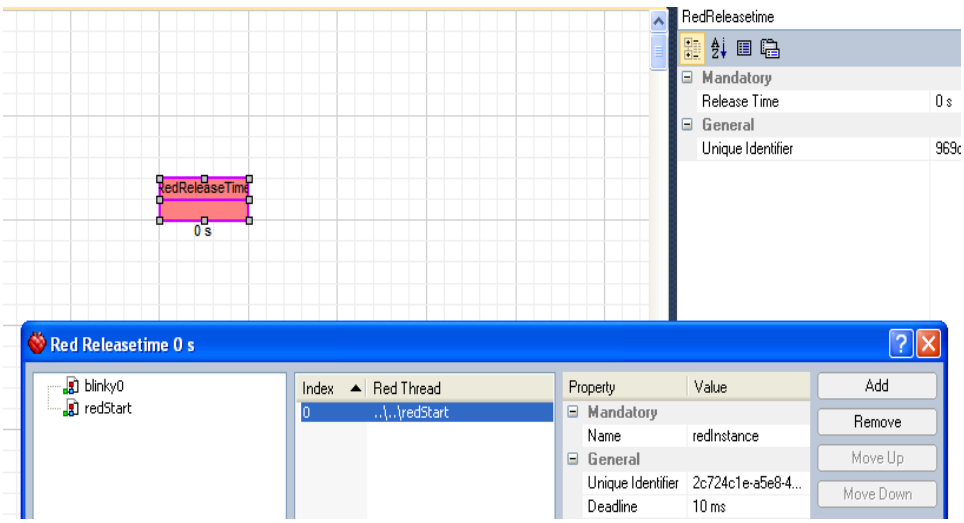


Figure 29: Red Releasetime for redSchedule with added thread redstart.

The blue threads (and green threads if applicable) are added the same way and their properties are set at the right side of the window. In the property field of “Entry”, the function names are declared for blue threads that are to be used when implementing the code (fig. 30). The whole setup for the threads is shown in table 3 on page 38.

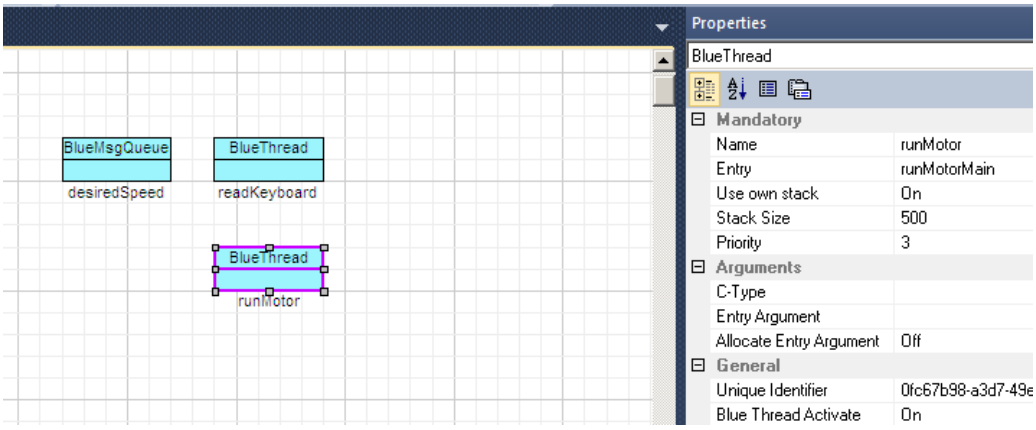


Figure 30: Blue thread runMotor.

Figure 31: Complete set-up for the real-time design.

The schedule redScheduleB contains five RedReleaseTime blocks. Each block releases the red threads blinky1 and write with the deadlines 2 ms and 4ms respectively after the release time. The third RedReleaseTime block, releases the thread just before the elevator reaches the top floor, in order to set the speed to -50 (fig. 32). This is a quite deterministic way of implementing the wanted behavior and is mentioned in the discussion-chapter. The last RedReleaseTime block, releases the threads just before the period time of the schedule runs out, in order to affirm that the position is zero when the elevator has reached ground floor. The two following figures show the release times with the red thread write and reschedule.

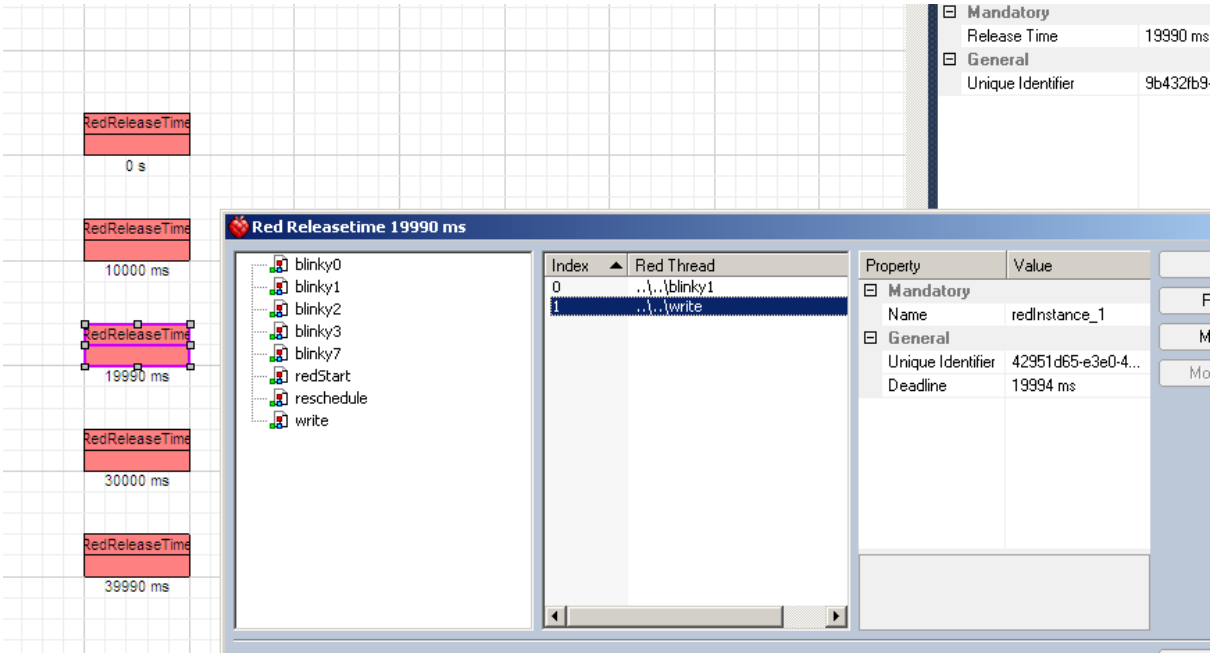


Figure 32: Third release time block with added threads.

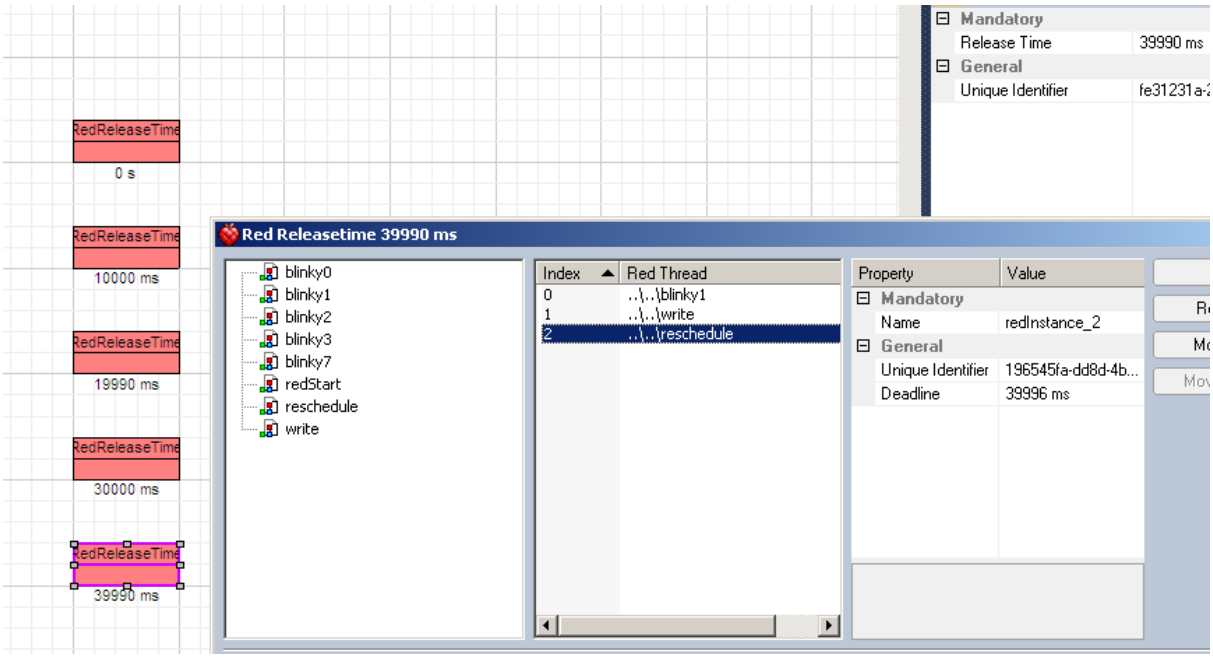


Figure 33: Fifth and last release time block with added threads.

Building the project successfully will result in the output shown in the figure below (fig. 34). The files are generated to the specified path and will later be included in the CodeWarrior project.

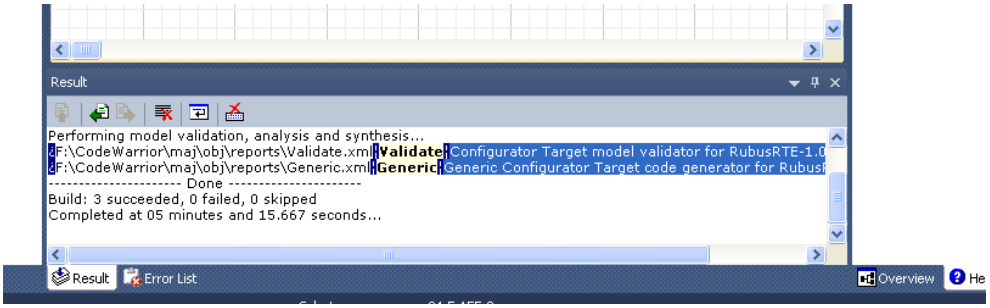


Figure 34: Building output.

Schedule	Period (ms)	Release time (ms)	Deadline (ms)
redScheduleStart	10	0	
		Thread:	
		redStart	10
redScheduleA	100	0	
		Thread:	
		blinky0	2
		write	4
redScheduleB	40000	0	
		Thread:	
		blinky1	2
		write	4
		10000	
		blinky1	10002
		write	10004
		19990	
		blinky1	19992
		write	19994
		30000	
		blinky1	30002
		write	30004
		39990	
		blinky1	39992
		write	39994
		reschedule	39996
redScheduleC	100	0	
		Thread:	
		write	2
		blinky7	4*
		blinky2	4*

*Table 5: red schedules with release time and threads. *Either led number 7 or 2.*

Now the threads have been modeled. The Inline window is found under “Components” in the Designer windows menu bar, and ‘ #include “red.h” ‘ is written (fig. 35). This will include the file red.h in the generated files that are added to the project in CodeWarrior.

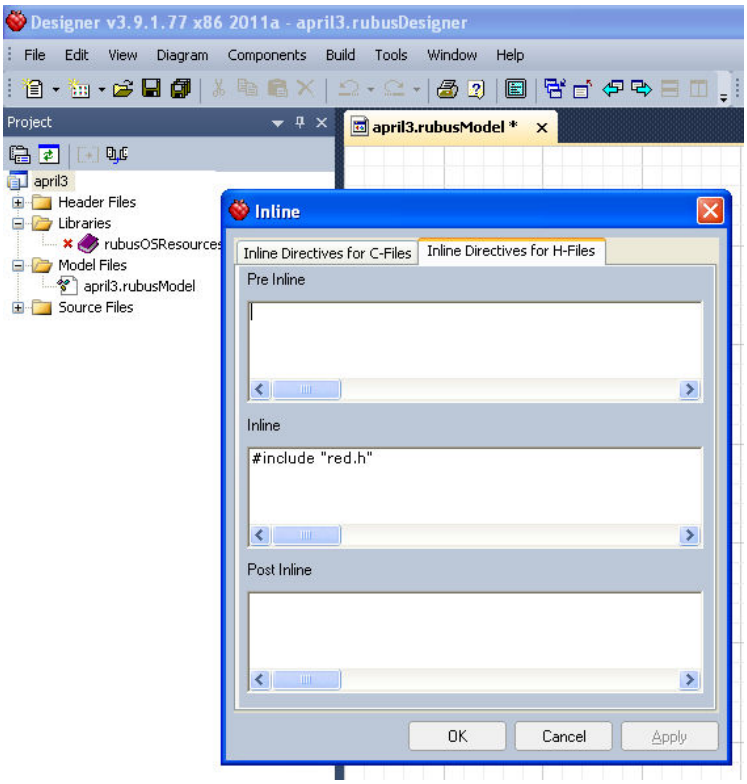


Figure 35: Inline window, Inline Directives for H-Files.

The properties of the project are configured (fig. 36) and set to the right path, in order for CodeWarrior to reach the files that is generated during a build-up. The project is built, and the files necessary for the CodeWarrior project are generated. The implementation of the code with the CodeWarrior Development Studio is described in the following chapter.

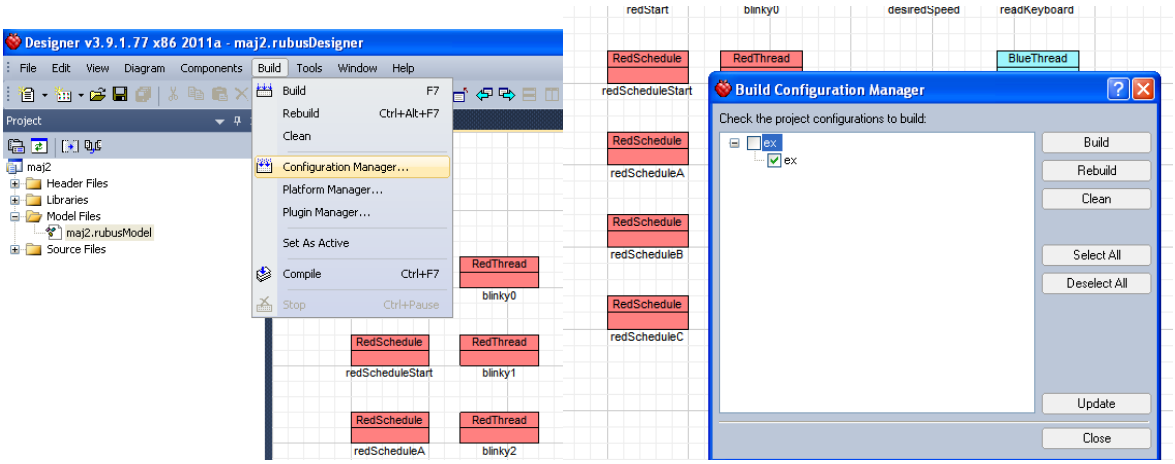


Figure 36 a,b: (a) Way to Build/Configuration Manager (b) Project configuration with object ex chosen.

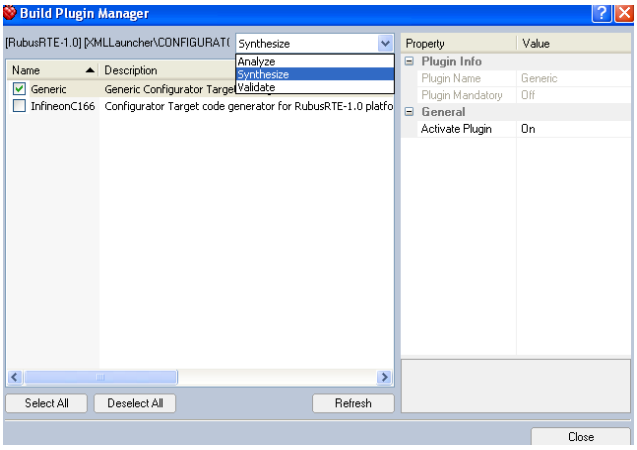


Figure 37: Build Plugin Manager window. Making sure the Generic is chosen and the property Activate Plugin is on.

7 CodeWarrior

The functions of the threads are implemented in CodeWarrior Development Studio. In the courses the students take use of a predefined project with regards to Rubus, and this is used also in this case. The project file can be found in the zip-file “RubusCWTemplate2011c”. Here are also all the necessary files found that are associated with Rubus, and the files associated with the microcontroller.

When a project is created, the folders and files are shown to the left in the window (appendix). For the realization of the schedules and threads, the files blue.c, red.c, red.h, error.c och main.c need to be created and included in the Application folder. The real-time design is imported to the programmer environment via the generated files, including these in the folder RubusGeneratedFiles.

Application	Contains the files associated with the threads, an error handling file and a main.c file.
RubusGeneratedFiles	Files that are generated when building a project in Rubus, are included. These contain features and definitions associated to the threads.
Rubus OS Library	A library assembly-file associated to the current version of Rubus.
RTE	Real-time operating files with services and communication handling.
Hal-Common	Hardware adaption layer for the microprocessor and communication, including CAN and UART.
Hal-Target	More adaption layer files. Microprocessor mcf5213 specific files.
Project Settings	Debugger- and linker command files.

Table 6: Project folders

Functions

As in the simulation, the functions wanted are readKeyboard, runMotor and Stop. Stop is represented by the redScheduleC. Beside these functions, a write function is suggested, that will write the position to the display. The code contained in the files can be found in Appendix C.

File	Function	Function description
main.c	main()	Initializes interface-card functions, hal and Rubus
	rubusIdleMain()	Keeps Rubus going as long as a thread is active, if not the function sends the system into an infinite loop
	timer()	Starts and reset a timer, and calculates the position of the elevator
blue.c	readKeyboardMain()	Reads button pushes and sends the requested speed to the blue threads
	runMotor()	Runs the elevator
red.c	writeMain()	Writes the position of the elevator to the display
red.h	-	Declares red prototypes
errors.c	blueError()	Writes an error code and sends the system into an infinite loop if an error occurs
	greenError()	
	redError()	

Table 7: Functions in the CodeWarrior project.

The generated code from Simulink is included by simply copying the header and source files to a folder in the catalog of the CodeWarrior project. The source-file for the subsystem, Elevator.c, is then included in the project and placed in the Applications folder. As an example-file, there is the file ert_main.c which contain the function rt_OneStep() and lines that include Elevator_private.h,

rtwtypes.h and Elevator.h. These are also added into the main.c file of the CodeWarrior project. The Elevator is initialized in function main() with Elevator_initialize();.

The timer function is implemented in the main.c, using the Basic Periodic Timer [4, part 1, page 6-2]. In order to use this, one has to make sure that the file bs_basic.h is included. Two variables, position and old_position is defined. When the position has been updated and the timer register is reset, the timer is started for the new speed.

```
#include <basic/bs_basic.h>

int position, old_position;

void timer(void)
{
     uint_t basicTimerPriority;
     extern int speed;
     extern int position, old_position;

     basicTimerPriority = 7;

     // position
     position = old_position + speed*MCF_PIT0_PCNTR
     MCF_PIT0_PCNTR = 0;

     // restart timer
     halBsTimerStart();
}
```

In file blue.c the motor speed is set in function runMotorMain(), and here the function timer() needs to be called on, in order to set the timer for the new speed if the speed has been changed. The variable char_t ch is replaced and made global in order for red.c to recognize it, for reasons that will be mentioned soon.

```
while (1) {

          if (blueMsgTimedReceive(&desiredSpeed,
          (uint8_t *)&speed, sizeof(int),&tp1) == R_OK)
               motor1_set_speed(speed);

          else {
```

```
                motor1_set_speed(0);

                blueSleep(&tp2);
           }
                // start timer for current speed
                if(speed != previous_speed)
                {
                     timer();
                }

           previous_speed = speed;
}//while
```

In red.h where the prototypes are declared, the following lines are added:

```
typedef struct {
int scheduleA;
int scheduleB;
int scheduleC;
} scheduleModeType;

void rescheduleMain (scheduleModeType *state);
void writeMain(int *args);
void timer(void);
void blinky1Main (int *state);
void blinky2Main (int *state);
void blinky3Main (int *state);
void blinky8Main (int *state);
```

File red.c is modified with the declarations for the new threads in function redStartMain(), a function rescheduleMain() and the function writeMain() which set the led number 3 if the elevator is on ground floor, or led number 8 if on top floor. It also changes the direction of the speed in order to let redScheduleB run the elevator down, and lastly it prints out the position. The variable char_t ch that was made global is used when setting the variables scheduleA, scheduleB and scheduleC. These variables are added with the thought that it might be advantageous in further development to recognize which schedule is running.

```
blinky1Args = 0;
blinky2Args = 0;
blinky3Args = 0;
```

```
blinky8Args = 0;
writeArgs = 0;
rescheduleArgs.scheduleB = 0;
rescheduleArgs.scheduleC = 0;
rescheduleArgs.scheduleA = 1;


void rescheduleMain (scheduleModeType *state)
{

     extern int position;
     extern char_t ch;
     timer();

     //switch to redScheduleC if position is 0 or 1000
     if (position == 0 || position == 1000) {

          redSetSchedule(&redScheduleC);
          rescheduleArgs.scheduleC = 1;
          rescheduleArgs.scheduleA = 0;
          rescheduleArgs.scheduleB = 0;

     }

     if(ch == '*')
     {
          redSetSchedule(&redScheduleA);
          rescheduleArgs.scheduleA = 1;
          rescheduleArgs.scheduleC = 0;
          rescheduleArgs.scheduleB = 0;
     }

     if(ch == '#')
     {
          redSetSchedule(&redScheduleB);
          rescheduleArgs.scheduleA = 0;
          rescheduleArgs.scheduleC = 0;
          rescheduleArgs.scheduleB = 1;
     }
}
```

```
void writeMain (int *args)
{
     extern int position;

     timer();

     // set led number 3 if on ground floor
     if(position == 0)
     {
          leds_turnON(2);
          motor1_set_speed(0);
     }

     // set led number 8 if on top floor
     else if(position == 1000)
     {
          leds_turnON(7);
          motor1_set_speed(0);

          // if redscheduleB is active, set the speed to -50
          if(rescheduleArgs.scheduleB == 1)
          {
               motor1_set_speed(-50);
          }
     }

     // write position
     printf("position   %i m", position);

}
```

The simplified project contains the same code with the exception that they do not declare redScheduleB and redScheduleC, or blinky1, blinky2, blinky3 or blinky8, or to these associated code. The writeMain-function set the led number 3 when the elevator is at ground or top floor.

8 Results

A didactic analysis of the course/tutorials has been done, recognizing that the so called “seven jumps” are to some degree being used in the course “Embedded Systems for Mechatronics”. Going through the currently used tutorials/lab exercises, one can see there is gain in using “the seven jumps”. However, when the situation calls the step-by-step manner is preferable, for example when learning to use a tool such as Rubus Designer. The opportunity to learn in a problem-based manner is given when modeling and implementing a system.

Repeating certain concepts in the moment of facing a problem in practice would serve the learning situation well. Giving high priority to learn to manage the given tools is advantageous, since the learning situation is very much fixed due to the amount of existing solutions.

An elevator model for a tutorial/lab exercise has been developed and implemented in Simulink. A real-time design has been suggested and a project in Rubus has been created. The schedules and threads are in place, but the implementation of the design needs to be tested.

Furthermore, a project has been created in CodeWarrior, which includes the Rubus real-time design and the code generated for Elevator controller from Simulink. As was the purpose for the thesis, including the Simulink generated code in the same project with code from Rubus has shown to be successful.

9 Discussion

There are certain points to consider regarding the real-time implementation and the overall thesis work. The time has of course been a limiting factor, as it often is, which is why certain parts of the work have been emphasized more than others.

Didactics

Some research has been done in the academic world in the field of problem-based learning associated to technological education. Most have been related to medical education though, and it is hard to find specifically PBL in mechatronics that give concise results referring to workshops and tutorials. There is opportunity to research the field further, since mechatronics is a relatively new area in the world of education.

Implementation

Regarding the implementation, for various reasons it has not been possible to test the functions properly. Testing the complete real-time design has not been possible, since the CodeWarrior program crashes when trying to link. Therefore, a simplified project has been used with only SceduleA, red threads blinky0 and write, and the blue threads with the message queue, when including the code generated from Simulink. However, when trying to run the simplified project, the USB connection is not found.

There are at least one alternative way to call to the function timer(). One can use the implemented red thread and call the function every time this is necessary, which is after every time the call to the function motor1_set_speed(speed) provided that the speed has changed since the last time. Or, what would be preferable is to continuously check the speed, and if the speed has change, call timer().

It has been necessary to comment away certain lines in can_mcf5213_isr.c and hal_xhp.c, since the compiler doesn't recognize the variables comXhpTx and comXhpRx and the handlers for these (appendix 3). Additionally, lines in file mcf5xxx.h have been modified. The char int8 and integer int32 variables have been declared as "signed char" and "long integer" respectively, in order to avoid compiler error due to previously declared variables. With these changes the program compile without problem.

In previous tutorials/exercises, the initiation of the encoders has worked without problems. Lately this initiation has caused trouble since a link error occurs, claiming the mcf5xxx_set_handler is undefined. Hence, it has not been possible to implement a regulator for the speed, and also impossible to get the actual position. There has been a constant concern whether or not more time should be given to solving these kinds of problems, or if more emphasis should be given to the real time design.

Real-time design

Regarding the implementation of the real-time design, a better way of monitoring the position would be preferred, avoiding deterministic coding as with the third release time block. When it comes to the position, there is a risk that the code is not sufficient when executing conditions, for example if(position == 0). In order to execute the code within the if-block, the position would have to be precisely zero, and this might not be the case if the position are assigned discrete values and zero is not among these. For this reason it might be wiser to compare with a lowest value, for example if(position<1), depending on the resolution of the assigned values.

10 Further work

For the time being, the tutorials/lab exercises are bringing up the various tools and skills needed in order to model and implement a real-time design. Some aspects though can be added in a final tutorial/workshop, such as logging and using CAN-communication. The real-time design is also possible to extend, using the component modeling in Rubus.

Elevator application

There are opportunities to expand the elevator model in various ways. A first step would be to add middle floors to the elevator, and invoke more functions associated to additional behavior, adding these in the Stateflow chart. In the current model the weight of the trolley is the same at all times. In a future model, the model could draw closer to reality, taking account of different weights.

Real-time design

Adjusting the real-time design to correspond to the expanded elevator application. Furthermore, there is opportunity to demonstrate the use of semaphores and deadlock.

Implementation

For the sake of the actual position of the elevator, solving the problem associated with the encoder would be desirable. The function timer() needs the time elapsed since a speed has been set in order to get the position. Rubus has predefined functions for timers, which can be used for this purpose.

11 Acknowledgements

The author of this thesis would like to thank the persons involved:

- **Bengt Eriksson**, for suggesting the thesis work and initial advice.
- **Martin Edin Grimheden**, supervisor, for helpful advice on the report.
- **Tahir Naseer**, for the help with small and big problems in Simulink and programming
- **Magnus Persson**, for assisting with the tools involved, overall advice and help in programming.
- all those who have been helpful in various way.

12 References

[1] MF2044 Embedded Systems for Mechatronics, II
http://www.kth.se/student/kurser/kurs/MF2044?l=en

[2] 4F1908 – Embedded Control Systems. Dokumentation för I/O-kort och motorkort, Dan Öhlund MMK Machine Design KTH 2007

[3] "Lärandets hur" Roar C. Pettersen, Studentlitteratur 2010

[4] "Problembaserat lärande" Henry Egedius, Studentlitteratur 1999

[5] "Evaluating the effects of redesigning a problem-based learning environment" Mien Segers, Piet Van den Bossche and Emily Teunissen, 2003. Faculty of Economics and Business Administration, Department of Educational Development and Research, University of Maastricht, the Netherlands

[6] MCF5213 ColdFire Integrated Microcontroller Reference Manual, Freescale Semiconductor 2011

[7] RubusOS RTOS for Dependable Real-Time systems, Part 1, Part 2, Arcticus systems 2009

Appendix A

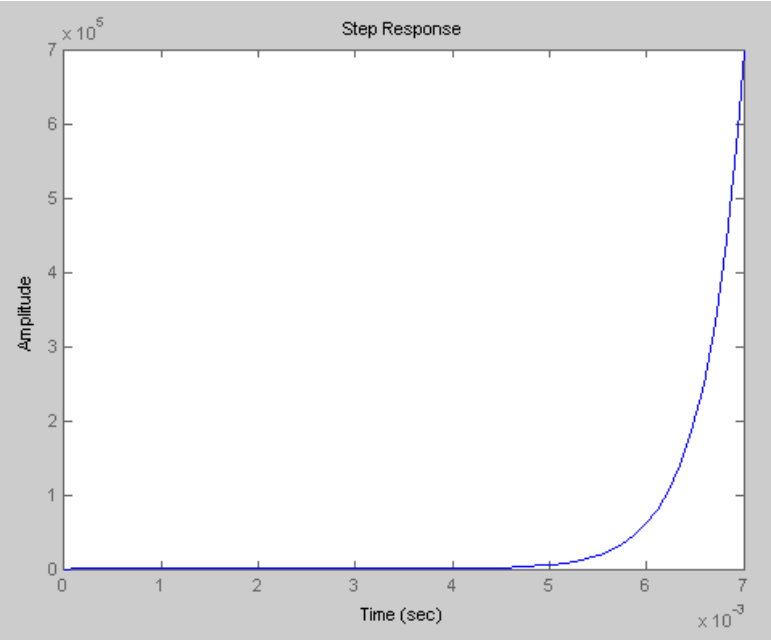
Matlab code and step response

Matlab code for the elevator application can be found in file april12.m. The step response is shown in the figure below.

```
n1=2;
n2=3;
n=n1+n2; %ratio of gearbox
m=8e-6;
Rt=0.05;%radius
d=3.8e-6; %viscous friction
Vmax=12;Vmin=-12;Imax=0.5;
R=24;%24;%friction
Ke=0.905*1e-
3/(2*3.14/60); %back emf
constant
Km=8.64e-3; %torque constant
L=750e-6; %rotor inductance
Jmotor=0.59e-7; %motor inertia
Jtot=Jmotor/n^2;
K=Ke+Km;
```

```
A=[-R/L -(Ke/L)*(n1/(n2*Rt));
Km/(Jtot-(Rt^2)*m) -d/(Jtot-
(Rt^2)*m)]
B=[1/L 0]'
C=[1 0]
D=[0]

%open loop system
sys = ss(A,B,C,D);
tfsys = tf(sys)
step(sys);
```



Appendix B

Course MF2044, Embedded Systems for Mechatronics, II

Learning outcomes

This course aims to equip the participants with fundamental knowledge and practical skills for the development of embedded systems with emphasis on correctness by construction, verification, and debugging. This understanding means that You after the course should be able to

1. examplify embedded systems and their applications, describe the special requirements placed in developing such systems and the differences among different application domains (e.g. automotive, automation and medtech).

2. describe and apply systematic approaches to system development including requirement specification, function design and realization, verification and validation.

3. classify and explain different types of functionalities, behaviors, their corresponding modeling techniques and implications on software, hardware, and real-time implementation.

4. apply your knowledge in control theory and software programming in the design and implementation of control applications on distributed computers.

5. describe, explain, and apply software platform technologies (real-time operating systems - RTOS).

6. describe and explain fundamental techniques for verification and debugging, including how to derive test cases, and apply a subset of these techiques.

7. analyze system requirements, derive the implied functional and nonfunctional constraints, and motivate architectural design and execution strategies using reference styles and patterns.

8. understand the trends and state-of-the-art approaches to model- and component-based development of embedded systems.

Course main content

The course includes
Lectures, where overview and inspiration are provided.
Laboratory exercises, where tools and techniques are introduced, and a set of practical excercises are carried out by the participants in groups.
Classroom exercises, where the participants can elaborate and practice theoretical parts of the course.

Each week of the course focuses on a specific theme. The exercises are modularized according to these themes. The exercises include the implementation of functionalities with RTOS in a single and distributed system. In parts of the exercises, the system designs will be modelled and analyzed using Matlab Simulink/Stateflow and other techniques.

Appendix C

CodeWarrior project

The following appendix contains files from the CodeWarrior project folder Application and files from the project that are inherent. The modified or added lines and lines that are associated to the code generation from Simulink are shown in bold.

Application files

main.c

```
#include <basic/bs_basic.h>
#include <hal.h>
#include <ros_xhp.h>
#include <blue/b_thread.h>
#include "r_ex_r.h"
#include <hal_xhp.h>
#include <can.h>
#include <can_mcf5213.h>
#include "support_common.h"
#include "interruptRoutines.h"
#include "LEDDriver.h"
#include "motorDriver.h"
#include "keyboardDriver.h"
#include "encoderDriver.h"
#include "mcf5xxx.h"
#include "Elevator_private.h"
#include "rtwtypes.h"            /* MathWorks types */
#include "Elevator.h"            /* Model's header file */
#include <basic/bs_basic.h>

void rubusIdleMain(void);
void halBsTimerIntrEntry(void);
void rt_OneStep(void);
```

```
void PITInit(int clockPrescaler, int modulus);
__declspec(interrupt) void pit_handler(void);
void rt_OneStep(void);
void timer(void);

int clockPrescaler, modulus;
int position, old_position;

void timer(void)
{
	uint_t basicTimerPriority;
	extern int speed;
	extern int position, old_position;

	basicTimerPriority = 7;

	// position
	position = old_position + speed*MCF_PIT0_PCNTR;
	MCF_PIT0_PCNTR = 0;

	// starta om timer
	halBsTimerStart();
}

__declspec(interrupt) void pit_handler(void) {

	// increment elevator rt
	rt_OneStep();

	// increment the Basic Clock and the Red Schedule Timer
	halBsTimerIntrEntry();

	MCF_PIT0_PCSR |= MCF_PIT_PCSR_PIF;
}

void PITInit(int clockPrescaler, int modulus) {//ok

	MCF_PIT0_PCSR = MCF_PIT_PCSR_PRE(clockPrescaler)
```

```
                        /*interrup enable*/
                                                | MCF_PIT_PCSR_PIE
                        /*reload module value when counter reaches 0*/
                                                | MCF_PIT_PCSR_RLD
                        /*reset counter when write to PMR*/
                                                | MCF_PIT_PCSR_OVW
                        /*enable PIT*/
                                                | MCF_PIT_PCSR_EN;

                        /*Configure the PIT module*/
                        MCF_PIT0_PMR = MCF_PIT_PMR_PM(modulus);

                        /*Set the interrupt priority and level*/
                        MCF_INTC_ICR55= MCF_INTC_ICR_IP(7) |
MCF_INTC_ICR_IL(7);//4

                        //Setup the interrupt registers for PIT
                        MCF_INTC_IMRH &= ~ MCF_INTC_IMRH_INT_MASK55;
}

void rt_OneStep(void){
 static boolean_T OverrunFlag = 0;

 /* Disable interrupts here */

 /* Check for overrun */
 if (OverrunFlag) {
  rtmSetErrorStatus(Elevator_M, "Overrun");
  return;
 }

 OverrunFlag = TRUE;

 /* Save FPU context here (if necessary) */
 /* Re-enable timer or interrupt here */
 /* Set model inputs here */

 /* Step the model */
```

```
  Elevator_step();

  /* Get model outputs here */

  /* Indicate task complete */
  OverrunFlag = FALSE;

  /* Disable interrupts here */
  /* Restore FPU context here (if necessary) */
  /* Enable interrupts here */
}


void main (void)
{
        leds_initialise();
        keyboard_initialise();
        motors_initialise();
        Elevator_initialize();
        //encoders_initialise();

        if (halInit() != R_OK)
        halHalt();
        if (bsRubusInit() != R_OK)
        halHalt();
        if (halStart() != R_OK)
        halHalt();

        bsRubusStart();
}

uint32_t rubusIdleLoop;
int_t stackRedUsed;
int_t stackBlueUsed;

void rubusIdleMain(void)
{
        rubusIdleLoop = 0;
```

```
	for (;;) {
		rubusIdleLoop++;
		if (redStackUsed(&stackRedUsed) != R_OK) {
			halHalt();
		}
		if (blueStackUsed(&stackBlueUsed) != R_OK)
		{
		halHalt();
		}
		SIM_BLUE_IDLE
	}
}
```

blue.c

```
#include <blue/b_signal.h>
#include <blue/b_msgqueue.h>
#include <r_ex_r.h>
#include <r_ex_b.h>
#include <hal.h>
#include <blue/b_sem.h>
#include "motorDriver.h"
#include "keyboardDriver.h"
#include "encoderDriver.h"
#include "LEDDriver.h"
#include <bs_basic.h>
#include <hal_ctypes.h>

static bsPosixTime_t const MotorThreadSleep={0L,500000000L};
static bsPosixTime_t const KeyboardThreadSleep={0L,200000000L};
static bsPosixTime_t const MotorThreadCommWait={5L,0L};

void timer(void);

char addValue;
int speed;
```

```
char_t ch; //made global

void readKeyboardMain(void) {

	int32_t tp;
	int_t rcode;
	blueSigSet_t set;
	extern char_t ch;
	extern int_t speed;
	static int last_ch;

	tp = bsTvToJiffies((bsPosixTime_t*)
	&KeyboardThreadSleep);

	for (;;) {
		ch = keyboard_get_char();

		if (ch != '!') {
			switch(ch) {
				case '0': speed = -124; break;
				case '1': speed = -100; break;
				case '2': speed = -75; break;
				case '3': speed = -50; break;
				case '4': speed = -25; break;
				case '5': speed = 25; break;
				case '6': speed = 50; break;
				case '7': speed = 75; break;
				case '8': speed = 100; break;
				case '9': speed = 124; break;
				case '*': redSetSchedule(&redScheduleA);
				break;
				case '#': redSetSchedule(&redScheduleB);
				break;
			}
			rcode = blueMsgSend(&desiredSpeed, (uint8_t
			*)&speed,sizeof(int));

		if (rcode != R_OK)
```

```
                if (rcode != -R_ERROR_FULL)

                blueError(1,(bsObject_t const *)blueSelf());

                }//if
                blueSleep(&tp);
        }//for
}

void runMotorMain(void) {

        int32_t tp1, tp2;
        extern int speed;
        int previous_speed;

        tp1 = bsTvToJiffies((bsPosixTime_t*)
        &MotorThreadCommWait);
        tp2 = bsTvToJiffies((bsPosixTime_t*)
        &MotorThreadSleep);

        motor1_enable();

        while (1) {

                if (blueMsgTimedReceive(&desiredSpeed,
                (uint8_t *)&speed, sizeof(int),&tp1) == R_OK)
                        motor1_set_speed(speed);

                else {
                        motor1_set_speed(0);

                        blueSleep(&tp2);
                }
                        // start timer for current speed
                        if(speed != previous_speed)
                        {
                                timer();
                        }
```

```
                previous_speed = speed;
        }//while
}
```

red.h

```
#ifndef _RED_H_
#define _RED_H_

typedef struct {
int scheduleA;
int scheduleB;
int scheduleC;
} scheduleModeType;

void redStartMain(void);
void rescheduleMain (scheduleModeType *state);
void writeMain(int *args);
void timer(void);
void blinky0Main (int *state);
void blinky1Main (int *state);
void blinky2Main (int *state);
void blinky3Main (int *state);
void blinky8Main (int *state);

#endif
```

red.c

```
#include <r_ex_bs.h>
#include <hal.h>
#include <r_ex_r.h>
#include "red.h"
#include "LEDDriver.h"
#include "stdio.h"

void timer(void);
```

```
void redStartMain(void) {

	blinky0Args = 0;
	blinky1Args = 0;
	blinky2Args = 0;
	blinky3Args = 0;
	blinky8Args = 0;
	writeArgs = 0;
	rescheduleArgs.scheduleB = 0;
	rescheduleArgs.scheduleC = 0;
	redSetSchedule(&redScheduleA);
	rescheduleArgs.scheduleA = 1;
}

void rescheduleMain (scheduleModeType *state)
{
	extern int position;
	extern char_t ch;
	timer();

	//switch to redScheduleC if position is 0 or 1000
	if (position == 0 || position == 1000) {

		redSetSchedule(&redScheduleC);
		rescheduleArgs.scheduleC = 1;
		rescheduleArgs.scheduleA = 0;
		rescheduleArgs.scheduleB = 0;

	}

	if(ch == '*')
	{
		redSetSchedule(&redScheduleA);
		rescheduleArgs.scheduleA = 1;
		rescheduleArgs.scheduleC = 0;
		rescheduleArgs.scheduleB = 0;
	}
```

```
        if(ch == '#')
        {
                redSetSchedule(&redScheduleB);
                rescheduleArgs.scheduleA = 1;
                rescheduleArgs.scheduleC = 0;
                rescheduleArgs.scheduleB = 0;
        }
}

void writeMain (int *args)
{
        extern int position;

        timer();

        // set led number 3 if on ground floor
        if(position == 0)
        {
                leds_turnON(2);
                motor1_set_speed(0);
        }

        // set led number 8 if on top floor
        else if(position > 19990)
        {
                leds_turnON(7);
                motor1_set_speed(0);

                // if redscheduleB is active, set the speed to -50
                if(rescheduleArgs.scheduleB == 1)
                {
                        motor1_set_speed(-50);
                }
        }
        // write position
        printf("position   %i m", position);
}
```

```
void blinky0Main (int *state) {

	if (*state == 0) {
		*state = 1;
		leds_turnON(0);
	}
	else {
		*state = 0;
		leds_turnOFF(0);
	}
}

void blinky1Main (int *state) {

	if (*state == 0) {
		*state = 1;
		leds_turnON(1);
	}
	else {
		*state = 0;
		leds_turnOFF(1);
	}
}

void blinky2Main (int *state) {

	if (*state == 0) {
		*state = 1;
		leds_turnON(2);
	}
	else {
		*state = 0;
		leds_turnOFF(2);
	}
}
```

```
void blinky3Main (int *state) {

	if (*state == 0) {
		*state = 1;
		leds_turnON(3);
	}
	else {
		*state = 0;
		leds_turnOFF(3);
	}
}

void blinky7Main (int *state) {

	if (*state == 0) {
		*state = 1;
		leds_turnON(7);
	}
	else {
		*state = 0;
		leds_turnOFF(7);
	}
}
```

Project files

Since the project doesn't make use of the can or xhp-communication it has been possible to do these changes made in the functions halXhpInit() and __declspec(interrupt)void canISR_Msg0(void). Only the functions where changes have been made are appended. The lines commented away are in italic and bold for clarity.

hal_xhp.c

```
/*********************************************
 halXhpInit()
*********************************************/
```

```
int8_t halXhpInit (void)
{
  comHandlerRxCB.flags    = _R_READ;
  comHandlerRxCB.ioSize   = xhpHandlerAttr.sizeRead;
  comHandlerRxCB.pIo      = xhpHandlerAttr.ioBufferRead;
  comHandlerRxCB.nIo      = 0;

  comHandlerTxCB.flags    = 0;
  comHandlerTxCB.ioSize   = xhpHandlerAttr.sizeWrite;
  comHandlerTxCB.pIo      = xhpHandlerAttr.ioBufferWrite;
  comHandlerTxCB.nIo      = 0;

  canMsgRx.header.id.id32 = CAN_RUBUS_ID_RX;
  canMsgRx.header.nBytes = 8;
  canMsgTx.header.id.id32 = CAN_RUBUS_ID_TX;

#ifdef _PROJECT_OS
  // canDeviceInit (CAN_XHP_NODE,comXhpTx.priority,2);
#else
  canDeviceInit (CAN_XHP_NODE,RTE_IT_Int_IrqTx.priority,2);
#endif
  canInit (CAN_XHP_NODE, &canBaud);

  canSetMask (CAN_XHP_NODE, 0, canMaskGlobal);
  canConfigMsgObj(CAN_XHP_NODE,CAN_RUBUS_OBJECT_READ,&canMsgRx.header,(_R_READ
| _R_APPEND));

#ifdef _PROJECT_OS
  // canSetIntrObj(CAN_XHP_NODE,CAN_RUBUS_OBJECT_READ,comXhpRx.priority);
  // canSetIntrObj(CAN_XHP_NODE,CAN_RUBUS_OBJECT_WRITE,comXhpTx.priority);
#else
  canSetIntrObj(CAN_XHP_NODE,CAN_RUBUS_OBJECT_READ,RTE_IT_Int_IrqRx.priority);
  canSetIntrObj(CAN_XHP_NODE,CAN_RUBUS_OBJECT_WRITE,RTE_IT_Int_IrqTx.priority);
#endif
  canEnable (CAN_XHP_NODE);
  return(R_OK);
}
```

can_mcf5213_isr.c

```
/*********************************************************
 CAN Interrupt Service
*********************************************************/

__declspec(interrupt)
void canISR_Msg0(void)
{
#ifdef _PROJECT_OS
 // comXhpRx.entry(comXhpRx.argument);
#else
 RTE_IT_ENTRY_Int_IrqRx(0);
#endif
}

/*********************************************************
 CAN Interrupt Service
*********************************************************/
__declspec(interrupt)
void canISR_Msg1(void)
{
#ifdef _PROJECT_OS
 // comXhpTx.entry(comXhpTx.argument);
#else
 RTE_IT_ENTRY_Int_IrqTx(0);
#endif
}
```

mcf5xxx.h

```
/*****************************************************************************/
/*
 * The basic data types
 */
typedef unsigned char      uint8;  /*  8 bits */
```

```
typedef unsigned short int  uint16; /* 16 bits */
typedef unsigned long int   uint32; /* 32 bits */

typedef signed char             int8;   /*  8 bits */
typedef short int          int16;  /* 16 bits */
typedef long int              int32;  /* 32 bits */

typedef volatile int8       vint8;  /*  8 bits */
typedef volatile int16      vint16; /* 16 bits */
typedef volatile int32      vint32; /* 32 bits */

typedef volatile uint8      vuint8;  /*  8 bits */
typedef volatile uint16     vuint16; /* 16 bits */
typedef volatile uint32     vuint32; /* 32 bits */
```