



MISKOLCI EGYETEM
GÉPÉSZMÉRNÖKI ÉS INFORMATIKAI KAR



TUDOMÁNYOS DIÁKKÖRI DOLGOZAT

Párhuzamos parkolás megvalósítása CompactRIO platformon

Forgács Zsófia
III. éves villamosmérnök hallgató

Konzulens:
Trohák Attila
egyetemi adjunktus
Automatizálási és Infokommunikációs Intézeti Tanszék

Miskolc, 2014

Tartalomjegyzék

1. CÉLKITŰZÉS.....	1
2. ELMÉLETI HÁTTER.....	2
2.1 A vezérlőegységünkről.....	2
2.2 Real-time rendszerek.....	3
2.2.1 A valós idejű rendszer jellemzői.....	3
2.2.2 Valós idejű terminológiák.....	4
2.2.3 Valós idejű operációs rendszerek.....	5
2.2.4 Valós idejű fejlesztőeszközök.....	6
2.2.5 A valós idejű rendszer komponensei.....	6
2.2.6 I/O csatornák elérése.....	8
2.2.7 NI Scan Engine.....	8
2.3 LabVIEW FPGA.....	10
2.3.1 A LabVIEW FPGA rendszer komponensei.....	11
2.3.2 A LabVIEW FPGA működése.....	12
3. AUTOMATIKUS PÁRHUZAMOS PARKOLÁS.....	13
3.1 A rendszer általános felépítése.....	13
3.2 A rendszer alapvető működéséhez szükséges komponensek.....	14
3.2.1 Előremenet, hátramenet és fék.....	14
3.2.2 Kormányzás.....	15
3.3 A parkolási folyamatot felépítő programrészek.....	19
3.3.1 Távolságtartás.....	19
3.3.2 Parkolóhely detektálás.....	19
3.3.3 Parkolási manőver.....	20
4. A FOLYAMAT GRAFIKUS ÁBRÁZOLÁSA.....	21
4.1 A gokart forgatása.....	21
4.2 Szenzoradatok megjelenítése.....	23
4.3 A parkolási manőver szemléltetése.....	24
5. ÖSSZEFOGLALÁS.....	25
6. FELHASZNÁLT IRODALOM.....	26

1. CÉLKITŰZÉS

Mint ahogy az autóiipari automatizálás területét képviselő mérnökök munkáján is észrevehető, napjainkban egyre nagyobb igény mutatkozik a vezetéstámogató rendszerek fejlesztésére. A fékező, sávtartó, és sebességtartó automatikákon túl számos egyéb, akár automatikus parkolási funkció is segítheti a vezető feladatát.

2013 őszén került meghirdetésre a „Go-Bosch Go-Kart!” elnevezésű országos csapatverseny, melynek keretein belül az ország öt egyeteme mérte össze tudását. A verseny célja egy gyári gokart felszerelése volt elektromos gáz- és fékrendszerrel, elektromos hátramenettel, illetve a járműnek bírnia kellett automatikus párhuzamos parkolási funkcióval is. Mivel a verseny teljes időtartama három év, és az ez évi versenykiírásban is szerepel a párhuzamos parkolás megvalósítása, így az ezen a területen történő fejlesztésekkel nem hagytunk fel.

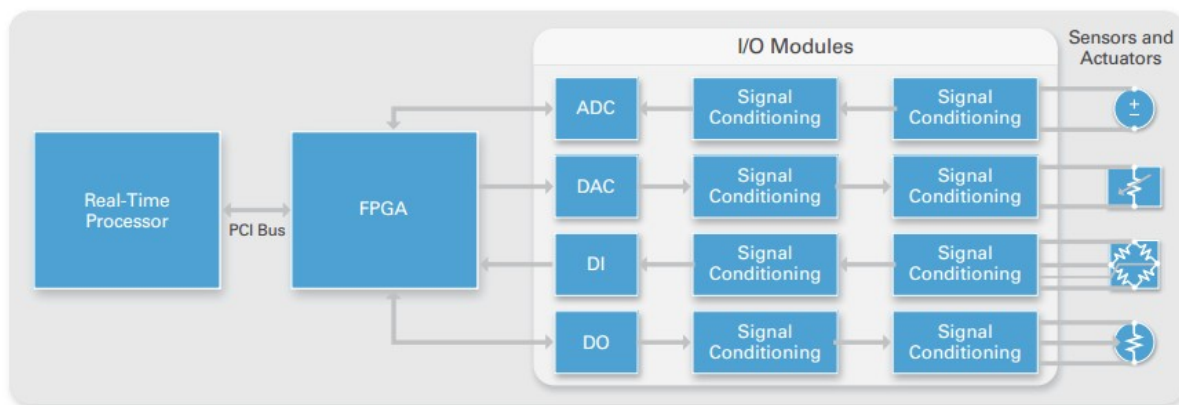
Dolgozatom ezt az automatizált folyamatot fogja ismertetni, bemutatva a rendszer felépítését és működését. A cél, hogy a gokart egy gombnyomás hatására elinduljon, haladása közben „figyelje” környezetét, majd ha megfelelő méretű parkolóhely áll a rendelkezésére, azt detektálja és az optimális parkolási görbével végrehajtsa a parkolást. Az alapkövetelményeken túl izgalmas feladatnak tartottam a parkolási folyamat grafikus megjelenítését ugyanazon platformon, amelyen a parkolási folyamat fő elemei is megvalósításra kerültek. Az utóbb leírt munkára vonatkozó fejezetben felfedem a korlátok szülte problémákat, melyekbe a tervezés során ütköztem és hogy hogyan jutottam el az alkalmazható végeredményig.

Mivel egyetemünk jó kapcsolatot ápol a National Instruments céggel, elérhető volt számunkra egy CompactRIO platform, mely megfelelt igényeinknek. A feladatok teljesítéséhez szükséges ismernünk az eszköz felépítését és annak controllerén futó operációs rendszer jellemzőit, így ezek bemutatásául részletes leírást nyújtok az elméleti háttérrel taglaló fejezetben.

2. ELMÉLETI HÁTTÉR

2.1 A vezérlőegységünkről

A gokartunk vezérlési feladatait a National Instruments CompactRIO eszközével valósítottuk meg. A választásunkban szerepet játszott, hogy az NI jó kapcsolatot ápol egyetemünkkel és csapatunk egyik fő támogatójaként rendelkezésünkre bocsátott több általuk gyártott terméket is. A vezérlőegység számos előnyt, mint például a modularitást magában hordozva megfelelt igényeinknek és a feladatok követelményeinek. A CompactRIO egy újrakonfigurálható beágyazott rendszer, amely három komponenst tartalmaz: egy processzort, amelyen valósídejű operációs rendszer (RTOS) fut, egy újrakonfigurálható FPGA-t, illetve cserélhető ipari I/O vagy kommunikációs modulokat.



2.1.1 ábra: A CompactRIO architektúrája

A CompactRIO LabVIEW grafikus fejlesztői környezetben programozható. A program írásakor a PLC programozásnál ismert funkcióblokk diagram nyelvhez hasonlóan itt is grafikus megjelenített blokkokkal dolgozunk, melyeknek a ki- és bemeneteit vezetékekkel összekötve írhatjuk meg a programunkat. A LabVIEW sajátossága azonban, hogy a különböző adattípusokat különböző színekkel, illetve különböző vezetékvastagságokkal jelöli, ezzel is segítve a felhasználót a programon belüli tájékozódásban és a megfelelő alkatrészek típusának kiválasztásában. Ezek a blokkok, mint virtuális műszerek működnek, erre utal a blokkok és a programfájl kiterjesztésének vi (virtual instrument) elnevezése.

2.2 Real-time rendszerek

2.2.1 A valós idejű rendszer jellemzői

A LabVIEW valós idejű (Real-Time) modul egyesíti a LabVIEW grafikus programozást a valós idejű operációs rendszer erejével, lehetőséget adva egy determinisztikus valós idejű alkalmazás építéséhez.

Hibás az értelmezés, miszerint a „valós idejű” elnevezés *gyorsat* jelent. A „real-time” elnevezéstől pontosabb lenne az „in-time”. Más szavakkal, egy valós idejű rendszer biztosítja a rendszerhívásokra adott válaszidőt meghatározott időben. Egy általános célú operációs rendszerben nem lehet biztosítani ezeket válaszidőket adott időperióduson belül, és a számítások lehet, hogy sokkal később vagy hamarabb befejeződnek.

Ahhoz, hogy egy rendszert valós idejűnek lehessen besorolni, minden részének valós idejűnek kell lennie. Például, ha egy alkalmazás valós idejű operációs rendszerben fut, lehet, hogy nem valós idejű rendszer terminológiák szerint viselkedik. Az alkalmazás függhet valami olyan dologtól, amely miatt az alkalmazás nem valós idejű rendszerként viselkedik.

A valós idejű rendszerek tárgyalásában a leggyakrabban használt terminológiák a determinizmus, a ciklusidő, jitter és beágyazottság. A terminológiák részletesebb ismertetése könnyíti a valós idejű rendszerek megértését.

2.2.2 Valósídejű terminológiák

A következő terminológiák vonatkoznak a valósídejű alkalmazásokhoz.

Ciklusidő: egy ciklus végrehajtásához szükséges idő. Sok alkalmazás, amely valósídejű operációs rendszert követel, mint például egy vezérlő alkalmazás, ciklikus. A ciklusok kezdetei között eltelt idő T , a ciklusidő.

Jitter: a ciklusidő eltérése a kívánt ciklusidőtől. Valósídejű operációs rendszereknél a ciklusidő változhat a ciklusok között. A maximális mennyiség, amivel eltérhet a ciklusidő a kívánt ciklusidőtől a maximum jitter.



2.2.2.1 ábra: A jitter értelmezése

Determinizmus: a determinizmus megmutatja, hogy mennyire biztosan tud válaszolni a rendszer külső eseményekre vagy műveleteket végrehajtani egy adott időlimiten belül. Ez a jitter nagyságát tükrözi. A magas determinizmus garantálja, hogy a számítások és műveletek végrehajtnak egy adott időn belül. A determinisztikus rendszerek kiszámíthatóak. Ez fontos egy vezérlő alkalmazásnál ami bemeneteket mér, bemeneteken alapuló számításokat végez és ezek eredményeinek értékeivel visszatér. A valósídejű rendszerek garantálják, hogy a számítások mindig időben befejeződnek.

Késleltetés: a szükséges válaszdő egy eseményre, vagy a bemenet és kimenet közötti idő. A determinisztikus rendszereknek magas késleltetése lehet. Helyesen implementált valósídejű rendszereknek valósídejű válaszája van, amely alacsony késleltetést garantál.

Beágyazottság: egy beágyazott rendszer egy komponens egy nagyobb rendszerben. A beágyazott rendszerek felhasználói felület mint például monitor, billentyűzet vagy egér nélkül működnek. Sok esetben a beágyazott alkalmazások a RAM és más erőforrások méretére, valamint arra a fizikai területre korlátozva hajtanak végre műveleteket, melyet a beágyazott alkalmazás le tud foglalni. A beágyazott rendszerek hardver eszközei lehetnek az ipari számítógépes rendszerek (pl. PXI/CompactPCI) vagy chip-en futó web szerverek is.

Időkritikus kód: az a kódrészlet, amelynek a kívánt ütemezésen kell végrehajtódnia. Az időkritikus kód olyan kód, ami nem kezelhet késleltetéseket a végrehajtás közben. Például, ha egy I/O hardver, amelynek specifikált időzítése van, pontosan akkor kell végrehajtódnia, amikor az elvárt. Egy példa egy nem-időkritikus kódra az adat kimentése egy fájlba. Az időkritikus kódok rendszerint nagyon magas prioritással bírnak.

Prioritás: a prioritás azt definiálja, hogy egy vi-nak vagy ciklusnak mikor kell végrehajtódnia a többi vi-okhoz és ciklusokhoz viszonyítva. Valós idejű programokban a helyesen konfigurált időzített struktúrákat prioritással kell létrehozni. Ez a prioritás egy sorrendet szolgáltat a valós idejű operációs rendszernek, amikor döntést kell hozni arról, hogy minek kell végrehajtódnia. A prioritás tartomány skálája 1-től 65535-ig definiált, ahol a nagyobb szám a magasabb prioritást jelenti.

2.2.3 Valós idejű operációs rendszerek

A LabVIEW alkalmazások Windows operációs rendszeren futnak, amely nem valós idejű operációs rendszer. A Windows nem biztosítja, hogy a kód mindig egy meghatározott időlimiten belül befejeződik. A kód végrehajtásához szükséges idő Windows operációs rendszerben sok dologtól függ, például attól, hogy hány program fut a háttérben (pl. egy vírusirtó vagy képernyővédő program). A Windowsnak szolgáltatnia kell megszakításokat eszközöktől, mint az USB port, billentyűzet, egér és más perifériák, amelyek a kód végrehajtását késleltethetik. A háttérben futó programok és az eszközök meghajtóinak felfüggesztésével növelhetjük annak valószínűségét, hogy a program determinisztikusan fusson Windowsban, ám ez se biztosítja azt, hogy a kód mindig valós idejűen hajtsódjon végre, mivel a Windows meg tudja szakítani a LabVIEW alkalmazást, még ha az időkritikus prioritást is használ.

LabVIEW valós idejű modullal az alkalmazásunk különálló valós idejű operációs rendszerben fut. Nem kell felfüggesztenünk programokat és nem kell újabb eszköz meghajtókat írunk ahhoz, hogy valós idejű viselkedést érzünk el. Egy valós idejű operációs rendszer lehetőséget ad a felhasználónak arra, hogy a különböző taskokat priorizálja, így tehát a legkritikusabb task mindig kap processzoridőt, amikor szükséges.

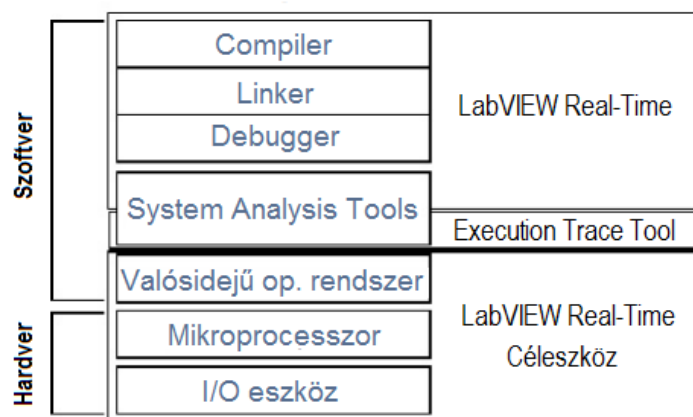
A National Instruments úgy tervezte a LabVIEW valós idejű modult, hogy a vi-okat két különböző valós idejű operációs rendszeren lehessen futtatni. A modul a vi-okat a hardver céleszközökön hajtja végre, az NI beágyazott rendszerekhez való eszközcsoomag (Embedded Tool Suite – ETS) vagy a Wind River VxWorks valós idejű operációs rendszerén futtatva. Az NI ETS és a Wind River VxWorks egy olyan valós idejű operációs rendszert szolgáltat, amely az NI valós idejű sorozaton fut és lehetővé teszi a determinisztikus viselkedést.

A valósídejű modulok nem támogatnak minden LabVIEW palettán elérhető funkciót ETS vagy VxWorks céleszközökön futó vi-ok esetében. Ezen korlát kiküszöbölésére a gokart fejlesztésénél is szükséges volt más alternatívákat kidolgozni, melyeket az érintett fejezetben részletesebben is kifejttek.

2.2.4 Valósídejű fejlesztőeszközök

A valósídejű fejlesztőeszközök olyan kód fejlesztő eszközöket tartalmaznak, mint a fordító (compiler), a linker és a hibakezelő (debugger). Továbbá, a rendszer analízis eszközök részletesebb belátást nyújtanak a valósídejű alkalmazások optimalizálásához.

A LabVIEW valósídejű modul alkalmazás fejlesztői környezet egy teljes fejlesztő és hibakezelő eszközként szolgál:



2.2.4.1 ábra: A LabVIEW Real-Time modul felépítése

A LabVIEW valósídejű modul platformok közös hardver és szoftver architektúrára alapszanak. Minden hardver céleszköz használ számítási komponenst, mint egy mikroprocesszor, RAM és egy I/O busz interfész. A beágyazott szoftver tartalmaz egy valósídejű operációs rendszert, meghajtó szoftvert és a LabVIEW Run-Time Engine egy specializált változatát.

2.2.5 A valósídejű rendszer komponensei

Egy valósídejű rendszer hardver és szoftver komponenseket tartalmaz. A szoftver komponensek közé sorolható a LabVIEW, az RT Engine és a LabVIEW projektek illetve vi-ok amiket létrehozunk. A hardver komponensek tartalmazzák a hoszt számítógépet, és egy valósídejű céleszközt. A következő bekezdések a valósídejű rendszer alkotóelemeit ismertetik.

Hoszt számítógép

A hoszt (gazda-) számítógép az a számítógép, amelyre a LabVIEW és a LabVIEW valósídejű modul fel van telepítve és amelyen a vi-okat fejlesztjük a valósídejű rendszerhez. A valósídejű rendszer vi-ok fejlesztése után letölthetjük és futtathatjuk a vi-okat a valósídejű céleszközökön. A

hoszt számítógép futtathatja azokat a vi-okat, amelyek a céleszközökön lévő vi-okkal kommunikálnak, így szolgáltatva egy felhasználói felületet.

LabVIEW

A vi-okat LabVIEW használatával fejlesztjük a hoszt számítógépen. A valós idejű modul kiterjeszti a LabVIEW adottságait kiegészítő eszközökkel, determinisztikus vi-ok létrehozásához, hibakezeléséhez és a használatához alkalmassá tételéhez.

Valós idejű rendszer motorja

A valós idejű rendszer motorja (RT Engine) a LabVIEW egy olyan verziója, ami a valós idejű céleszközön fut. Azokat a vi-okat futtatja, amelyeket letöltöttünk a céleszközzre. A valós idejű rendszer motor a következő okokból szolgáltat determinisztikus valós idejű működést:

- Valós idejű operációs rendszeren fut, ami biztosítja, hogy a LabVIEW végrehajtó rendszer és más szolgáltatások kövessék a valós idejű műveletet.
- Valós idejű hardveren fut. A valós idejű céleszközök úgy lettek tervezve, hogy csak azokat a valós idejű alkalmazásokhoz szükséges vi-okat és eszköz meghajtókat futtassa, amelyek meggátolják más alkalmazások beavatkozását a valós idejű vi végrehajtásában.
- A valós idejű céleszközök nem használnak virtuális memóriát, mivel a virtuális memória előre nem látható konfliktusokat okozhat.

Valós idejű céleszköz

Egy valós idejű céleszköz egy valós idejű hardvert azonosít, ami a valós idejű rendszer motort és az elkészített vi-okat futtatja. Egy hálózaton lévő valós idejű eszköz egy hálózaton lévő hardver platform beágyazott processzorral és valós idejű operációs rendszerrel, ami a valós idejű rendszer motort és a vi-okat futtatja. Lehetőség van külön hoszt számítógépet használni a vezérlő vi-okkal való kommunikációra egy hálózaton lévő valós idejű eszközön Ethernet kommunikáción keresztül. A mi általunk használt céleszköz az NI CompactRIO, amely egy újraprogramozható vezérlő- és adatgyűjtő rendszer, olyan alkalmazásokra tervezve, melyek nagy teljesítményt és megbízhatóságot igényelnek.

2.2.6 I/O csatornák elérése

A CompactRIO (későbbiekben cRIO) két módot ajánl fel az I/O csatornák eléréséhez. Ha a fejlesztői számítógépen LabVIEW Real-Time és LabVIEW FPGA is van, ki lehet választani, hogy melyik programozási módot szeretnénk használni. Ezek a Scan- illetve a LabVIEW FPGA interfész mód.

Scan interfész mód

A Scan interfészen keresztüli programozás lehetővé teszi számunkra, hogy a C szériás modulokat közvetlenül használjuk a LabVIEW Real-Time modulból. A modulok, amelyeket Scan interfészen keresztül használunk, a projekt fában közvetlenül a Chassis rész alatt találhatók meg és az I/O csatornák I/O változókként jelennek meg a modulok alatt. Az I/O változók használatához „drag and drop” elven lehet őket behúzni a LabVIEW valós idejű vi-okba.

Scan interfész módban nincs szükség LabVIEW FPGA fejlesztés vagy program kommunikációjára az FPGA és a hoszt vi-ok között. Nincs szükség továbbá a vi-ok FPGA-ra való fordítására várni a futtatásuk előtt.

LabVIEW FPGA interfész mód

A LabVIEW FPGA interfész mód azt teszi lehetővé, hogy a C szériás modulokat LabVIEW FPGA vi-okból használjuk. Az ebben a módban használt modulok közvetlenül az FPGA Target rész alatt találhatók meg a projekt fában és az I/O csatornák I/O eszközökként jelennek meg a modulok alatt. Az I/O csatornák eléréséhez FPGA I/O Node-okat kell konfigurálni az FPGA vi-okon.

A LabVIEW FPGA interfész mód alkalmazásával használható a LabVIEW FPGA programozása, amellyel alkalmazásunkhoz nagyobb rugalmasságot, testreszabási lehetőséget és determinisztikus időzítést rendelhetünk.

2.2.7 NI Scan Engine

Az NI Scan Engine hatékony elérést tesz lehetővé az adat csatornák, mint például az I/O csatornák beállításaihoz, egy olyan beolvasást használva, amely az adatokat egy globális memóriában tárolja és minden értéket egy beolvasási periódusban frissít.

A Scan- és LabVIEW FPGA interfész mód közötti egyik különbséget szemlélteti egy NI által elvégzett mérés eredményei. A mérés az NI Scan Engine szintjeit mutatta be cRIO eszközön. A méréssorozat egyik eleme a leggyorsabb frissítési frekvenciát mérte a csatornák számának függvényében. Ez a mennyiség azt a leggyorsabb ciklust mutatja meg, amelyet a rendszer adatvesztés nélkül tud elérni.

Csatornák száma	Scan mód [Hz]	FPGA mód [Hz]
1	1000	7100
16	1000	3500
80	399	974

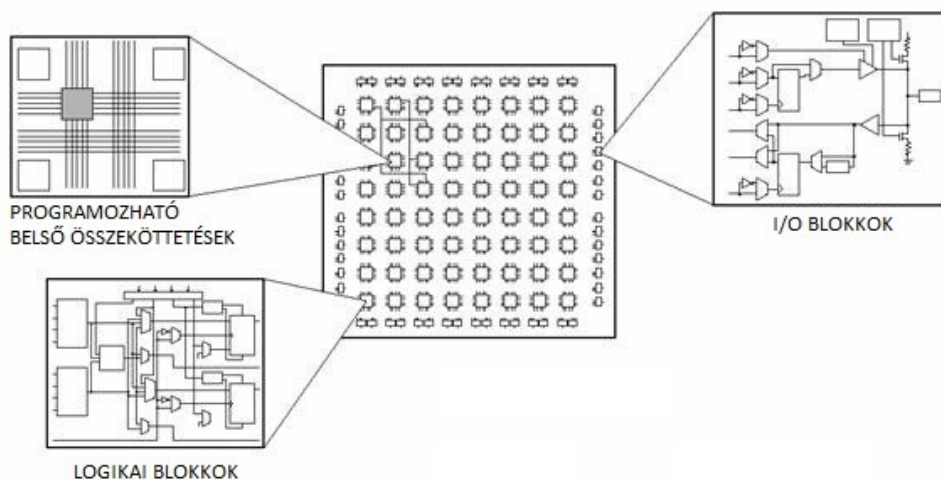
2.2.7.1 táblázat: A Scan és FPGA interfész frissítési frekvenciái

A Scan és LabVIEW FPGA interfészen keresztüli programozási módok között azonban számos egyéb különbség fedezhető fel. Scan módban nem használjuk fel az FPGA-t és a modulok egyes csatornáit közvetlenül hívjuk meg a fejlesztői felületen. Ilyenkor a program futtatásakor a cRIO kontroller periodikusan olvassa be az aktuális adatokat. Ez a mód egyes esetekben előnyös, azonban nem minden modul adatait képes feldolgozni (pl. a később tárgyalt CAN kommunikációs kártyát).

A mi döntésünk az FPGA interfészen keresztüli programozás, melynek oka egyrészt a Scan interfész utóbb említett hiányossága, másrészt mert ez a mód egyedi előnyökkel is rendelkezik. Ha ezt a módot használjuk a fejlesztésre a vezérlőegységünk nagyobb hányadát kihasználva, sokkal nagyobb gyorsaságot tudunk elérni, amelyet a fenti táblázat is alátámaszt.

2.3 LabVIEW FPGA

A programozható logikai kapumátrixok (Field Programmable Gate Array – FPGA) szilícium chippek logikai kapukkal, melyek alapesetben nincsenek összekapcsolva. A megfelelő szoftver használatával az FPGA kapuk konfigurálásával meghatározhatjuk az FPGA funkcióit. Szoftveresen fejleszthetőek számítási taskok, melyeket olyan konfigurációs fájlkká (bit stream) kell lefordítani, melyek a kapuk szükséges összekapcsolásairól tartalmaznak információt. Továbbá, az FPGA teljesen újrakonfigurálható és azonnal új „személyiséget” nyer, ha egy eltérő áramköri konfigurációt töltünk le. Az FPGA-kat gyakran használják feldolgozó egységként, ahol a fejlesztési idő és költségek, illetve egy alkalmazásspecifikus integrált áramkör (ASIC) gyártása korlátozott. Az FPGA-k olyan alkalmazásokban is megtalálhatóak, ahol szükséges őket helyileg újrakonfigurálni.



2.3.1 ábra: Az FPGA chip elemei

Az FPGA előnyei:

Az FPGA-k szoftveres úton való újrakonfigurálhatóságuk révén nagyon rugalmasan kezelhető eszközök. A mikroprocesszorokkal vagy alkalmazásspecifikus integrált áramkörökkel szemben az FPGA-k funkcióit módosítani lehet. Az FPGA-k speciális taskokat is hatékonyan tudnak végrehajtani. Mivel az FPGA végrehajtási műveleteit programozható belső kapcsolatok által vezérelhetjük, a felhasználók valódi párhuzamos taskokat tudnak implementálni az FPGA-n belül, amelyek szimultán és egymástól függetlenül fognak végrehajtódni. Végül, mivel az alkalmazások operációs rendszer nélkül lettek implementálva a hardver egységbe, az FPGA-k megbízhatóan hajtják végre az alkalmazásokat.

2.3.1 A LabVIEW FPGA rendszer komponensei

Habár az FPGA-k számos előnnyel rendelkeznek, a programozásuknál kihívásokba lehet ütközni. Programozásukhoz tipikusan egy hardver leíró nyelvet kell használnunk (hardware description language – HDL), mint a VHDL vagy Verilog. Ezek a nyelvek kevésbé bonyolult feladatoknál is nagy szakértelmet igényelhetnek és a mérnökök gyakran korlátokba ütközhetnek a tesztelési és vezérlési teendőknél.

A LabVIEW FPGA rendszer alapvető komponensei a LabVIEW FPGA modul, az NI-RIO meghajtó és egy NI-RIO eszköz, amely RIO céleszközként is ismert. A LabVIEW FPGA modul engedélyezi a LabVIEW használatát a cél FPGA-hoz az NI újrakonfigurálható I/O (RIO) hardveren, így a mérnökök alacsony szintű fejlesztő eszközök nélkül élvezhetik az FPGA teljesítményének és rugalmasságának előnyeit.

LabVIEW FPGA modul

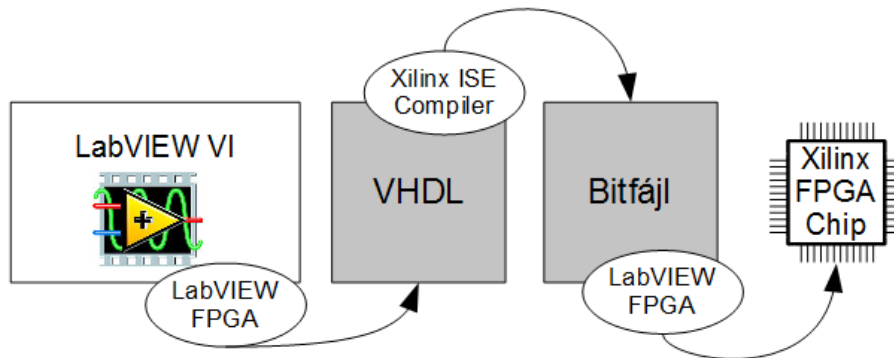
LabVIEW és az FPGA modul használatával bármely hardver leíró nyelv előzetes ismerete nélkül tudunk fejleszteni vezérlő és mérő-adatgyűjtő alkalmazást. A vi-okat a LabVIEW FPGA modullal kell létrehozni és célként a RIO hardveren az FPGA-t kell megjelölni. A LabVIEW-ban készített vi-ok definiálják az FPGA logikai kapuinak logikai kapcsolatait. Lehetőség van továbbá a hoszt számítógép számára is létrehozni olyan vi-okat, melyek az FPGA céleszközzel való kommunikációért felelnek.

LabVIEW FPGA céleszközök

A LabVIEW FPGA modult a National Instruments RIO eszközeinek széles választékával lehet használni. (Nem RIO eszközökkel vagy más gyártó termékeivel viszont nincs lehetőség alkalmazni.) A CompactRIO vezérlőegységünk tehát kompatibilis a LabVIEW FPGA modullal.

2.3.2 A LabVIEW FPGA működése

A LabVIEW FPGA modul lefordítja a LabVIEW vi-t az FPGA hardverre. Ekkor a grafikus kód egy szöveges VHDL kóddá fordítódik, majd a Xilinx ISE fordító eszköz leképezi logikai áramköri megvalósítását. A végeredmény egy bitfájl, amely tartalmazza azokat az utasításokat, melyek szükségesek a programkódunk FPGA céleszköztre való letöltéshez. Az alkalmazás futtatásakor a bitfájl az FPGA chip-be töltődik és újrakonfigurálja a kapumátrix logikai kapcsolatait. Az FPGA chip-en nincs operációs rendszer, de lehetőség van vezérelni azt egy hoszt alkalmazással.



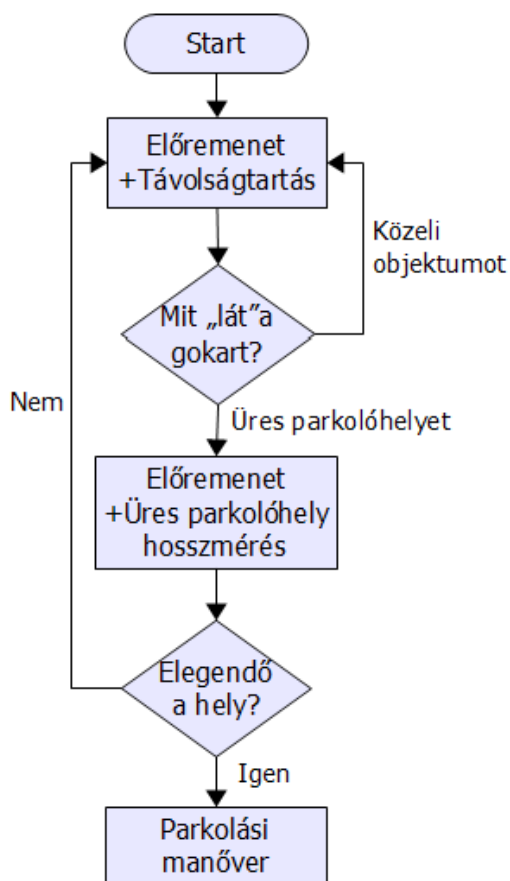
2.3.2.1 ábra A LabVIEW FPGA működése

3. AUTOMATIKUS PÁRHUZAMOS PARKOLÁS

3.1 A rendszer általános felépítése

A párhuzamos parkolási folyamat több részfeladatból áll. Ezek többek között az előremenet, a gokart egyenesben tartása, az ultrahang szenzorok által szolgáltatott jelek feldolgozása, a megtett távolság mérése és a parkolási manőver végrehajtása.

A folyamat indítása a gokart kormányán lévő kezelőpanelen elhelyezkedő gomb megnyomásával történik. Ekkor a gokart aktiválja az előremenetet, előre haladása közben pedig több részfunkciót is ellát. Az ultrahang szenzorok azt a centiméter adatot szolgáltatják felénk, amekkora távolságra objektum érzékelhető tőlük. Ezt kihasználva, a jobb oldalon lévő szenzorok jeleit figyelve vizsgáljuk, hogy a gokart milyen távolságra van a jobbra lévő objektumoktól és hogy elhagyta-e az előre meghatározott tartományt. Amennyiben kilépett az ideális tartományból, bekapcsol egy vissz szabályozó funkció. Abban az esetben, ha az ideális tartomány alatti jelet érzékel (az objektumhoz túl közel került), balra kormányoz, ha a tartomány feletti jelet kap (az objektum túl távoli), jobbra kormányoz. Ha kiütkezően nagy távolságadatot kapunk, az egy szabad parkolóhely kezdetét is jelentheti, ezért a később definiált feltételek szerint elkezdődik a szabad parkolóhely hosszának mérése (ekkor az utóbb tárgyalt, objektumoktól való távolságtartás nem működik, a gokart egyenesen halad). Ha ismét közeli objektumhoz érkezünk, megvizsgáljuk, hogy a mért parkolóhely hossza elegendő-e a parkolási manőver végrehajtásához. Amennyiben nem, az eltárolt hosszadat kinullázódik, és tovább haladunk a távolságtartási funkcióval. Ha a parkolóhely hossza megfelelő, a gokartunk megáll és elindítja a parkolási manőverét egy olyan minta alapján, melynek a később tárgyalt feltételek szerint dinamikusan változnak a megfelelő paraméterei. A folyamat egészét szemléletli a 3.1.1 folyamatábra:



3.1.1 ábra: A teljes automatikus párhuzamos parkolás folyamata

3.2 A rendszer alapvető működéséhez szükséges komponensek

3.2.1 Előremenet, hátramenet és fék

Hardveres kivitelezés:

Ezen funkciók mindegyike egy hajtóműházban kerültek megvalósításra. A hátramenet funkciót szolgáló fő hardveres egységünk egy ablaktörő motor, melyet egy elektromágneses tengelykapcsoló segítségével lehet a hajtásláncba kapcsolni illetve leválasztani. Mindez egy 1:9 áttételű fogaskerék kapcsolaton keresztül történik. Az elektromos fékezési funkció kinematikus rövidzárral megy végbe a hajtóműben, szintén az 1:9 áttételű fogaskerék kapcsolaton keresztül. Ehhez egy másik elektromágneses tengelykapcsolóra van szükségünk.

Szoftveres megvalósítás:

Az elektromos fékezés funkcióhoz tartozó tengelykapcsoló vezérlőpanelén szoftvertervezési szempontból 2 fontos bemenetünk van. Az egyik a vezérlő nulla vezetője, a másik az a vezérlő jel, amelyet a vezérlőegységünkől adunk ki. Ha ezen a bemeneten a az áramkör magas jelszintet kap (a TTL magas jelszintnek megfelelő 5V-ot), a tengelykapcsoló működésbe lép és a fékezési funkció aktiválódik. Alacsony jelszint esetén (0V, a TTL alacsony jelszintnek megfelelően) az elektromos fék oldott állapotban van és a gokartunk képes az előre- vagy hátramenetre.

Az elektromos hátramenet vezérlőpanelén irányítási szempontból 3 lényeges bemenetünk van. A vezérlő nulla vezetője (általunk nem használt bemenet) és két vezérlő jelünk, melyek egyike a jobbra forgatásért, a másik pedig a balra forgatásért felel, így képesek vagyunk az elektromos hátra- és előremenetet is működtetni. A forgásirány váltásért egy H-híd felel, melyet PWM jellel is vezérelhetünk, mi azonban az elektromos fék megvalósításához hasonlóan működtetjük a bemeneteket. Magas jelszint esetén (5V) elindul a forgatás, alacsony jelszint esetén (0V) megáll. Lényeges, hogy az előre illetve hátramenet funkciók csak akkor működnek, ha a tengelykapcsolójára magas jelszintet adunk.

A három funkcióhoz tartozó változók jegyzéke:

Funkció	LabVIEW változó	Adattípus	Kimeneti jel	
			Magas jelszint	Alacsony jelszint
Fék	FEK	BOOL	0V	5V
Előre forgatás	ELORE	BOOL	0V	5V
Hátra forgatás	HATRA	BOOL	0V	5V
Tengelykapcsoló	TKAPCS	BOOL	0V	5V

3.2.1.1 táblázat: Az előre- és hátramenet és fék funkcióhoz szükséges változók

Miután az FPGA vi-on meghívtuk a szükséges csatornákat, a programunkban ezeket a funkciókat egyszerű feltétel alapján történő kiválasztással hozhatjuk létre az RT vi-on. Ha a megfelelő bool típusú LabVIEW változók igaz értéket kapnak akkor 5V, míg ha hamis értéket kapnak, 0V-ot adunk a kimeneti csatornákra.

3.2.2 Kormányzás

A kormányzáshoz felhasznált hardveres komponensünk egy szögelfordulás mérő és egy ütfűró-csavarozó, mely egy 1:13 fogaskerék áttételen keresztül csatlakozik a kormányrúdhhoz. A csavarozó gyári vezérlését eltávolítva és azt egy saját tervezésű vezérlő panellal helyettesítve lehetővé vált számunkra a szoftveresen történő beavatkozás a kormányzási folyamatba. A H-hidat tartalmazó panelnek három bemenetét használjuk az automatizált kormányzás megvalósításához. Két PWM jellel vezérelhető bemenet felel a jobbra illetve balra forgatásért, a harmadik bemenet pedig egy fék funkciót lát el. Az utóbb említett bemenetre az egyes jobbra-balra szabályozási szakaszok után adunk a TTL magas jelszintnek megfelelő jelet, hogy a kormányunkat rögzítse az adott helyen.

A kormányzáshoz használt változók jegyzéke:

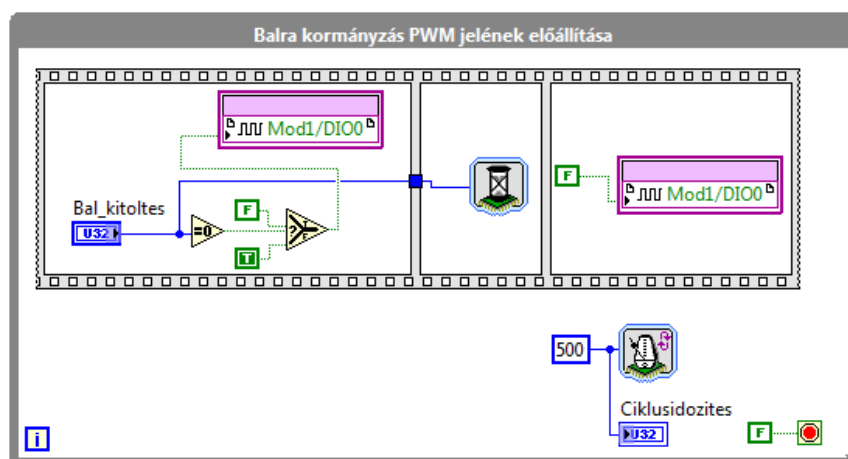
Funkció	LabVIEW változó	Adattípus	Kimeneti jel	
			Magas jelszint	Alacsony jelszint
Balra forgatás	KBALRA	BOOL	0V	5V
Jobbra forgatás	KJOBBRA	BOOL	0V	5V
Kormány fék	KFEK	BOOL	0V	5V
			Bemeneti jel	
Kormányzóg	KSZOG	FXP(fixed-point)	4...20 mA	

3.2.2.1 táblázat: A kormányzáshoz felhasznált változók

A kormányzás megvalósításához elsősorban a két PWM jelünket kell létrehozni az FPGA vi-on. LabVIEW segítségével ez könnyen megoldható két Flat Sequent struktúrával, melyek az egyes „kereteiben” lévő programrészletek egymás után, szekvenciálisan hajtódnak végre. A jobbra vagy balra forgatásért felelős kimeneti csatornát meghívva egy három keretből álló Flat Sequent struktúrába, melynek az első keretében hamis, a harmadik keretében pedig igaz értéket rendelünk a csatornához, közéjük pedig meghatározott késleltetést adunk, a kimeneti csatornánkon előáll a PWM jel. A PWM jel kitöltési tényezője annak függvényében változik, hogy a Flat Sequent struktúrában lévő késleltetésnek a külső ciklusidőzésnek hányad részét adjuk meg:

$$k = \frac{\text{Flat Sequent késleltetés}}{\text{külső ciklusidőzés}}$$

A balra kormányzás PWM jelének előállításért felelős grafikus programrészletet mutatja a 3.2.2.1 számú ábra:



3.2.2.1 ábra: PWM jel előállítása LabVIEW fejlesztői környezetben

Az első „keretben” szükséges volt egy feltétel megadása arra az esetre, ha 0%-os kitöltési tényezőt (forgatás nélküli állapotot) szeretnénk elérni. Ha a késleltetésünk 0, akkor hamis értéket adunk a kimenetre, más esetben állandó magas jelszint jelent meg a kimeneti csatornán.

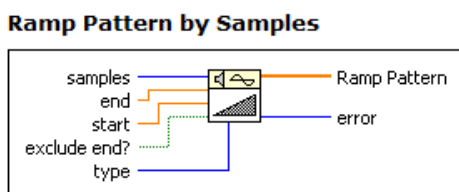
A következő lépésben az FPGA változók RT vi-on való lekezelést kellett megoldani. Mivel az FPGA vi-on a külső ciklusidőzítés hányadának megfelelő idő mértékegységű adattal dolgozunk, a későbbi könnyebb kezelhetőség érdekében az RT vi-on ezt százalék értékévé váltjuk át. Ekkor a létrehozott RT változók felvehető értékeit 0-tól 100-ig korlátozzuk, az FPGA változóba pedig a következő képlet alapján számított, majd előjel nélküli egész számmá alakított értékeket olvassuk be:

$$\text{bal kitöltés} = \frac{\text{bal RT változó}}{100} \cdot \text{FPGA ciklusidőzítés}$$

A továbbiakban a kormányzásunk a bal illetve jobb RT változó értékétől függ, melyeknek lehetőség van manuálisan vagy egy meghatározott szabályozás szerint állítani az értéküket.

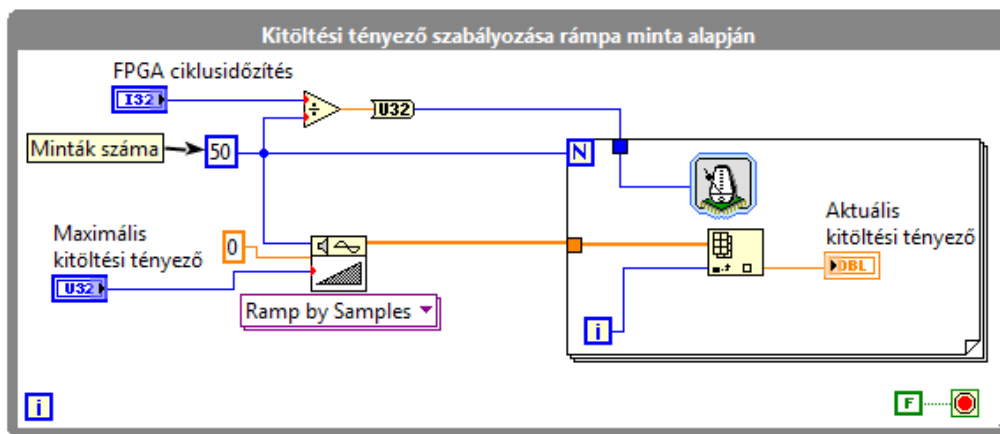
A szabályozás bemutatásához szükség szerű ismertetni a másik hardver egységet, amely több folyamatunkat is segíti. A kormányrúdra felszerelésre került egy szögelfordulás mérő, mely a 4...20 mA tartományban szolgáltat jelet. Az FPGA vi-on történő meghívása után az RT vi-on értékét ezerrel felszorozva és egész számmá alakítva 4 és 20 közötti értéket kapunk. A kormány 90°-os elfordulásra képes mindkét irányba, s az eszköz pozicionálása után számunkra feldolgozható adatok 8 és 16 közötti egész számok (1 kormánysszög érték változás így 22,5°-os kormánysszög elfordulást jelent).

A kormány forgatásának szabályozása aszerint történik, hogy az elérendő kormányyszög és annak az aktuális értéke között mekkora kormányyszög elfordulást kell megtennie a kormánynak. A kezdeti és végpont között arányosan csökkentjük a vezérlő PWM jel kitöltési tényezőjét, tehát a kormány mozgásának sebességét. Ezt a Ramp Pattern vi segítségével valósítottuk meg, melynek a be- és kimeneteit a következő ábra mutatja:



3.2.2.2 ábra: A Ramp Pattern vi be- és kimenetei

Az „end” és „start” bemenetek a rámpa mintában lévő vég- és kezdeti értékek. A végérték a mi esetünkben 0, a kezdőérték pedig annak a függvényében változik, hogy mekkora kormányyszög utat kell megtennie a kormánynak, legnagyobb esetben 100. A „samples” bemenettel a két végérték közötti minták számát adhatjuk meg. A Ramp Pattern egy tömböt fog adni, melyet ha a minták számával megegyező ciklusszámú for ciklusba vezetünk az indexelés engedélyezése nélkül, akkor a tömb egyes elemeit sorra ki tudjuk olvasni, a for ciklus iteráció számát egy Index Array „index” bemenetére kötve a kimeneti értékeket (a tömb adott indexű elemeit) felhasználva. A for ciklusba időzítést adva, minden egyes ciklus aktuális értékét egy előre meghatározott ideig tudjuk olvasni. Ezt kihasználva, ha ez az időzítés az FPGA vi-on lévő ciklusidőzítés és a rámpa minta mintaszámának a hányadosa, akkor biztosítva van, hogy egy FPGA vi ciklusidő alatt végbemenjen a szabályzásunk.



3.2.2.3 ábra: Rámpa minta alapján történő kitöltési tényező szabályozás

Azt, hogy a fenti szabályozás értékei a balra- vagy jobbra forgatásért felelős PWM jel kitöltési tényezőjét írja felül, egy 3 állapotú vezérlő dönti el. Egyik állapotában a balra, másik állapotában a jobbra forgatás PWM jel kitöltési tényezőjének módosítását engedélyezi, a harmadik állapotban pedig mindkét kitöltési tényezőt 0%-ra állítja.

Az eddig leírtak szerint már van egy PWM jelet generáló programrészünk, amelynek a kitöltési tényezőjét egy további programrészben arányosan csökkentve változtathatjuk. Az pedig, hogy a szabályozás mekkora értékről induljon, azaz a rámpa mintáknak mi legyen a kezdeti értéke, az elérendő és az aktuális kormánysszög különbségétől függ. Nagy kormányút megtételekor gyorsabban, azaz nagyobb kitöltési tényezőről indul, kis kormányútnál pedig kisebb kitöltési tényezőről. Ezek az értékek között is felírható egy arányosság, de a konkrét értékek tesztek során határozhatóak meg. Így a szabályozásunk kezdő értékei 100% és 60% között változnak arányosan a kormánysszög különbség függvényében.

Összegezve, az eddigiek szerint megírt programunkkal a kormányzást kétféle módon tudjuk működtetni. Manuálisan, az FPGA vi-on a két logikai állapot közé ültetett késleltetés közvetlen változtatásával, illetve szabályozottan, ha az elérendő kormánysszög értéket változtatjuk és hozzárendeljük a háromállapotú vezérlőnk egyik állapotát.

3.3 A parkolási folyamatot felépítő programrészek

3.3.1 Távolságtartás

A távolságtartás funkciót a már korábban említett cél érdekében, a gokart más objektumoktól való megfelelő távolságtartományban tartása miatt kell megvalósítanunk. Ehhez elsősorban az ultrahang szenzorok lesznek a segítségünkre, melyeknek a jeleit CAN hálózaton keresztül kapjuk meg. A CAN adatokat elsődlegesen az FPGA vi-on kezeljük le, ahol a megfelelő azonosítóval rendelkező adatokat szűrjük, majd kijeleztetésükkel elérhetővé tesszük az RT vi-okon is. A kapott adatok centiméter mértékegységek, amely a tesztelés és programozás szempontjából is kedvező.

A távolságtartási funkció szerint, ha a gokart előre haladása közben kilép a jobbra lévő objektumoktól való ideális távolságtartományból, amely 40-60 centimétert jelent számunkra, a kormányzás meghatározott feltételek szerint működésbe lép. 40 cm alatti értékeknél a gokart túl közel kerül az objektumhoz, így a balra kormányzás aktiválódik, míg 60 cm feletti értékeknél jobbra fog kormányozni. Ekkor elsőként elforgatjuk a kormányt középállásból adott irányba a megfelelő kormányzó értékkel. Ezután várunk, hogy a gokart megtegyen egy kívánt hosszúságú utat (melynek mérése a következő bekezdésben kerül leírásra), majd ha azt elérte, visszakormányozunk középállásba. Fontos, hogy ez a funkció csak akkor működik, ha a gokartunk „lát” valamit, nem kiütköző értékeket kap a szenzoroktól. Ezek az értékek a 150 cm feletti értékek vagy a 0, ami azt jelentheti, hogy a maximálisan érzékelhető 250 cm-es távolságtartományban nem érzékelhető objektum. Ezen értékek jelezhetik egy üres parkolóhely kezdetét.

3.3.2 Parkolóhely detektálás

A távolságtartó funkciónál említett határfeltételek teljesülése esetén, ha az ultrahang szenzorok 150 cm feletti vagy 0 értékeket érzékelnek, a távolságtartás befejeződik és elkezdődik a kormány középállásában megtett út hosszának mérése. Ezt egy induktív szenzor segítségével tesszük meg, melyet a gokart hajtóműben egy fogaskerék előtt helyeztünk el. A szenzor felfutó éleket számol, amikor elhalad előtte egy fogaskerék fog. Az út hosszának mérése akkor fejeződik be, ha a jobb első, majd a jobb hátsó szenzor is ismét érzékel közeli objektumot, tehát az üres parkolóhely végéhez érkeztünk. Ha ez a távolság elegendő a beparkolásához (legalább 300 cm), akkor a gokart még tovább halad előre 50 cm-t, majd megáll és elkezdi a parkolási manőverét. Az 50 cm-es további előrehaladás szükségességét a tesztelések és az ideális parkolási manőver számításai során állapítottuk meg. Amennyiben viszont az üres parkolóhely nem éri el a 300 cm-t, az úthossz mérési változónk kinullázódik és a gokart távolságtartó funkcióval halad tovább.

3.3.3 Parkolási manőver

Amennyiben megfelelő méretű parkolóhelyet talált a gokart és megtette a további 50 cm-t, elkezd a parkolási manőverét. A parkolási görbénk mindig hasonló jellegű lesz, ezért egy előre felvett minta alapján történő újraszámolással meg lehetett oldanunk ezt a feladatot. Tesztelések útján meghatároztuk azt a legkisebb parkolóhely hosszúságot, amelyre a gokart biztonságosan be tud parkolni. Ezt követően egy manuálisan vezérelt parkolási folyamat során rögzítettük azoknak a változóknak az értékeit, melyek szükségesek a parkolás végrehajtások. Ilyen változók a hátramenet és a tengelykapcsoló igaz vagy hamis-, illetve a kormányszög numerikus értékei. A rögzített értékeket egy szöveges fájlba mentettük ki. Ha a párhuzamos parkolást automatikusan hajtjuk végre, a programunk a paraméterezett minta visszajátszásánál a parkolóhely hosszának és a szenzoradatok függvényében korrigálásokat hajt végre.

4. A FOLYAMAT GRAFIKUS ÁBRÁZOLÁSA

Az automatikus parkolási folyamatra vonatkozó követelmények teljesítése után egy olyan program fejlesztését terveztem megvalósítani LabVIEW segítségével, mely képes a teljes parkolási folyamatot, annak valós idejű működése közben grafikusán ábrázolni. A feladatok elvégzéséhez a LabVIEW Graphics & Sound palettáján lévő vi-okat használtam fel. Az elképzelés szerint a kijelzőn a gokart forgatásának, a szenzoradatoknak és a parkolási manőver folyamatának megjelenítését kellett megoldanom. A továbbiakban a fejezet ennek a három funkciónak a megvalósítását fogja tárgyalni.

4.1 A gokart forgatása

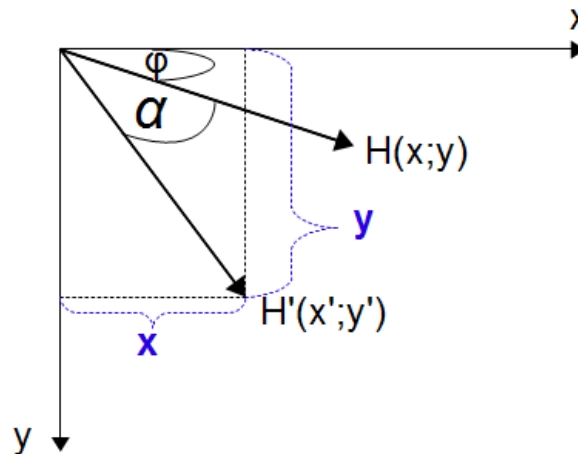
Mivel a fejlesztői környezet palettáin nem található olyan vi, amellyel egy képet beolvasva egy bemenet segítségével forgathatjuk azt, így annak algoritmusát saját kezűleg kellett megírni. A program összességében úgy működik, hogy egy képet pixelről pixelre beolvasunk, egy függvény segítségével változtatjuk azok helyzetét, majd az így módosított képet kijejeztetjük. A megjelenítés méretaránya szerint egy pixel egy centimétert jelent.

Az alap képünk a gokart 3D modelljének bmp kiterjesztésű felülnézeti képe, melyet első lépésként beolvasunk, majd az Unflatten Pixmap vi segítségével egy két dimenziós tömbbé konvertáljuk. Ekkor lehetőségünk nyílik két egymásba ültetett for ciklussal a kép minden egyes képpontját sorról sorra illetve oszlopról oszlopra beolvasnunk. A belső ciklusban található az a függvény, amely a kép forgatásáért, tehát az egyes képpontok helyzetének módosításáért felel.

A függvényhez szükséges bemenő változók között a legfontosabb az orientáció érzékelőnk által szolgáltatott jel, mely képes megadni a gokart azzal az egyenessel bezárt szögét, melyet előzetesen referenciaként felvettünk. Ezt az értéket a programunkban radiánná átváltva használjuk, mivel a később említett szögfüggvény vi-ok radiánnal számolnak. További változóink egy két elemű cluster, mely a forgatás középpontjának x és y koordinátáját adja meg, illetve az aktuálisan beolvasott képpont x és y koordinátája, melyeket az egymásba ültetett for ciklusok iteráció száma határoz meg. A képpontok új helyzetének meghatározásához elsőként az aktuálisan beolvasott képpont és a forgatás középpontját összekötő vektor helyvektorát kell képeznünk, a megfelelő koordinátájuk különbségével:

$$\begin{aligned} \text{forgatás középpontja} &: C(c_x; c_y) \\ \text{aktuális képpont} &: P(p_x; p_y) \\ \text{helyvektor koordináta} &: H(p_x - c_x; p_y - c_y) \end{aligned}$$

Ezután a fordulás szöge és a helyvektor abszolút értékének ismeretében kiszámolható a képpont az a helyzete, amelyet az origó körüli forgatással érne el.



4.1.1 ábra: A képpont helyvektorainak és azok tengellyel bezárt szögeinek viszonya koordináta rendszerben

A fenti ábrából is látható, hogy mivel a helyvektor hossza állandó és a forgatás szöge adott lesz számunkra, az új helyvektor koordinátákat könnyen megkaphatjuk szögfüggvények alkalmazásával. A kiszámítandó értékek az ábrán kiemelt x és y értékek. Ha vesszük ezek az értékek és az elforgatott helyvektor által határolt derékszögű háromszöget, akkor az elforgatott helyvektor koordinátáit a következő számításokkal határozhatjuk meg:

A helyvektor abszolút értéke: $h = \sqrt{x^2 + y^2}$

Ha φ az eredeti helyvektor és az x tengely által bezárt szög, α pedig az eredeti és az elforgatott helyvektor által bezárt szög, akkor legyen $\alpha' = \alpha + \varphi$.

A korábban említett derékszögű háromszög szögfüggvényei: $\cos \alpha' = \frac{x}{h}$ és $\sin \alpha' = \frac{y}{h}$,

melyekből az elforgatott helyvektorok koordinátái: $x = h \cdot \cos \alpha'$ illetve $y = h \cdot \sin \alpha'$.

Végül, a képpontok tényleges új helyzetét úgy kaphatjuk meg, ha az előbb kiszámolt koordinátákhoz hozzáadjuk a forgatás középpontjának koordinátáit. Az így meghatározott elforgatott képpont: $P'(x + c_x; y + c_y)$.

A függvényünk szerint kiszámolt új koordinátákat egy Index Array vi segítségével olvassuk be az új képbe, melynek „array” bemenetére a bmp képből leképzett két dimenziós tömböt kötjük, ekkor a vi felismeri, hogy egy képadatokból álló tömbről van szó, és az „index” bemenetein megjelennek a col (oszlop) és row (sor) feliratok. Ezekre a bemenetekre az új koordinátákat kötve és a for ciklusokon kívül a képadatokból álló tömbből egy Draw Unflattened Pixmap vi-al a megfelelő formátumra hozva azt, a képünk kijelezhetővé válik.

4.2 Szenzoradatok megjelenítése

Ezen funkció működésekor a kijelzőn a gokartunktól jobbra eső tartományon olyan sávokat kell látnunk, melyeknek vastagsága az egy ciklus alatt megtett úttól, a hossza (kezdő pontjainak helyzete) pedig a szenzoradatok nagyságától függ. Nagyobb szenzoradatoknál a gokarttól távolabb kezdődő, rövidebb, míg kisebb szenzoradatok esetén a gokarttól közelebb kezdődő, hosszabb sávok megjelenítése a cél. Ehhez egy olyan alap képet használtam fel, mely két mintából áll. A kép egyik felén a szenzoradatokat illusztráló mintát, a másik felén pedig az a terepet szemléltető mintát kell látnunk, melyen a gokartunk halad (pl. aszfalt minta, a lenti kép esetén egyszerű színes kitöltés, szebb nyomtatást tervezve). A képforgatáshoz hasonlóan, itt is pixelről-pixelre való leképezést alkalmaztam azzal a különbséggel, hogy ebben a részben csak az oszlop indexeket változtatom a sávok mozgatásáért felelős függvénnyel.

Első lépésként a szenzoradatokból egy olyan dinamikusan változó tömböt hoztam létre, melyben egy ciklus alatt egy szenzoradat annyiszor adódik hozzá a tömbhöz, ahány cm-t haladt előre a gokart. Ez könnyen megoldható egy olyan for ciklus alkalmazásával, melynek a ciklusszámára az egy főciklus alatt megtett út centiméter adatát kötöm, a for cikluson belül pedig a vizsgált ultrahang szenzor változóját hívom meg. A for cikluson kívül így létrehoztunk egy olyan résztömböt, melyet minden ciklusban hozzáadunk az összes adatot tartalmazó tömbünkhöz. Egy shift register segítségével tudjuk elérni azt, hogy a teljes tömbünk minden ciklus során növelje az elemszámát, ne veszítse el a korábban hozzáadott résztömböket. Ahhoz, hogy a teljes n db. elemszámú tömbünk a 0. elemtől kezdve frissüljön és ne az n . indexű elem után adja hozzá a szenzorok által szolgáltatott értékeket, egy Reverse 1D Array vi-t kell alkalmazni. Így létrehoztuk a kívánt mód szerint változó tömbünket.

A képpontról-képpontra való leképezést itt is a korábban alkalmazott egymásba ágyazott for ciklusok használatával valósítjuk meg, a különbség csak a függvényünkben mutatkozik majd. Ennél a funkciónál a bemenő paramétereink az alap kép sorindexei (belső for ciklus i iteráció száma), a sorok száma (ugyanezen for ciklus N ciklusszáma) illetve a szenzoradatokból álló tömb, melynek egyes elemeit a már többször használt Index Array használatával olvassuk be az oszlop koordináták indexei szerint. A függvény egy egyszerű if-else feltételvizsgálat alapján működik, melyben azt vizsgáljuk, hogy az adott sorban az aktuálisan beolvasott képpont az x tengelyen az eltolni kívánt értéktől nagyobb-e. Amennyiben nagyobb, az kimeneti koordináta az aktuális x koordináta és a szenzoradat különbsége, ellenkező esetben a sorok számából kivonva a szenzoradatot és ahhoz hozzáadva az aktuális koordinátát kapjuk meg a keresett értéket.

A Formula Node-ban megírt programkódunk így a következő:

```
int32 outx;  
if (s==0) {  
    outx=x+250; }  
else  
if (x<s)  
    { outx=xmax-s+x; }  
else  
    outx=x-s;
```

Az első feltételvizsgálatra azért van szükség, mert ha a szenzoroktól 0 értéket kapunk, az az üres parkolóhelyet is jelentheti. Ekkor azért 250 képponttal toljuk el a sávunkat, mert az a legnagyobb centiméter adat amit a szenzorunk „láthat”, s így a kijelzett képünk ezen felét is így méreteztem.



4.3.1 ábra: Részlet a grafikus megjelenítésből

4.3 A parkolási manőver szemléltetése

Az előző két folyamat grafikus ábrázolásának megvalósítása után már csak a parkolási manőver „lekövetése” a feladat. Amikor megfelelő hosszúságú üres parkolóhelyet találtunk, egy kétállású belső változó igaz értékre billen. Ez a feltétel felel a szenzoradatok tömbbe írásáról is. Amennyiben az értéke hamis, a tömb elemeit a szenzoradatok megjelenítésénél írtak szerint módosítjuk, ha viszont igaz, az utolsó hamis állapotbeli tömböt csatoljuk vissza egy Feedback Node segítségével. Ekkor a szenzoradatokat szemléltető sávok „mozgása” leáll és elkezdődik a parkolási görbe megjelenítése. Az ehhez szükséges orientáció érzékelő jel alapján történő forgatási programrész továbbra is aktív, az egyedüli megoldandó feladat a gokart jobbra illetve balra történő mozgatása a kijelzőn. Ez tulajdonképpen a szenzoradatok megjelenítésénél használt x tengely mentén történő képpont eltolási algoritmus párja, itt viszont nem a sorok, hanem az oszlopok mentén változtatjuk a pixelek koordinátáit.

5. ÖSSZEFOGLALÁS

A vezérlőegységünk felépítése, a valós idejű rendszerek jellemzőinek és a LabVIEW FPGA sajátosságainak ismeretében felépíthetővé válnak számunkra az automatikus párhuzamos parkolási folyamathoz szükséges alapvető funkciók és a teljes folyamatért felelős automatizálási célokat teljesítő elemek. A helyes működéshez nélkülözhetetlen volt az egyes programrészek alaposan átgondolt tervezése, illetve rengeteg tesztelés, melyekkel az elképzeléseinket alátámasztottuk, vagy a fellépő problémák miatt azok új megvilágításba kerültek. Elkészült továbbá a parkolási folyamatot szemléltető grafikus megjelenítést szolgáló programrész, mely a későbbiekben felhasználhatóvá válik számunkra. Az idei évben ugyanis feladat lesz többek között egy telemetrián keresztüli adatok, mint például a sebesség, gyorsulás vagy kormányászög értékek, illetve a parkolási folyamat egyes státuszainak megjelenítése.

Csapatunk folyamatosan új fejlesztési lehetőségeket fedez fel az eddig kialakított rendszerünkben, ezért alkalmazunk többek között egy tavaly még nem használt, egyik csapattagunk által fejlesztett orientáció érzékelő. Ezt nem csak a grafikus megjelenítésben, de a gokart egyenesben tartásához is felhasználjuk az üres parkolóhely keresésénél. A CAN hálózat optimalizálása által az ultrahang szenzoraink a tavalyi két méter helyett, már három méter feletti távolságokat is detektál. Reméljük, hogy a további fejlesztéseink által a verseny következő évében is dobogóra állhatunk.

6. FELHASZNÁLT IRODALOM

LabVIEW Real-Time 1 Course Manual

LabVIEW FPGA Course Manual

NI LabVIEW for CompactRIO Developer's Guide

(<http://www.ni.com/pdf/products/us/fullcriodevguide.pdf>)

Go Kart, Go Bosch M05 végleges dokumentáció – ME-Kart