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; Real-Time; Control Systems; ...

1 Introduction

The RobonAUT is an annual contest of the Department of Automation and Applied Informatics at the Budapest University of Technology and Economics. Each year many teams, consisting three students each design and build autonomous model cars that battle on an obstacle course, and race each other on a race track. The robots must be completely autonomous, lack any remote control and any kind of external intervention during the race is punished. The team whose car gathers the most points wins the contest in the end[1].

Even with such simple robots, there is 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 "in line" 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.

Often a low-level approach to the problem pays off in smaller 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 and solutions is no easy task. It might present more problems than solve, 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 neglected in the industry for its initial setup complexity, and the high price of design softwares discourage its use even further. Against all these odds though its popularity is growing, but was never considered a way to deal with small projects before.

We wanted to demonstrate that a scaled-down approach is possible using such tools. Because the project is relatively small, the functional software and the operating system responsible for the core services can be integrated manually, but most of the advantages of the methodology can be retained. We used `MATLAB-Simulink` to create a *Rapid Prototyping* environment, where 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 functionalities, designed and verified in a swift visual model-based environment to overcome the challenges the competition presents. The development cycle became shorter, and the syntactic reliability of the resulting functional 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 of the system, 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?) `FreeRTOS` (hard real-time operating 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 that was 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 in scaling up the work force that gave birth to a number of safety standards, internal regulations and coding guidelines[2], and led to a huge amount of overhead in human labour on 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 model. This means, 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 this all sounds very good, 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. Often the usefulness is limited by the lack of hardware-support as well.

3 Development of the software

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.

Figure 1 depicts the system architecture and stages of deployment. A typical system developed in `MATLAB` can be divided to the following objects.

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.

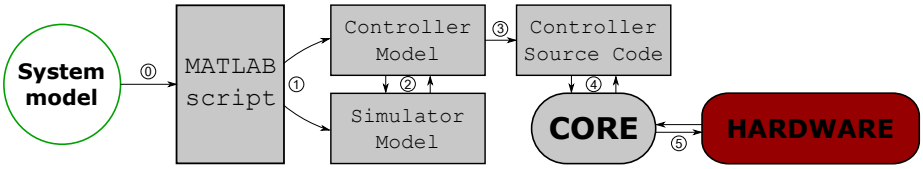

Figure 1: System architecture

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 loadout[**?**, 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 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 loadout. 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 center of the front optical array and the center of the track
δ	Angular error: The angle between the centerline 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 centerline

Illustrating the system can help to visualize the relationships:

Formulating the system model the first step is to implement the system descriptor script. In the following we'll build a steering controller.

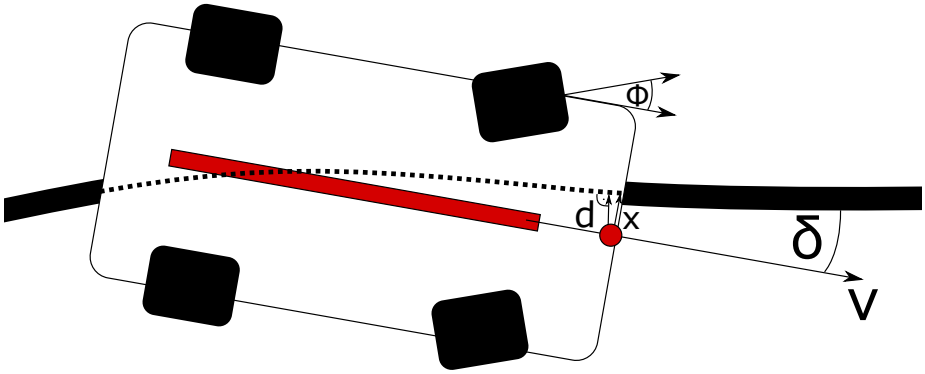

Figure 2: System model

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 the equilibrium ($d = 0; \delta = 0; v = 1$) (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 closer to 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 nonlinear dynamic equations.

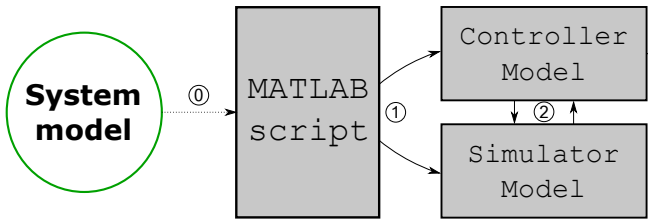

Figure 3: Symbolic design

Furthermore, this way we can hold on to a lot more information that can later be used to generate a nonlinear state observer, Extended Kalman Filter[appendix?], or Hybrid Linear Controller[appendix?].

$$A_J = \begin{bmatrix} \frac{\partial \dot{d}}{\partial d} & \frac{\partial \dot{d}}{\partial \delta} \\ \\ \frac{\partial \dot{\delta}}{\partial d} & \frac{\partial \dot{\delta}}{\partial \delta} \end{bmatrix} \quad (4) \qquad B_J = \begin{bmatrix} \frac{\partial \dot{d}}{\partial \Phi} \\ \\ \frac{\partial \dot{\delta}}{\partial \Phi} \end{bmatrix} \quad (5)$$

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 their effectiveness 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 get the feedback matrix.

If we check the results we can see that the system has been stabilized.

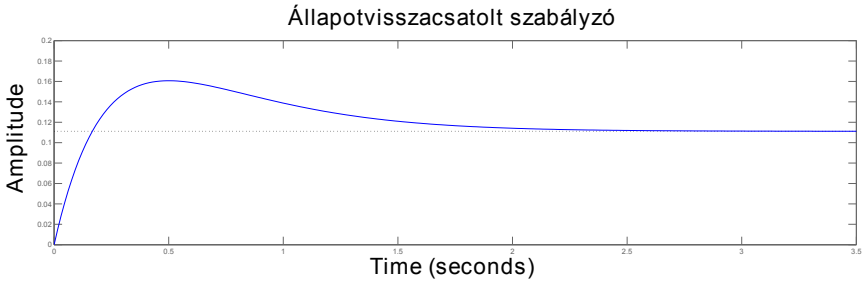

Figure 4: Instabil rendszer állapotvisszacsatolt stabilizációja

Direct and inverse measurement models Unfortunately, a full-state feedback loop can rarely be realized directly, and the controller we have just created would not be able to control the actual system. Certain states can not be observed directly, and can only be estimated by a state observer. Simulating the sensor readings based on the state variables can be done 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 geometrical projection to the optosensor array:

$$x = \frac{d}{\cos(\delta)} \quad (6)$$

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

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

$$\hat{c} = \dot{x} \tag{8}$$

$$\hat{\delta} = \arctan\left(\frac{c}{v}\right) - \Phi \tag{9}$$

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

Note: δ can be directly measured if the robot is equipped with multiple optosensor arrays.

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 go through with the model based design. To test the behavior of the system, the controller and the simulator must be implemented in `Simulink` as well. Figure 5 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 must be designed.

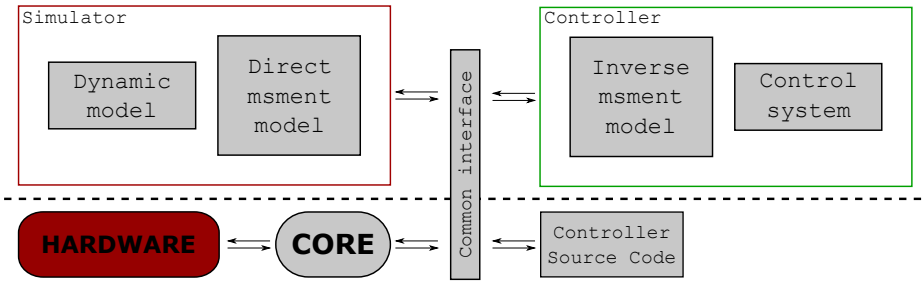

Figure 5: Simulation environment and analogy of deployment

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.

Simulator The controller that was designed earlier is based on the linear approximation of the system, but the simulator must always represent the full

nonlinear 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 6 shows the simulation of the dynamics designed to test the control system (without inputs). A common programing 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 (e.g. it generates the signal of each individual optosensor and puts them into a vector (same as a 1-dimensional array), just like the signal that would be 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.

$$J = \begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_n}{\partial x_1} & \cdots & \frac{\partial F_n}{\partial x_n} \end{bmatrix} \tag{10}$$

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.

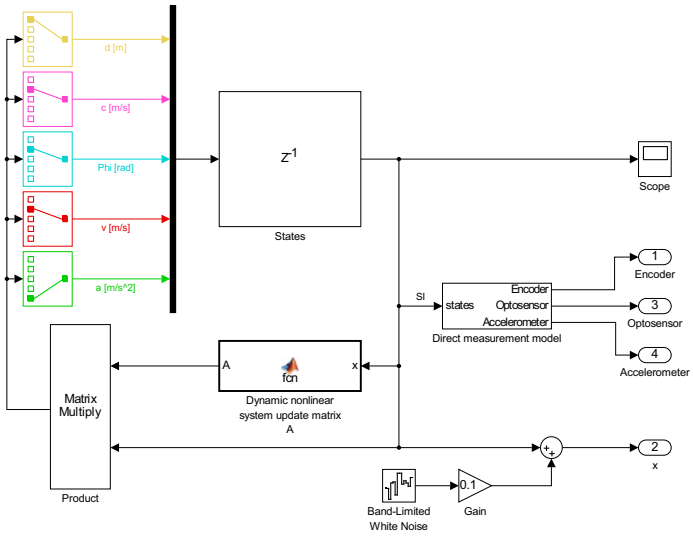

Figure 6: Jacobi-mátrix alapú nemlineáris szimulátor blokkdiagram

3.4 Implementation of the control system

Az irányítás tetszőlegesen bonyolult lehet, a RobonAUT-hoz készült megoldás 4 fő kritikus (működéshez elengedhetetlen) részre bontható:

Inverz mérési modell A korábban vázolt inverz mérési modell megvalósítása, célja a **visszacsatolt, mérhető állapotok** előállítása a szenzorok jelei alapján, tehát **priori információk nélkül**. Mivel a verseny során többféle jelre kellett szabályozni (vonaldetektálás, bal/jobb éldetektálás, bal/jobb távolságérzékelő szenzor jele), ezekre mind létre kell hozni az inverz mérési modellt, majd a megfelelő visszacsatolt állapotot adni tovább a jelfeldolgozónak.

Jelfeldolgozó és állapotbecslő Az inverz mérési modell által előállított **visszacsatolt állapotokat** dolgozza fel **priori információk** segítségével. Ez a mi esetünkben egy Kálmán-szűrőt jelentett, ahol a mérési jel a visszacsatolt állapot, a priori információ pedig a rendszer modellje. Bár a Kálmán-szűrő támogatja az inverz mérési modell közvetlen integrálását a szűrőbe, ezt a megoldást nem találtam célszerűnek, mivel a szenzorsor jelvektorából a vonal pozíciójele nem állítható elő zárt mátrixos alakban, valamint többféle jelentősen eltérő lehetséges bemenet között kell kapcsolni.

Szabályzó A szabályzó a szkript alapján számított paraméterek alapján implementálható, ez függ a választott szabályzó típusától és a tervezés módjától.

Magas szintű irányítás, állapotgép A magas szintű irányítás felel a szabályzás és állapotgép alapján történő vezérlés közötti kapcsolásért és a szabályzó bemeneti jelének kiválasztásáért, az aktuális állapot szerint. Ezenkívül referenciajeleket határozhat meg a sebességnek és a kormányszabályzónak.

3.5 Preparing MATLAB Coder for code generation

Az elkészült szabályzóból `C` (vagy `C++`, `Verilog` és `PLC`) kódot generálhatunk, melyet beágyazhatunk a rendszerbe. A generált kód egyfajta statikus osztályként működik, vannak belső tárolói és tagfüggvényei, de nem szükséges a példányosítása.

Az első lépés a kódgenerálás előkészítéséhez a megfelelő **solver** beálítása. Ezt a **Simulation/Configuration parameters... (Ctrl + E)** menüpontban tehetjük meg a **Solver** menüpontban. A típust Fixed-step-re kell állítani, a solvert pedig discrete-re, hogy ne legyenek folytonos állapotok a modellünkben. Ezután ne felejtsük el megadni a mintavételi idejét a rendszernek, másodpercben. Ezt a beállítást célszerű már a modell létrehozásakor, a legelső lépésként elvégezni, ugyanis egy folytonos környezetben épített modell nem feltétlenül fog működni diszkrét solverrel! (Alternatív módon **Atomic Subsystem** blokkba helyezhetjük a fordítani kívánt rendszert, amennyiben mindenképpen folytonos idejű szimulátorral szeretnénk dolgozni. Ekkor nem szükséges a solver módosítása, viszont gondoskodni kell a mintavételi idő váltásról.)

A tényleges kódgeneráláshoz szükség van egy MATLAB által támogatott `C/C++` compilerre is. A teljes lista megtekinthető a MathWorks weboldalán.[1]

Target beállítása A kódgenerálásra millióféle különböző beállítás létezik, ezek közül az STM32F4-Discovery kártyához tartozót mutatjuk be, mivel ez volt a célplatform a RobonAUT verseny során.

A **Hardware Implementation** menüpontban a **Device Vendor** legördülő listát állítsuk **ARM Compatible**-re, a **Device type**-ot pedig **ARM Cortex**-re. A **Code Generation** menüpontban állítsuk át a **System target file**-t **ert.tlc**-re (ezzel az általános MATLAB-Embedded codert hívjuk meg). Az **Interface** menüpontban a **Code Replacement Library**-t állítsuk **GCC ARM Cortex-M3**-re. Amennyiben lebegőpontos számábrázolást is használunk a programban, kapcsoljuk be ezeknek a támogatását. Kihasználatlanul viszont nem javasolt mindent bekapcsolni, mivel csak felesleges típusdefiníciók jönnek létre. Ha valamit elfelejtettünk beállítani, Build közben hibajelzéssel ide fog visszairányítani a MATLAB. A **Report** menüpontban bekapcsolható a "Create code generation report", ami egy dokumentációt is generál a kód mellé. A **Code Generation** menüponthoz visszatérve bekapcsolhatjuk az optimalizációkat is, így javíthatunk a kód futásteljesítményén, valamint a generálás sebességén növelhetjük, ha nem kérünk make-filet, illetve buildet egyből, hanem csak a kódot állítjuk elő (úgyis csak arra van szükségünk itt). A kezelhetőséget

[1] http://www.mathworks.com/support/compilers/

javítja, ha a **Code placement**-ben a **Code packaging**-et **Compact**-ra állítjuk, így kevesebb forrásfájlt generál, és könnyebb kezelni, ha csak egy generált rendszerünk van.

Tl;dr

- **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
- **MATLAB Command Line** → **mex -setup**

A megfelelő beállítás után a Simulink modellben keressük meg azt a blokkot, melyből kódot szeretnénk generálni. Egy nagy rendszerben akár több részben is lehet kódot generálni, pl. kapcsolható működés, vagy eltérő mintavételi idő esetén hasznos. Jelen esetben az Irányító rendszer blokkját szeretnénk felhasználni, melynek bemenete a szenzoroktól kapott közvetlen jel, kimenete pedig a szervójel. A blokkon jobb kattintás, majd **C/C++ Code** → **Build This Subsystem** (Régebbi MATLAB verziókban MATLAB Coder, illetve Real Time Workshop (RTW) menüpontokat kell keresni). A generált fileok a `MATLAB Current Working Fo` be kerülnek, nem a modell mappájába.

4 Integration

4.1 A generált kód felépítése

Amennyiben compact code placement beállítással generáltuk le a fileokat, csupán 4 számunkra hasznos file keletkezik:

- "subsystem neve".c
- "subsystem neve".h
- rtwtypes.h
- ert_main.c

A továbbiakban feltételezzük, hogy a generált subsystem neve "Controller" volt, az egyszerűbb olvashatóságért.

A **Controller.c** és **Controller.h** fileokban van definiálva a rendszer működése. A kódból hozzávetőlegesen kiolvasható a modell működése. A **rtwtypes.h** fileban kerülnek definiálásra a rendszer által használt típusok, az **ert_main.c** pedig bemutatja a használatukat.

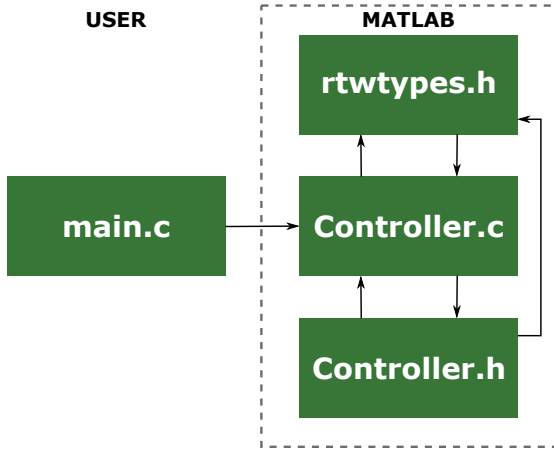

Figure 7: A generált fileok kapcsolata

4.2 Simulink modell kezelése C kódban

A generált kód tökéletes megfelelője a Simulinkben futtatott modellnek. A generált fileok egy objektumot definiálnak, saját típusokkal és tagfüggvényekkel.

A rendszer inicializálását a Controller_initialize() függvénnyel tehetjük meg. Ez a modell futásának elídításának felel meg, beállítja a 0 időpillanatot a rendszerben, a kimenetek felveszik az alapértelmezett értéküket. A rendszerrel csupán a be és kimeneti interfészen keresztül kommunikálhatunk, valamint hatást gyakorolhatunk rá a `Step` függvény meghívási idejének változtatásával (ezt nem részletezzük, és ne is nagyon erőltessük). A bemeneti interfészt a Controller_U struktúra implementálja, ennek a változónak a tagjai felelnek meg egy-egy bemeneti jelnek.[2]

Miután beállítottuk a bemeneteket, a Controller_step() függvény meghívásával **léptethetjük** a modellt. A lépés megegyezik a generált rendszer **1 mintavételi idejű** lépésével. A Step függvény lefutása közben frissülnek a rendszer belső állapotai, valamint Controller_Y-ban a kimenetek. A kimenetet hasonlóléppen állítja elő a rendszer, Controller_Y struktúrában tárolva.

A bemenetek megadása, léptetés, kimenet kiolvasása az az elemi lépéssorozat, melyet rendszeresen **időzítve** végrehajtunk. Ha a modellnek időtől függő belső állapotai is vannak[3], **pontos időzítéssel** kell biztosítanunk a periodikus meghívást. Ha a rendszer túlnyomó része MATLAB-ban készült, ez csupán timer időzítővel is biztosítható, ám ha más feladatokat is el kell látnia a rendszernek, valószínűleg egy Hard Real-Time operációs rendszerre is szükségünk lesz.

Rögtön felhívnám a figyelmet, ha eddig nem vált volna nyilvánvalóvá, hogy a generált forrásfileok közül **csak** az ert_main.c-t szabad módosítani. Ha úgy érezzük, hogy a többi generált kódba kell kézzel belenyúlni, akkor valamit el-

[2] Többdimenziós jel esetén tömbként történik az átadás. 2 vagy több dimenzió esetén ne feledjük, hogy a MATLAB **oszlopfolytonosan** kezeli a tömböket.

[3] Ez minden esetben igaz, P szabályzónál bonyolultabb irányítás esetében

rontottunk. Továbbá meg lehet találni a megfelelő megfeleltetéseket a generált kód és a simulink modell között, de ez többnyire felesleges és **megbízhatatlan**. A legjobb fekete dobozként tekinteni a rendszerre, ha pedig valamilyen belső változóra szükség van debuggoláshoz, vegyük a fáradságot és vezessük ki kimenetre[4].

4.3 Példarendszer integrációja periodikus meghívással

A példarendszerünk bemenete a vonal pozíciója, kimenete pedig a kívánt kormányszög. A főprogramunkból a következőképpen tudjuk meghívni a modellt:

Ezt a kódot kell egy olyan függvénybe beletennünk, melynek tudjuk biztosítani a periodikus meghívását.

A vonal pozíciót feldolgozhatjuk C-ben is akár, de minek, ha úgyis köré építünk egy MATLAB rendszert? Ehhez pointerrel tudjuk átadni az adattömböt, Simulinkben pedig vektor bemenetet kell beállítanunk.

5 Outlook and prospects

A MATLAB 2013b verziójától elérhető direkt hardware-támogatás az STM32F4 Discovery fejlesztőkártyához, melyet a MATLAB Hardware Support oldaláról tölthetünk le. A támogatás segítségével villámgyorsan tesztelhetjük az elkészített szoftvert, soros kábel segítségével pedig akár Processor-in-the-Loop tesztet is végrehajthatunk[**?**,].

5.1 Gyors Prototípustervezés

A RobonAUT elsősorban gyors prototípustervezési munkát igényel. Rövid idő alatt, minél hatékonyabb párhuzamosítással és a lehető legkevesebb teszteléssel kell sok funkcióval rendelkező, többé-kevésbé megbízható rendszert építeni. A magas és az alacsony szintű irányítás fejlesztése teljesen párhuzamosan zajlik, ha megfelelően kiaknázzuk a lehetőségeket.

Példa Készítsünk Simulinkben egy LED-villogtató programot 5 perc alatt!

Miután feltelepítettük a support package-et[5], hozzunk létre egy új Simulink modellt, majd húzzuk be az **Embedded Coder Support Package for STMF4-Discovery Board** library-ből a szükséges blokkokat. Az ábrán látható módon állítuk össze a kapcsolást, majd konfiguráljuk a modellt az 1. részben leírtakhoz hasonlóan, de most a **Code Generation** menüben **Target Hardware**-nek válasszuk az `STM32F4-Discovery`-t, valamint ne csak kódot generáljunk (Pipa ki a **Generate code only** checkbox-ból).[6]

[4] A Goto/From simulink blokkpárral ezt elegánsan megtehetjük, a vezetékek összekuszálása nélkül

[5] Ekkor indul a stopper

[6] Cheat sheet újoncoknak: A `Constant` blokk signal type-ját `boolean`-re, a `Pulse Generator` Pulse type-ját pedig `Sample based`-re kell állítani. A blokkokról pedig senki sem tudja, hogy

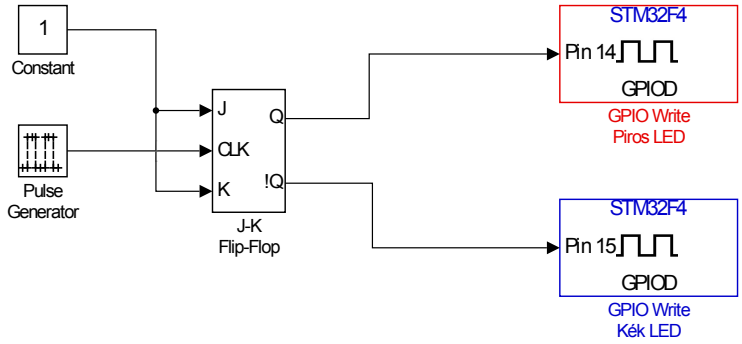

Figure 8: LED villogtató Simulink-modell

A teljes rendszert a **Code** menü **C/C++ Code** menüpontjából tudjuk buildelni. Ha mindent jól csináltunk, a jutalmunk a végén egy `.hex`, egy `.elf` és egy `.bin` file lesz a Working Directory-ben. Ezután az `STM ST-LINK Utility` segítségével felprogramozhatjuk a kártyát a generált fileok segítségével. Természetesen nem csak a ledeket tudjuk villogtatni, hanem az összes GPIO-t, gombot és analóg I/O-t kezelni tudjuk tetszőlegesen bonyolult modell köré építve. A GPIO Read és Write blokkok beállításához az `ST-Microelectronics` megfelelő segédletet nyújt[?].

Hasonló elv alapján épült fel egy standard servo jeleket feldolgozó projekt is, melyet egy távirányítós autó végfokozatának PWM-es vezérlésére használtam.A Simulink modell szintén letölthető `innen`, és mélyebb betekintést ad a modell alapú tervezés nyújtotta lehetőségekbe. Egy ilyen feladat elkészítése is inkább munkapercekben, mint -órákban mérhető. Korábban említettük, hogy akár PIL tesztelésre is lehetőség van. Ebben a dokumentumban erre az alkalmazási területre nem térünk ki, de egy későbbi bővített kiadásban előfordulhat, ha igény mutatkozik a témára.

6 Konklúzió

[3] Bízunk benne, hogy sikerült meghozni a kedvet az `STM32F4-Discovery` fejlesztőkártya kreatív használatához, és népszerűsíthettünk egy olyan fejlesztési irányzatot, ami a rendszerszemléletet helyezi a végeláthatatlan kódolás elé.

---

melyik almenüben vannak, érdemes használni a keresőt. Ne felejtsük el a GPIO blokkok konfigurálását sem! Ha semmiképpen sem akar működni, akkor a kész modell letölthető `innen`.

Acknowledgments

The author would like to express his thanks to István Vajk [7] for his support as a scientific advisor. This work has been supported by the ... [8]

A Generating the Jacobi matrix

```
% 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 Call sw

```
    /* Pass inputs */
    Controller_U.Position = line_position;

    /* Update model */
    Controller_step();

    /* Receive outputs */
    servo_position = Controller_Y.Servo;
```

[7] Please mention the name of your advisor in the Acknowledgements section.
[8] Please mention the institution or organization that has supported your research work.

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

[3] W. Weinrebe, A. Kuijpers, I. Klaucke, and M. Fink, "Multibeam bathymetry surveys in fjords and coastal areas of west-greenland," *AGU Fall Meeting Abstracts*, p. A1152, Dec 2009. Provided by the SAO/NASA Astrophysics Data System.