Development of an Autonomous Ground Vehicle for the RobonAUT 2014 Contest

Attila Fodor Csorvási Gábor
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
{csorvagep, attila.fodor.89}@gmail.com

Abstract. RobonAUT is a local robot development contest of the Faculty of Electrical Engineering and Informatics of BME. This paper describes the development process of an Autonomous Ground Vehicle which was one of the successful competitors of the 2014 contest. During software development of the embedded control system the possibilities of rapid prototyping have been exploited extensively. The paper demonstrates the power of model based design of control systems and embedded software, and describes the rapid deployment method in detail, which was used to integrate the MATLAB model with FreeRTOS on the target hardware.

Keywords: RobonAUT; Autonomous Robot; MATLAB; Simulink; Control Systems; Rapid Prototyping, FreeRTOS

1 Introduction

RobonAUT is an annual contest of the Department of Automation and Applied Informatics of the Budapest University of Technology and Economics. Each year many teams design and build autonomous model cars to compete on an obstacle course, and rival each other on a race track. The robots must be completely autonomous, lacking any remote control, and any sort of external intervention during the race is punished. The team whose car scores the highest wins the contest [1].

Even with such simple robots, there are enormous amounts of work, including the design of the system hardware carrying the sensor array, the software capable of recognizing and handling obstacles, a client software to ensure a safe testing environment and an efficient control system to keep the the car stable while sweeping through the racetrack [2]. As the fame of the competition grows, the expectations towards the cars heighten and teams become more ferocious to win. This all causes the pressure on the students to enlarge, while they have to hold their own in other challenges throughout the semester.

Usually a low-level approach to the problem is the best approach to small projects like this. Although good ideas people have used successfully in the past must not be forgotten, innovation is crucial. Therefore we decided to look beyond the boundaries of classic software development, to find a solution that enables us to concentrate on the important aspects, instead of being lost in

the thousand lines of source code. However, utilizing complex technologies and integrating various pieces of design software is not an easy task. It might present more problems than solutions, because the available time for the development is very limited.

Although these tools can greatly enhance the productivity of the developers, Model-Based Design is still scarcely used in the industry, because of its initial setup complexity, and the high price of design software discourage its use even further. Against all these odds its popularity is increasing, but it was never considered a way to deal with small projects before.

We wanted to demonstrate that a scaled-down approach is possible using this method. The functional software and the operating system responsible for the core services can be integrated manually, and most of the advantages can be retained while keeping the technology simple. We used MATLAB-Simulink to create a *Rapid Prototyping* environment, in which using a simulated model of the system the software development can be started weeks before an actual hardware prototype. Using automated code-generation for the target hardware, the core functions can be augmented with features designed and verified in a swift visual model-based environment, to overcome the challenges the contest presents. The development cycle becomes shorter, and the result is a syntactically correct code free of any compile or runtime errors. This is guaranteed by the code generation software.

Outline

- The paper will demonstrate how the focus of the software development has shifted from the actual coding to a more visual and mathematical representation, and how to harness its traits to quickly develop a reliable system.
- It will be described how to integrate the MATLAB source with the Core system, and deploy it on the target hardware, the STM32F4-Discovery developer board.

2 Development Methods

Classic Software Design Approach In every software development project, a visual sketch of the system is often created in order to aid the design process, and act as a visual guide during the documentation. However, implementing whatever design created earlier is not always trivial. The primary and most widespread embedded programming language, C, has been designed as a "Professional Programmers Language" [3], and it supports only very basic syntax checks, but nothing ensures that the program will behave as the programmer intended. The designer must choose to take the risks of unexpected errors, or a thorough testing must be conducted in order to ensure the correct and reliable functioning of the system. While the testing of simple systems can be finished quickly, dangers arise when the same method is used in a large multi-developer

project. Most of the industry have responded to this by scaling up the work force that gave birth to a number of strict project safety standards, internal regulations and coding guidelines [4], and led to a huge amount of overhead in human labour in each software development project.

Model-Based Design As described earlier, the model-based design is based on solving the problems in a more visual environment. During the development, there is limited connection between the designed software and the target hardware. Many of the solutions can be verified using the simulated model of the system, resulting in an accelerated development process[5]. Due to the heavy reliance on abstract models and their connections, even if significant changes occur in the specifications, only minimal amount of modification is required to the code that formulates the final software. This advantage can not be over emphasized, because the final specifications of the system are rarely known, especially in prototype development. Although it has its own limitations as well, model-based design relies on complex tools to do most of the work, and their operation and integration into an existing project is difficult and become less practical or limited because of the lack of sufficient support as well.

Background For the **RobonAUT 2014** contest, we used MATLAB and Simulink to implement the control system and the state machine. We chose the MathWorks product family, because Simulink has extensive support for model and simulation-based development, and allows the generation of C source code that can be later used on any hardware that is capable of running FreeRTOS or any other hard real-time operating systems. We have have used these technologies before successfully in industrial environment, which allowed us to follow out the concept.

3 Model-based Design of the System

Figure 1 shows the basic system connections and stages of deployment. A typical system designed in MATLAB can be divided to the following parts.

1. A MATLAB Script that defines the system model and computes the simulator and controller parameters.

2. Based on these parameters we can build the simulator and control software in Simulink that interact with each other.

3. If the system response in the simulation is correct, the controller is ready for code generation and field-testing.

4. The generated source code is invoked by the core operating system, the FreeRTOS in this case

5. All communication with the hardware in implemented by the Core.

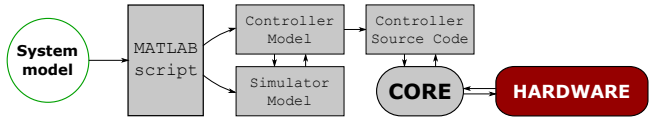

Figure 1: Connections and dependencies

Although in most cases the core unit of a complex software will be an operating system, it is not mandatory. Therefore we use the *Core* expression in this paper, but it refers to the FreeRTOS operating system in our case.

3.1 Functional Description

During the race, the track is marked by a dark line on light ground. The car follows this trail that guides it through certain checkpoints. In between there are obstacles that need to be passed in a specific way. Each obstacle is marked in a unique manner that allows the car to unambiguously determine, based on simple on-board sensors, which function it has to execute in the current segment.

The recommended sensor outfit[6] is a number of optoelectronic sensors beneath the front axis or bumper of the car, and a couple of infrared proximity sensors at the front and two sides of the car. Certain teams extended this setup with a camera for image-recognition, a second optosensor array or ultrasound proximity sensors, but this paper will not consider any of these layouts.

3.2 Building a Model

The first step in model-based design is to create a model of the system, based on the functional description and known sensor fitting. To follow the line, the signals of the optical array must be processed and a control system must use this input to keep the car on the track. Let's define the following state variables:

d	Position error: The shortest distance between the centre of the front optical array and the centre of the track
δ	Angular error: The angle between the centreline of the body and the tangent of the track
Φ	Steering angle: The effective steering angle, according to Ackermann-steering
κ	Current curvature of the track $(1/R)$
x	Line position: the position of the track line along the front optical array
c	Line speed: derivative of x, the change of the track line position
v	Car speed: The current speed of the car along the centreline

Other variables with temporary significance might be defined in the text before their usage. Figure 2 helps to visualize the basic geometrics of the car, and the connections of the state variables.

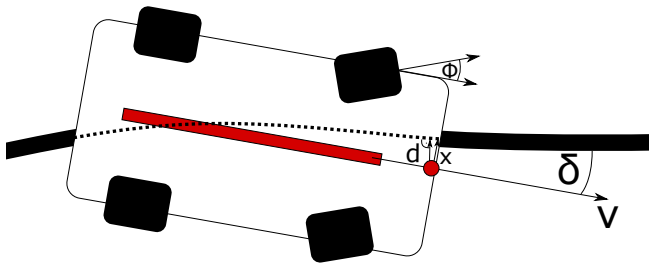

Figure 2: System model

Formulating the System Model Knowing the dynamical system equations (1, 2), and parameters (L: wheelbase), we can formulate the **A** and **B**[1] matrices based on their linear approximations in an equilibrium state ($d = 0; \delta = 0; v = 1$) (3).

$$\dot{d} = sin(\delta + \Phi) \cdot v \tag{1}$$

$$\dot{\delta} = \frac{v}{L} \cdot tan(\Phi) + \kappa \tag{2}$$

$$\begin{bmatrix} \dot{d} \\ \dot{\delta} \end{bmatrix} = \begin{bmatrix} 0 & \delta \cdot v \\ 0 & 0 \end{bmatrix} \begin{bmatrix} d \\ \delta \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ \frac{v}{L} & 1 \end{bmatrix} \begin{bmatrix} \Phi \\ \kappa \end{bmatrix} \tag{3}$$

Instead of hard-coding the system matrices into the script, we can apply a different method that is nicer for the model-based approach, by using the MATLAB Symbolic Toolbox. This package allows the automatic generation of the Jacobi-matrices of the system, so the inputs are narrowed down to the system parameters and non-linear dynamic equations. Furthermore, this way we can hold on to a lot more information that can later be used to generate a non-linear state observer, Extended Kalman Filter [7], or Hybrid Controller [8] for additional features in a complex system.

Control System Once an accurate system model is available, it's possible to formulate a controller. It's relatively easy to implement multiple controllers and compare them on the simulated system to determine, which would fit the robot the best. Already knowing the state-space form of the system, a full-state

[1] The system is in discrete-time, but the continuous-time notation will be used throughout the paper, for better readability.

feedback controller can be formulated quickly to stabilize the system. Using the Ackermann formula, we can obtain a suitable feedback matrix **K**, thus creating a stable new system with the feedback.

$$\dot{\underline{x}} = (A - BK) \cdot \underline{x} \tag{4}$$

Direct and Inverse Measurement Models Unfortunately, a full-state feedback loop can rarely be realized directly, hence the controller we have just created would not be able to control the actual system. Certain states can not be measured but only estimated by a state observer. The controller must be prepared to do that. Simulating the sensor readings based on the state variables is possible with the *Direct measurement model*, while in order to estimate the states based on the sensor readings we need the *Inverse measurement model*.

Knowing the geometrical layout of the car, the expected signals can be simulated using inverse geometric projection to the optosensor array:

$$x = \frac{d}{\cos(\delta)} \tag{5}$$

After the direct model has been determined, the inverse model can be formulated as well:

$$\hat{d} = x \cdot \cos(\delta) \tag{6}$$

$$\hat{c} = \dot{x} \tag{7}$$

$$\hat{\delta} = \arctan\left(\frac{c}{v}\right) - \Phi \tag{8}$$

The direct model is usually part of the simulation only, however, it is possible to proof-check the sensor readings during run-time. This can be especially useful, if the states are estimated by sensor fusion. The inverse measurement model is always the first layer of the control loop, and it is rarely found in the simulated environment.

Note: δ can be directly measured if the robot is equipped with multiple optosensor arrays, but of course a different inverse measurement model is still necessary.

3.3 Building the Simulation Environment

When the system model and controller are available, it is time to build the simulation environment. In case of a simple system, at this point the controller could be implemented directly in C, but if the system is more complex, it is better practice to follow through the model-based design. To test the behaviour of the system, the controller and the simulator must be implemented in Simulink as well. Figure 3 shows the simulation-based development concept. Because the

Controller model and the Controller source code are identical, the key is to design a simulator that is deceptively similar to the real system, from the point of view of the controller. This is the reason why Direct and Inverse Measurement (msment) models are required.

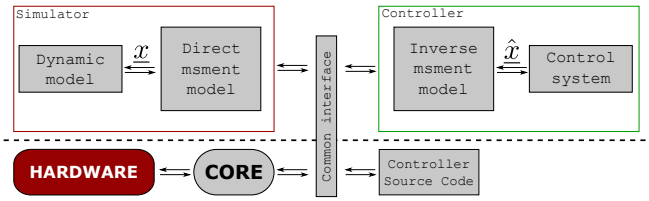

Figure 3: Simulation environment and analogy of deployment

Between the Simulator and Controller model, and between the Core and the Controller Source Code the actual sensor signals travel through a common interface, in the same form. The modules are interchangeable, without any functional loss. The modules themselves in the inside are working with the real (and estimated) state variables. The measurement models are mere encoding and decoding interfaces.

3.4 Simulator Model

The controller that was designed earlier is based on the linear approximation of the system, but the simulator must always represent the full non-linear system, or the whole model-based design effort is pointless. The system can be created with basic Simulink blocks. However, exploiting the technique described earlier to generate the Jacobi matrices of the system, we can use them directly to generate a full representation.

Figure 4 shows the simulation of the dynamics designed to test the control system (without inputs). A common programming analogy can describe how a discrete time Simulink Model works. The **States** block is the central element of the model, which stores the actual state variables. It is a *storage unit*, which acts as a variable in this case. It outputs the value of the input of the previous time-step (which can be considered as a single execution of commands in a loop). The **Direct measurement model** simulates the sensor readings for the given state variables, in the same form as the expected inputs from the hardware, by generating the signal of each individual optosensor and putting them into a vector (same as a 1-dimensional array), just like as it is received from the Core.

Based on the system dynamics and the current value of the state variables, the states of the car can change. For example, a nonzero speed results in a change of traveled distance for the next time-step, even when there are no inputs to the system. The product of the state variables ($\mathbf{x}$) and the system update matrix ($\mathbf{A}$) results in a vector filled with the new states. In case of a linear system, $\mathbf{A}$ is constant. In case of a nonlinear system however, $\mathbf{A}$ depends on the current states.

Consider the above example with nonzero speed ($\mathbf{v}$), a straight track, the distance from the track ($\mathbf{d}$) and angular deviation from the track (δ). While the change of $\mathbf{d}$ is a linear function of $\mathbf{v}$, it is also a trigonometric function of δ. The resulting function is nonlinear, which means $\mathbf{A}$ is a nonlinear function of δ.

The Jacobian is a matrix of the first-order partial derivatives of the system [9, p. 294]. It basically tells the effect of an individual state variable on the system. The summary of these effects result in the full system update.

$$J = \begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_n}{\partial x_1} & \cdots & \frac{\partial F_n}{\partial x_n} \end{bmatrix} \longrightarrow A_J = \begin{bmatrix} \frac{\partial \dot{d}}{\partial d} & \frac{\partial \dot{d}}{\partial \delta} \\ \\ \frac{\partial \dot{\delta}}{\partial d} & \frac{\partial \dot{\delta}}{\partial \delta} \end{bmatrix}; \quad B_J = \begin{bmatrix} \frac{\partial \dot{d}}{\partial \Phi} \\ \\ \frac{\partial \dot{\delta}}{\partial \Phi} \end{bmatrix} \tag{9}$$

If the system is linear, the elements of the matrix are constants. If the system in nonlinear, some elements of the matrix are functions of the state variables themselves.

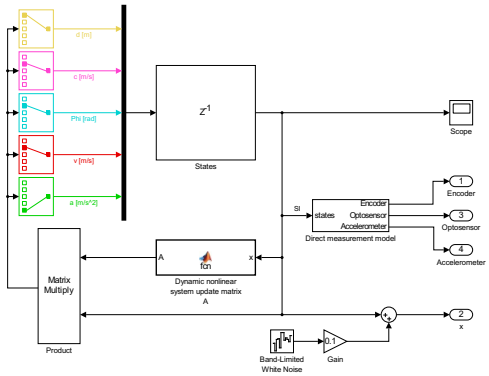

Figure 4: Simulator model based on Jacobian matrices

The **Dynamic nonlinear system update matrix A** block contains a simple MATLAB script that builds the current system update matrix based on the Jacobian functions and the current states. Once we possess this, the system update phase can be executed just like in a linear case, with a matrix product. The resulting vector is the new set of state variables.

3.5 Implementation of the Control System

To describe the implementation of the entire control system is out of the scope of this paper, however, key aspects of the software will be highlighted. The visual development of the software requires a different way of thinking than working with traditional source code. Therefore, we encourage the reader to design his or her own control system, in order to get a first hand experience in model-based design.

Inverse Measurement Model The implementation of the inverse measurement model described earlier. During the race, multiple signals provide information about the track (optosensor array, side proximity sensors), therefore an inverse measurement model must be derived and realized for each of them, to provide the system states for the controller.

Controller Although basic processing might be required to enhance the raw signal quality before the inverse measurement model, any serious processing should be implemented in the next stage. Using model-based design, several controller implementations can be evaluated quickly by testing and comparing them using the dynamic simulation of the model. A word of caution though: an inferior simulator can negatively influence the results of the comparison. Always introduce the major physical limitations to the system, for example the non-zero transition time of the steering servo between states.

3.6 High Level Control and State Machine

MATLAB has an extension called Stateflow dedicated to the development of state machines. It integrates seamlessly with Simulink Models, and supports a wide range of features including subcharts, temporal logic and code generation as well, making it a very powerful tool. Figure 5 shows the top layer of the state machine implementation of the obstacle course in Stateflow. It is primarily used to detect certain sections during the race based on the external signals, and take over the control of the car at some points to perform an action like the automated parking.

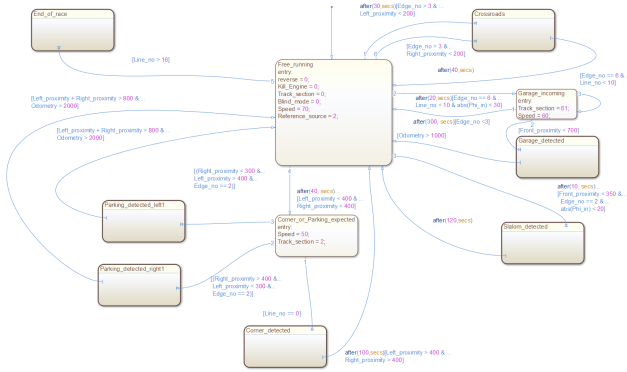

Figure 5: State machine diagram of the obstacle course

4 Integration

If the controller is implemented and tested, it's ready for C code generation (or C++, Verilog or PLC). Find the correct subsystem that acts as the controller, and build the model with the `MATLAB Embedded Coder`. In a large system, it is possible to generate separate source code implementations of different subsystems. Though the integration complexity increases, it can provide enhanced functionality, and excellent reusability.

4.1 Structure of the Generated Code

If during the code generation *Compact code placement* was used, only four files are generated. Figure 6 shows the connection between these files.

- "system_name".c
- "system_name".h
- rtwtypes.h
- ert_main.c

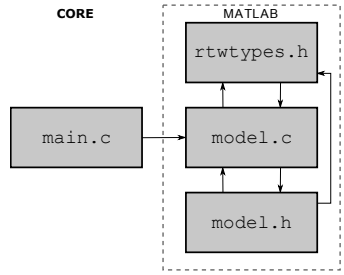

Figure 6: Relationship of the generated files

In the following we assume that the name of the Controller block was "model". The functions of the system are defined in the `model.c` and `model.h` files. The `rtwtypes.h` contains the unique type definitions of the code. The `ert_main.c` is an exemplary main function that demonstrates the use of the others.

4.2 Using the Controller Source Code

The generated source code is functionally equivalent to the Simulink model. A number of functions and variable structures are defined that implement an easy interface of control the module, and communication with the system is only possible through this unique interface provided by the generated code. The system can be initialized using the *model_initialize()* function. It sets the 0 time in the model, and the outputs assume their initial values. The inputs are handled by the *model_U* struct. It contains fields each corresponding to a system input, matching it's name and type. If the inputs are set, calling the *model_step()* function runs the model once, for the period of one time sample. The outputs are stored in a structure similar to the input storage, called *model_Y*.

4.3 Deployment with FreeRTOS on STM32F4-Discovery Board

As in many controlled processes, the timing of the controller program execution is crucial, as well as the processing of the I/O ports, converting the analogue signals and sending status information to the supervisor computer over a wireless connection. It can be implemented with integrated timer peripherals and interrupt sequences, but in case of a multi task system, the application of a *Real Time Operating System* allows a more high-level approach. There are numerous implementations of this operating system family. We chose FreeRTOS, a popular open-source operating system in the industry, because of its detailed documentation and earlier work experiences with the operating system on STM32F4-Discovery. In addition, it provides useful features for application debugging.

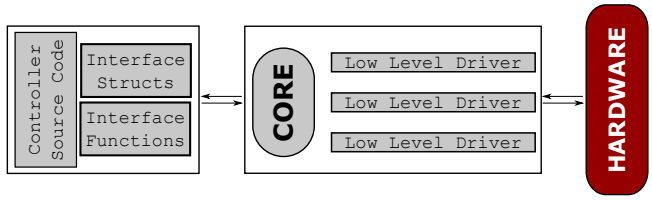

Figure 7: Connections between modules

Figure 7 illustrates the usage of the technologies. In our solution the FreeRTOS is responsible for the task scheduling, and provide a communication interface between the tasks. This way we can ensure that the sensor readings and the controller task are conducted with precise timing. Through the operating system direct communication with the I/O ports is simple, and it handles the processing of the optoelectronic and proximity sensor signals with the Analogue to Digital converters of the board.

Setting up the system requires the inclusion the FreeRTOS source files in the project and some additional low level drivers to read the sensors, and initialize the actuators. Once the Core system is ready, the generated source files are included and periodically called by a timer, and its input and output *structs* are connected to the electronics through the low-level drivers.

5 Conclusion and future work

The most important result is the successful application of the method. The system was designed and implemented in MATLAB-generated code integrated with a FreeRTOS Core, based on the methods described in this paper. Despite the odds and a challenging competition during the **RobonAUT 2014**, our car have scored a podium finish. It demonstrated complete reliability, though there have been stability issues in the faster sections, due to a flaw in the controller design. We plan to participate in the **RobonAUT 2015** competition as well,

and utilize the direct hardware support of MATLAB for the STM32F4-Discovery, to improve the performance both of the vehicle and team.

Acknowledgments

The authors would like to express their thanks to all of the colleagues of the Department of Automation and Applied Informatics and everyone else involved in the organization of the RobonAUT competition.

References

[1] Department of Automation and Applied Informatics, Budapest University of Technology and Economics, *RobonAUT 2014 Versenyleírás és szabályzat*, v 1.4 ed., January 2014.

[2] *Autonóm robot vonalkövetése, különböző szabályozó algoritmusok vizsgálata*, BME Conference of Scientific Students' Associations, November 2010.

[3] N. Parlante, *Essential C*. Stanford CS Education Library, April 2003.

[4] MIRA Ltd., *MISRA-C:2004 Guidelines for the use of the C language in critical systems*, October 2004.

[5] J. Reedy and S. Lunzman, "Model based design accelerates the development of mechanical locomotive controls," October 2010.

[6] D. Kiss and S. Kolumbán, *RobonAUT 2014 Szabályozástechnikai szeminárium*. Department of Automation and Applied Informatics, Budapest University of Technology and Economics, October 2013.

[7] G. A. Terejanu, "Extended Kalman Filter Tutorial," tech. rep., Department of Computer Science and Engineering, University at Buffalo, Buffalo, NY 14260.

[8] P. J. Antsaklis and X. D. Koutsoukos, "Hybrid systems control," tech. rep., Department of Electrical Engineering, University of Notre Dame, March 2001.

[9] F. Wettl, *Lineáris Algebra*. Typotex kiadó, 2011.