

# Gyors prototípustervezés STM32F4-Discovery fejlesztőkártyán Simulink segítségével

Csorvási Gábor

Automatizálási és

Alkalmazott Informatikai Tanszék

Budapesti Műszaki és Gazdaságtudományi Egyetem

Budapest, Magyarország

Email: csorvagep@gmail.com

Fodor Attila

Automatizálási és

Alkalmazott Informatikai Tanszék

Budapesti Műszaki és Gazdaságtudományi Egyetem

Budapest, Magyarország

Email: attila.fodor.89@gmail.com

**Abstract**—Methodology description and general technological documentation of the development of an Autonomous Ground Vehicle for the RobonAUT 2014 competition at Budapest University of Technology.

## I. BEVEZETÉS

A dokumentum abból a célból jött létre, hogy segítséget nyújtson STM32F4-Discovery board alapú projektek fejlesztéséhez. A szükséges lépések nagyrészt igyekszünk bemutatni valós alkalmazások segítségével, melyeknek túlnyomó többsége a 2014-es **RobonAUT** versenyre történő felkészülés közben került kivitelezésre. A problémák számunkra is újak voltak, így nem ígérhetjük, hogy a legjobb megoldásokat fogjuk prezentálni itt, de a bemutatott eljárásról kijelenthetjük, hogy hasznosnak és megbízhatónak bizonyult.

Az írást három fő részre osztottuk, először bemutatásra kerülnek a MATLAB-Simulink modell alapú tervezés lépései és a fordítható kód generálása. Ezután a generált kód integrálására és élesztésére térünk rá FreeRTOS operációs rendszer alatt, legvégül pedig valós gyors prototípustervezési technikák kerülnek bemutatásra, a MATLAB közvetlen STM32F4-Discovery hardware támogatását felhasználva.

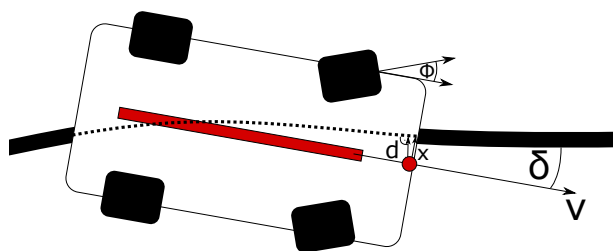


Fig. 1. A probléma sematikus ábrája felülnézetből

A **RobonAUT** verseny során egy olyan robot szoftverét és hardverét kell elkészíteni, amely lehetővé teszi a távirányítós autóból átalakított platformot, hogy egy ismeretlen pályán végighaladjon, és ott akadályokat teljesítsen, a szabályzatban meghatározott módon. A továbbiakban csak a szoftver környezettel foglalkozunk és feltételezzük, hogy megfelelő logikai jelek zajjal terheltlen bár, de rendelkezésre állnak a szenzorokból, illetve az autó várakozásainknak megfelelően reagál a kimeneti jelekre, pl. a kormánysszög beállítására.

A dokumentumban próbáljuk a szakirodalomban elterjedt jelrendszert alkalmazni, de az alábbi táblázat segítségével magyarázzuk a rendszeresen előforduló jelöléseket és összefüggéseket. Amennyiben valami hiányzik innen, az a szöveghelyeztetben kerül definiálásra.

$d$	Pozícióhiba: Az első optikai szenzor sor középpontjának távolsága a vonaltól
$\delta$	Szöghiba: Az első optikai szenzorsorra merőleges egyenes (középvonal) és a pályavonal érintője által bezárt szög
$\Phi$	Kormánysszög: A kerekek síkjainak a középvonallal bezárt szögeinek átlaga, Ackermann-kormányzás szerint
$\kappa$	A pálya pillanatnyi görbülete ( $1/R$ )
$x$	Vonalpozíció: az első optikai szenzor sor által érzékelt vonalak középpontjának előjeles távolsága a szenzorsoron a középponttól mérve.
$c$	Vonalsebesség: $x$ első deriváltja, a vonalpozíció mozgásának sebessége
$v$	Sebesség: Az autó pillanatnyi sebessége a középvonal mentén

## II. MODELL ALAPÚ FEJLESZTÉS

A modell alapú fejlesztés a probléma vizuális, rendszerszintű megközelítésén alapul. Segítségével a fejlesztés során a hangsúly a technológiák implementálásáról átkerül a problémák gyökereinek feltárására, és matematikai megoldások keresésére. A fejlesztés során csak kevés kapcsolat van a tényleges rendszerrel, mivel a megoldások helyességét az eszközről alkotott modell segítségével, szimulációval ellenőrizzük, illetve egy-egy mérföldkő elérésekor elég megbizonyosodni arról, hogy a rendszer valóban jól működik a valóságban is.

Fontos kiemelni, hogy a megfelelő módszertanú fejlesztés eredmény nem csak a megoldás egy adott feladatra, hanem egy automata elkészítése, mely a feladat specifikációjának (józan kereteken belüli) ismert megváltozása esetén is módosítás nélkül elő tudja állítani a megoldást. Ennek a tulajdonságnak a hasznosságát nem lehet túlhangsúlyozni, mivel a tervezés indulásának pillanatában ritkán ismert pontosan az irányítani kívánt végleges rendszer, különösen gyors prototípustervezés során.

Kijelenthető, hogy a modell alapú fejlesztés irányelveit megfelelően követve, szimulációs környezetek intenzív használatával rendkívül gyorsan lehet fejleszteni, a modellek kellően pontos ismerete (illetve az ismert modell pontosságának megfelelő becslése) esetén.

#### A. Szoftverkörnyezet

A **RobonAUT 2014** versenyre az irányítást és állapotgépet MATLAB és Simulink segítségével implementáltuk. A választásunk azért esett a MathWorks termékcsaládjára, mivel a Simulink segítségével hatékonyan lehet szimulációs környezetben tesztelni, valamint támogatja az általános C kód generálását. Ezt később közvetlenül felhasználhatjuk tetszőleges hardveren, mely a FreeRTOS (vagy egyéb, hasonló) operációs rendszer futtatására képes.

#### B. Tervezési fázis

Az irányító rendszer 3 részre osztott logikai elrendezést követ. Először a MATLAB Script-ek segítségével létrehozhatjuk azt az automatikát, ami a modell megfelelő paramétereit alapján generálja a szimulációs környezetet ill. paramétereit (rendszermodell), kiszámítja az irányító rendszert és annak működését befolyásoló tényezőket, valamint tesztelés céljából bemeneteket és környezeti hatásokat generálhatunk a szimulációs ellenőrzés céljából.

a) *Rendszermodell definiálása:* Ez a folyamat általában az első lépés mindenféle rendszer vagy irányítás tervezésekor. Az összefüggéseket és kezdeti, becsült paramétereit megadva létrehozhatunk egy modellt, mellyel később dolgozhatunk. A továbbiakban a sebességfüggetlen vonalkövető szabályzón demonstráljuk a lépéseket.

A dinamikai összefüggéseket (??, ??), és a rendszer paramétereit (L: autó tengelytávolsága) ismerve felírhatjuk az **A** (állapotátmeneti) (??) és **B** (állapotfrissítési) (??) mátrixát az egyensúlyi pont körül ( $d = 0; \delta = 0; v = 1$ ) linearizálva.

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

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

$$\begin{bmatrix} \dot{d} \\ \dot{\delta} \end{bmatrix} = \begin{bmatrix} 0 & \delta \cdot v \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \Phi \\ \kappa \end{bmatrix} \quad (3) \quad \begin{bmatrix} \dot{d} \\ \dot{\delta} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ \frac{v}{L} & 1 \end{bmatrix} \begin{bmatrix} \Phi \\ \kappa \end{bmatrix} \quad (4)$$

A rendszermodell közvetlen megadása helyett szemléletileg jobb megoldást kapunk a MATLAB Symbolic toolbox használatával. A szimbolikus matematikát támogató szoftvercsomag segítségével automatikusan generálhatjuk a rendszer Jacobi-mátrixait, így leszűkíthetjük a program bemeneteit a rendszer paramétereire és nemlineáris összefüggéseire. További előny, hogy lényegesen több információt tartunk a kezünkben így, amivel például nemlineáris állapotbecslőt is készíthetünk.

$$A_J = \begin{bmatrix} \frac{\partial \dot{d}}{\partial d} & \frac{\partial \dot{d}}{\partial \delta} \\ \frac{\partial \dot{\delta}}{\partial d} & \frac{\partial \dot{\delta}}{\partial \delta} \end{bmatrix} \quad (5) \quad B_J = \begin{bmatrix} \frac{\partial \dot{d}}{\partial \Phi} \\ \frac{\partial \dot{\delta}}{\partial \Phi} \end{bmatrix} \quad (6)$$

```
% 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) *Szabályzó tervezése:* A rendszer modelljének ismerete után megtervezhetjük a szabályzót. A szkript nem képezi a program szerves részét, ezért akár többféle szabályzót is kipróbálhatunk itt a PID-től az adaptív Fuzzy irányításig (és tovább), melyeknek a hatékonyságát összehasonlíthatjuk még a tervezési fázisban, így a legmegfelelőbb szabályzóval dolgozhatunk tovább.

A rendszer állapotteres alakját ismerve az egyik legegyszerűbb megoldás a stabilizálás teljes állapot-visszacsatolásos szabályzó segítségével. Ehhez felvesszük a megfelelő **C** és **D** mátrixokat a szabályozni kívánt kimenet előállításához, majd az Ackermann-képlettel számított **K** pólusáthelyező mátrixon keresztül visszacsatoljuk a rendszert.

```
% 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);
```

Az eredményeket leellenőrizve ábrázoljuk az eredetileg instabil rendszer választát, és elégedetten hátradőlhetünk, mert működik.

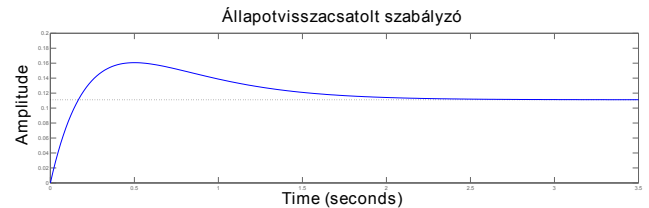


Fig. 2. Instabil rendszer állapotvisszacsatolt stabilizációja

c) *Direkt és inverz mérési modell:* A rendszer viselkedésének ismerete és egy erre létrehozott szabályzó a legritkább esetben elég a hatékony szabályzáshoz. Bizonyos állapotok csak korlátozottan, esetleg egyáltalán nem mérhetők, így állapotmegfigyelővel kell előállítanunk. További probléma, hogy a legtöbb állapotot is csak közvetve tudjuk mérni, zajjal terhelve. A szenzorok érzékeléseinek előállása az állapotváltozók függvényében a **Direkt mérési modell**, illetve a megfigyelhető állapotok becslése a szenzorok jelei alapján az **Inverz mérési modell**.

A kormány szabályzás esetében az optikai szenzorok jelei egyszerűen szimulálhatók a rendszer állapotait és geometriai elrendezését ismerve. Az optikai szenzor mentén a vonal pozícióját a szenzorra való vetítéssel kaphatjuk meg:

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

A direkt modell és a rendszermodell ismeretével pedig előállítható az indirekt modell is:

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

$$\hat{c} = \dot{x} \quad (9)$$

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

A direkt modell általában csak a szimuláció részét képezi, de a szenzorok jeleinek hitelességvizsgálatát is elvégezhetjük vele futási időben, amennyiben szenzorfüziónál állítjuk elő a becsült állapotokat. Az inverz modell az irányító rendszer legelső fokozata, a szimulációban ritkán indokolt a felhasználása.

Megjegyzendő, hogy  $\delta$  közvetlenül is mérhető, ha két optikai szenzort helyezünk el az autón.

### C. Szimulációs környezet kiépítése

A megfelelő modell és szabályzó megtervezése után elkezdhetjük a szimulációs környezetet összeállítani. Egy egyszerűbb rendszer esetében akár közvetlenül implementálni lehetne a szabályzót C-ben, ám ha a rendszer összetettebb, jobban járunk ha a modell alapú fejlesztés elvét követjük. A RobonAUT versenyre az autó irányítórendszerének implementálásához ezt az utat választottuk. A döntésben szerepet játszott a MATLAB kiváló integrált állapotgép és szimuláció támogatása, a földrajzi távolság a tesztkörnyezet (Truggy) és a lusta fejlesztő lakhelye között, illetve a technológiai kihívás vonzása is.

*d) Felépítés:* A rendszert két fő logikai részre bonthatjuk, az **Irányító rendszerre** és a **Szimulátorra**, illetve egy harmadik blokk interfészt biztosít e kettő között. A MATLAB (így a Simulink is) gyengén típusos nyelv, de lehetőség van a típusok erős állítására is. Ez beágyazott rendszereknél különösen hasznos, mivel a MATLAB egészen meglepő típusokat tud előállítani magának, ami nem előnyös sem a futási sebesség, sem a kézbentartathóság szempontjából, különösen ha fixpontos számábrázolással szeretnénk implementálni a szabályzót.

*e) Szimulátor:* A megállapított rendszermodell alapján felépíthetjük a szimulátort. Hiába terveztük a szabályzót a linearizált modellhez, a szimulátornak mindig a legelső lépésben meghatározott, potenciálisan nemlineáris összefüggések alapján kell felépülnie, különben csak magunkat csapjuk be a hibás szimulációval, és az egész technológia nem ér semmit. A nemlineáris rendszerek felépítéséhez kiváló segédletet nyújt a Szabályozástechnika tárgya 5-6. gyakorlata, illetve annak felkészülési anyaga[?, p. 319-354], így erre nem térünk ki. Amennyiben a rendszermodellt a Jacobi mátrixok segítségével generáltuk, célszerűbb közvetlenül azt

felhasználni a szimulációban a blokkokból felépített rendszer helyett, így nem csak paraméter-, hanem rendszerszintű változás esetén sem kell kézzel utólag módosítani a szimulátort.

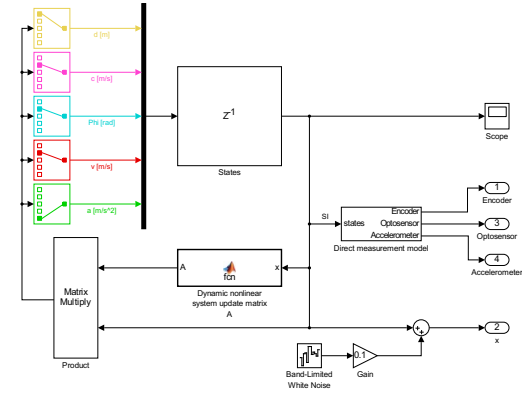


Fig. 3. Jacobi-mátrix alapú nemlineáris szimulátor blokkdiagram

Ne feledjük, a szimulátor kimenete nem csupán az állapot, hanem a direkt mérési modell által előállított szimulált szenzorjelek, melyeket mesterségesen zajjal is terhelhetünk, hogy eggyel kevesebb meglepetés érjen a tényleges teszteléskor.

### D. Irányító rendszer implementálása

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

*f) 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 jelle kellett szabályozni (vonaldetektálás, bal/jobbs éldetektálás, bal/jobbs távolságérzékelő szenzor jele), ezekre mind létre kell hozni az inverz mérési modellt, majd a megfelelő visszacsatolt állapotot adni tovább a jelfeldolgozónak.

*g) 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.

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

*i) 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 kormány szabályzóknak.

### E. MATLAB Coder beállítása

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** beállítása. 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 egyáltalán nem legyenek folytonos állapotok a modellünkben. Ezután ne felejtjük 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.<sup>1</sup>

j) *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 számbábrázolást is használunk a programban, kapcsoljuk be ezeknek a támogatását. Kihaszíratlanul 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** menüpontba 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 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.

k) *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**

### F. Kódgenerálás

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 Folder-be kerülnek, nem a generált rendszer mellé.

## III. INTEGRÁCIÓ

### A. 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.

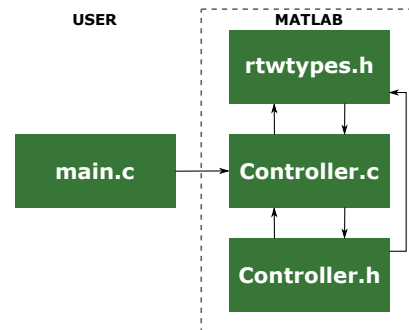


Fig. 4. A generált fileok kapcsolata

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

### B. 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 előidézésének

<sup>1</sup><http://www.mathworks.com/support/compilers/>



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.<sup>2</sup>

Miután beállítottuk a bemeneteket, a `Controller_step()` függvény meghívásával **léptethetjük** a modellt. A lépés meg-e egyezik 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 hasonlóképpen állítja elő a rendszer, `Controller_Y` struktúrában tárolva.

A bemenetek megadása, léptetés, kimenet kiolvasása az az elemi lépéssorozat, melyet rendszeresen **időzítva** végrehajtunk. Ha a modellnek időtől függő belső állapotai is vannak<sup>3</sup>, **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.

### C. A FreeRTOS operációs rendszer

A FreeRTOS egy szabadon felhasználható Real-Time operációs rendszer. Módosított GPL licence alatt fejlesztik, miszerint üzleti felhasználása esetén szerepelhet zárt forráskódú projektben, viszont a kernelt nyílt forráskódúvá kell tenni, és kérésre kiadni.<sup>4</sup>

A FreeRTOS segítségével gyors és hatékony futást érhetünk el, és biztosíthatjuk a pontos időzítést, a többi adatgyűjtő taszk mellett is.

### D. Példarendszer integrációja periodikus meghívással

A következő példa szemlélteti a generált kód integrálásának módját egy egyszerű kormányvezérlő alkalmazással.

## IV. HARDWARE-TÁMOGATÁS

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.

### A. Gyors Prototípustervezés

A RoboAUT 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.

<sup>2</sup>Többdimenziós jel esetén tömbként történik az átadás. 2 vagy több dimenzió esetén ne feledjük, hogy a MATLAB **oszlopfolytonosan** kezeli a tömböket.

<sup>3</sup>Ez minden esetben igaz, P szabályozónál bonyolultabb irányítás esetében

<sup>4</sup>Létezik támogatott változata is, (...)

l) *Példa:* Készítsünk Simulinkben egy LED-villogtató programot 5 perc alatt!

Miután feltelepítettük a support package-et<sup>5</sup>, hozzunk létre egy új Simulink modellt, majd húzzuk be az **Embedded Coder Support Package for STM32F4-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).<sup>6</sup>

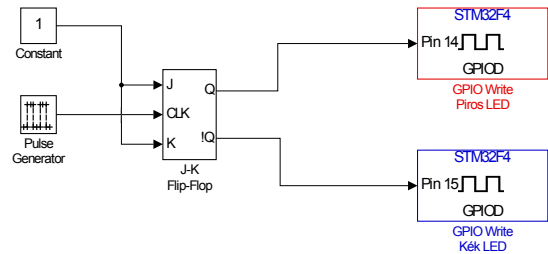


Fig. 5. LED villogtató Simulink-modell

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 felforraszthatjuk 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[?]. 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.

## V. KONKLÚZIÓ

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.

## KÖSZÖNET

(...)

## REFERENCES

- [1] Lantos-Kiss-Harmati: , Szabályozástechnika gyakorlatok, Budapest, BME-IIT, 2007.
- [2] STMicroelectronics: STM32F4-Discovery User Manual, 2014 January

<sup>5</sup>Ekkor indul a stopper

<sup>6</sup>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 melyik almenüben vannak, érdemes használni a keresőt. Ne felejtjük el a GPIO blokkok konfigurálását sem! Ha semmiképpen sem akar működni, akkor a kész modell letölthető innen.