Development of an Autonomous Ground Vehicle for the RobonAUT 2014 Contest

Csorvási Gábor Fodor Attila
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; AACS Workhsop; MATLAB; Simulink; Control Systems

1 Introduction

The 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 battle on an obstacle course, and compete each other on a race track. The robots must be completely autonomous, lack any remote control and any sort of external intervention during the race is punished. The team whose car scores the highest wins the contest[?].

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 tasked to keep the the car stable while sweeping through the racetrack. 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 a small projects like this. Although good ideas people have used successfully in the past must not be forgotten, innovation is crucial. For these reasons, 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 a complex technology, and integrating several design softwares is no 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 softwares discourage its use even further. Against all these odds though 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 technology. Because the project is relatively small, the functional software and the operating system responsible for the core services can be integrated manually, and most of the advantages of the methodology 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 conductive software development could be started weeks before an actual hardware prototype. Using automated code-generation for the target hardware, through a predefined interface the core services can be augmented with the proper functionalities, designed and verified in a swift visual model-based environment, to overcome the challenges the contest presents. The development cycle became shorter and the syntactic reliability of the resulting system is guaranteed by the code generation software.

Summary of Contributions

- 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 use the `MATLAB Coder` auto code generation tool and how to integrate the 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 in the documentation. However, implementing whatever design created earlier is not always trivial. The primary and most widespread embedded programming language, the `C`, has been designed as a "Professional Programmers Language", 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, many

dangers surface when the 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[1], 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 real hardware, as many of the solutions can be verified using the simulated model of the system. In addition, with the use of the correct methods the result is not only the solution for the problem, but an automation that can generate this solution for a given system. Meaning that even if significant changes occur in the specifications, only minimal amount of modification is required to the code that formulates the 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 the 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 generic `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. The authors have used these technologies before successfully in industrial environment that allowed us to follow out the concept.

3 Development of the software

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

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.

[1] ISO26262, Misra-C, etc.

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.

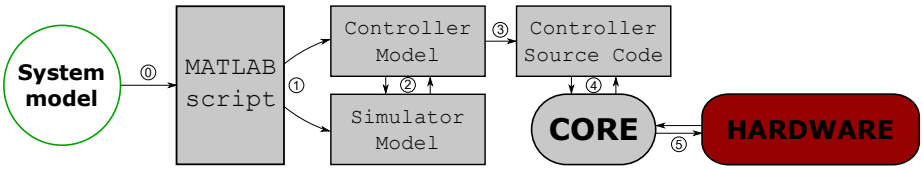

Figure 1: Connections and dependencies

Although in most cases the core unit of a complex software will be an operating system, it is not required. 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 line that guides it through certain checkpoints. Between checkpoints, 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[**?**, sensors]s a number of optoelectronic sensors beneath the front axis or bumper of the car, and a couple of infrared proximity sensors to cover the front and the two sides of the car. Certain teams extended this setup with a camera for image-recognition, a second optosensor array, ultrasound proximity sensors and other methods. We sticked to the basic recommended sensor outfit, and this paper will not consider any other 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.

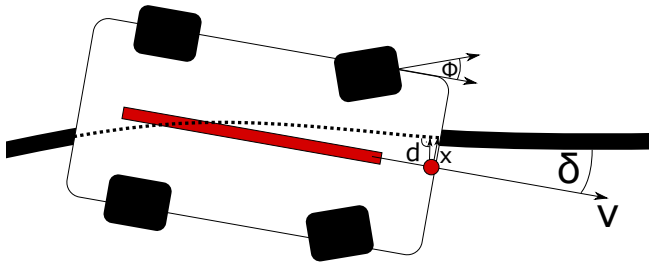

Figure 2: System model

Formulating the system model Knowing the dynamical system equations (1, 2), and parameters (L: wheelbase), we can formulate the **A** (system transition) and **B** (system update) matrices linearly approximated 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[see Appendix A], 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, or Hybrid Linear Controller.

$$A_J = \begin{bmatrix} \frac{\partial \dot{d}}{\partial d} & \frac{\partial \dot{d}}{\partial \delta} \\ \\ \frac{\partial \dot{\delta}}{\partial d} & \frac{\partial \dot{\delta}}{\partial \delta} \end{bmatrix} \tag{4}$$

$$B_J = \begin{bmatrix} \frac{\partial \dot{d}}{\partial \Phi} \\ \\ \frac{\partial \dot{\delta}}{\partial \Phi} \end{bmatrix} \tag{5}$$

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 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.

$$\dot{\underline{x}} = (A - BK) * \underline{x} \tag{6}$$

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 reading we need the *Inverse measurement model*.

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

$$x = \frac{d}{\cos(\delta)} \tag{7}$$

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

$$\hat{d} = x \cdot \cos(\delta) \tag{8}$$

$$\hat{c} = \dot{x} \tag{9}$$

$$\hat{\delta} = \arctan\left(\frac{c}{v}\right) - \Phi \tag{10}$$

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 required.

3.3 Building the simulation environment

When a sufficient 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 model based development environment. 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 controller's point of view. This is the reason why Direct and Inverse Measurement (msment) models are required.

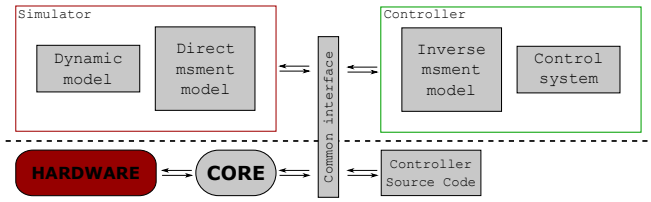

Figure 3: Simulation environment and analogy of deployment

Between the simulator, and 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.

Although `MATLAB` is a dynamic typing language, it is good practice to fix the data type of the signals, especially if the software is developed for an embedded system. The default signal type is *Double*, but it should be overridden to *Single*, or *Integer* to spare the limited processing capabilities of the system.

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 this 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 exact 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 the signal that 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 and the system update matrix ($\underline{\underline{\mathbf{A}}}$) results in a vector filled with the new states. In case of a linear system, $\underline{\underline{\mathbf{A}}}$ is constant. In case of a nonlinear system however, $\underline{\underline{\mathbf{A}}}$ 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 $\underline{\underline{\mathbf{A}}}$ is a nonlinear function of δ.

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

$$J = \begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_n}{\partial x_1} & \cdots & \frac{\partial F_n}{\partial x_n} \end{bmatrix} \tag{11}$$

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.

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 traditional source code writing. Therefore, we encourage the reader to design

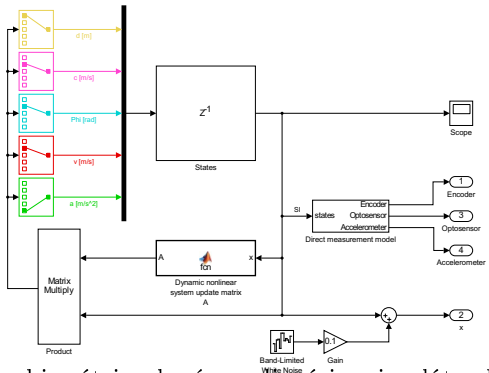

Figure 4: Jacobi-mátrix alapú nemlineáris szimulátor blokkdiagram

his or her own control system based on the supplied information and examples, 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.

Signal processing and state estimation Although basic processing might be required to enhance the raw signal quality before the inverse measurement model, serious processing should be implemented after. If the Jacobian matrices are known, they can be used to create an Extended Kalman Filter (EKF) that serves a dual purpose as a powerful filter and statistically optimal state estimator.

Controller 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 nonzero transition time of the steering servo between states.

3.6 High level control, state machine

`MATLAB` has an extension called `Stateflow` dedicated to the development of state machines. It supports a wide range of features including subcharts, temporal logic and code generation of course, 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 obstacles to perform an action like slalom or automated parking.

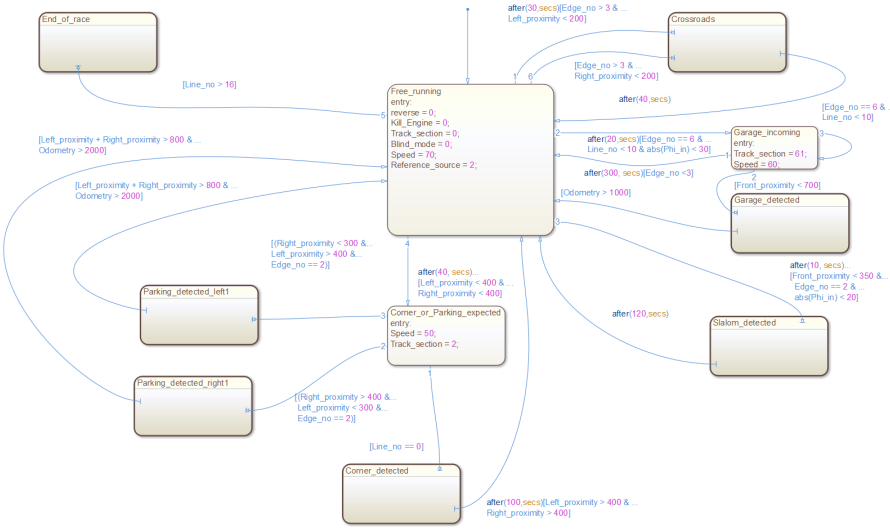

Figure 5: State machine diagram of the obstacle course

3.7 Code generation

If the controller is implemented and tested, it's ready for `C` code generation (or `C++`, `Verilog` és `PLC`). It acts as some sort of a static object, as it has its inner states and member functions, but does not need to be instantiated. Appendix C describes the requited settings of the system model to produce a suitable controller source code. Then find the correct subsystem that acts as the controller, and build the model with the `MATLAB Coder`. In a large system, it's possible to generate separate source code implementations of different subsystems. Though the integration complexity increases, it can provide enhanced functionality, or excellent reuseability[should cite something?].

4 Integration

4.1 Structure of the generated code

If during the code generation *Compact code placement* was used, only four files are generated:

- "system_name".c
- "system_name".h

- rtwtypes.h
- ert_main.c

In the following we assume that the name of the Controller block was "controlsystem".

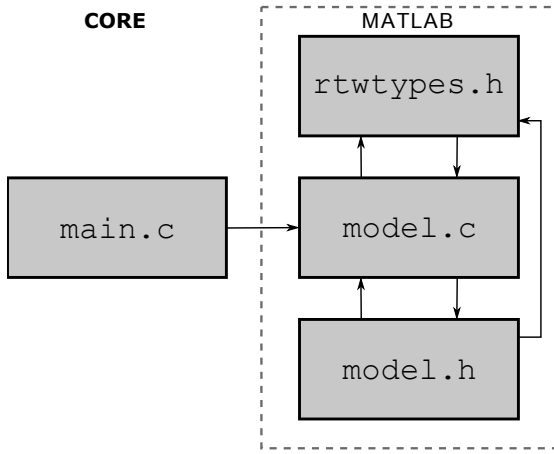

Figure 6: Relationship of the generated files

The functions of the system are defined in the **controlsystem.c** and **controlsystem.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 a perfect equivalent of 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 *controlsystem_initialize()* function. It sets the 0 time in the model and the outputs obtain their initial values. The inputs are handled by the *controlsystem_U* struct. It contains fields each corresponding to a system input, matching it's name and type. If the inputs are set, calling the *controlsystem_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 *controlsystem_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 processing the I/O lines, converting the analogue

signals and sending status information to the supervisor computer over a wireless connection. It can be implemented with the 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.

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 lines are simple, and it handles the reading of the optoelectronic and proximity sensors and their processing on the Analogue to Digital converters of the board.

Setting up the system is quite easy in this point. We only have to include the FreeRTOS source files into our project.[2] The next step is to write some low level driver to read the sensors, and initialize the actuators so this way we can connect these to the control system. It is also necessary to set up the communication between the tasks, so we can read the state of the controller and the other tasks real time.

5 Results and future work

The most important result is the successful application of the method. The system was designed and implemented in `MATLAB`-generated code embedded into 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 performance 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`.

6 Conclusion

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.

[2] A detailed description is available for many processors in the FreeRTOS website: http://www.freertos.org/

Appendix

A Generating the Jacobi matrices

```
% Definition of system equations
d_dot = sin(Delta + Phi) * v;
Delta_dot = v/L * tan(Phi) + Kappa;

% Computing the Jacobi matrices
A_sym = [0      diff(d_dot, Delta);
         0      0                 ];

B_sym = [diff(d_dot, Phi);
         diff(Delta_dot, Phi)];

% Substitution of approximation points
A = double(subs(A_sym, [L, d, Delta, Phi, v],...
    [L_car, 0, 0, 0, 1]));
B = double(subs(B_sym, [L, d, Delta, Phi, v],...
    [L_car, 0, 0, 0, 1]));
```

B Linear full state feedback controller design

```
% Definition of remaining state-space matrices
C = [1 0];
D = 0;
sys = ss(A,B,C,D);

% Controller design
P = [-3 -3];
K = acker(A,B,P);
sys_c = ss(A-B*K, B, C, D);
```

C Preparing MATLAB Coder for code generation

The first step is to set up the solver. Open *Simulation/Configuration parameters* (Ctrl + E) and set the *type* to *Fixed-step* and the *Solver* to *discrete*, in order to avoid having continuous-states in the controller. Next, set the sampling time of the model. Note: Set these preferences right after the creation of the empty Simulink project, because they have great effect on the simulation as well. A model built as a continuous time will not function properly with a discrete solver. Never build the module you plan to use for code generation in continuous time, and always set a discrete solver. Building the whole model like this is recommended, if deployment is intended. Alternatively, if a continuous time simulator is required, place the controller in an **Atomic Subsystem** that allows the special discrete-time treatment of the subsystem inside it.

For the code generation, a MATLAB supported compiler is required. The full list of compilers can be checked on the MathWorks website. The recommendation of the authors is the Windows SDK compiler for Windows. Supported compilers: http://www.mathworks.com/support/compilers/

To set a compiler as active, or check the active compiler, use the `mex -setup` command. The instructions will guide through the settings.

Settings for STM32F4-Discovery Target In **Hardware Implementation** menu set **Device Vendor** to **ARM Compatible**. Set the **Device type** to **ARM Cortex**. In **Code Generation** menu set **System target file** to **ert.tlc** (invokes the embedded coder). In **Interface** menu set **Code Replacement Library** to **GCC ARM Cortex-M3**. In case the generated model contains floating point types, turn on their support. If the support is turned off, but the model contains floating point types, a code generation-time error will appear, pointing to these settings. This is true for other related interface support issues. Turning off the unnecessary or unintended supports is recommended as they point out model errors in an early stage. Useful when designing fixed-point tools.

In **Report** menu it is optional to set the *Create code generation report*. It generates a documentary report of the generated code, containing links from the source code to the model. Useful for debugging. In **Code Generation** menu it is optional to set the **Code optimizations**. The performance will increase for the cost of increased duration of code generation. It is recommended to turn off the generation of a make file and not building the model, because only the source files are used. In **Code placement** it is optional and recommended to set the **Code packaging** to **Compact**. Less source files will be generated and the integration will become somewhat easier.

To generate the source code, right clock on a subsystem or block, then **C/C++ Code** → **Build This Subsystem** (In older MATLAB versions look for Real Time Workshop). The generated source files are placed in the current MATLAB folder.

Summary:

- **MATLAB Command Line** → **mex -setup**
- **Solver** → Fixed step, discrete
- **Hardware Implementation** → Device type → ARM Cortex
- **Code Generation** → System Target File → ert.tlc
- **Interface** → Code Replacement Library → GCC ARM Cortex-M3