

Szegedi Tudományegyetem
Informatikai Intézet

Ipari folyamat szimulációja és irányítása
programozható logikai vezérlővel

Diplomamunka

Készítette:

Miklós Árpád

mérnök informatikus
szakos hallgató

Témavezető:

Dr. Kincses Zoltán

egyetemi adjunktus

Szeged

2021

Ipari folyamat szimulációja és irányítása programozható logikai vezérlővel

Diplomamunka mérnök-informatikus MSc szakos hallgató számára

Témavezető: Dr. Kincses Zoltán

Témakör: ipari informatika, rendszer szimuláció, irányítás

Műszaki Informatika Tanszék

A feladat leírása, a munka célja:

A hallgató feladata egy ipari folyamat emulációjának és irányításának elkészítése. A munka célja, egyrészt az Országos Ajtonyi István Irányítástechnikai Programozó Versenyen szereplő technológia emulációjának elkészítése egy HIL szimulátor eszköz segítségével. A munka másik célja az így elkészített rendszer irányításának megvalósítása egy programozható logikai vezérlővel. Az elkészült munka később jól alkalmazható a későbbi PLC versenyekre történő felkészítésben.

A munkavégzés fontosabb lépései:

- A HIL szimulátor eszköz és a hozzá tartozó szoftverek megismerése, ismertetése
- Az emulálni kívánt technológia megismerése
- A programozható logikai vezérlő és a hozzá tartozó szoftver megismerése, ismertetése
- A HIL emuláció elkészítése
- Az elkészült szimuláció irányításának megvalósítása programozható logikai vezérlővel
- Hibakezelés
- Tesztelés
- A dolgozat megírása

A fejlesztéshez rendelkezésre álló erőforrások:

- OMRON CJ2M PLC, I/O egységek és a programozásához szükséges szoftver
- Lucas Nülle I/O interfész PRO/TRAIN-hez
- Lucas Nülle BORIS szoftver csomag
- Lucas Nülle PRO/TRAIN

A jelentkezés feltételei:

- Érdeklődés a PLC alapú irányítások és vizualizációjuk iránt
- Angol nyelvtudás

Tartalmi összefoglaló

- **A téma megnevezése:**

Egy ipari folyamat emulálása hardware-in-the-loop (HIL) szimulátor segítségével és az emulált ipari folyamat irányítása programozható logikai vezérlővel.

- **A megadott feladat megfogalmazása:**

Meg kell valósítanom egy megfelelően összetett ipari folyamat emulálását és annak irányítását programozható logikai vezérlővel azért, hogy szemléltessem a HIL szimulátorral támogatott fejlesztést és tesztelést. Továbbá, az alap feladatkiírásán túl, meg kell terveznem és meg kell valósítanom egy alternatív megoldást az emulációra, amelyik egyben kompatibilis megoldásához biztosított HIL szimulátorral.

- **A megoldási mód:**

- A feladat megoldásához biztosított hardverek és szoftverek megismerése.
- A kiválasztott ipari folyamat emulációjának megvalósítása és tesztelése.
- Az emulált ipari folyamat irányításának megvalósítása PLC segítségével.
- A HIL szimulátor hardveres és szoftveres elemeinek a tanulmányozása.
- A HIL szimulátorral kompatibilis hardver és szoftver fejlesztése és tesztelése.

- **Alkalmazott eszközök, módszerek:**

- Lucas-Nülle I/O interfész, Omron CJ2M-CPU32 PLC, MikroElektronika EasyPIC v7 fejlesztőlap, WinFACT 7 BORIS szimulációs szoftver, CX-Programmer 9.1, mikroC PRO for PIC 6.6.2, IAR Embedded Workbench for Arm 7.10.1, Visual Studio Code 1.54.3 és Qt Creator 4.14.1 fejlesztői környezetek, Altium Designer 17.1 elektronikai tervező szoftver, Eltima Serial Port Monitor 6.0.235 segédprogram
- dokumentumelemzés, megfigyelés, kísérlet, mérés, fejlesztés, tesztelés

- **Elért eredmények:**

Megvalósítottam a kiválasztott ipari folyamat emulálását a szimulációs szoftverrel és elkészítettem a hozzá tartozó irányítást. Megvizsgáltam az I/O interfész és a BORIS közötti kommunikációt az I/O interfész felnyitása nélkül, majd ez alapján megterveztem egy helyettesítő elektronika alapjait és írtam egy könyvtárat az I/O interfésszel való kommunikációhoz, amivel újra megvalósítottam az ipari folyamat emulálását QML-ben.

- **Kulcsszavak:**

ipari informatika, rendszer emuláció, HIL, irányítás, PLC, technológia-visszafejtés

Tartalomjegyzék

Tartalmi összefoglaló.....	2
Bevezetés.....	6
1. Felhasznált hardverek és szoftverek	7
1.1. Az ipari folyamat emulálásának eszközei.....	7
1.1.1. A Lucas-Nülle I/O interfész	7
1.1.2. A WinFACT 7 BORIS szimulációs szoftver.....	8
1.1.3. A Flexible Animation Builder beépülőmodul	9
1.2. Az ipari folyamat irányításának eszközei	10
1.2.1. Az Omron CJ2M programozható logikai vezérlő	10
1.2.2. A CX-Programmer programozószoftver	11
1.3. A technológia-visszafejtés eszközei	11
1.3.1. Az Eltima Serial Port Monitor segédprogram	12
1.3.2. A MikroElektronika EasyPIC v7 fejlesztőlap	12
1.3.3. A mikroC PRO for PIC fejlesztői környezet.....	13
1.4. Az alternatív emuláció megvalósításának eszközei.....	14
1.4.1. A Qt Creator fejlesztői környezet.....	14
1.4.2. Az Altium Designer elektronikai tervezőszoftver.....	15
1.4.3. Az IAR Embedded Workbench for Arm fejlesztői környezet	16
2. Az emulálni kívánt ipari folyamat.....	17
3. Interaktív vizualizáció készítése a FAB eszköztárával	18
3.1. A nyomógombok megvalósítása.....	19
3.2. A mozgítás irányának és a szerszám működésének jelzése	21
3.3. A szerszám animációinak megvalósítása.....	22
4. Az ipari folyamat emulációja és irányítása.....	28
4.1. Az emuláció megvalósításának bemutatása.....	28
4.1.1. A cellák megvalósításának alapjai.....	28
4.1.2. A robotok mozgatása.....	30
4.1.3. A karosszériák mozgatása	32
4.1.4. A daru mozgatása	33
4.1.5. A cellák ellenőrzőlogikái.....	34
4.2. Az irányítás megvalósításának bemutatása.....	35
5. A BORIS projektek hordozhatósági problémájának megoldása	39

6. Az ipari folyamat emulálásának alternatív megoldása.....	42
6.1. Az I/O interfész működésének behatásmentes visszafejtése	42
6.1.1. BORIS és az I/O interfész közötti kommunikáció lehallgatása	42
6.1.2. Az I/O interfész működésének utánzása a BORIS számára	44
6.1.3. Az I/O interfész működtetése a BORIS használata nélkül.....	47
6.2. Az emuláció megvalósítása alternatív eszközökkel.....	50
6.2.1. Az alternatív vizualizáció megvalósításának alapjai	51
6.2.2. A karosszériák mozgatása	52
6.2.3. A futószalagok megvalósítása és működtetése.....	54
6.2.4. A daru megvalósítása és működtetése	55
6.2.5. A robotok megvalósítása és működtetése.....	57
7. Az I/O interfészt helyettesítő elektronika	59
7.1. Az elektronika programozott működése	60
7.2. A kimenetek és a bemenetek felépítése	61
8. Konklúzió.....	63
Irodalomjegyzék	65
Nyilatkozat	66
Mellékletek	67
M1 A CO3715-1H típusjelzésű I/O interfész.....	67
M2 A FAB kezelőfelülete	67
M3 A BORIS kezelőfelülete	68
M4 A CJ2M-CPU32 típusú PLC	69
M5 A CX-Programmer kezelőfelülete	69
M6 A Serial Port Monitor kezelőfelülete	70
M7 Az EasyPIC v7 fejlesztőlap	71
M8 A mikroC PRO for PIC kezelőfelülete	71
M9 A Qt Creator kezelőfelülete	72
M10 Az Altium Designer kezelőfelülete.....	72
M11 Az IAR Embedded Workbench for Arm kezelőfelülete	73
M12 A karosszéria gyártósor emulációjának legfelső szintje	74
M13 A karosszéria gyártósor vizualizációja.....	75
M14 A karosszéria gyártósor irányításának legfelső szintje	76
M15 A BORIS Corrector osztályának teljes forráskódja	77

M16	Az I/O interfész működését utánzó prototípus programja	80
	A prototípus megszakításkezelő és main függvényeinek forráskódja	80
	A prototípushoz tartozó soros port kezelésének forráskódja	80
	A BORIS utasításait kiszolgáló központi logika forráskódja	82
M17	A BORIS Controller teljes osztálydiagramja.....	85
M18	A karosszéria gyártósor alternatív vizualizációja	86
M19	A BORIS Controller integrálása az alternatív emulációba	87
	Az integráció megvalósítása C++ oldalon	87
	Az integráció megvalósítása QML oldalon.....	88
M20	A helyettesítő elektronika kommunikációját lebonyolító program.....	90
M21	A helyettesítő elektronika oszcilloszkópos vizsgálatainak eredményei.....	93
	Bemenet	93
	Kimenet.....	94

Bevezetés

A minőség korunk egyik legmeghatározóbb hívószava. Olyan követelmény, amelyre a piaci versenyképesség fenntartása érdekében folyamatosan kiemelt figyelmet kell szentelni. Habár a fogalma az elmúlt évtizedek során sokat változott, köznap értelemben a minőség annak a mércéje, hogy egy termék vagy szolgáltatás milyen mértékben elégíti ki a vele szemben támasztott elvárásokat vagy igényeket. Az ipari automatizálásban a minőség általában olyan tulajdonságokhoz köthető, mint a megbízhatóság, a biztonság, a karbantarthatóság és a teljesítmény, amelyeknek a biztosításában kiemelt szerepe van a rendszeres és átfogó tesztelésnek.

A hardware-in-the-loop (HIL) tesztelés egyike azon módszereknek, amelyek az ipari automatizálásban is jól alkalmazhatók tesztelési és fejlesztési célokra. Ez a fajta eljárás a vizsgált eszközt olyan fizikai környezetben működteti, amelynek a jeleit egy virtuális rendszer biztosítja oly módon, hogy azok a valós rendszerrel megegyezőnek tűnjenek. Így nem csak költséghatékonyabbá tehető a tesztelés, hiszen nincs szükség egy valódi rendszerre, de olyan szélsőséges körülmények tesztelését is lehetővé teszi, amelyeknek a valóságban akár súlyos anyagi károkkal járó vagy emberéleteket is követelő következményei lehetnek, ha egy teszt elbukik.

Ez a dolgozat egy összetettebb ipari folyamat segítségével mutatja be a HIL tesztelés és a HIL szimulátorok működését a gyakorlatban. Ennek érdekében az 1. fejezetben először azok a hardverek és szoftverek kerülnek bemutatásra, amelyek a feladat megoldásához fel lettek használva. Ezt követi az emulálni kívánt ipari folyamat rövid bemutatása a 2. fejezetben. A 3. fejezet a HIL szimulátor szoftverével történő animációkészítésre korlátozódik, ugyanis a HIL szimulátor megismerése Görbedi Ákos kollégával közösen végzett munka volt, és az ő diplomamunkája részletesen ismerteti az említett szoftver azon részeit, amelyek ebben a dolgozatban nem kerülnek részletezésre. Az emulált ipari folyamatnak a megvalósítása a 4. fejezetben kerül bemutatásra, míg a bemutatott megvalósítást is érintő hordozhatósági probléma és annak megoldása az 5. fejezetben kerül ismertetésre. Végezetül bemutatásra kerül az emulált ipari folyamat alternatív megvalósítása és a HIL szimulátor hardverének helyettesítő elektronikája a 6. és a 7. fejezetekben. A feladat kidolgozása közben megszerzett tapasztalatok és az elért eredmények pedig a 8. fejezetben kerülnek összefoglalásra.

1. Felhasznált hardverek és szoftverek

A feladat megoldásához felhasznált hardverek és szoftverek a részfeladatok alapján jól csoportosíthatók. Az ipari folyamat emulálásához a HIL szimulátor szoftvere és hardvere lettek igénybe véve, míg az irányításhoz egy széles körben alkalmazott PLC és a programozását lehetővé tevő programozószoftver. A HIL szimulátor technológiájának visszafejtéséhez elsősorban egy olyan szoftver került felhasználásra, amelyik képes a PC és az I/O interfész közötti kommunikáció lehallgatására, majd egy fejlesztőlap a hozzá tartozó fejlesztői környezettel a kommunikáció megfigyeléséből származó adatok helyességének az igazolására. Az alternatív HIL szimulátor és az ipari folyamat alternatív emulálásának megvalósításához a legnépszerűbbnek számító tervezőszoftverek és fejlesztői környezetek lettek felhasználva.

1.1. Az ipari folyamat emulálásának eszközei

A HIL szimulátor két különálló gyártó egymással kompatibilis termékeiből tevődik össze. Az emulációhoz használt szoftver alapvetően egy széleskörűen felhasználható grafikus fejlesztői környezet, amelyik támogatja a bővítmények használatát is, míg a PLC-vel kommunikáló hardver egy olyan eszköz, amelyiknek a meghajtóprogramja bővítményként beépül ugyanebbe a szoftverbe.

1.1.1. A Lucas-Nülle I/O interfész

A CO3715-1H típusjelzésű I/O interfész (M1 melléklet) a Lucas-Nülle GmbH egyik terméke, amelyet a PRO/TRAIN for Windows szoftveréhez fejlesztett ki azzal a céllal, hogy a segítségével összeköthesse a számítógépen futó emulált folyamatot az azt irányító PLC-vel.

Ez az eszköz úgy lett megalkotva, hogy minden olyan PLC típust támogasson, amelyik képes az iparban használatos jelszintekkel üzemelni. A be- és kimenetei 4 mm-es biztonsági aljzatokra, illetve DC-37 csatlakozókra lettek kivezelve, emellett állapotjelző LED-ekkel is rendelkeznek. A számítógéphez soros porton keresztül csatlakoztatható az erre a célra szolgáló DE-9 csatlakozó segítségével. Az eszköz további műszaki jellemzői a következők:

- 16 digitális bemenet: +24 V DC / 10 mA
- 16 digitális kimenet: +24 V DC / 100 mA
- 2 analóg bemenet: 0...+10 V / 11 bit
- 4 analóg kimenet: 0...+10 V / 11 bit

- tápellátás: +24 V DC / 1 A
- adatátvitel: RS-232
- méretek: 297 mm × 227 mm × 60 mm (magasság × szélesség × mélység)
- súly: 1 kg

Ennek az eszköznek a működtetéséhez a megfelelő tápellátás biztosításán és a számítógéphez való csatlakoztatásán túl szükség van egy speciális meghajtóprogramra is, amellyel beépül a 1.1.2 alfejezetben bemutatásra kerülő szoftverbe. Ezt a meghajtóprogramot a PRO/TRAIN for Windows szállítja.

Mivel a PRO/TRAIN for Windows a feladat megoldásához nem került felhasználásra, csak a vele együtt települő meghajtóprogram, ezért ez a szoftver ebben a dolgozatban nem kerül bemutatásra.

1.1.2. A WinFACT 7 BORIS szimulációs szoftver








A BORIS (M3 melléklet) az Ingenieurbüro Dr. Kahlert terméke és a WinFACT moduláris programrendszer alapmodulja [1]. Az elnevezése a Block-Oriented Simulation System kifejezésből ered, és elsősorban dinamikus rendszerek szimulációjára szolgál, de a megfelelő hardver eszközökön keresztül valós folyamatokhoz is csatlakoztatható.

A BORIS kezelőfelületét az egyszerűség és az átláthatóság jellemzi, aminek köszönhetően az alapvető használata könnyen elsajátítható. Egy egyszerű szimuláció megvalósításához gyakorlatilag elegendő a megfelelő blokkokat lerakni a munkalapra, majd a megfelelő módon összekötni azokat, és elindítani a szimulációt. A lerakható blokkoknak szinte mindegyike megtalálható a Rendszerblokk Eszköztáron (System Block Toolbar), amelyről a beállításoktól függően egérekattintással vagy vonszolással lehet azokat lerakni a munkalapra. Ezeknek a blokkoknak az összekötése mindig egy kimenetre való kattintással kezdődik és egy bemenetre való kattintással végződik. Az összeköttetések átláthatóvá tételéről egy beépített automatikus elrendező gondoskodik, amelyik a blokkok összekötése és mozgatása során igyekszik a legjobb elrendezést megtalálni. Mindemellett lehetőség van külön-külön minden egyes összeköttetés színét is megváltoztatni, amely szintén az átláthatóság növelését szolgálja.

A szimulált folyamatok vezérlésére a BORIS több eltérő lehetőséget is kínál. Legegyszerűbb közülük a kézi vezérlés, amelyet a Vezérlő Eszköztár (Control Toolbar) tesz lehetővé. Ennek az eszköztárnak a funkciói a következők:



Standard szimuláció elindítása

-  Szimuláció befejezése
-  Léptető üzemmód aktiválása és deaktiválása
-  Szimuláció léptetése
-  Végtelenített szimuláció elindítása
-  Szimulációs paraméterek módosítása
-  Töréspont beállítása
-  Töréspont törlése

Végtelenített szimuláció esetén a szimulációs folyamat addig fut, amíg valamilyen hatás (pl. a Szimuláció befejezése gomb megnyomása) meg nem állítja azt. Ezzel szemben a standard szimulációk időtartama előre meghatározott és a szimulációs folyamat futása ennek letelésekor automatikusan megáll. A standard szimulációk időtartama és a lépések nagysága más egyebek mellett a szimulációs paraméterek között módosíthatók.

A BORIS rendszerblokkjainak a könyvtára kétféleképpen is bővíthető. Egyfelől a blokkok egy tetszőleges csoportjából bármikor létre lehet hozni egy úgynevezett szuperblokkot, amelyik ezt követően tetszőleges számú munkalapra is letehető. Másfelől lehetőség van egyéni bővítmények hozzáadására is, amelyeket bármelyik programozási nyelven meg lehet valósítani, amennyiben a fordítója képes a megvalósításokból natív DLL (Dynamic Link Library) állományokat építeni.

1.1.3. A Flexible Animation Builder beépülőmodul

A Flexible Animation Builder (FAB) beépülőmodul (M2 melléklet) egy vizualizáció tervező és megjelenítő eszköz, amelyet az Ingenieurbüro Dr. Kahlert fejlesztett ki a BORIS szoftveréhez azért, hogy leegyszerűsítse az interaktív vizualizációk, animációk és felhasználói felületek létrehozását [2].

A BORIS alapból nem rendelkezik olyan eszközökkel, amelyekkel vizualizációkat lehetne készíteni a szimulációkhoz, helyett olyan adatmegjelenítő blokkokat kínál, mint a táblázatok, a diagramok vagy a mérőeszközök. Természetesen a bővítmények a szállított funkcionalitásaikhoz biztosíthatnak saját vizualizációkat is, viszont ezeket a vizualizációkat értelemszerűen a bővítmény fejlesztéséhez használt eszközökkel kell megvalósítani és esetlegesen módosítani. Mindezekkel szemben a FAB egy teljesen üres vásznat biztosít a felhasználók számára, amelyre az eszköztárából igény szerint vihetők fel az elemek, a kínálata pedig többek között a következőkből áll:

- primitív alakzatok: körök, vonalak, téglalapok stb.
- médiatartalmak: képek, videók, hangok stb.
- szimbólumok és animációk: motorok, tartályok, keverőlapátok stb.
- adatmegjelenítők: táblázatok, diagramok, mérőeszközök stb.
- vezérlők: nyomógombok, beviteli mezők, jelölőnégyzetek stb.

A FAB blokkokhoz legfeljebb 50 darab bemenetet és ugyanennyi kimenetet lehet definiálni, amelyeken keresztül a vizualizálni kívánt adatok (pl. folyadékszint, szelepek állapota stb.) összeköthetők a vizualizációval, és amelyeken keresztül a felhasználói interakciók (pl. nyomógombok állapota) összeköthetők a szimulációval.

Mivel a feladat megoldásában a FAB kiemelt szerepet töltött be és Görbedi Ákos diplomamunkája elsősorban a BORIS használatának a bemutatására koncentrál, az interaktív vizualizáció készítés ebben a dolgozatban részletesen is bemutatásra kerül a 3. fejezetben.

1.2. Az ipari folyamat irányításának eszközei

Az irányítás megvalósításához, összhangban a feladat megoldásához rendelkezésre álló erőforrásokkal, egy CJ2M-CPU32 típusjelzésű PLC került felhasználásra, amelynek a programozása a hozzá tartozó programozószoftver segítségével valósult meg.

1.2.1. Az Omron CJ2M programozható logikai vezérlő

A CJ2M-CPU32 programozható logikai vezérlő (M4 melléklet) az Omron Corporation terméke és a CJ2 termékcsalád tagja, amelyet elsősorban csomagolási és általános gépipari automatizálási folyamatok elvégzéséhez fejlesztettek ki [3].

Ez az eszköz a termékcsaládjára jellemző módon moduláris felépítésű, ami lehetővé teszi, hogy a megfelelő bővítőmodulok segítségével az igényekhez lehessen alakítani. Emellett egy beépített USB-porttal is rendelkezik, aminek köszönhetően egyszerűen és különleges beállítások nélkül lehet a számítógéphez csatlakoztatni. Az eszköz főbb műszaki jellemzői a következők:

- programmemória mérete: 10 Klépés
- adatmemória mérete: 64 Kszó
- utasítás-végrehajtási idő: 0,04 μ s
- kommunikáció: EtherNet/IP, Ethernet TCP/IP, USB
- tápellátás: +5 V DC / 0,7 A

- méretek: 90 mm × 62 mm × 84,5 mm (magasság × szélesség × mélység)
- súly: 190 g

A feladat megoldásához a felhasznált PLC digitális I/O egységekkel is ki lett bővítve az I/O interfészhez való csatlakoztatás érdekében. A bemeneti egység típusa CJ1W-ID211, ami 16 darab egyenáramú bemenettel rendelkezik, a névleges feszültsége 24 V, a névleges árama pedig 7 mA. A kimeneti egység CJ1W-OD212 típusú, ami 16 darab tranzisztoros kimenettel (PNP) rendelkezik, a névleges feszültsége a bemeneti egységhez hasonlóan 24 V, a névleges árama pedig 0,5 A.

1.2.2. A CX-Programmer programozószoftver

A CX-Programmer (M5 melléklet) az Omron Corporation teljes PLC kínálatát lefedő programozószoftver és a CX-One szoftvercsomag szerves része [4]. Sok más programozási rendszerhez hasonlóan az IEC 61131-3 szabvány [5] előírásainak megfelelően lett kialakítva, emellett az integrált PLC szimulációs eszközeivel lehetővé teszi a programok tesztelését még a letöltés előtt.

A CX-Programmer több kényelmi funkcióval is rendelkezik, amelyek a programok írását igyekeznek még egyszerűbbé és gyorsabbá tenni. Így például a memória kiosztása és felügyelete automatizált, aminek köszönhetően elegendő a szimbólumok típusát meghatározni, a tárolásukról már maga a szoftver gondoskodik. Támogatja az azonos típusú adatokból álló adatblokkokat (tömböket) és az eltérő típusú adatokból álló adatblokkokat (struktúrákat) is, amelyek akár a funkcióblokkok be- és kimeneti változóiként is megadhatók. Az intelligens programbevitel a szimbólumok kiválasztását teszi egyszerűbbé azáltal, hogy egy prediktív stílusú böngészőt biztosít a felhasználó számára, amiben a szimbólumok nevének beírása közben megjelennek a lehetséges találatok. Képes továbbá a soros, az USB és az EtherNet/IP portokon keresztül kapcsolódó eszközökkel automatikusan összekapcsolódni, megkönnyítve ezzel a programozást és a hibakeresést.

Ennek a szoftvernek a 30 napos próbaverziója ingyenesen is használható. Ez az idő alatt minden funkciója korlátozásmentesen igénybe vehető, ugyanakkor a 30 nap elteltével megszűnik a mentés és a nyomtatás lehetősége.

1.3. A technológia-visszafejtés eszközei

A technológia visszafejtés valójában két további részfeladatra lenne bontható, ami alapján a felhasznált hardvereket és szoftvereket tovább lehetne csoportosítani. Az

adatgyűjtéshez ugyanis egy független lehallgatószoftver került felhasználásra, míg a begyűjtött adatok igazolása az I/O interfészt helyettesítő elektronika működő prototípusának megépítésével történt, amihez egy PIC fejlesztőlap és a hozzá tartozó fejlesztői környezet került felhasználásra.

1.3.1. Az Eltima Serial Port Monitor segédprogram

A Serial Port Monitor (M6 melléklet) az Eltima IBC által kifejlesztett segédprogram, amely a soros portok aktivitásának nyomon követésére és elemzésére szolgál [6]. A képességeinek köszönhetően kiválóan alkalmazható a hibakeresésben, a kommunikációs protokollok fejlesztésében, a technológia-visszafejtésben és az oktatásban.

Ennek a szoftvernek a segítségével lehetőség van a soros portok minden aktivitását rögzíteni, beleértve a felénk küldött vezérlőkódokat (IOCTL), a rajtuk áthaladó teljes adatforgalmat még akkor is, ha az érintett soros portok más alkalmazások által már meg lettek nyitva, illetve azokat az adatokat is, amelyeket az alkalmazások csak megkíséreltek a portokra írni a ténylegesen a portokra íródott adatok mellett.

A Serial Port Monitor a rögzített adatok vizualizálásra öt különböző nézetet is biztosít, amelyek a terminál nézet (Terminal view), a vonali nézet (Line view), az adatforgalom nézet (Dump view), a táblázat nézet (Table view) és a Modbus nézet (Modbus view). Mindezeknek a tartalma bármikor elmenthető HTML, egyszerű szöveg vagy CSV formában, illetve összehasonlíthatók a korábbi munkamentekből származó adatokkal is. A megfigyelés mellett a Serial Port Monitor képes adatokat is küldeni a megfigyelt soros portra úgy, mintha azokat a megfigyelt alkalmazás küldte volna. Ennek segítségével megvizsgálható a soros port és a csatlakoztatott eszköz reakciója is.

Érdemes megjegyezni, hogy a bemutatott funkciók egy része csak Professional és Company licenszekkel érhető el, viszont a 14 napig tartó próbaidőszak alatt minden funkció korlátozásmentesen kipróbálható.

1.3.2. A MikroElektronika EasyPIC v7 fejlesztőlap

Az EasyPIC v7 (M7 melléklet) a MikroElektronika d.o.o. hetedik generációs fejlesztőlapja, amelyet a Microchip Technology Inc. 8-bites PIC mikrovezérlőihez alakítottak ki főként kezdő felhasználók és hobbisták számára, illetve oktatási célokra.

Ezen a fejlesztőlapon számos modul kapott helyet, amelyek a legkülönbözőbb alkalmazások fejlesztéséhez használhatók, beleértve a grafikus megjelenítést, az USB és

soros kommunikációt, a hőmérsékletmérést és egyébeket. Rendelkezik két mikroBUS aljzattal is, amelyeken keresztül ezernyi új funkcionalitás adható a fejlesztőlaphoz Click Board bővítőkártyák segítségével. Mindezek mellett rendelkezik egy beépített mikroProg programozóval és áramkörön belüli hibakeresővel is, amelyik a Microchip Technology Inc. több mint 387 mikrovezérlőjét támogatja [7].

Az EasyPIC v7 kapcsolási rajza szabadon hozzáférhető csak úgy, mint a csatlakoztatható bővítőkártyáké is, így a fejlesztőlapon kipróbált megoldások viszonylag kevés ráfordítással átvezethetők a célhardverek világába. Mindezt a bővítőkártyákhoz szintén szabadon hozzáférhető függvénykönyvtárak és példaprogramok teszik még egyszerűbbé.

1.3.3. A mikroC PRO for PIC fejlesztői környezet

A mikroC PRO for PIC (M8 melléklet) a MikroElektronika d.o.o. integrált fejlesztői környezete, amelyet a Microchip Technology Inc. 8-bites PIC mikrovezérlőinek C nyelven történő programozásához fejlesztettek ki [8].

Ez a fejlesztői környezet több, mint 808 különböző PIC mikrovezérlő programozását támogatja, illetve mindegyik támogatott mikrovezérlőhöz rendelkezik egy konfigurációs felülettel is, amelyen keresztül a mikrovezérlő könnyedén testre szabható. A perifériák és a gyakran csatlakoztatott hardverkomponensek használatához számos függvénykönyvtárral rendelkezik, amelyeknek a használata a beépített sűgő és a szoftverrel együtt szállított példaprogramok segítségével könnyedén megismerhető.

A modern fejlesztői környezetektől elvárható módon a mikroC PRO for PIC szintén rendelkezik olyan funkciókkal, amelyek kényelmesebbé és hatékonyabbá teszik a fejlesztői munkát. Így például rendelkezik intelligens kódkiegészítővel, ami a begépelte karakterekhez automatikusan felkínálja a lehetséges függvényeket, konstansokat, struktúrákat, változókat és egyéb kódelemeket. A függvényhívások helyes paraméterezését a beépített paramétersegédlet támogatja oly módon, hogy a begépelte karakterek felett megjeleníti a függvény szignatúráját. Mindezek mellett az átláthatóság növelése érdekében képes a kódblokkok összezsukására és szétnyitására is.

A mikroC PRO for PIC a hozzá tartozó mikroProg programozó és áramkörön belüli hibakereső segítségével natív támogatást nyújt a hardveres hibakereséshez. Ezzel az eszközzel képes a mikrovezérlők programjait lépésről lépésre futtatni, megjeleníteni a regiszterek, az EEPROM-ok stb. aktuális értékeit, illetve lehetőséget biztosít a töréspontok igény szerinti beiktatására és eltávolítására is.

Ennek a szoftvernek a demó verziója korlátlan ideig használható ingyenesen, viszont a demó verzióval legfeljebb csak 2 Kszó nagyságú PIC programok fordíthatók.

1.4. Az alternatív emuláció megvalósításának eszközei

Az alternatív emuláció három különböző fejlesztési terület prominens szoftvereinek a felhasználásával valósult meg. Az I/O interfészt helyettesítő elektronika kapcsolási rajza és nyomtatott áramköre az Altium Designer tervezőszoftverrel lett megrajzolva. Ugyanennek az elektronikának a be- és kimeneteit vezérlő, illetve a számítógéppel kommunikáló mikrovezérlő programja az IAR Embedded Workbench for Arm fejlesztői környezetben lett megvalósítva. Az alternatív I/O interfésznek a számítógépen futó illesztőprogramja, illetve az alternatív vizualizáció pedig a Qt Creator fejlesztői környezetben valósult meg.

1.4.1. A Qt Creator fejlesztői környezet

A Qt Creator (M9 melléklet) egy keresztplatformos integrált fejlesztői környezet, amelyet a Qt Company fejlesztett ki a Qt alkalmazás-keretrendszerrel folytatott fejlesztés igényeire szabva. Köszönhetően a beépített Qt-specifikus eszközeinek és kényelmi funkcióinak, jelenleg ez a Qt-alapú alkalmazások elsődleges fejlesztői környezete [9].

A Qt Creator segítségével tulajdonképpen bármilyen C++ projekt fejleszthető, a használatához ugyanis a Qt keretrendszer használata nem szükségszerű, sőt még a hozzá tartozó qmake és Qbs fordítórendszerek használata sem, mivel natívan támogatja a CMake, a GNU Autotools stb. fordítórendszereket is. A kódminőség biztosítása érdekében beépített kódelemzőkkel és szoftvertesztelési eszközökkel rendelkezik, emellett támogatja a legnépszerűbb verziókezelő rendszereket is, mint például a Git, a Subversion, a ClearCase vagy a Mercurial. Habár a Qt Creator fő programozási nyelve a C++, a megfelelő bővítmények segítségével bármikor felruházható más programozási nyelvek támogatásával is, vagy bármi egyébvel, így például helyesírás-ellenőrzővel vagy diagram készítővel is.

Ebben a fejlesztői környezetben a Qt moduljaira épülő grafikus felhasználói felületeknek mindkét típusa szerkeszthető grafikusán. A beépített űrlapszerkesztő a hagyományos űrlap alapú alkalmazások felhasználói felületének fejlesztését teszi lehetővé, míg a modern, főleg okostelefonokra és táblagépekre jellemző grafikus felületek fejlesztésére a beépített Qt Quick tervező szolgál.

A Qt Creator egy nyílt forráskódú és teljesen ingyenes szoftver, aminek a használata sem hobbi, sem üzleti célokra nem igényel külön licenszt. Mindemellett nagyon jól dokumentált és számos példaprojekt érhető el hozzá, amik még könnyebbé teszik a használatának az elsajátítását.

1.4.2. Az Altium Designer elektronikai tervezőszoftver

Az Altium Designer (M10 melléklet) egy sokrétű elektronikai tervezőszoftver, amit az Altium Limited fejlesztett ki a nyomtatott áramkörök tervezésének támogatására. Alapvetően négy fő funkcionális területet foglal magába, nevezetesen a kapcsolási rajzok készítését, a nyomtatott áramkörök tervezését, a programozható logikai kaputömbök (FPGA) fejlesztését és az adatok kezelését [10].

Ez a szoftver az elektronikai termékek fejlesztésének gyakorlatilag minden aspektusát támogatja. Ez azt jelenti, hogy az Altium Designer környezetében a fejlesztési folyamat minden állomásának megtalálhatóak az eszközei, kezdve az első vázlatok megrajzolásától egészen a gyártásig.

Az Altium Designer környezetében minden fejlesztés kiindulópontja egy projekt, ami összefogja mindazokat a dokumentumokat, amelyek együttesen meghatározzák a megvalósítani kívánt terméket. Ezeknek a dokumentumoknak az alapvető építőkövei az úgynevezett komponensek, amelyek a való világ elektronikai alkatrészeinek szoftveres megfelelői. A komponensek a különböző dokumentumokban más-más reprezentációval rendelkeznek, ennek megfelelően egy komponens a kapcsolási rajzokon szimbólumként, a nyomtatott áramkörökön tokozásként (footprint), a szimulációkban SPICE definícióként, a vizualizációkban pedig háromdimenziós modellként jelenik meg.

Ahhoz, hogy egy alkatrészt használni lehessen a fejlesztés során, szükség van az azt reprezentáló komponensre. Általában ezek a komponensek megtalálhatóak a rendelkezésre álló komponenskönyvtárak valamelyikében, tekintve, hogy az elérhető komponensek száma többszázszáz nagyságrendű. Azonban, ha egy komponens mégsem szerepelne egyik komponenskönyvtárban sem, lehetőség van megrajzolni azt, amihez az elérhető rajzelemek mellett a már meglévő komponensek is felhasználhatók.

Az Altium Designer a sokrétűsége mellett több olyan funkcióval is rendelkezik, amelyek kényelmesebbé és hatékonyabbá teszik a fejlesztést. A kapcsolási rajzok esetén például támogatja a szimbólumok egy csoportjának a kiszervezését külön munkalapra, illetve képes az összeköttetések hibáinak a felderítésére is. A nyomtatott áramkörök esetén a nyomvonalak megrajzolását a beépített interaktív útválasztás teszi sokkal

egyszerűbbé, emellett lehetőséget biztosít a teljesítményeloszlás elemzése is. A gyártáshoz szükséges dokumentumok előállítására pedig egy beépített anyagjegyzék generátort is biztosít a felhasználók számára.

Ez a szoftver alapvetően ipari alkalmazásra lett kifejlesztve széles licenz kínálattal, de van elérhető próbaverziója is. A próbaverzió mellett elérhetőek a korlátozottabb képességű változatai is, mint például az Altium CircuitMaker, ami az Altium Designer kapcsolási rajz készítőjét és nyomtatott áramkör tervezőjét foglalja magába, viszont teljesen ingyenes.

1.4.3. Az IAR Embedded Workbench for Arm fejlesztői környezet

Az IAR Embedded Workbench for Arm (M11 melléklet) az IAR Systems AB integrált fejlesztői környezete, amely több, mint 7000 különböző ARM mikrovezérlő programozását teszi lehetővé C és C++ nyelveken [11].

Ez az egyik legnépszerűbb fejlesztői környezet az ARM mikrovezérlőkhöz, amit elsősorban a széleskörű kompatibilitásának, az egyszerű kezelőfelületének és a magába foglalt többesernyi példaprojektnek köszönhet. Külön érdekessége a beépített ARM utasításszimulátor, amely lehetővé teszi a program futtatását és a hibakeresést magában a fejlesztőkörnyezetben. Ezzel a szimulátorral a lépésről lépésre történő futtatáskor azok az ARM utasítások is láthatóak, amikre az aktuális kódsorok fordulnak. Természetesen a hibakeresést közvetlenül a célrendszeren is támogatja, amihez az elterjedt áramkörön belüli hibakeresők szinte bármelyike használható.

Az IAR Embedded Workbench for Arm a kódminőség biztosítása érdekében két beépített eszközt is tartalmaz. A C-STAT statikus kódelemző a forráskód lehetséges hibáit képes a felszínre hozni a kódszinten végzett elemzésekkel, ugyanakkor azt is képes ellenőrizni, hogy a kód megfelel-e a kiválasztott szabványoknak. A C-RUN futás idejű elemző ezzel szemben a célrendszeren vagy az utasításszimulátorban futó programot vizsgálja aritmetikai hibák, túlcordulás, túlindexelés, memóriaszivárgás stb. után kutatva, amelyeknek képes meghatározni a pontos helyét is, ha bekövetkeznek.

Ezt a szoftver ingyenesen is lehet használni korlátlan ideig, de az ingyenes változat korlátozásokat tartalmaz. Ezzel szemben a 30 napos próbaverzióban minden funkció korlátozásmentesen elérhető, viszont a próbaidőszak után már csak megvásárolt licensszel használható tovább.

2. Az emulálni kívánt ipari folyamat

Az emulálni kívánt ipari folyamat egy karosszéria gyártósor, amely egy korábbi Országos Ajtonyi István Irányítástechnikai Programozó Verseny egyik gyakorlati feladata volt. Az emulálni kívánt technológia az irányításához kiadott feladatlap alapján került megtervezésre és megvalósításra a BORIS eszköztárának segítségével.

A gyártósor a gépjármű karosszériák gyártásának egy szakaszát mutatja be, ahol három fő művelet folyik, nevezetesen ponthegeesztés, festés és leválogatás. A rendszer ezeket a műveleteket három egymást követő cellában végzi, amelyekben futószalagok mozgatják a karosszériákat. A cellák mindegyikében két-két érzékelő (fénykapu) jelzi a karosszériák hollétét. Az első érzékelő jelzi a cellába történő érkezést, a második pedig a munkadarab esetleges megállításának pozíciójában található.

Az első cellában két ponthegeesztésre alkalmas robot helyezkedik el, amelyek a rajtuk található hegesztőpisztolyokat is beleértve külön-külön vezérelhetők. A második cellában két festésre alkalmas robot található, amelyek szinkron üzemmódban működnek. Ez azt jelenti, hogy az egyik robot másolja a másik robot mozgását, így ez a két robot irányítási szempontból egynek tekinthető. Ugyanebben a cellában egy operátor is tevékenykedik, aki szabadon beléphet a robotok mozgásterébe azzal a céllal, hogy ellenőrizze a festés minőségét és esetlegesen kézzel korrigálja azt. A harmadik cellában egy daru helyezkedik el, melynek feladata a karosszériák típus szerinti leválogatása.

Az emuláció szempontjából a feladat nehézsége a vizualizáció megvalósítása. A karosszériáknak ugyanis követniük kell a valóságban őket mozgató elemek mozgását ahhoz, hogy valóságosnak tűnjenek, ehhez pedig számon kell tartani, hogy az animációban éppen egy futószalagon helyezkednek-e el, a futószalag mozog-e, a daru megfogója záródott-e a karosszéria körül, átemelés történik-e vagy az átemelés éppen befejeződött-e. További nehézséget jelent a FAB grafikus elemeinek a szegényes kínálata, azokkal ugyanis a vizualizáció nem valósítható meg felismerhető módon.

Irányítási szempontból az operátor jelenléte jelenti a legnagyobb nehézséget. A gyártósor irányítása mellett ugyanis folyamatosan figyelni kell az operátor tartózkodási helyét, mivel bármikor beléphet a robotok munkaterébe, amire az irányításnak gyorsan és megbízhatóan kell reagálnia.

Az emuláció elsődleges célja az irányítás tesztelése és az esetleges hibák felderítése biztonságos körülmények között. Ennek érdekében az emulációba számos korlátozást és ellenőrzést kell beépíteni, amelyek jelzik az irányítás hibáit.

3. Interaktív vizualizáció készítése a FAB eszköztárával

A FAB egy grafikus felülettel rendelkező beépülőmodul, amely lehetővé teszi a BORIS felhasználói számára, hogy a szimulációkhoz egyéni vizualizációkat valósítsanak meg. Nélküle a BORIS csak nagyon korlátozott mértékben és nem módosítható módon képes az adatok vizualizálására.

A FAB valójában önálló szerkesztőprogramként működik, saját eszköztárral, beállító felülettel, fájl típussal és szerkesztőablakkal. A BORIS szimulációkkal egyedüli kapcsolata csak a Rendszerblokk Eszköztáron is megtalálható Flexible Animation Builder blokkon keresztül van. Egy új vizualizáció megvalósításához pontosan ezt a blokkot kell lerakni a munkalapra, majd a dupla kattintással is megnyitható szerkesztőablakában a *Dialog...* gombra kattintani és az így megjelenő szerkesztővel megkezdeni a munkát.

Az interaktív vizualizációk készítésének szemléletesebb bemutatása érdekében ebben a fejezetben egy kellően összetett feladat kerül kidolgozásra lépésről lépésre. Ez a feladat lényegében egy mini asztali CNC marógép kézi vezérlésének a vizualizálása, aminek a következő elfogadási feltételeknek kell megfelelnie:

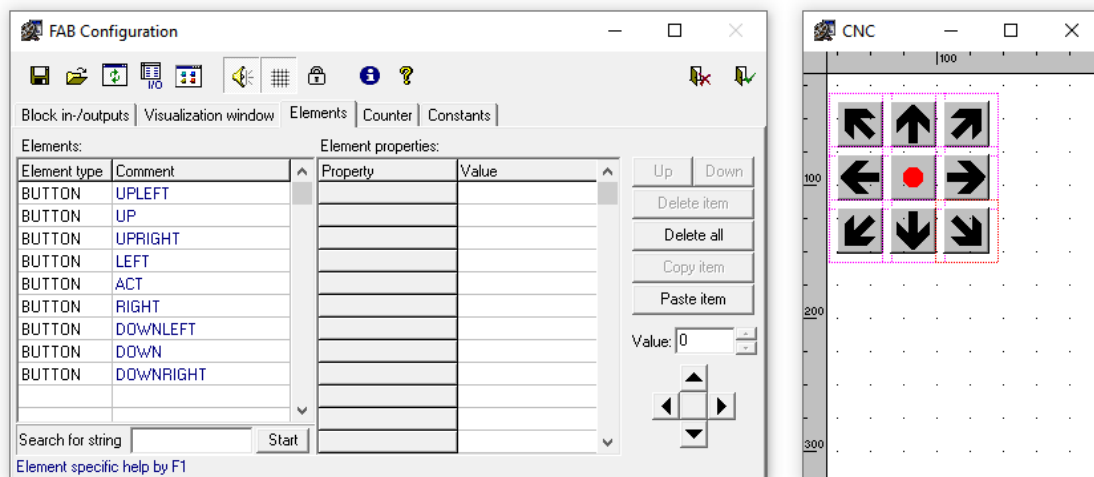
- A felhasználónak nyolc iránygomb és egy nyomógomb áll a rendelkezésére, amelyeknek a nyomva tartásával pozícionálja és működteti a szerszámot.
- A felhasználó által nyomva tartott nyomógomb-kombináció négy biten van kódolva úgy, hogy ha mind a négy bit magas logikai szinten van, akkor az a szerszám működtetését jelenti, ellenkező esetben a bitek magas logikai szintjeinek a jelentése a legmagasabb helyiértékű bittől a legalacsonyabbig $-Y$, $+Y$, $-X$ és $+X$.
- A folyamatban levő mozgató irányát négy darab LED jelzi a felhasználó számára, azzal, hogy ha mind a négy LED világít, akkor az a szerszám működését jelzi.
- A szerszámnak az X és Y tengelyek mentén történő mozgása előjelek megadásával történik, ami negatív irány esetén -1 , pozitív irány esetén $+1$, egyébként pedig 0 .
- A mozgató csak a végállások között lehetséges, vagyis azok a mozgató előjelek, amelyek hatására a szerszám túlmozdulna végállásain, mellőzésre kerülnek.
- A szerszám működtetése logikai szintek segítségével történik, aminek megfelelően a magas jelenti a működés megkezdését, az alacsony pedig a befejezést.

Tekintettel arra, hogy ezzel a példával a vizualizációkészítés bemutatása a cél, a kidolgozásban a nyomógombok állapotait szintén a szimuláció alakítja át előjelekre és a

szerszámot működtető logikai szintre. A működő kidolgozás a mellékelt adathordozón az */emulated-system/emulation/examples/hmi* könyvtárban található.

3.1. A nyomógombok megvalósítása

A nyomógombok részét képezik a FAB építőelemeinek, ezért ahhoz, hogy a vizualizáció egy új nyomógommbal bővüljön, elegendő megkeresni azt az Elements ablak Controls fülén és rákattintani. Amikor egy új elem kerül a FAB vásznára, egyúttal egy új bejegyzéssel bővül az FAB Configuration ablak Elements fülén található Elements lista is. Ebben a listában az aktuálisan kiválasztott elem tulajdonságai tetszés szerint módosíthatók a lista melletti Element properties felületen. A feladat ezen részének kidolgozásához kilenc darab nyomógombot kell hozzáadni a vizualizációhoz célszerűen 3×3-as elrendezésben, és az átláthatóság szempontjából az a jó, ha a nyolc iránygomb körül veszi a szerszámot működtető nyomógombot (3.1.1. ábra).



3.1.1. ábra: Nyomógombok elrendezése a vizualizációban

A nyomógombok jeleinek kimenetekké alakításához érdemes megfigyelni azt a tényt, hogy a nyolc iránygomb értelemszerűen négy főirányra és négy mellékirányra bontható, ahol a főirányok az *X* és *Y* tengelyek mentén történő pozitív és negatív irányú mozgásokat jelentik, a mellékirányok pedig azokat a főiránypárokat, amelyek egy *X* és egy *Y* irányú mozgásból állnak. Ezek alapján ugyanis a főirányok megfeleltethetők a nyomógombok állapotát reprezentáló bitsorozatnak, a mellékirányok pedig kifejezhetők két főiránnyal ugyanezen a bitsorozaton. Továbbá észre kell venni azt is, hogy a szerszám működtetésének bitsorozata a négy főirány egyidejű lenyomásával is kiváltható.

Első megközelítésre talán az tűnhet a legészszerűbb megoldásnak, ha a FAB blokk négy kimenettel rendelkezne, mindegyik főirányhoz eggyel, a mellékirányok és a

szerszámtól működtető nyomógomb pedig ugyanezeket a kimeneteket használnák, esetleg a főirányok nyomógombjainak tulajdonságait felhasználva. A FAB azonban nem ennyire rugalmas. Igaz, hogy egyes tulajdonságok értékei kifejezésekkel is megadhatók, de ezekben a kifejezésekben csak a blokk be- és kimeneteire lehet hivatkozni, elemek tulajdonságaira nem, továbbá egy elem csak egy kimenetet képes befolyásolni. Mindezek fényében viszont az a legkézenfekvőbb irány, ha minden nyomógomb különálló kimenetet kap, azonban azt kihasználva, hogy a kimenetek lehetnek rejtettek is és az értékeik ettől függetlenül felhasználhatók akár egy látható kimenet értékének az előállításában is, teret ad egy elegánsabb kerülőmegoldásnak.

A kerülőmegoldáshoz először is szükség van egy segédelemre, amelyik képes a hozzá tartozó kimenet értékét kifejezéssel is megadni. Erre közvetlenül sajnos egyik elem sem képes, de az Elements ablak Controls fülén található Checkbox például, ha ki van jelölve, akkor az OnValue tulajdonságának értékét írja a kimenetére, ami már kifejezéssel is megadható. Ezen a ponton fontos rávilágítani a FAB kifejezések azon hiányosságára is, hogy egyáltalán nem támogatják a logikai műveleteket. Jelen esetben viszont mindaddig, amíg egyszerre csak egy nyomógomb tartható lenyomva, ezáltal egyszerre csak az egyik kimenete lehet nullától eltérő, az is megoldja a problémát, ha a nyomógombok mindegyike a teljes bitsorozatot írja ki a kimenetére (3.1.1. táblázat), a segédelem pedig egyszerűen csak összeadja ezeket.

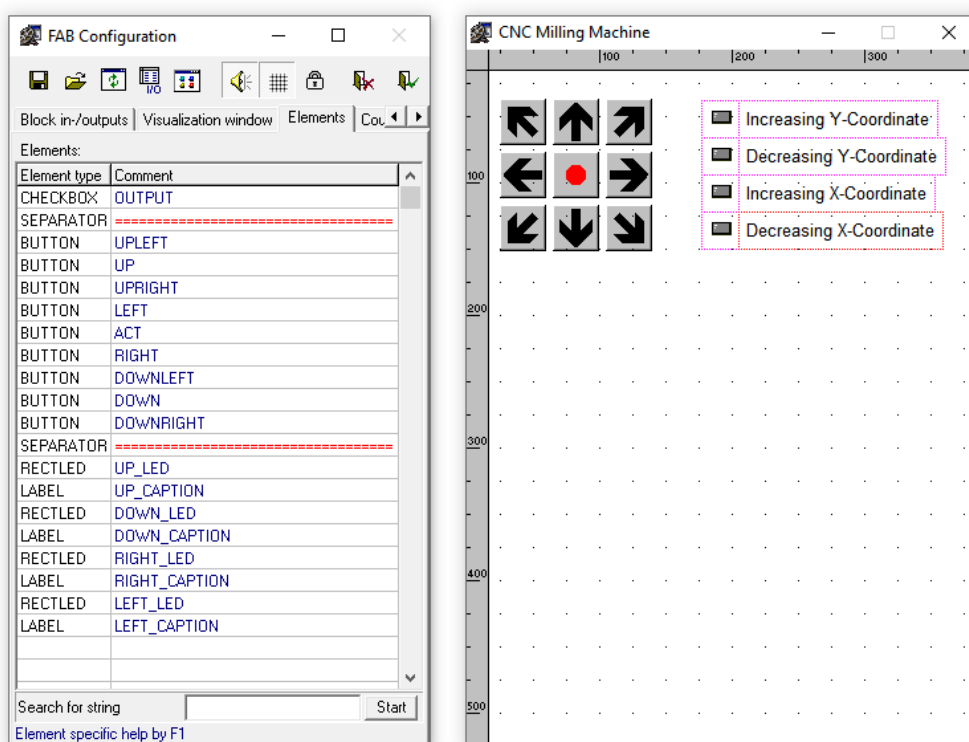
Element	Output	OnValue
OUTPUT	1	O2+O3+O4+O5+O6+O7+O8+O9+O10
UPLEFT	2	10
UP	3	8
UPRIGHT	4	9
LEFT	5	2
ACT	6	15
RIGHT	7	1
DOWNLEFT	8	6
DOWN	9	4
DOWNRIGHT	10	5

3.1.1. táblázat: A kimenet előállításában résztvevő elemek tulajdonságai

A segédelemet miután ki lett jelölve, célszerű elrejtetni. Ennek a megtételére nincs egyetemes módszer, de az elem méreteinek a nullára állításával elérhető a kívánt hatás.

3.2. A mozgatás irányának és a szerszám működésének jelzése

A mozgatás irányának és a szerszám működésének jelzéséhez szükséges LED-ek szintén megtalálhatók a FAB építőelemei között az Elements ablak Display fülén. Első lépésként ezekből kell négy darabot hozzáadni a vizualizációhoz függőleges elrendezésben (3.2.1. ábra), majd a jelzett állapotok könnyebb megértése érdekében érdemes mindegyiket ellátni egy-egy címkével is, amire legjobban az Elements ablak Graph fülén található Static text használható.



3.2.1. ábra: A LED-ek és a címkék elrendezése a vizualizációban

Mivel a megjelenítendő állapotok megegyeznek a nyomógombok által előállított kimenet bitjeivel, a feladat ezen részének kidolgozásához célszerű ennek a kimenetnek az értékét felhasználni és az egyes bitjeit összekötni a megfelelő LED-ekkel. Tekintettel arra, hogy a FAB kifejezések nem támogatják a logikai műveleteket, az egyes bitek értékét eltolás és maszkolás helyett ki kell számítani a (3.2.1) egyenlet segítségével, ahol az x a kimenet aktuális értéke, az n pedig a kiszámítandó bit sorszáma.

$$b_n = \left\lfloor \left(\frac{x}{2^n} \right) \bmod 2 \right\rfloor \quad (3.2.1)$$

A FAB kifejezések több matematikai függvény használatát is lehetővé teszik, így például a szignumfüggvényt is, de érdekes módon a maradékképzést egyáltalán nem támogatják,

ezért ahhoz, hogy a (3.2.1) egyenlet leírható legyen FAB kifejezésként, el kell végezni a (3.2.2) behelyettesítést [12], aminek az eredménye a (3.2.3) egyenlet lesz.

$$a \bmod b = a - b \left\lfloor \frac{a}{b} \right\rfloor \quad (3.2.2)$$

$$b_n = \left\lfloor \frac{x}{2^n} - 2 \left\lfloor \frac{\frac{x}{2^n}}{2} \right\rfloor \right\rfloor$$

$$b_n = \left\lfloor \frac{x}{2^n} - 2 \left\lfloor \frac{x}{2^{n+1}} \right\rfloor \right\rfloor \quad (3.2.3)$$

Egy LED akkor fog világítani, ha az Input tulajdonságának értéke nagyobb vagy egyenlő az OnValue tulajdonságának értékénél. Amennyiben az Input tulajdonsága a (3.2.3) egyenlet szerint változnak, aminek az értéke 0 vagy 1 lehet, akkor az OnValue tulajdonságát is 1-re kell állítani. Figyelembe véve, hogy az OnValue tulajdonság nem követeli meg az Input tulajdonságtól, hogy egész szám legyen, a (3.2.3) egyenletben a végső egészre kerekítést el is lehet hagyni. Mindezeket felhasználva, illetve a bitek sorszámainak és a kimenet referenciájának behelyettesítése után a LED-ek releváns tulajdonságai a 3.2.1. táblázat szerint fognak alakulni.

Element	Input	OnValue
UP_LED	$(O1 / 8) - (2 * \text{INT}(O1 / 16))$	1
DOWN_LED	$(O1 / 4) - (2 * \text{INT}(O1 / 8))$	1
RIGHT_LED	$O1 - (2 * \text{INT}(O1 / 2))$	1
LEFT_LED	$(O1 / 2) - (2 * \text{INT}(O1 / 4))$	1

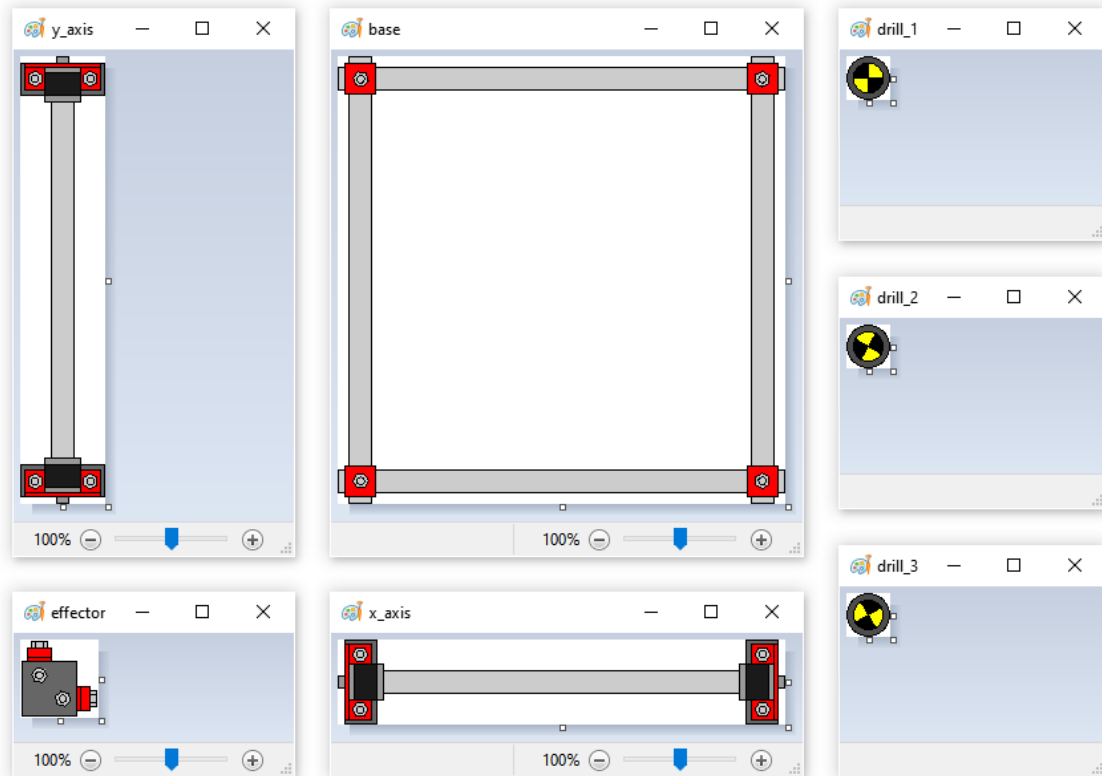
3.2.1. táblázat: A mozgatus irányát és a szerszám működését jelző LED-ek tulajdonságai

A FAB kifejezésekben az INT operátor az operandusának a törtrészét lefelé kerekíti függetlenül a törtrész nagyságától, másszóval az INT operátor úgy kerekíti egészre az operandusát, hogy az egészrészt megőrzi, a törtrészt pedig elhagyja.

3.3. A szerszám animációinak megvalósítása

A FAB elsősorban a saját építőelemeinek és animációinak a használatát támogatja és meglehetősen korlátozottak a képességei az egyéni animációk készítése terén, annak ellenére, hogy a FAB betűszó valójában rugalmas animációkészítő (Flexible Animation Builder) jelent. Valójában a megjelenítésen kívül semmi egyébbe nem tud hozzájárulni az egyéni animációkhoz, vagyis a megjelenített képektől kezdve az animáció tulajdonságain át a mozgatus logikájáig mindent kívülről kell biztosítani hozzá.

A szerszám animációinak megvalósításához első lépésként a CNC marógép minden egyes megjelenítendő részét, illetve a szerszám működését vizualizáló mozgóképnek minden egyes képkockáját meg kell rajzolni. Erre a feladatra bármelyik rajzóprogram felhasználható, egyedül arra kell ügyelni, hogy a képkockák és a mozgatás szempontjából különálló elemek külön képfájlba kerüljenek (3.3.1. ábra).

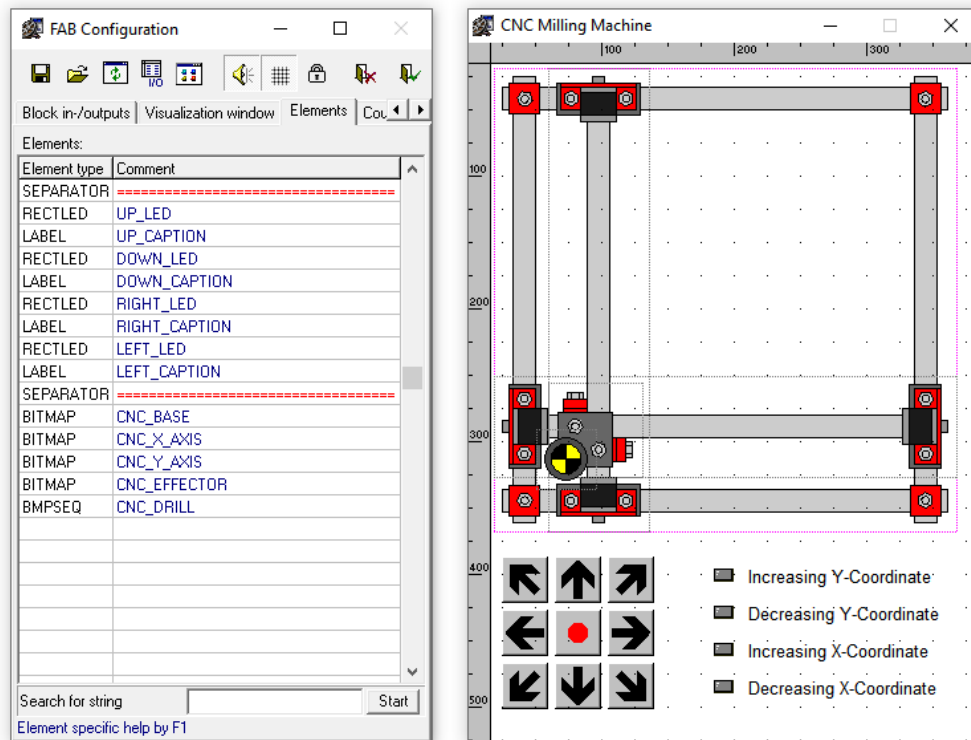


3.3.1. ábra: A CNC marógép megrajzolt elemei

A feladat ezen részének kidolgozásához az elkészült rajzokat az Elements ablak Graph fülén található Bitmap segítségével hozzá kell adni a vizualizációhoz a 3.3.2. ábra szerinti elrendezésben, illetve a szerszám működésének képkockáit hozzá kell adni egy Bitmap sequence elemhez, amelyik ugyanennek az ablaknak a Display fülén található. Az elrendezés mellett a rétegezésre is ügyelni kell, ugyanis az az elem, amelyik lejjebb van, eltakarja a felette levőket, ezért az elemeknek a FAB Configuration ablak Elements fülén található Elements listában szintén a 3.3.2. ábra szerinti sorrendet kell követniük.

A képek mozgatása és a mozgókép lejátszása legkönnyebben számlálók segítségével valósítható meg, amelyeknek az aktuális értékeit felhasználva indexelhetők a képkockák, illetve kifejezések írhatók a mozgó elemek X és Y koordinátáinak kiszámítására. A FAB ugyan rendelkezik beépített számlálókkal, amelyek az I51, I52 és I53 virtuális bemenetekre vannak kötve, de ezeket az időzítőket nem lehet sem

megállítani, sem a sebességüket módosítani, és mivel ezek a szimuláció ciklusidejével számolnak, túl gyorsak ahhoz, hogy egy animációt működtessenek. Éppen ezért ezen a ponton már a BORIS segítségére van szükség, hogy megvalósítsa az animációkhoz szükséges számlálót és összekösse azokat a FAB blokk bemeneteivel.



3.3.2. ábra: A feladatot kidolgozását jelentő vizualizáció

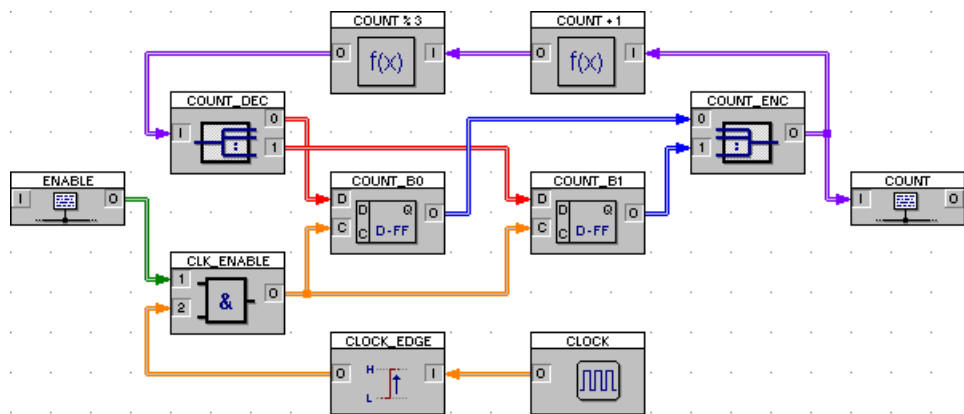
A BORIS blokkjai között megtalálható számláló (Counter) több szempontból is meghaladja a FAB számlálóinak a képességeit, így például bármikor újraindítható és a számlálás iránya is változtatható, viszont a működéséhez órajel generátorra van szüksége és a FAB számlálóival szemben ennek nem lehet megadni egy felső korlátot, ami felett újraindulna a számlálás. Mindezek ellenére ezzel a blokkal már meg lehetne valósítani az animációk működtetését, a hiányosságai ugyanis további blokkok hozzáadásával bármikor pótolhatók. Ami viszont ellene szól, az az, hogy sokszor egyszerűbb a megfelelő kimenetet előállító számlálót összerakni, mint ezt a blokkot kiegészíteni, ugyanis egy számláléhoz mindössze egy memóriára van szükség az aktuális érték eltárolásához, és egy számláló logikára a következő érték előállításához.

A képkockák indexelését végző számlálónak (3.3.3. ábra) a (0, 1, 2) sorozatot kell folyamatosan ismételnie mindaddig, amíg az engedélyező bemenete (ENABLE) magas logikai szinten van.

Ehhez a számlálóhoz elegendő egy 2-bites memóriát megvalósítani az aktuális érték eltárolásához, hiszem a sorozat mindegyik eleme kifejezhető két biten. A memóriát flip-flopok és egy órajel generátor segítségével kell megvalósítani, a számlálás felfüggesztését és folytatását pedig az órajel kapuzásával. Ahhoz, hogy az aktuális értékből minden alkalommal elő lehessen állítani a számláló következő értékét, a BORIS blokkjaival a (3.3.1) függvényt kell megvalósítani, ahol a c_n az aktuális értéket jelenti, a c_{n+1} pedig a számláló következő értékét.

$$c_{n+1} = (c_n + 1) \bmod 3 \quad (3.3.1)$$

Tekintettel arra, hogy a BORIS sem támogatja a maradékképzést, az áltáthatóság érdekében érdemes a (3.3.1) függvényt két különálló blokkal megvalósítani, ahol az első $(COUNT + 1)$ csak növeli a bemenetét eggyel, a második $(COUNT \% 3)$ pedig a (3.2.2) behelyettesítést alkalmazva elvégzi a maradékképzést.



3.3.3. ábra: A képkockák indexelését végző számláló

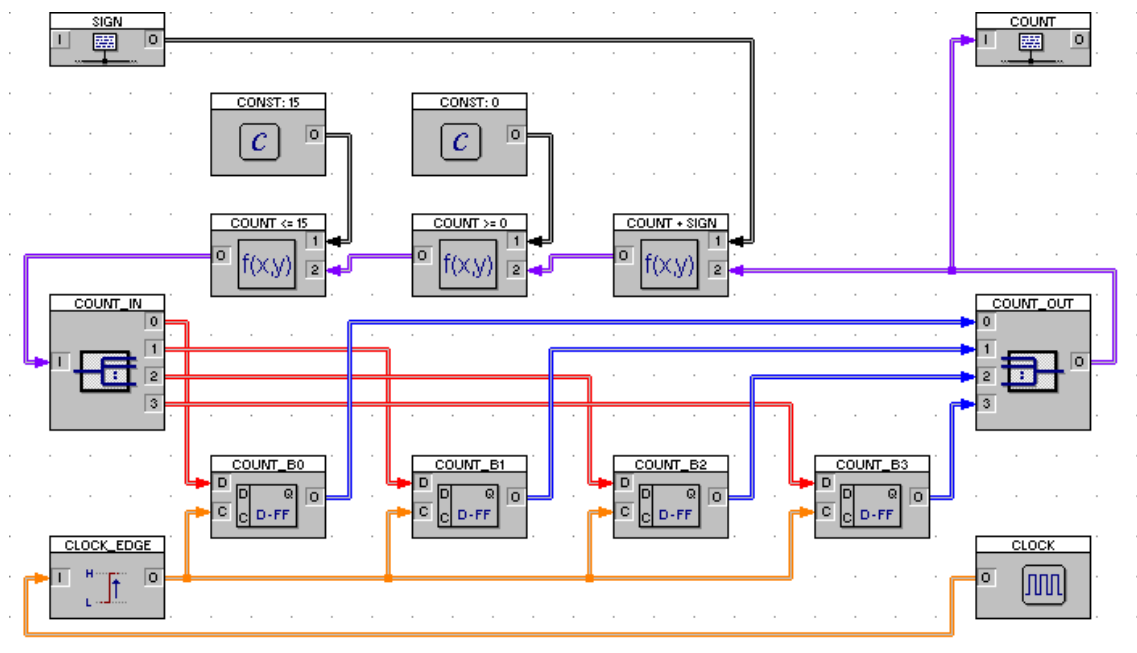
A mozgókép meghajtásához meg kell oldani, hogy a mozgókép képkockáit a vizualizációban ennek a számlálónak a kimenete (COUNT) indexelje. Ehhez először is létre kell hozni egy új bemenetet a FAB blokkon, majd ezen keresztül össze kell kötni a számláló kimenetét a vizualizációval. Amint ez megvalósult, az indexeléshez még meg kell adni a mozgókép Input tulajdonságának a számlálóval összekötött bemenet sorszámát, illetve be kell állítani az indexek sorozatának a minimumát és a maximumát a RangeMin és a RangeMax tulajdonságok segítségével.

A mozgatóst működtető számlálónak (3.3.4. ábra) két érték között kell oda-vissza számolnia attól függően, hogy a bemenetén levő előjel pozitív vagy negatív, illetve meg kell állítania a számlálást, ha a számláló következő értéke kívül esik a számlálási intervallumon vagy a bemenetén levő érték nulla.

A kidolgozás egyszerűsítésének érdekében elegendő a számlálót a $[0, 15]$ intervallumra megvalósítani, mivel már egy ekkora intervallummal is elfogadható eredményt lehet kapni, miközben a számláló aktuális értékeinek az eltárolásához elegendő egy mindössze egy 4-bites memóriát megvalósítani. A képkockák indexelését végző számláléhoz hasonlóan a memóriát itt is flip-flopokkal és egy órajel generátorra kell megvalósítani, viszont az órajel kapuzása ezúttal elhagyható, ugyanis a számlálás felfüggesztése és folytatása a számláló következő értékét előállító logika által implicit megvalósul. Ezt a logikát a (3.3.2) függvény írja le, ahol a c_n az aktuális értéket jelenti, az s az előjel bemenet értékét, a c_{n+1} pedig a számláló következő értékét.

$$c_{n+1} = \min(\max(c_n + s, 0), 15) \quad (3.3.2)$$

Ebből a függvényből jól látható, hogy ha az s értéke nulla, a c_{n+1} értéke egyenlő lesz a c_n értékével, így folytonos órajel mellett sem fog megváltozni a kimenet értéke. Ugyanez akkor is igaz, ha a c_n értéke 0 és az s értéke -1 , vagy a c_n értéke 15, az s értéke pedig 1. A BORIS rendelkezik úgy a min, mint a max függvényekkel, de csak a két bemenettel rendelkező függvény blokkok esetén, így a (3.3.2) függvényt szükségszerűen három különálló blokkal kell megvalósítani.



3.3.4. ábra: A mozgatót működtető számláló

A szerszám mozgatásához először is létre kell hozni két példányt ebből a számlálóból, ezt követően ki kell bővíteni a FAB blokkot két új bemenettel, végül pedig össze kell kötni létrehozott számlálók kimeneteit a két új bemenettel. Innentől kezdve a vizualizációban a két új bemenet tetszőleges módon felhasználható a mozgó elemek x és

y tulajdonságaihoz tartozó kifejezésekben. Annak köszönhetően, hogy a számlálók egy független intervallumon számlálnak, a vizualizációban a megfelelő szorzó és kezdőérték megadásával bármelyik elem mozgásához felhasználhatók. A 3.3.2. ábra szerinti elrendezésben minden mozgó elem x és y tulajdonságai már rendelkeznek a megfelelő kezdőértékkel, így ezekhez csak hozzá kell adni a megfelelő bemenet értékét a megfelelő szorzóval. Ahhoz, hogy az elemek összhangban mozogjanak, a számlálókkal összekötött bemenetek értékeire mindenhol ugyanazokat a szorzókat kell alkalmazni. Jelen esetben ezek a szorzók a szerszám végállásai közötti távolságok tizenötöd részei.

A mozgás szempontjából a $[0, 15]$ intervallum valójában azt jelenti, hogy a szerszám mindössze 15 különböző helyzetet tud felvenni az X vagy az Y tengelyek mentén, ami egy szemmel láthatóan szaggatott animációt eredményez. Természetesen az animáció minőségén bármikor javítani lehet az intervallum és az órajel frekvenciájának együttes növelésével, viszont ezzel együtt a futtatókörnyezetre mért terhelés is növekedni fog. Ez a terhelés azt jelenti, hogy a futtatókörnyezetnek gyakrabban kell a teljes vizualizációt újrarajzolnia, ami mindaddig nem okoz gondot, amíg az újrarajzolás kevesebb időt igényel, mint a számláló periódusideje. Ellenkező esetben azonban az újrarajzolási feladatok feltorlódnak, ami azt a látszatot kelti, mintha az animáció késne vizualizálandó szimulációhoz képest.

4. Az ipari folyamat emulációja és irányítása

Az emulálni kívánt ipari folyamat részletes leírásául a XX. Országos Irányítástechnikai Programozó Verseny feladatlapja szolgált, ami egy karosszéria gyártósor működését írja le és részletes instrukciókat ad az irányítás megvalósításához.

Ezen a feladatlapon minden információ rendelkezésre állt a megvalósításához, így az igazi kihívást nem a feladat megértése, sokkal inkább a HIL szimulátor ilyen célú felhasználása jelentette.

4.1. Az emuláció megvalósításának bemutatása

A versenyfeladatot jelentő karosszéria gyártósor alapvetően nem egy túl összetett ipari folyamat, a megvalósításának az összetettségét nem is PLC-től érkező jelek feldolgozása és a megfelelő kimenetek előállítása jelenti, hanem a vizualizálásának a nagyfokú animációigénye.

Mivel a 3. fejezetben már részletesen bemutatott FAB jelentős hiányosságokkal rendelkezik az animációk terén, az elkészült emuláció összetettsége olyan mértékű lett, hogy nem lenne célszerű minden egyes pontját részletesen bemutatni. Ennek okán ebben az alfejezetben a fókusz a megvalósítás koncepciójának az ismertetésére korlátozódik az implementációs részletek mellőzése mellett. Így viszont a koncepciók megértéséhez elengedhetetlen az az előtudás, amit egyrészt Görbedi Ákos diplomamunkája, másrészt a 3. fejezetben bemutatott interaktív vizualizációkészítés hordoznak.

A megvalósított emuláció és a BORIS használatának könnyebb megértését megcélzó példaszimulációk a mellékelt adathordozón az */emulated-system/emulation* könyvtárban találhatók, az emuláció legfelső szintjének teljes diagramja pedig megtalálható a mellékletek között az M12 sorszám alatt, a vizualizációról készült képernyőfotó pedig az M13 mellékletben tekinthető meg.

4.1.1. A cellák megvalósításának alapjai

A megvalósítandó vizualizációtól függően a FAB egy igen jól alkalmazható szoftver is lehet, ugyanis számos előre elkészített interaktív és animált elemet biztosít a BORIS szimulációkhoz. A karosszéria gyártósor megvalósításához azonban ezek közül szinte egyik sem volt megfelelő.

Az első nehézséget a megfelelő nézet megválasztása jelentette. Az eredeti versenyfeladat mozgásai ugyanis 3-dimenziós térben történnek, ahol a robotok mozgása legjobban felülnézetben jeleníthető meg, a karosszériák haladása oldalnézetből, a daru

mozgása pedig előlnézetből. Az első próbálkozás az oldalnézet volt, mivel a FAB már rendelkezik egy ilyen nézettel rendelkező futószalag animációval, de a robotok ebben a nézetben annyira felismerhetetlenek lettek volna, hogy a felülnézet vált a legjobb választássá (4.1.1. ábra). Robotkar, vagy bármi arra emlékeztető, előre animált elem, azonban nincs a FAB eszköztárában semmilyen nézetben sem, így ezek végül az Elements ablak Process fülén található Cylinder elemekkel lettek helyettesítve. Természetesen a felismerhetőbb eredmény érdekében ezek a Cylinder elemek jelentős testre szabáson estek át, amik nagyjából a következőkben foglalhatók össze:

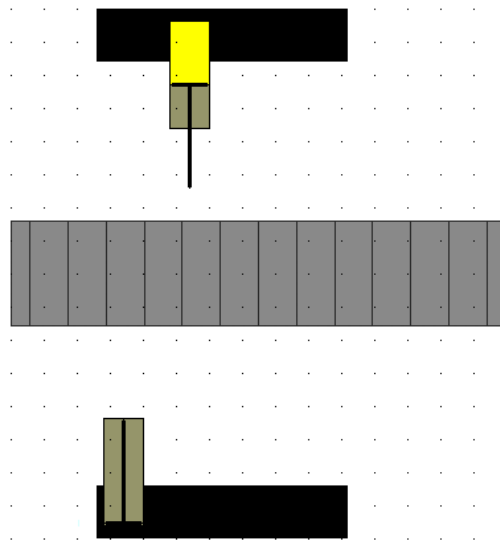
1. A dugattyúrúd végén levő lapka (SliderSize) el lett tüntetve nulla méret megadásával, illetve a működtető közeg (RGBFill) és az első kamra (RGBBack) színei a robotkar burkolatát jelképező színekre lettek módosítva.
2. A robotkar X irányú mozgása a (4.1.1) kifejezéssel lett megadva, ahol az x_{be} a vizualizációtól független X koordinátát jelöli.

$$x = x_{kezdő} + (x_{be} \cdot 2) \quad (4.1.1)$$

Az Y irányú mozgás ezzel szemben a dugattyú állapotát százalékban meghatározó tulajdonsághoz (SliderPos) lett társítva a (4.1.2) kifejezésnek megfelelően.

$$SliderPos = y_{be} \cdot 2 \quad (4.1.2)$$

3. A robot teste, amin a robotkar jobbra és balra tud haladni, egy négyszög (Rectangle) elemmel lett megvalósítva. Ennek az elemnek a színe (RGBFill) feketére lett festve, rétegezés szempontjából pedig a robotkar takarásába került.



4.1.1. ábra: Az egyik futószalag és a két hegesztőrobot vizualizációja

A futószalag felülnézeti animációjának megvalósításához minden képkockát külön meg kellett rajzolni, majd egy bitkép sorozathoz (Bitmap sequence) kellett

hozzáadni. Ez az elem egyébként mindössze 10 képkocka abszolút útvonalának megadását teszi lehetővé (BMPFile[1...10] or Id) és csak kizárólag BMP formátumban. Ezzel az elemmel az animáció kétféleképp működtethető:

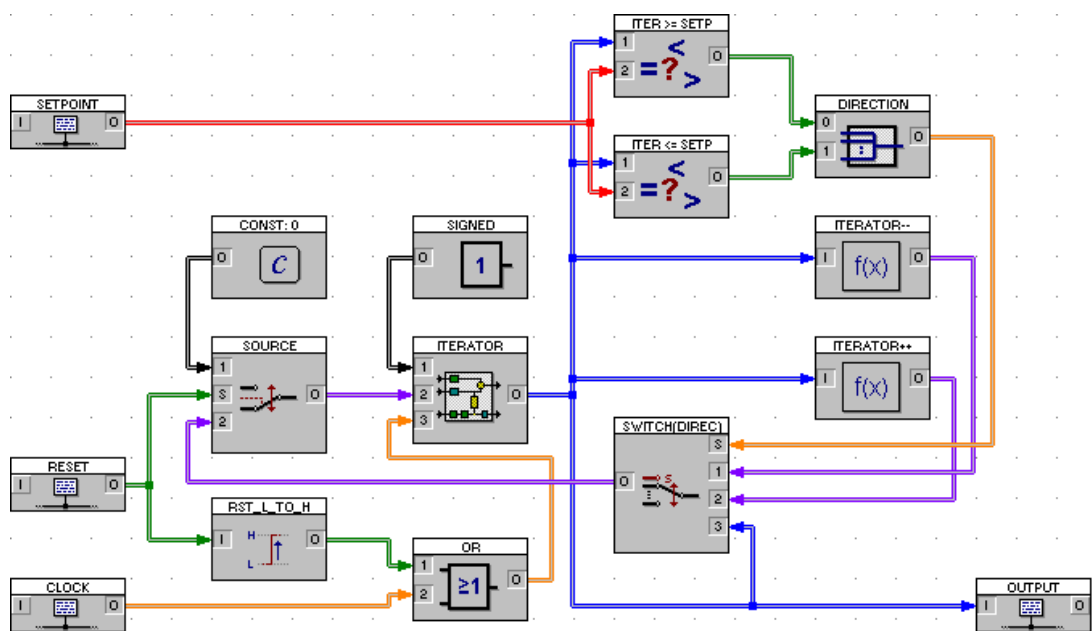
- Végtelenített üzemmódban, ahol a képkockák közötti váltás sebessége a megadott késleltetéstől (Delay) függ. Ehhez meg kell adni a képkockák számát (BMPCount) és nullára állítani a bemenetet (Input).
- Vezérelt üzemmódban, ahol a képkockák közötti váltást egy külső jel vezérli. Ehhez a bemenetnek (Input) meg kell adni a FAB blokk azon bemenetének sorszámát, amelyről az indexelést fogadja majd.

Egy olyan futószalag, amelyik külső jellel indítható el és állítható meg értelemszerűen csak a második megoldással valósítható meg.

Az első két cella vizualizációja egyébként csak a robotok színeiben tér el, illetve a robotok mozgásaiban, ami a technológia leírásából adódik, minden másban egymás másolatai. A harmadik cellában levő daru vizualizációja viszont független a celláktól, és inkább a karosszériákkal áll kapcsolatban, ugyanis hatással van azok helyzetére.

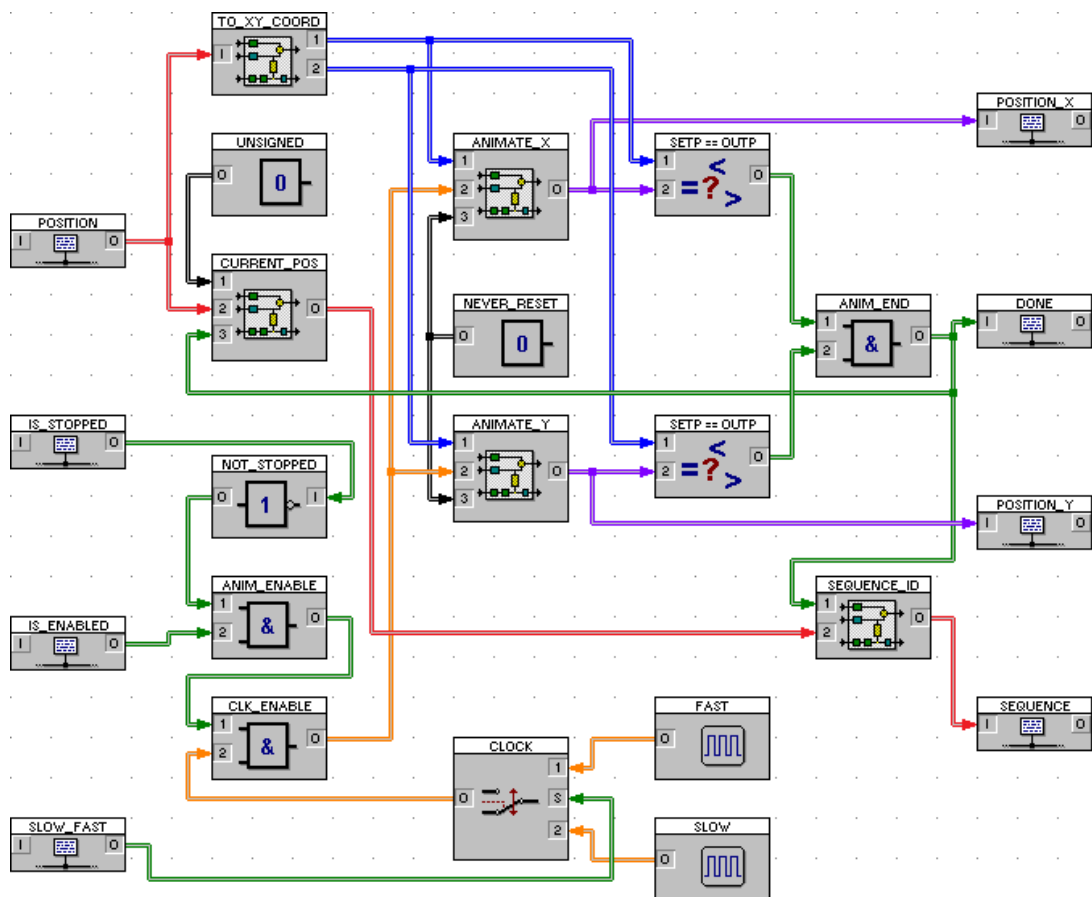
4.1.2. A robotok mozgatása

Majdnem minden mozgás alapját egy egyéni számláló valósítja meg. Ez a számláló nem növekvő vagy csökkenő számlálás végez, hanem egy előre beállított érték felé számlál az éppen aktuális értékétől (4.1.2. ábra).



4.1.2. ábra: Az animációk számlálójának egyszerűsített blokkdiagrammja

Ennek a számlálónak a magja egy 16-bites, előjeles memória, amelyik a megadott órajel minden felfutó élére eltárolja a bemenetére adott értéket. Ez az érték valójában a saját kimenetének eggyel nagyobb, kisebb vagy változatlan számértéke, amit a meghatározott célérték és a memória aktuális értéke közötti viszony dönt el. Ha a memóriában levő érték nagyobb a célértéknél, a memóriába minden órajelre a benne levő értéktől eggyel kisebb kerül, ha kisebb, akkor pedig a benne levő értéktől eggyel nagyobb. Abban az esetben, ha a memóriában levő érték és a célérték megegyezik, a memóriába a már benne levő számérték íródik vissza. Ez látszólag egy algebrai hurok, ahol a bemenet függ a kimenettől és fordítva, azonban a memóriát alkotó flip-flopoknak van kezdeti állapota, a bemeneteiken levő értékek pedig csak az órajel felfutó élre írják felül a tárolt tartalmaikat.



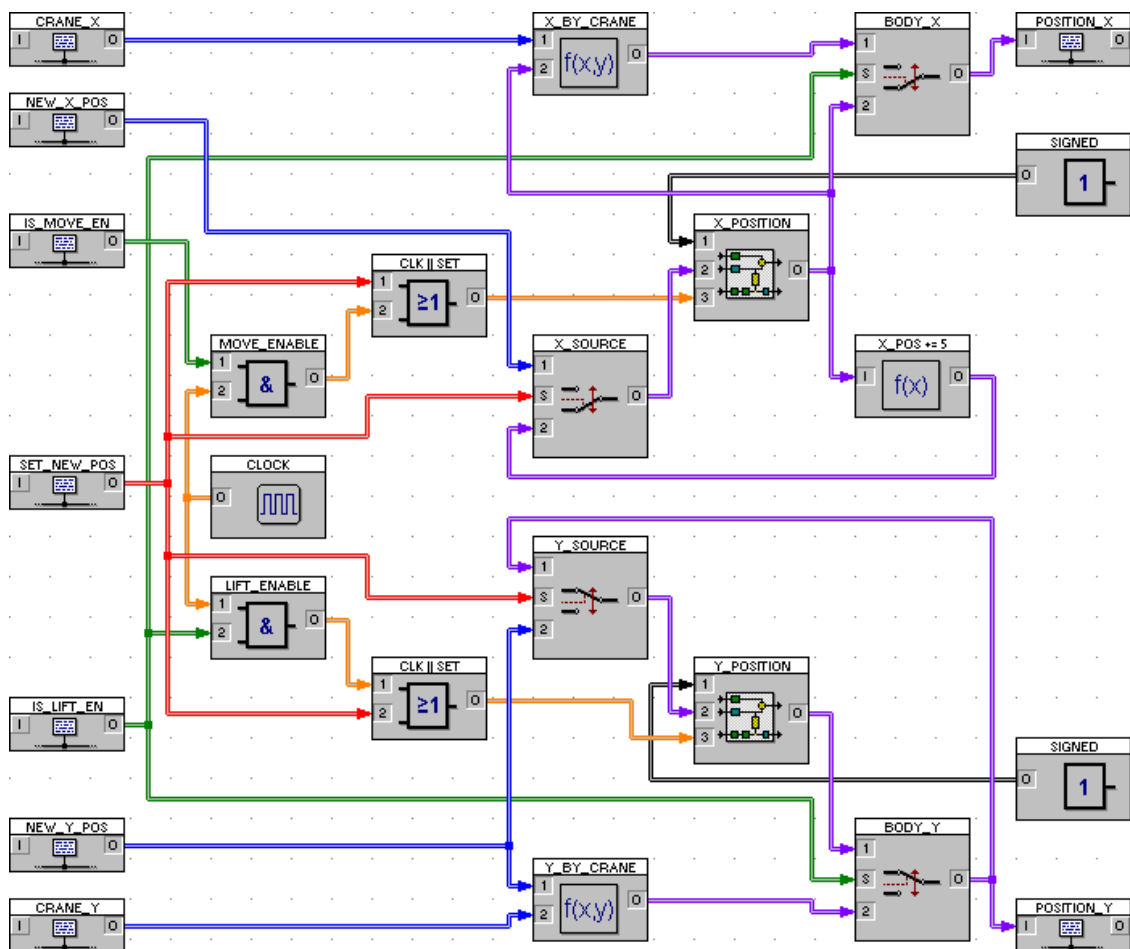
4.1.3. ábra: A robotok mozgásának egyszerűsített blokkdiagrammja

A robotok mozgása (4.1.3. ábra) két számlálót igényel (ANIMATE_X és ANIMATE_Y), amelyek az X és Y irányú mozgásokért felelősek. Ezeknek a számlálónak a célértékeit a TO_XY_COORD elem biztosítja, amelyik a vezérlőtől fogadott új pozíciókat a vizualizációtól független X és Y koordinátákká alakítja. A vizualizációban ezekből a koordinátából a 4.1.1 alfejezetben ismertetett módon lesznek

tényleges koordináták. A mozgatás két sebességfokozatát két különböző frekvenciájú órajel generátor teszi lehetővé (FAST és SLOW), amik között a sebességválasztó bemenet (SLOW_FAST) kapcsol. Minden robot megőrzi a már elért aktuális pozícióját, vagyis, amíg mozgásban van, ez az érték a korábban elért pozíciót jelenti. Erre azért van szükség, mert ez alapján érvényesíteni lehet a vezérlőtől érkező következő pozíciót, illetve nyomon lehet követni a robot mozgását.

4.1.3. A karosszériák mozgatása

A technológia legnehezebben vizualizálható elemeit a karosszériák jelentik, ugyanis a harmadik cellában annak a látszatát kell kelteni, mintha a daru ténylegesen képes lenne megfogni azokat és magával vonszolni. Ugyanakkor, amíg a futószalagokon mozognak, az érzékelőkkel is kölcsönhatásba kell kerülniük, hogy az irányítás számára detektálható legyen a hollétük, illetve behelyezhetőnek kell lenniük minden egyes cellába. Mindezek csak vizualizációban elfoglalt tényleges koordináták felhasználásával voltak kivitelezhetők, ezért a mozgatásuk erősen függ a vizualizációtól.

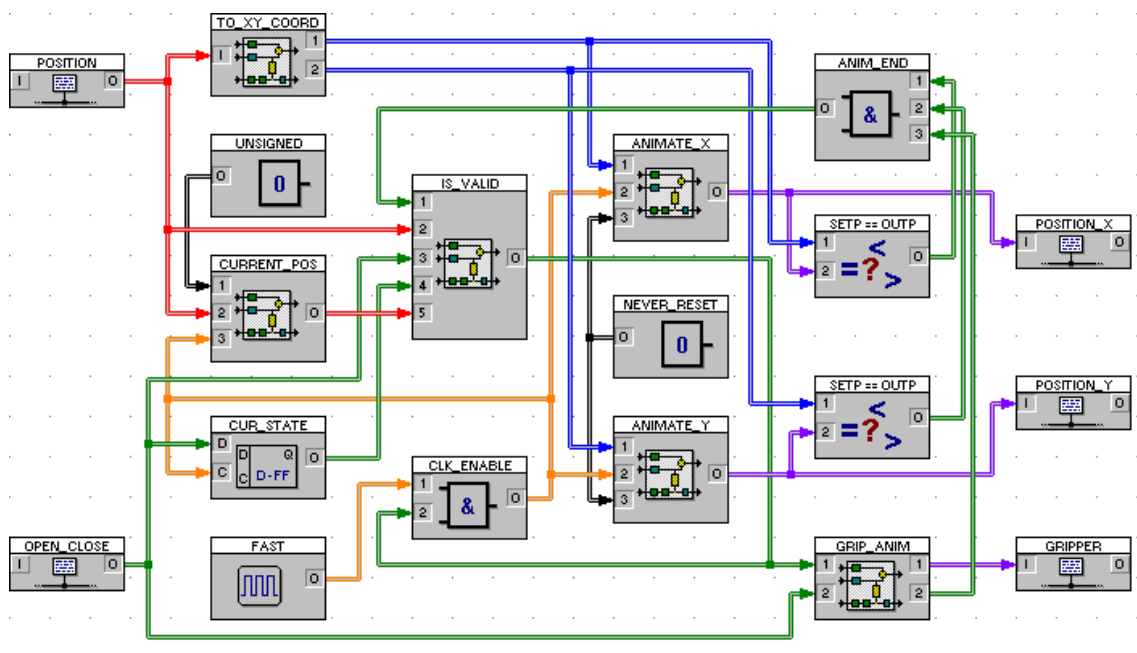


4.1.4. ábra: A karosszériák mozgatásának egyszerűsített blokkdiagrammja

A karosszériák mozgatásához (4.1.4. ábra) minden esetben rendelkezésre állnak az behelyezési koordináták (NEW_X_POS és NEW_Y_POS), illetve a daru mozgásának vizualizációtól független koordinátái (CRANE_X és CRANE_Y). Az, hogy a mozgatás végül melyik koordináták alapján történik meg, attól függ, hogy a karosszériára a mozgatási feltételek (IS_MOVE_EN) vagy az átemelési feltételek (IS_LIFT_EN) teljesülnek-e. A mozgatási feltételek teljesülése esetén a karosszéria az utolsó behelyezési koordinátától kezdve egyenes vonalú mozgást végez az X tengely mentén. Ennek a mozgásnak a pillanatnyi koordinátája kerül felhasználásra az érzékelők működtetésére, annak meghatározására, hogy a karosszéria éppen melyik futószalagon van, illetve arra is, hogy a daru megfogójának zárásakor átemelési pozícióban van-e. Az átemelési feltételek teljesülése esetén a karosszéria a daru megfogójának koordinátáit felhasználva mozog. Mivel ezek a koordináták már függetlenek a vizualizációtól, át kell alakítani olyan koordinátákká, amik már a vizualizációtól függenek. Ehhez minden esetben a megfogás pillanatában elfoglalt helyzetét használja fel.

4.1.4. A daru mozgatása

A daru mozgatása a legegyszerűbb mégpedig több okból is. Egyrészt daruból csak egy van, így nincs szükség választóbitekre, másrészt miután a daru megkezdte a mozgást, minden esetben be kell fejeznie azt, amiben még a vészstop sem akadályozhatja meg, harmadrészt a darunak csak egy sebességfokozata van.



4.1.5. ábra: A daru mozgatásának egyszerűsített blokkdiagrammja

A daru mozgatása (4.1.5. ábra) a robotokéhoz hasonlóan a vizualizációtól független X és Y koordináták segítségével történik, amelyeket a vezérlőtől érkező új pozíciókból a TO_XY_COORD elem hoz létre. Ezekből a vizualizációban szintén tényleges koordináták lesznek, akárcsak a robotok esetén, viszont ebben az esetben ezekre a karosszériák miatt van szükség. A karosszériák mozgatása ugyanis akkor természetes, ha a begyűjtési pozícióknak megfelelően mozognak tovább a daruval.

A daru megfogójának a működtetését külön szuperblokk végzi (GRIP_ANIM). A megfogó valójában két egymás felett elhelyezkedő mozgókép, amelyeknek a rétegződése úgy van kialakítva, hogy a villa a karosszériák alá, a karok pedig a karosszériák fölé kerüljenek. Ilyen módon a vizualizációban egy karosszériának a felvétele valóban azt a látszatot kelti, mintha a daru megfogója körbefogná azt.

4.1.5. A cellák ellenőrzőlogikái

A karosszéria gyártósornak három cellája van. A technológia feladatlapja részletesen kitér arra, hogy melyik cellában milyen irányítási szabályok érvényesek, amelyeket az irányítás megvalósításakor be kell tartani. Ezek röviden a következők:

1. Az egyes típusú karosszérián az egyes jelű robot, a kettes és hármas típusú karosszérián a kettes jelű robot végez ponthegeesztést. Az egyes jelű robotnak a (0, 4, 1, 4, 5, 2, 5, 6, 3, 6, 0) sorozat szerinti robotkonfigurációkat kell felvennie a hegesztés során, a kettes jelű robotnak pedig a (0, 6, 3, 6, 5, 2, 5, 4, 1, 4, 0) sorozat szerintieket. A futószalag csak a hegesztés befejezése után indítható.
2. A festőrobotoknak a (0, 1, 2, 3, 4, 0) sorozat szerinti robotkonfigurációkat kell felvenniük a festés során. A futószalag csak a festés befejezése után indítható.
3. A karosszériák típus szerinti leválogatását a daru végzi. Minden felvételi vagy lerakási pont megközelítése és elhagyása a hozzá tartozó megközelítési ponton keresztül történik. A daru egyszerre csak egy mozgást végezhet, vagyis emelkedhet, süllyedhet, vízszintesen haladhat, a nyithatja a megfogót vagy zárhatja a megfogót.

Tekintettel arra, hogy a HIL szimulációk általában tesztelési célok ellátására szolgálnak, ezeknek a szabályoknak a betartását az emuláció ellenőrzi. Természetesen mindegyik cellának megvan a maga ellenőrzőlogikája.

A robotok esetén a mozgatás (4.1.3. ábra) rendelkezik egy SEQUENCE_ID blokkal, amelyik minden pozíció felvételekor megkísérli az aktuális pozíciót ráilleszteni a három lehetséges sorozat valamelyikének a következő elemére. Amennyiben az aktuális pozíciót egy sorozatban nem egyezik az aktuális elemmel, az adott sorozatra

alaphelyzetbe állítja az indexet, ellenkező esetben növeli az indexet eggyel. Amennyiben valamelyik sorozat utolsó elemére is egyezést észlelt, visszaadja a sorozat azonosítóját. A robot által felvett pozíciók akkor kerülnek ellenőrzésre, amikor a karosszéria elhagyja az adott cellát, vagyis a cella érzékelőjének a kimenetén lefutó él detektálható.

A daru esetén a daru mozgásának az ellenőrzése az aktuális és a következő pozíciók összehasonlításával történik. Amennyiben az aktuális és a következő pozíció eltér, megtörténik az ellenőrzés. Ilyenkor a következő pozíció akkor érvényes, ha a daru:

- az alacsony és a hozzá tartozó magas pozíciók között mozog,
- két magas pozíció között mozog,
- alacsony pozíción nyitja vagy zárja a megfogókat,
- nem végez éppen valamilyen más mozgást.

Ezek mellett a minden egyes célállomásának megvan a maga ellenőrző logikája, amelyek felügyeli, hogy a leválogatás során a karosszéria a megfelelő platformra kerüljön.

Abban az esetben, ha az ellenőrzőlogikák bármelyike hibát jelezne, az emuláció hibajelzést küld a felhasználónak és terminálja önmagát.

4.2. Az irányítás megvalósításának bemutatása

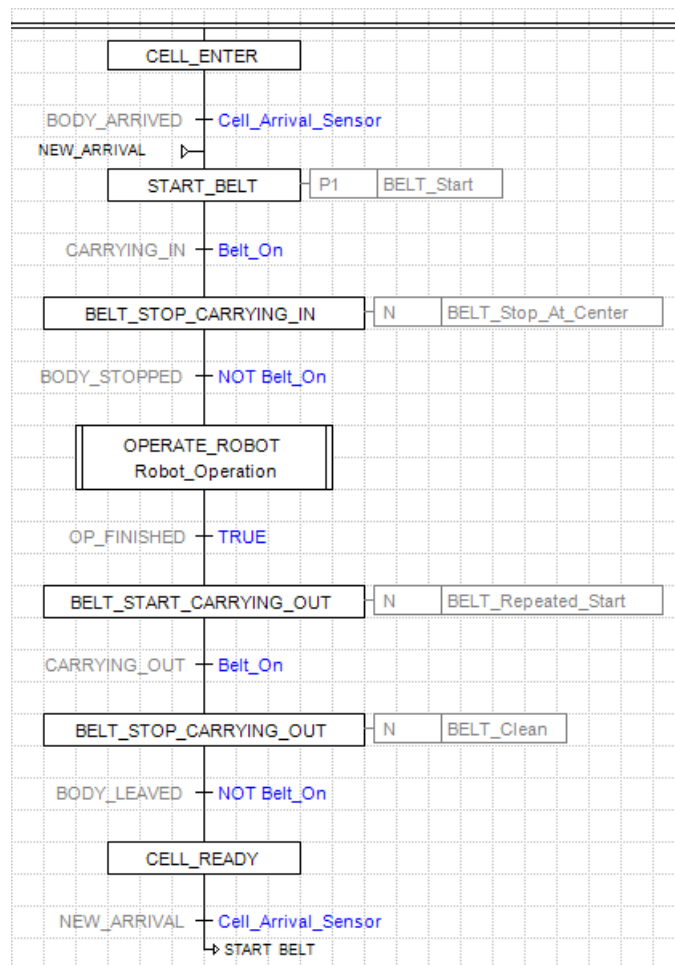
Az irányítás megvalósítása gyakorlatilag a versenyfeladat megoldását jelentette, csak ezúttal az emulált rendszer megfelelő működésének a tesztelése volt a cél, így az irányítás megvalósítása gyakran tartalmazott szándékos hibákat, hogy hibajelzésekre kényszerítse az emulációt. Erre azért volt szükség, mert ezáltal le lehetett tesztelni azt, hogy az emuláció az általa figyelt hibákat valóban észleli-e és ha igen, akkor küld-e hibajelzést, illetve, ha igen, akkor a megfelelő hibakóddal küldi-e azt.

A karosszéria gyártósor jellegzetessége, és egyúttal az irányítás megvalósításban a kihívás, hogy a cellák némiképp függetlenül üzemelnek egymástól. Egyedül a karosszériák létesítenek kapcsolatot a cellák között azáltal, hogy a karosszériák egyik cellából a másikba történő átmozgatásának az egyik feltétele az az, hogy a fogadó cella üres legyen. Mindez azt jelenti, hogy a cellákat egymástól közel függetlenül kell egyidejűleg irányítani. Az ilyen feladatok megoldására nagyon jó eredménnyel alkalmazható az SFC (Sequential Function Charts) grafikus programozási nyelv, ami azon túl, hogy jelentősen növeli a PLC programok áttekinthetőségét, támogatja a szimultán szekvenciák létrehozását is.

A megvalósított irányítás négy szimultán hurkot tartalmaz, amelyek egyenként a három cella irányításáért és az operátor biztonságáért felelnek. Az első két cella

irányítása (4.2.1. ábra) a kiadott robotkonfigurációkon és a használt be- és kimeneteken kívül minden másban megegyeznek. Leegyszerűsítve az irányítás a következő főbb műveletek ismételt végrehajtásából áll:

1. Futószalag indítása a cellába érkezést detektáló szenzor jelére.
2. Futószalag megállítása a megmunkálási pozícióba érkezést detektáló szenzor jelére.
3. A cellában levő karosszéria megmunkálása a megfelelő robot irányításával.
4. Futószalag indítása a megmunkálás befejeződésének jelére.
5. Futószalag megállítása a következő cellába érkezést detektáló szenzor jelére.



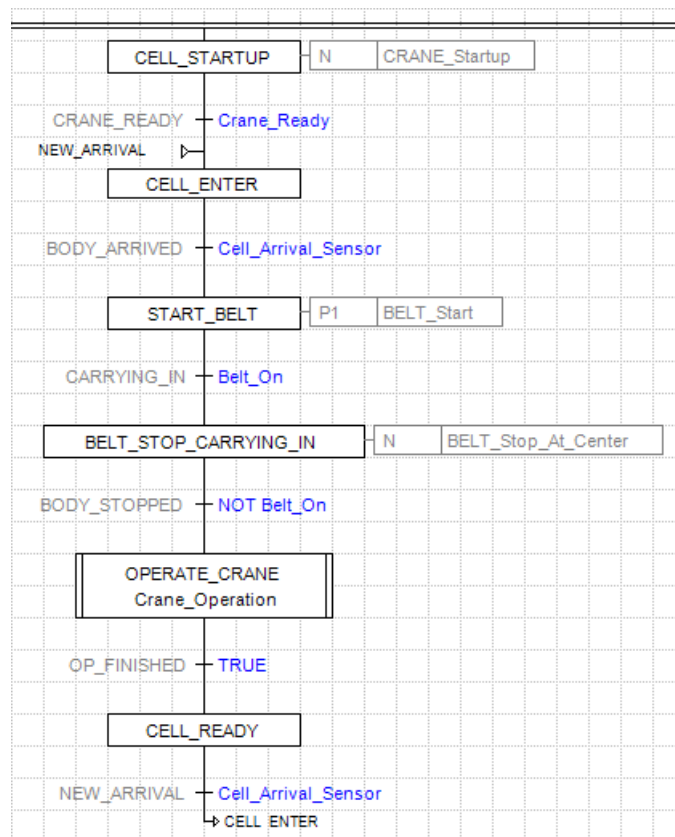
4.2.1. ábra: Az első két cella irányításának összevont és egyszerűsített diagramja

A harmadik cella irányítása (4.2.2. ábra) lényegében hasonló az előző két cella irányításához, azzal a szembetűnő különbséggel, hogy ebben a cellában a daru vezérlése után a futószalag már csak egy új karosszéria érkezésének a jelére lesz újból elindítva, egyébként nem, mivel ebből a cellából a karosszériák a daru által távoznak. Az ehhez a cellához tartozó irányítás leegyszerűsítve a következő:

0. A daru felemelése a karosszériák fogadására szolgáló kiinduló pozícióba.

1. Futószalag indítása a cellába érkezést detektáló szenzor jelére.
2. Futószalag megállítása a felvételi pozícióba érkezést detektáló szenzor jelére.
3. A cellában levő karosszéria áthelyezése a típusának megfelelő célállomásra.

Ennek a cellának még egy érdekessége, hogy külön inicializáló művelettel rendelkezik, aminek még az első karosszéria megérkezése előtt kell végrehajtódnia, viszont csak egyetlen egyszer.

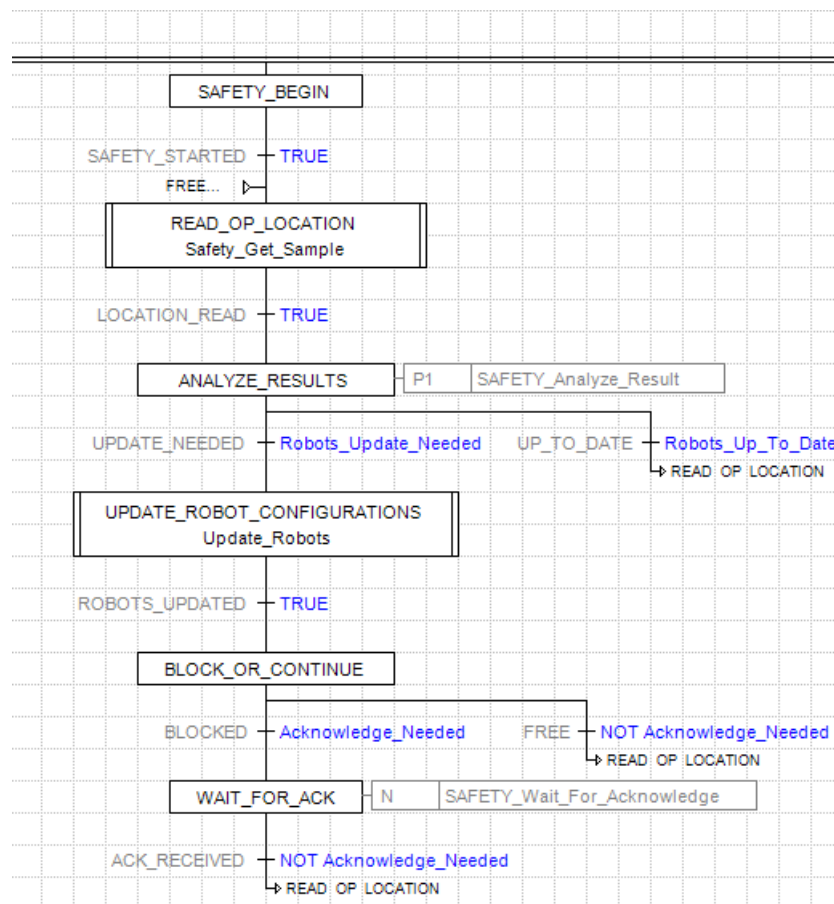


4.2.2. ábra: A harmadik cella irányításának egyszerűsített diagramja

Az operátor biztonságos munkavégzését lehetővé tevő logika (4.2.3. ábra) teljesen eltér a cellák irányításaitól. Míg az irányítás lényegében nem tartalmaz elágazásokat csak végrehajtja a műveleteknek egy előre definiált szekvenciáját, addig a biztonságért felelős logika több elágazást is tartalmaz a megvalósításának minden szintjén, ugyanis a folyamatos futása során a körülményeket vizsgálva döntéseket kell hoznia és szükség esetén bele kell avatkoznia az irányításba. Legfelső szinten leegyszerűsítve három hurkot tartalmaz, amelyek a következő műveletek végrehajtásából állnak:

1. Az operátor nem változtatott a tartózkodási helyén.
 - a. Az operátor mozgását jelző bit mintavételezése.
 - b. Az operátor tartózkodási helyének a megállapítása.
2. Az operátor tartózkodási helye megváltozott, de nincs szükség nyugtázásra.

- a. Az operátor mozgását jelző bit mintavételezése.
 - b. Az operátor tartózkodási helyének a megállapítása.
 - c. A robotok konfigurációinak módosítása az új állapotoknak megfelelően.
3. Az operátor tartózkodási helye megváltozott, és a folytatás nyugtázást igényel.
- a. Az operátor mozgását jelző bit mintavételezése.
 - b. Az operátor tartózkodási helyének a megállapítása.
 - c. A robotok konfigurációinak módosítása az új állapotoknak megfelelően.
 - d. Várakozás a nyugtázása.



4.2.3. ábra: Az operátor biztonságáért felelős logika egyszerűsített diagramja

A technológia teljes leállítására szolgáló vészstop az operátor mozgását jelző bit mintavételezése során kerül ellenőrzésre, és amennyiben az aktív, a mintavételezés azonnal megszakad, és a technológia a robotok konfigurációinak módosítási szakaszába lépve teljes leállításra kerül. A mintavételezésbe iktatott késleltetések miatt a vészstop megnyomása legfeljebb 100 milliszekundumon belül váltja ki a kívánt hatást.

A megvalósított irányítás a mellékelt adathordozón az */emulated-system/control* könyvtárban található, az irányítás legfelső szintjének teljes diagramja pedig megtalálható a mellékletek között az M14 sorszám alatt.

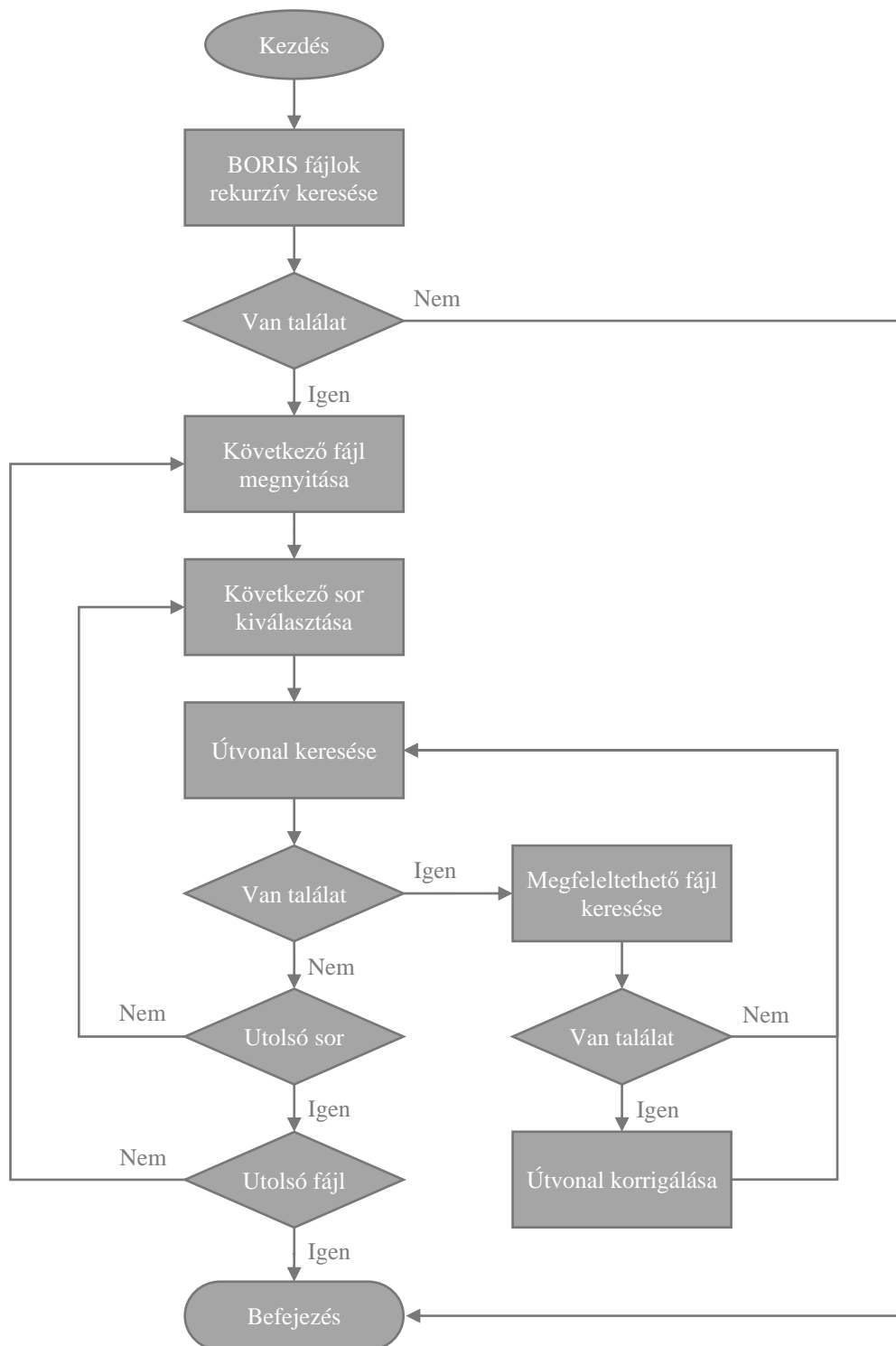
5. A BORIS projektek hordozhatósági problémájának megoldása

A fájlrendszerekben a fájlok vagy a könyvtárak helyét, vagyis azt, hogy milyen könyvtárakon át navigálva találhatóak meg, az úgynevezett elérési útvonal [13] határozza meg, amelynek két változata van. Az abszolút elérési útvonal a gyökérkönyvtárból indulva sorolja fel a fájlig vezető könyvtárak neveit, így a fájlrendszeren belül mindig ugyanazt a helyet jelöli. A relatív elérési útvonal ezzel szemben egy abszolút elérési útvonalhoz viszonyítva sorolja fel a fájlig vezető könyvtárak neveit, így ebben az esetben a fájlrendszeren belül jelölt hely viszonyításfüggő.

A több fájlból álló összefüggő adatok esetén elterjedt és általánosan követendő gyakorlat a relatív elérési útvonalak használata a belső hivatkozásokhoz. Enélkül ugyanis nem lehetne az ilyen adatokat szabadon mozgatni a fájlrendszerben, mivel a belső hivatkozások sérülnének. A feladat megoldásához használt BORIS és FAB verziók azonban nem így működnek, ehelyett minden hivatkozást abszolút elérési útvonalak formájában tárolnak, még akkor is, ha történetesen a hivatkozott és a hivatkozó fájlok egymás mellett helyezkednek el.

A BORIS nem rendelkezik úgynevezett projektfájllal, vagyis egy olyan központi fájllal, amely általában a projekt gyökérkönyvtárában helyezkedik el, tárolja a projekthez tartozó fájlok elérési útvonalait, és minden olyan beállítást, amelyek ezeket a fájlokat valamilyen formában érintik. Ennek fényében viszont olyan fogalom, hogy „BORIS projekt” valójában nem létezik, vagyis tárolás szempontjából a szuperblokkokra bontott összetett szimulációk is csak egymás mellé elhelyezett BORIS fájlok, amelyek az abszolút elérési útvonalakkal tárolt hivatkozásaik miatt még csak nem is hordozhatók. Ezt a problémát igyekszik megoldani a BORIS Corrector segédprogram, amely egy kvázi-projekt gyökérkönyvtárából megfuttatva bejárja az alkönyvtárakat, begyűjti a BORIS specifikus fájlok elérési útvonalait, majd megkísérli az ezekben a fájlokban található hivatkozások útvonalait korrigálni.

A BORIS Corrector a forráskód szintjén gyakorlatilag egyetlen C++ osztály (M15 melléklet), amelyik teljesen automatikusan végzi a feladatát, így elegendő mindössze lepdányosítani és meghívni az `execute()` metódusát ahhoz, hogy megtörténjen a korrigálás, ahogyan azt a 4.2.1. ábra is szemlélteti egy folyamatábra segítségével.



4.2.1. ábra: A BORIS specifikus fájlok korrigálásának leegyszerűsített folyamata

A megvalósításnál a fő szempont az egyszerűség volt, viszont ennek hatására a jelenlegi verzió nem lett felkészítve minden határesetre. Így például, ha a hivatkozások egyike olyan fájlra hivatkozik, amelyik a BORIS projekt gyökérkönyvtárán kívül található, miközben azonos névvel és megfeleltethető relatív elérési útvonallal létezik egy fájl a BORIS projekt gyökérkönyvtárán belül is, akkor az a hivatkozás a korrigálás

hatására elromlik, ahogyan azt a 4.2.1. példa is mutatja. Ahhoz, hogy ez a hiba ne történjen meg egy már egy jóval összetettebb algoritmusra lenne szükség, amely képes következtetni a BORIS projekt gyökérkönyvtárának korábbi elérési útvonalára, és ezzel az információval végezni a hivatkozások korrigálását.

```
// Korrigálás előtt:  
C:\Program Files (x86)\Kahlert\WinFACT 7\Examples\Demo8.sbl  
D:\Studies\BORIS Project\Examples\Demo8.sbl  
  
// Korrigálás után:  
E:\Archived\2021\Studies\BORIS Project\Examples\Demo8.sbl  
E:\Archived\2021\Studies\BORIS Project\Examples\Demo8.sbl
```

4.2.1. példa: Korrigálási hiba azonos végződésű elérési útvonalak esetén

Tulajdonképpen a BORIS Corrector az olyan projektek számára jelent segítséget, ahol verziókezelő rendszerek segítségével teszik lehetővé azt, hogy egyidejűleg többen is dolgozzanak rajta. Ilyen esetben az egyik lehetőség egy megállapodás a projekt gyökérkönyvtárának abszolút elérési útvonalában, a másik lehetőség pedig a BORIS Corrector kibővítése egy olyan funkcionalitással, amellyel a hivatkozásokat közös alakra hozhatná a verziókezelő rendszerbe való feltöltés előtt.

A BORIS Corrector teljes forráskódja megtalálható a mellékelt adathordozó */boris-corrector* könyvtárában, a hozzá tartozó tesztek pedig a */boris-corrector/test* könyvtárban.

6. Az ipari folyamat emulálásának alternatív megoldása

Már a BORIS és a FAB tanulmányozásakor körvonalazódni látszott, hogy ez a két szoftver nem arra a feladatra lett tervezve, mint amire az emulálni kívánt ipari folyamat megvalósítása kényszeríti. Tulajdonképpen már akkor feltételezni lehetett, hogy ha ezeknek a felhasználásával sikerül is maradéktalanul megvalósítani az ipari folyamat emulálását, a végeredmény várhatóan nem lesz túl biztató. Ezeknek a tapasztalatoknak a hatására merült fel olyan alternatív megoldás megvalósításának a gondolata, amely egyszerűbben és hatékonyabban képes megoldani az eredeti feladatkiírás szerint feladatot.

6.1. Az I/O interfész működésének behatásmentes visszafejtése

Mivel maga az I/O interfész oktatási célokra remekül felhasználható, az alternatív megoldás egyik elsődleges kritériuma az volt, hogy kompatibilis legyen I/O interfésszel. Ehhez mindenképpen meg kellett ismerni az eszköz működését, azonban az anyagi károk elkerülése érdekében mindezt úgy kellett megtenni, hogy az eredeti eszközt ne kelljen felnyitni hozzá. Vagyis az I/O interfészről szükségszerűen feketedoboz (black-box) módszerrel kellett a lehető legtöbb információt megszerezni.

6.1.1. BORIS és az I/O interfész közötti kommunikáció lehallgatása

Az I/O interfész felépítéséről a legtöbbet a közte és a BORIS között zajló kommunikáció árult el. Ennek a megismeréséhez azt kellett megfigyelni, hogy a BORIS egy szimuláció elindításakor milyen paraméterekkel nyitja meg azt a soros portot, amelyre az I/O interfész csatlakozik, és ezt követően milyen adatok haladnak át ezen a porton. Mindezeknek a megfigyelése a soros port aktivitásainak a monitorozásával történt, ami láthatóvá tette a soros port megnyitásának minden lépését, a csatlakoztatott I/O interfész azonosítását, illetve az ezt követő teljes kommunikációt. Ezek a monitorozási adatok rögzítve lettek, és továbbra is elérhetők a mellékelt adathordozó */boris-io/investigation/wiretapping* könyvtárában.

A soros port megnyitásával kapcsolatban az derült ki, hogy az I/O interfész olyan soros porton keresztül szólítható meg, amelyik a következő paraméterekkel lett megnyitva:

- baudráta: 9600 baud
- szóhosszúság: 8 bit
- paritás: nincs

- stopbitek száma: 2 (vagy 1)
- adatfolyam-vezérlés: nincs

A monitorozási adatokból az is jól lázik, hogy a BORIS és az I/O interfész között folyamatos adatcsere zajlik, amelyet mindig a BORIS kezdeményez, az I/O interfész pedig csak válaszol rá. Ebben az adatcsereben nem volt nehéz ismétlődő mintázatot találni, ugyanis mint kiderült rendkívül egyszerű adatkapcsolat van a két végpont között, ami az azonosítás utasításán túl mindössze két utasítással valósítja meg az I/O interfész digitális be- és kimeneteinek kezelését, ahogy azt a 6.1.1. táblázat is szemlélteti.

Utasítás	Leírás	Válasz
0x02	Az interfész típusát lekérdező utasítás.	Minden esetben: "S03715-1H.110", azaz 'S', 'O', '3', '7', '1', '5', '-', '1', 'H', '.', '1', '1', '0', '\0'
0xB9	Az interfész 16 digitális bemenetének állapotait lekérdező utasítás.	Két byte, ahol az első byte a 15-8 sorszámu bemenetek állapotait tartalmazza, a második pedig a 7-0 bemenetek állapotait. Például, ha a bemenet állapota 1000000000001111, akkor a válasz: 0x80, 0x0F
0xBA	Az interfész 16 digitális kimenetének állapotait beállító utasítás.	Nincs válasz, a PC a 0xBA utasítás után két byte adatot küld, ahol az első byte a 15-8 sorszámu kimenetek beállítandó állapotait tartalmazza, a második pedig a 7-0 kimenetek beállítandó állapotait.

6.1.1. táblázat: A BORIS utasításai I/O interfész felé

A megfigyelés eredményeiből azt a konklúziót lehetett levonni, hogy az I/O interfész nem egy felprogramozott eszköz, mint egy PLC, amelyik a programja alapján önállóan és valós időben üzemel, hanem gyakorlatilag egy átalakító, amelyik a soros portra küldött állapotokat párhuzamosítja a kimenetei által, illetve bemeneteiről párhuzamosan beolvasott jelszinteket soros adatként küldi vissza. Ez a megoldás azon túl, hogy könnyen másolható, arra is magyarázatot ad, hogy miért változó az I/O interfész késleltetése, ami komoly nehézségeket okozott a választott ipari folyamat emulálásában és irányításában. Mivel a be- és kimenetek állapotai a soros porton utaznak, azoknak a késleltetése a számítógép operációs rendszerétől is függ, és mivel a Windows nem egy valós idejű operációs rendszer, ez a késleltetés sem lesz állandó. Ennek következtében az I/O interfész képtelen az olyan emulációk megfelelő kiszolgálására, amelyekben valamelyik kimenet vagy bemenet változásának a frekvenciájával, esetleg egy logikai szint bizonyos időtartamig való megtartásával is számolni kell.

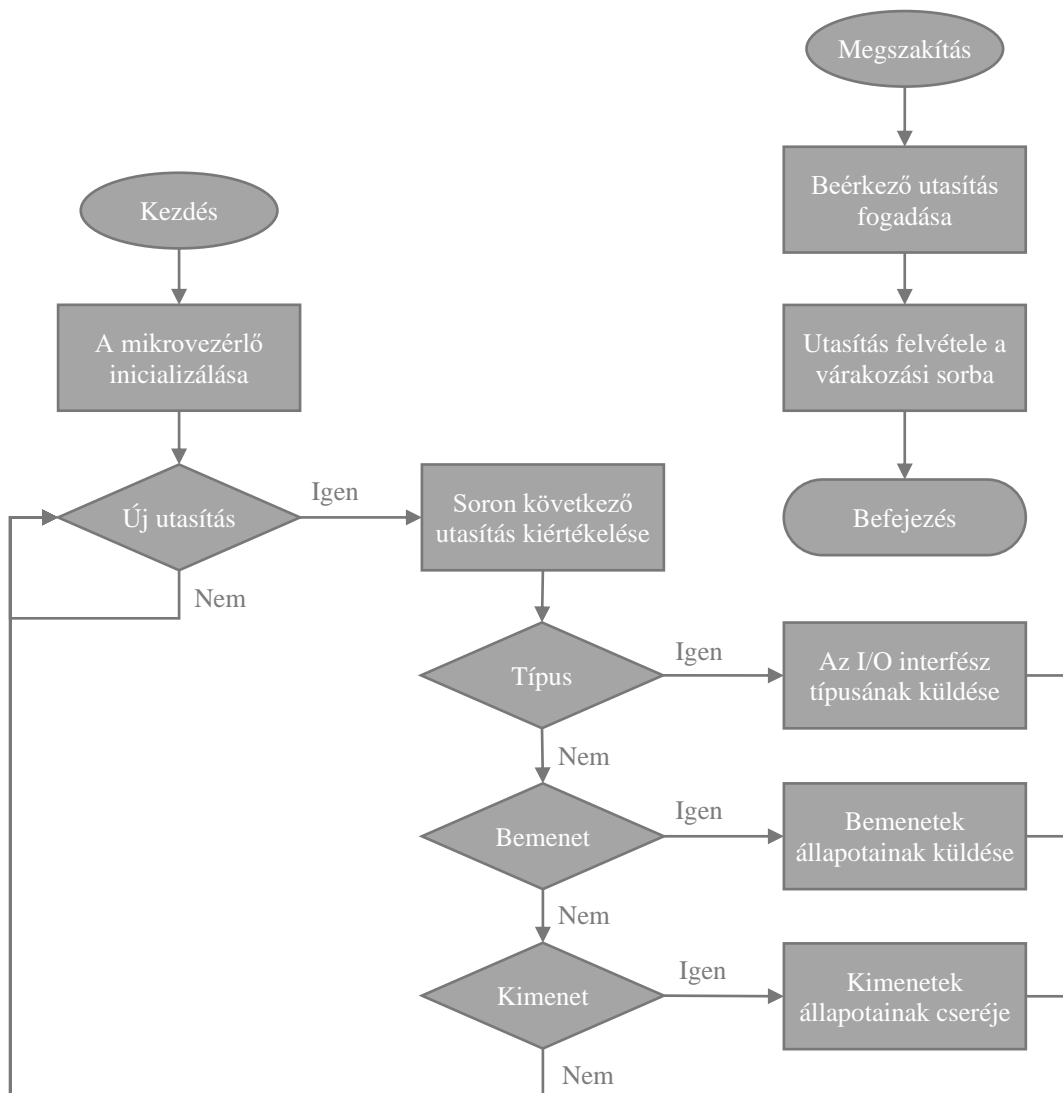
A kommunikáció felderítéséből származó adatok mindkét oldalról igazolva lettek, vagyis a kommunikációt sikerült fenntartani úgy is, hogy az I/O interfész helyettesítve lett egy másik eszközzel, miközben a BORIS úgy kezelte, mint az I/O interfésszel kommunikálna, és úgy is, hogy a BORIS helyett egy másik szoftver kommunikált az I/O interfésszel.

6.1.2. Az I/O interfész működésének utánzása a BORIS számára

Az I/O interfész helyettesítéséhez szükség volt egy olyan eszközre, amelyik képes a soros porton keresztüli kommunikációra és képessé tehető arra, hogy a 6.1.1. táblázat szerinti utasításokra a BORIS által várt válaszokat adja. Habár ez az eszköz akár egy másik számítógép is lehetett volna, a hitelesebb eredmény érdekében erre a feladatra egy PIC18F45K22 mikrovezérlő került felhasználásra, amelyik egy korábban hobbi célból megvásárolt EasyPIC v7 fejlesztőlapon kapott helyet. Ennek a fejlesztőlapnak megvolt az az előnye is, hogy a segítségével a kommunikáció egyaránt tesztelhető volt RS-232 adatátvitellel is és USB-to-UART átalakítóval is. Mivel a cél az volt, hogy a BORIS úgy tekintsen a helyettesítő eszközre, mintha az az I/O interfész lenne, elegendő volt a kommunikációra fókuszálni, ezért a mikrovezérlő be- és kimenetei egy PLC helyett a fejlesztőlapon elhelyezkedő nyomógombokkal és LED-ekkel lettek összekötve.

A megvalósításkor az elsődleges szempont az volt, hogy a mikrovezérlő a lehető leggyorsabban reagáljon a BORIS utasításaira, ezért a megvalósításban a soros portról beérkező adatoknak a beolvasása és értelmezése az ezeket fogadó puffer megszakítására történik meg. Az ilyen módon fogadott utasítások végrehajtása már a mikrovezérlő fő programjában futó végtelen hurokban valósul meg, hogy a megszakítás lekezelése minél rövidebb időt vegyen igénybe. Ahhoz viszont, hogy a mikrovezérlő mindezekre képes legyen, a megfelelő módon be kell konfigurálni, majd pedig inicializálni kell.

A mikrovezérlő programjának (M16 melléklet) valójában egy tekintélyes részét éppen az inicializáció teszi ki, ugyanakkor a központi logika egyáltalán nem bonyolult, ahogyan azt a 6.1.1. ábra is szemlélteti egy folyamatábra segítségével. Ebből a folyamatábrából jól látszik, hogy a megszakítást lekezelő folyamat a beolvasott és értelmezett utasításokat egy várakozási sorba helyezti, a fő folyamat viszont ciklikusan ellenőrzi ezt a várakozási sort és a benne levő utasításokat a beérkezési sorrendben végre is hajtja.



6.1.1. ábra: A I/O interfész utánzásának leegyszerűsített folyamata

Minden beérkező utasítás egy `RequestEntry` típusú adatként (6.1.1. példa) kerül bele a várakozási sorba, ami tartalmazza magát az utasítást (`type`), illetve a hozzá tartozó esetleges argumentumokat (`dataHigh` és `dataLow`) is.

```

typedef struct request_entry_t {
    enum RequestType type;
    uint8_t dataHigh;
    uint8_t dataLow;
} RequestEntry;
  
```

6.1.1. példa: A BORIS utasításainak struktúrája a mikrovezérlő programjában

Ezeket az adatokat a fogadó puffer megszakítását lekezelő függvény hozza (6.1.2. példa) létre a beérkező adatbyte-okból. Mivel ez a függvény egyenként hívódik meg minden adatbyte beérkezésekor, ezért az egy byte-nál hosszabb adatok beolvasása és értelmezése valamivel összetettebb, mint az egy byte hosszúságúaké.

```

void Serial_handleInterrupt() {
    uint8_t next_byte = UART1_Read();

    switch (next_byte) {
    case 0x02:
        BorisProdigy_setRequest(REQUEST_GET_DEVICE_TYPE, 0, 0);
        break;
    case 0xb9:
        BorisProdigy_setRequest(REQUEST_GET_INPUT, 0, 0);
        break;
    case 0xba:
        receiveCounter = 2;
        break;
    }

    if (receiveCounter && next_byte != 0xba) {
        receiveBuffer[2 - receiveCounter] = next_byte;
        if (--receiveCounter == 0)
            BorisProdigy_setRequest(REQUEST_SET_OUTPUT,
                                    receiveBuffer[0], receiveBuffer[1]);
    }

    RC1IF_bit = false;
}

```

6.1.2. példa: A BORIS által küldött nyers adatbyte-ok értelmezése

Annak köszönhetően, hogy a fogadott utasítások a végrehajtás helyén már csak `RequestEntry` típusú adatokként vannak jelen, nagy mértékben leegyszerűsíti a végrehajtásukat, így ugyanis a teljes végrehajtási folyamat egyetlen elágazásra és a várakozási sor mutatójának a léptetésére korlátozható (6.1.3. példa).

```

if (requestQueue.begin == requestQueue.end) return;

switch (requestQueue.requests[requestQueue.begin].type) {
case REQUEST_GET_DEVICE_TYPE:
    Serial_sendData(serialNumber, 14);
    break;
case REQUEST_GET_INPUT:
    port_data[0] = PORTA;
    port_data[1] = 0 | (PORTC & 0x0F) | ((PORTE & 0x0F) << 4);
    Serial_sendData(port_data, 2);
    break;
case REQUEST_SET_OUTPUT:
    LATB = requestQueue.requests[requestQueue.begin].dataHigh;
    LATD = requestQueue.requests[requestQueue.begin].dataLow;
    break;
}

requestQueue.begin = (requestQueue.begin + 1) %
    REQUEST_QUEUE_SIZE;

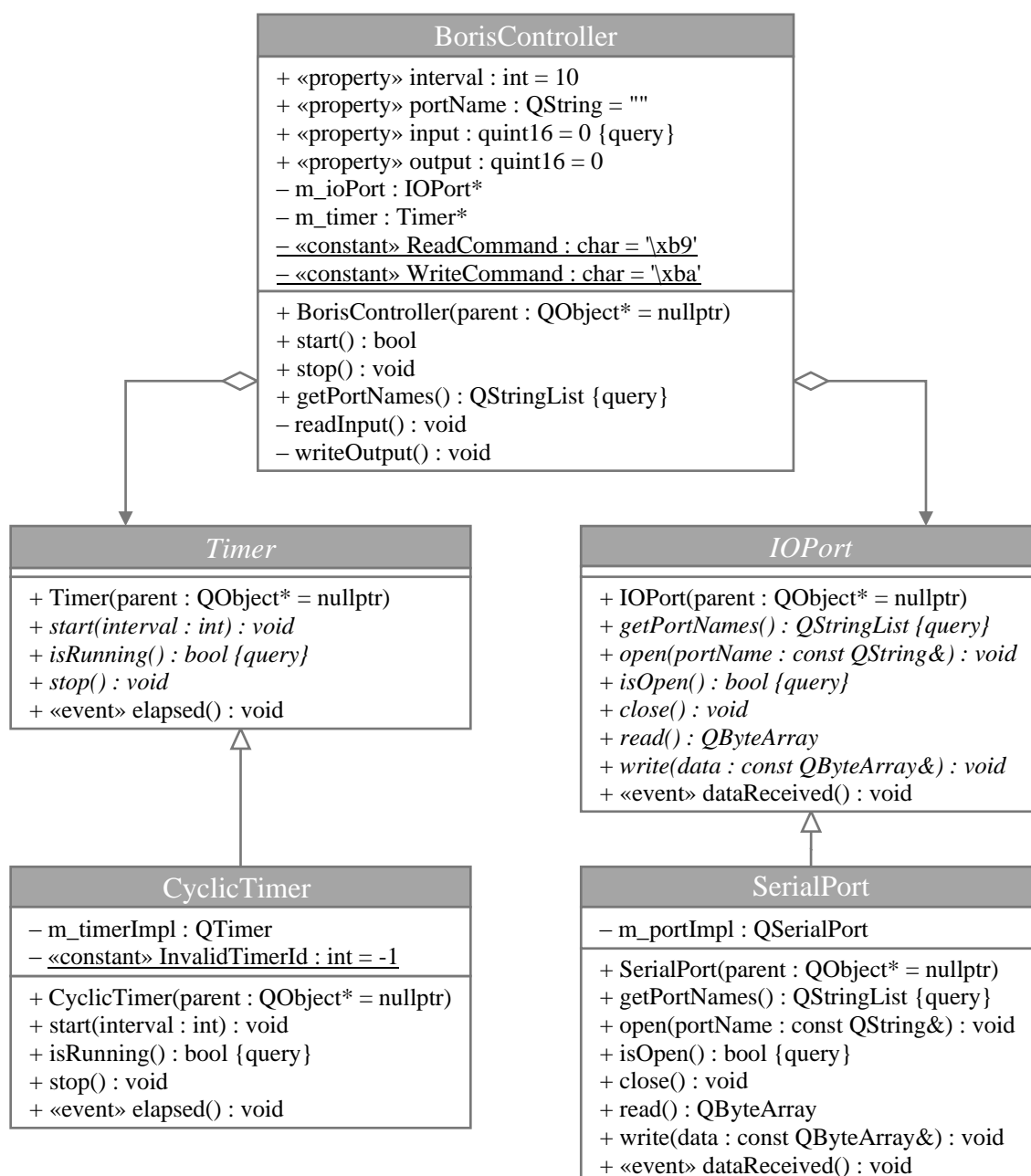
```

6.1.3. példa: A BORIS által küldött utasítások végrehajtása

Az elkészült mikrovezérlő program megtalálható a mellékelt adathordozó */boris-io/investigation/prototype* könyvtárában, a tesztelésről készült videófelvétel pedig a */multimedia/20160426_145453.mp4* lejátszásával tekinthető meg.

6.1.3. Az I/O interfész működtetése a BORIS használata nélkül

A BORIS utasításainak és a soros port megnyitásához szükséges paraméterek feltérképezésével, minden információ adott volt egy olyan függvénykönyvtár kifejlesztéséhez, amelynek segítségével bármikor fejleszthető olyan szoftver, amelyik képes az I/O interfész működtetésére a BORIS megléte nélkül.



6.1.2. ábra: Az I/O interfészt működtető függvénykönyvtár egyszerűsített osztálydiagramja

A megvalósított függvénykönyvtár a Qt alkalmazás-keretrendszerrel valósult meg a Qt Quick alapú alkalmazásokhoz fejlesztésének igényeihez alakítva. A legfőbb osztálya a `BorisController` (6.1.2. ábra), ami gyakorlatilag az I/O interfész szoftveres reprezentációja. Ennek az osztálynak a segítségével lehet felépíteni vagy bontani a folyamatos kommunikációt az eszközzel, illetve az `output` és az `input` tulajdonságai segítségével lehet kezelni a be- és kimeneteit.

A `BorisController` a kapcsolat létesítéséhez egy `IOPort` implementációt, a ciklikus adatcseréhez pedig egy `Timer` implementációt foglal magába. Alapértelmezés szerint ezek az implementációk a `CyclicTimer`, ami ténylegesen megvalósítja a ciklikus időzítő logikát, és a `SerialPort`, ami pedig a soros porton keresztüli kapcsolódást valósítja meg. Ezek az implementációk viszont bármikor lecserélhetők anélkül, hogy ez bármilyen különbséget is jelentene a szoftver többi komponense számára. Így például a soros porton keresztüli kapcsolódás bármikor helyettesíthető USB, Ethernet vagy bármilyen kapcsolódást lehetővé tevő implementációval, az időzítő pedig egy tesztimplementációval, ami csak külső utasításra jelez. A függvénykönyvtár teljes osztálydiagramja (M17 melléklet) jól szemlélteti, hogy a fejlesztés és a tesztelés támogatásának érdekében több eltérő `IOPort` implementáció is megvalósult, amelyek lehetővé teszik a soros port előre programozott utánzását.

Az I/O interfésszel való folyamatos kommunikáció megkezdéséhez elegendő az alapértelmezett konstruktorával létrehozni egy `BorisController` példányt, majd a `portName` tulajdonságának beállítani az I/O interfészhez csatlakozó soros port nevét, végül pedig meghívni a `start()` metódusát. Ennek hatására az időzítője elkezd 10 milliszekundumonként kiváltani az `elapsed` eseményt, amitől a `writeOutput()` metódusa kiírja a portra az `output` tulajdonságának az értékét (6.1.4. példa).

```
void BorisController::writeOutput() {
    const quint16 newOutput = output();

    QByteArray bytesToWrite;
    bytesToWrite += WriteCommand;
    bytesToWrite += static_cast<char>((newOutput >> 8) & 0xff);
    bytesToWrite += static_cast<char>(newOutput & 0xff);
    bytesToWrite += ReadCommand;

    m_ioPort->write(bytesToWrite);
}
```

6.1.4. példa: A kimenet új értékeinek elküldése az I/O interfésznek

Amint az I/O interfész válaszol, a portja kivált egy `dataReceived` eseményt, amitől a `readInput()` módszere beolvassa a beérkező választ és beállítja azt az `input` tulajdonságának értékeként (6.1.5. példa).

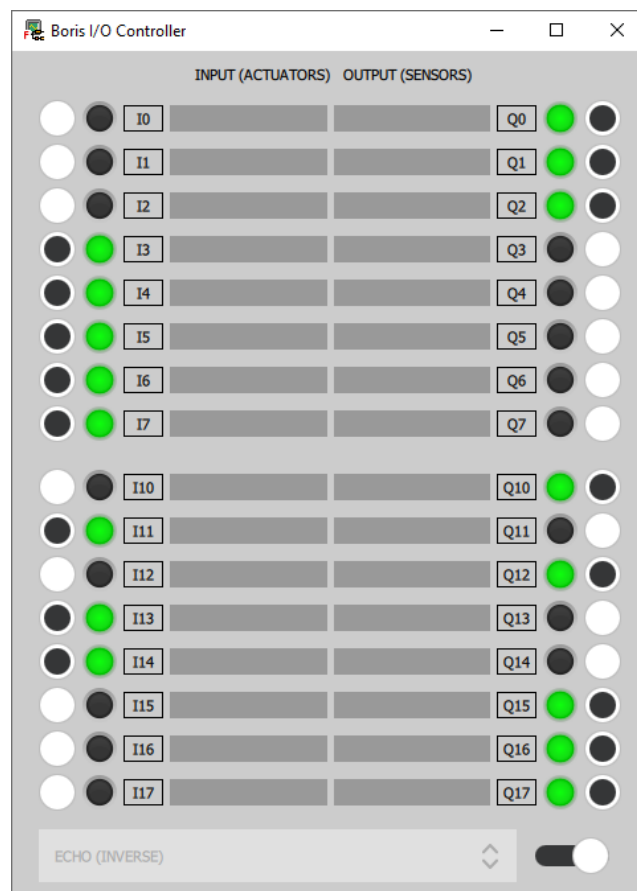
```
void BorisController::readInput() {
    const QByteArray readBytes = m_ioPort->read();

    if (readBytes.length() == 2) {
        quint16 newInput = 0;
        newInput |= static_cast<unsigned char>(readBytes[0]) << 8;
        newInput |= static_cast<unsigned char>(readBytes[1]);

        input(newInput);
    }
}
```

6.1.5. példa: A bemenet új értékeinek fogadása az I/O interfésztől

Az I/O interfészt működtető függvénykönyvtár köré egy egyszerű grafikus felhasználói felület (6.1.3. ábra) is készült, ami az I/O interfészt fedőlapjának a megjelenését igyekszik idézni. Ez a felület elsősorban a függvénykönyvtár kézi tesztelését hivatott megkönnyíteni, de azt is jól példázza, hogy hogyan építhető be az I/O interfész működtetésének a képessége más alkalmazásokba.



6.1.3. ábra: Az I/O interfészt működtető helyettesítőszoftver

A függvénykönyvtárnak, a hozzá tartozó teszteknek, illetve a hozzá fejlesztett grafikus felhasználói felületnek a közös projektje megtalálható a mellékelt adathordozón a */fake-boris* könyvtárban. Ennek a közös projektnek a lefordítása esetén mindhárom komponens kimenete elkészül.

6.2. Az emuláció megvalósítása alternatív eszközökkel

A BORIS és az I/O interfész közötti kommunikáció lehallgatása és az ebből származó adatok helyességének sikeres ellenőrzése egészen új lehetőségeket teremtett az BORIS segítségével már emulált ipari folyamat elegánsabb megvalósítására.

Az elegánsabb alternatíva megvalósítását elsősorban az motiválta, hogy a BORIS segítségével megvalósított emuláció nagyságrendekkel összetettebb lett, mint maga az irányítás, aminek a fejlesztését és a tesztelését elvileg segíteni hivatott. Márpedig, ha egy teszt túlságosan is összetett, az ösztönösen is kételyeket ébreszt a hitelességét illetően. Éppen ezért az alternatív megvalósítás elsődleges célja az volt, hogy a jelenleginél jóval egyszerűbb és karbantarthatóbb módon valósítsa meg az ipari folyamat irányítását.

A felhasználói felületek fejlesztése hosszú múltra tekint vissza. Azóta számos segédeszköz látott már napvilágot, amelyek egyszerűsíteni igyekeztek ezen a feladaton. Megfigyelve ezeknek a segédeszközöknek a fejlődését észrevehető, hogy az többnyire valamilyen leírónyelv segítségével igyekeznek csökkenteni azon a kódtömegben, ami a vizuális elemek megjelenítésével jár. A Qt User Interface Creation Kit (Qt Quick) egyike ezeknek a segédeszközöknek, amelyik elsősorban az okostelefonokon, táblagépeken stb. futó alkalmazások grafikus felhasználói felületeinek a megvalósításához lett kifejlesztve, illetve a keresztplatformos alkalmazások felhasználói felületeinek az egységesítésére. Ebben a komponensben a különféle vizuális hatások, animációk és átmenetek központi szerepet töltenek be, amitől akár egyszerűbb játékok grafikus felületének fejlesztésére is alkalmas. Külön érdekessége, hogy a gyakorlatban leginkább elterjedt XML helyett egy egyedi leírónyelvet biztosít a felhasználói felületek szerkezetének definiálására, ami első ránézésre leginkább a CSS és a JavaScript ötvözetének tűnik. Ez a leírónyelv a Qt Modeling Language (QML), amiben a UI elemek megjelenítését a CSS leírásokhoz hasonlóan tulajdonság-érték párok definiálják, de ezeknek a tulajdonságoknak az értékei JavaScript kifejezésekkel is megadhatók, jelentős dinamikát biztosítva ezzel a grafikus felület számára.

A karosszéria gyártósor emulációjának alternatív megvalósítása a mellékelt adathordozó */emulated-system/remake* könyvtárában található. A vizualizációról készült képernyőfotó pedig az M18 mellékletben látható.

6.2.1. Az alternatív vizualizáció megvalósításának alapjai

Az I/O interfészt működtető függvénykönyvtár megvalósítása az első lépés volt afelé, hogy a BORIS használata nélkülözhetővé váljon az I/O interfész használatához. Fontos azonban kiemelni, hogy a BORIS számos előre elkészített blokkot, és egy gazdagon felszerelt szerkesztőfelületet is biztosít, amivel szemben a függvénykönyvtár csak egy komponens, ami beépíthető egy új alkalmazásba. Ami viszont a függvénykönyvtár mellett szól az az, hogy a Qt hatékonyabb vizualizációfejlesztést tesz lehetővé, mint a BORIS, emellett a Qt-alapú implementációk több platformra is lefordíthatóak és önállóan futtathatóak.

Az alternatív vizualizáció megvalósításának első állomása a fejlesztéshez létrehozott projekt megfelelő bekonfigurálása volt. Mivel a függvénykönyvtár statikusan linkelt, azaz a benne található osztályok, függvények, konstansok stb. már a fordítás során beleépülnek a célalkalmazásba, a dinamikus betöltésére és felszabadítására nem kellett figyelmet fordítani, és elegendő volt a függvénykönyvtárnak a pontos helyét és a nevét meghatározni a fordító számára (6.2.1. példa).

```
CONFIG += c++17
QT += quick serialport

SOURCES += main.cpp
RESOURCES += qml.qrc
LIBS += -L$PWD/libs/BorisAccess/ -lBorisAccess
```

6.2.1. példa: A projektfájl konfigurációjának egyszerűsített változata

Miután a fordító tudtára lett adva, hogy linkelésnél a szimbólumokat az megadott függvénykönyvtárban is keresse, a `BorisController` a hozzá tartozó többi osztállyal együtt a részévé váltak a projekt C++ oldali részének, de ez önmagában még kevés volt ahhoz, hogy a QML oldal számára is elérhetőek legyenek. A Qt Quick alkalmazások belépési pontja ugyanis szintén a C++ oldali `main` függvény, azonban erről az oldalról automatikusan semmit sem visznek magukkal. Ahhoz, hogy a `BorisController` használhatóvá váljon a QML kódban, először a C++ oldalon le kellett példányosítani, majd a grafikus felhasználói felület motorjának segítségével környezeti referenciaként hozzá kellett adni azt a QML oldalhoz (6.2.2. példa).

```

#include <BorisController.h>
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <QQmlContext>

using namespace Diplomamunka;

int main(int argc, char* argv[]) {
    QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
    QGuiApplication application(argc, argv);
    QQmlApplicationEngine engine;

    BorisController boris;
    engine.rootContext()->setContextProperty("boris", &boris);
    engine.load(QUrl("qrc:/main.qml"));

    return application.exec();
}

```

6.2.2. példa: A BORIS Controller hozzáadása a QML alkalmazáshoz

Onnantól, hogy a `BorisController` példánya elérhetővé vált a QML oldalon is, a C++ oldali kóddal nem maradt már több tennivaló, és el lehetett kezdeni a vizualizáció teljes körű implementálását.

A könnyebb fejlesztés és a kézi tesztelés biztosítása érdekében a C++ oldal később úgy lett kibővíve, hogy Debug konfiguráció esetén a `BorisController` egy olyan `IOPort` implementációval legyen példányosítva, amelynek a visszaadott értékeit tetszés szerint lehet módosítani, majd az így létrehozott példány legyen átadva a QML oldalnak. A kibővített `main` függvény és a QML oldalnak átadott példány integrálásának forráskódja megtekinthetők az M19 mellékletben.

6.2.2. A karosszériák mozgatása

Az alternatív emulációban a karosszériák leginkább egy videójáték irányított karakteréhez hasonlíthatók. Egyfelől rendelkeznek azzal a képességgel, ami lehetővé teszi a számukra a képernyőn történő előre haladást, másfelől képesek helyváltoztatást elszenvetni a környezet más objektumaitól.

Az előre haladásra akkor van szükség, amikor a karosszériák a futószalagokon utaznak, hiszen ebben az esetben a futószalag elemek nem mozdulnak csak lejátsszák az aktivitásukat ábrázoló mozgóképeket, ezért a karosszériáknak saját maguknak kell gondoskodniuk a helyváltoztatásaikról. Ezt a feladatot egy `NumberAnimation` elem látja el, ami a karosszériák `x` tulajdonságainak a változását animálja, ahogy azt a 6.2.3. példa is mutatja.

```

function moveTo(destination) {
    var duration = Math.abs(destination - x) * 10

    animation.stop()
    animation.to = destination
    animation.duration = duration
    animation.start()
}

function stop() {
    animation.stop()
}

NumberAnimation on x {
    id: animation

    onFinish: moveFinished(carBody)
}

```

6.2.3. példa: A karosszériák mozgásának megvalósítása

Ezzel szemben, amikor a karosszériák leválogatása történik, a daru megfogója ténylegesen képes megfogni a karosszériát és magával vonszolni. Ezt a képességet a Qt Quick azon alapelve teszi lehetővé, ami szerint egy elem mindig a szülő elemével együtt mozog. Mivel minden elemnek saját koordinátarendszere van ezért, amellet, hogy a karosszériáknak dinamikusan kell változtatgatniuk a `parent` tulajdonságaikat, minden alkalommal el kell végezniük a koordinátáik transzformálását is, ahogy azt a 6.2.4. példa is mutatja.

```

function attachTo(item) {
    var mappedPosition = parent.mapToItem(item, x, y)

    x = mappedPosition.x
    y = mappedPosition.y
    parent = item
}

function attachedTo(item) {
    return parent === item
}

```

6.2.4. példa: A karosszériák csatolásának megvalósítása

Mivel a karosszériák definiálják azokat a műveleteket, amelyek a mozgásukat lehetővé teszik, szükségszerűen nekik kell feliratkozniuk a cellákba való behelyezés eseményeire és a daru megfogójának eseményeire.

6.2.3. A futószalagok megvalósítása és működtetése

A futószalagok az alternatív emulációban úgy lettek megvalósítva, hogy képesek legyenek a karosszériák önálló mozgására. Ennek köszönhetően a karosszériák mozgására nem kell külön figyelmet szentelni, elegendő ráhelyezni azokat valamelyik futószalagra és elindítani a futószalagot.

Annak érdekében, hogy a karosszériák képesek legyen átgördülni egyik futószalagról egy másikra, minden futószalag referenciát tart fenn a soron következő futószalagra. Így, amikor egy karosszéria eléri egy futószalag végét, a futószalag egyszerűen csak továbbadja azt a soron következő futószalagnak (6.2.5. példa).

```
function onMoveFinished(body) {  
  body.detectable = false  
  body.attachTo(nextCell)  
  body.detectable = true  
}  
  
onChildrenChanged: {  
  carBodies.forEach(body =>  
    body.moveFinished.disconnect(onMoveFinished))  
  
  carBodies = Array.from(children).filter(item =>  
    item instanceof CarBody)  
  carBodies.forEach(body =>  
    body.moveFinished.connect(onMoveFinished))  
  
  activeChanged()  
}  
  
onActiveChanged: {  
  if (active) {  
    carBodies.forEach(body =>  
      body.moveTo(width - body.width / 2))  
  } else {  
    carBodies.forEach(body =>  
      body.stop())  
  }  
}
```

6.2.5. példa: Karosszériák mozgásának és átadásának kezelése a futószalagokon

A futószalagok nincsenek egyetlen elemre korlátozva, vagyis akárhány karosszériát képesek egyszerre mozgatni. Az egyidőben ugyanazon a futószalagon levő karosszériákat a futószalagok a `carBodies` tömbjeikben tartják számon. Ezeknek a tömböknek az elemei a futószalagok gyermek elemeinek a változásaikor frissülnek, vagyis amikor egy karosszéria egyik futószalagról átgördül egy másikra, akkor mindkét futószalag `carBodies` tömbje automatikusan frissül. A futószalagok és a karosszériák

közötti szülő-gyermek viszonyának megvan továbbá az az előnye is, hogy a futószalagok a saját koordináta-rendszerükben mozgathatják a karosszériákat. Ezt azt jelenti, hogy egy futószalag bárhol is helyezkedjen el a vizualizációban, mindig ugyanarra a koordinátára kell küldenie az általa mozgatott karosszériákat.

A Qt Quickben az elemek tulajdonságainak a változásai többnyire összefűzhetők egymással, ezáltal az egyik tulajdonság változása automatikusan kiváltja a hozzá fűződő tulajdonságok változását és így tovább. Erre a funkcionalitásara épülnek a futószalagok által megvalósított érzékelők is (6.2.6. példa), amelyek a `carBodies` tömb változásaira a benne levő összes karosszériát letesztelik átfedés után kutatva. Amennyiben egy érzékelő és egy karosszéria között átfedés van, akkor az adott érzékelő érzékeli az érintett karosszériát.

```
Sensor {
    id: entrySensor

    activeColor: "yellow"
    active: carBodies.some(body => entrySensor.detects(body))
}

Sensor {
    id: arrivalSensor

    activeColor: "green"
    active: carBodies.some(body => arrivalSensor.detects(body))
}
```

6.2.6. példa: Karosszériák érzékelése a futószalagokon

A futószalagok részét képezik a `ConveyorBelt` elemnek, amelyik lehetővé teszi a sorbarendezt, egymással összekötött és megfelelő animációval ellátott futószalagok létrehozását. Ugyanez az elem gondoskodik arról is, hogy a karosszériák pontosan arra a futószalagra legyenek ráhelyezve, amelyik felett vannak, illetve ez tartja számon az utoljára behelyezett karosszéria típusát is.

6.2.4. A daru megvalósítása és működtetése

A legtöbb Qt Quick elem rendelkezik egy saját beépített állapotgéppel, amelyik két részből tevődik össze. Az állapotok (`states`) az elem különböző állapotainak a definiálására szolgál. Egy állapot járhat tulajdonságváltozással, szülőelem-változással stb., amelyek akkor mennek végbe, amikor az elem az adott állapotba kerül. Az átmenetek (`transitions`) az állapotok közötti átmeneteket definiálják, azaz képesek különféle animációkat és más akciókat társítani az állapotváltozásokhoz.

Az alternatív emulációban a daru megvalósítása állapotokkal definiálja az egyes darupozíciókat, amelyek között az átmenetek valósítják meg a feladatlapon definiált időtartamoknak megfelelő hosszúságú animációkat (6.2.7. példa).

```
states: [
  BridgeState {
    name: "0"
    bridge: bridge
    distance: destinationHome
  },
  BridgeState {
    name: "1"
    bridge: bridge
    distance: destinationTwo
  },
  BridgeState {
    name: "2"
    bridge: bridge
    distance: destinationOne
  },
  BridgeState {
    name: "3"
    bridge: bridge
    distance: destinationThree
  }
]

transitions: [
  BridgeTransition {
    duration: 6000
  },
  BridgeTransition {
    from: "0"
    to: "2"
    duration: 12000
  },
  BridgeTransition {
    from: "1"
    to: "3"
    duration: 12000
  },
  BridgeTransition {
    from: "2"
    to: "3"
    duration: 18000
  }
]
```

6.2.7. példa: A daru állapotai és átmenetei

A daru több együttesen mozgó elemből áll, amelyek esetén a rétegeződés, azaz egymás eltakarásának a sorrendje, meggátolta a szülő-gyermek viszony használatát a mozgások összekapcsolására. Ehelyett az elemek jellegzetes élei (jobb oldal, bal oldal,

vízszintes közép, függőleges közép stb.) lettek egymáshoz rögzítve, ami ugyanúgy képes az együttes mozgás biztosítására.

Ahhoz, hogy a daru fel tudjon emelni egy karosszériát, a megfogójának pontosan a karosszéria felett kell elhelyezkednie. A megfogás pillanatában a karosszéria a megfogó gyerek elemévé válik, ezért a megfogóval együtt fog mozogni. A lerakáshoz mindig szüksége van egy másik elemre, amelyik az új szülő eleme lesz a karosszériának. Ez alapértelmezésben maga az ablak, amiben az animáció fut.

6.2.5. A robotok megvalósítása és működtetése

A robotok teljesen független szereplői az alternatív emulációnak, ugyanis egyáltalán nem lépnek kölcsönhatásba más objektumokkal, egyszerűen csak vizualizálják a vezérlőtől kapott utasításokat, és válaszjeleket állítanak elő az irányításhoz. A működésük nagyon hasonló a daruéhoz, csak valamivel bonyolultabb annál, aminek az az oka, hogy míg a daru csak egy sebességfokozattal rendelkezik és mindig befejezheti az aktuális mozgást, addig a robotok mozgás közben gyorsíthatók, lassíthatók vagy teljesen meg is állíthatók. Sőt, a megállítást követő engedélyezés során a mozgást ott kell folytatniuk, ahol abbahagyták. Ettől függetlenül a robotok esetén is az egyes pozíciók állapotokhoz vannak rendelve, így az összetettebb viselkedés megvalósításának a terhe az átmeneteket érinti.

Sajnos a Qt Quick nem kimondottan támogatja a futó animációk sebességének a megváltoztatását. Persze az animációk bármikor megállíthatók és új paraméterekkel újból elindíthatók, de a legtöbb animációnál az időtartamot kell meghatározni, és ez implicit módon hat a sebességre. Ezt az időtartamot a kívánt sebességből és a hátralevő út hosszából nem túl bonyolult kiszámítani, viszont ezekhez az adatokhoz nem mindig egyszerű hozzájutni és nemkívánatos függőségeket eredményezhet a forráskódban. Egyedül egy olyan animáció van a Qt Quickben, ami a sebesség meghatározását igényli, a `SmoothedAnimation`, de ez az animáció egészen váratlanul viselkedik, ha futás közben megállítják, ugyanis a tulajdonságok módosítását ekkor is véghez viszi, viszont ekkor már animáció nélkül és azonnal. Ennek a problémának a kerülőmegoldása végül egy speciális átmenet lett (6.2.8. példa), ami folyamatosan rögzíti a robot animálható tulajdonságainak pillanatnyi értékeit, majd visszaírja a `"pause"` állapotba lépés előtt utoljára rögzített értékeket a robot megfelelő tulajdonságaiba.

```

Transition {
    property var robotArm

    readonly property real distance:
        robotArm.paused ? distance : robotArm.distance
    readonly property real lowering:
        robotArm.paused ? lowering : robotArm.lowering
    readonly property real rotation:
        robotArm.paused ? rotation : robotArm.rotation

    to: "pause"

    PropertyAction {
        target: robotArm
        property: "distance"
        value: distance
    }

    PropertyAction {
        target: robotArm
        property: "lowering"
        value: lowering
    }

    PropertyAction {
        target: robotArm
        property: "rotation"
        value: rotation
    }
}

```

6.2.8. példa: Speciális átmenet a robotok mozgásának a felfüggesztésére

A robotok multiplexelését a `RobotController` végzi, ami mindössze annyit tesz, hogy átadja a tulajdonságainak az értékeit azoknak a robotoknak, amelyek a `robotNumber` tulajdonságának értékével azonos `number` értékkel rendelkeznek, amennyiben az `enabled` tulajdonsága legalább 50 milliszekundumig `true` értéken volt tartva.

7. Az I/O interfészt helyettesítő elektronika

A 6.1 alfejezetben bemutatott vizsgálatok eredményeként létrejött egy prototípus, amely alkalmas az I/O interfész digitális be- és kimeneteinek a működését utánozni azzal a különbséggel, hogy ez a prototípus nem rendelkezik olyan csatlakozókkal, amiken keresztül összeköthető lenne egy PLC-vel vagy más ipari vezérlőegységgel. Annak érdekében, hogy az I/O interfész működésének visszafejtéséből végül egy kompakt és sokszorozható oktatóeszköz legyen, meg kellett tervezni a prototípushoz a megfelelő a be- és kimeneti áramköröket is, illetve azt a végleges beágyazott rendszert, ami ezeket végül használni is fogja.

Mivel az I/O interfész vizsgálatának behatásmentessége elsődleges szempont volt, az általa megvalósított be- és kimeneti áramköröket nem lehetett megfelelően szemügyre venni ahhoz, hogy le lehessen másolni. Ugyanígy az is rejtély maradt, hogy az eredeti eszköz milyen mikrovezérlővel működik vagy milyen tápegységgel rendelkezik. Mindezek hiányában egy teljesen egyedi elektronikát kellett kidolgozni, ami csak annyiban köthető az eredeti I/O interfészhez, hogy ugyanazokkal az utasításokkal működtethető. Az elkészült eszköz (6.2.1. ábra) műszaki jellemzői a következők:

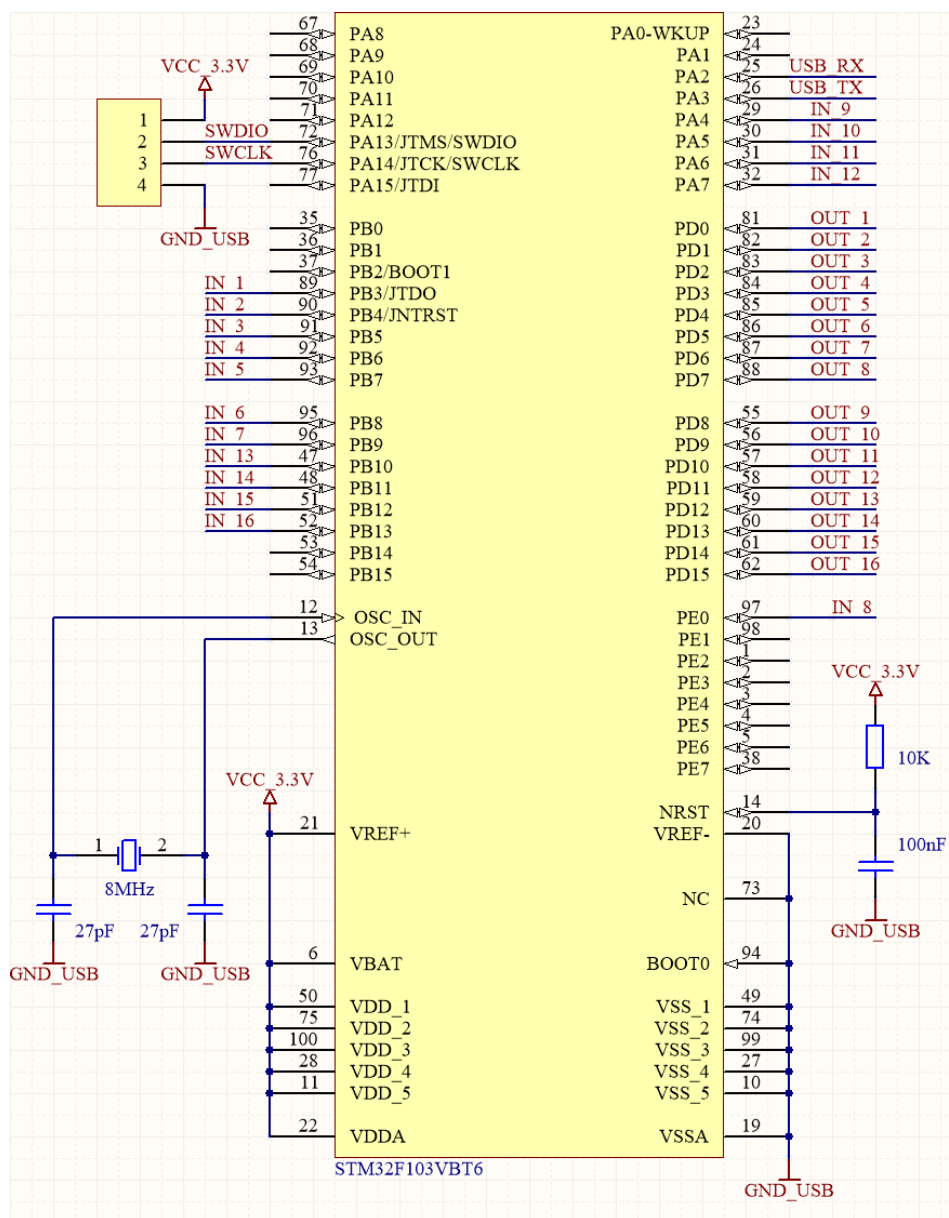
- 16 digitális bemenet: +24 V DC / 10 mA
- 16 digitális kimenet: +24 V DC / 600 mA
- tápellátás: +24 V DC / 1 A
- kommunikáció: USB to UART
- méretek: 105 mm × 90 mm × 70 mm (magasság × szélesség × mélység)
- súly: 225 g



6.2.1. ábra: Az I/O interfészt helyettesítő elektronika

7.1. Az elektronika programozott működése

A helyettesítő elektronika a 6.1.2 alfejezetben bemutatott prototípushoz képest már egy STM32F103VBT6 mikrovezérlővel lett felszerelve (7.1.1. ábra). Erre a váltásra egyrészt azért került sor, mert a prototípusnál még nem merült fel a használandó mikrovezérlő típusa, mivel akkor csak a kommunikáció felderítéséből származó adatok igazolására volt a cél és a feladat elvégzésére egy már meglévő eszköz lett felszánálva.



7.1.1. ábra: A helyettesítő elektronika mikrovezérlőjének egyszerűsített kapcsolási rajza

Másrészt az STM32 mikrovezérlőkkel nagyobb sebesség érhető el, mint a prototípus által használt PIC mikrovezérlővel, amivel tovább csökkenthetőek a számítógép által képviselt késleltetésen túli késleltetések.

Ennek a mikrovezérlőnek a fő programja szintén ábrázolható a 6.1.1. ábra szerinti folyamatábrával, de a megvalósításba bekerült néhány módosítás. Egyfelől a beérkező utasítások nem kerülnek külön értelmezésre, ugyanis a prototípus esetén az csak egy plusz elágazást jelentett, és mivel a három utasítás közül csak az egyik rendelkezik argumentumokkal, a várakozási sor is feleslegesen foglalta a memóriát. Ehelyett az új megvalósításban csak egy byte tömb van, amibe az UART fogadó pufferének megszakításakor belekerülnek az érkező adatbyte-ok, az értelmezést maga a fő program végzi (7.1.1. példa) egy végtelen hurokban.

```
if (datanumber != 0) {
    switch (receiveddata[0]) {
        case 0x02:
            UART_StrKuld("SO3715-1H.110");
            datanumber = 0;
            break;
        case 0xb9:
            portin = read_port_input();
            UART_AdatKuld(portin >> 8);
            UART_AdatKuld(portin);
            datanumber = 0;
            break;
        case 0xba:
            if (datanumber == 3) {
                portout = receiveddata[1];
                portout = (portout << 8) + receiveddata[2];
                GPIO_Write(GPIOD, (0xFFFF - portout));
                datanumber = 0;
            }
            break;
        default:
            datanumber = 0;
            break;
    }
}
```

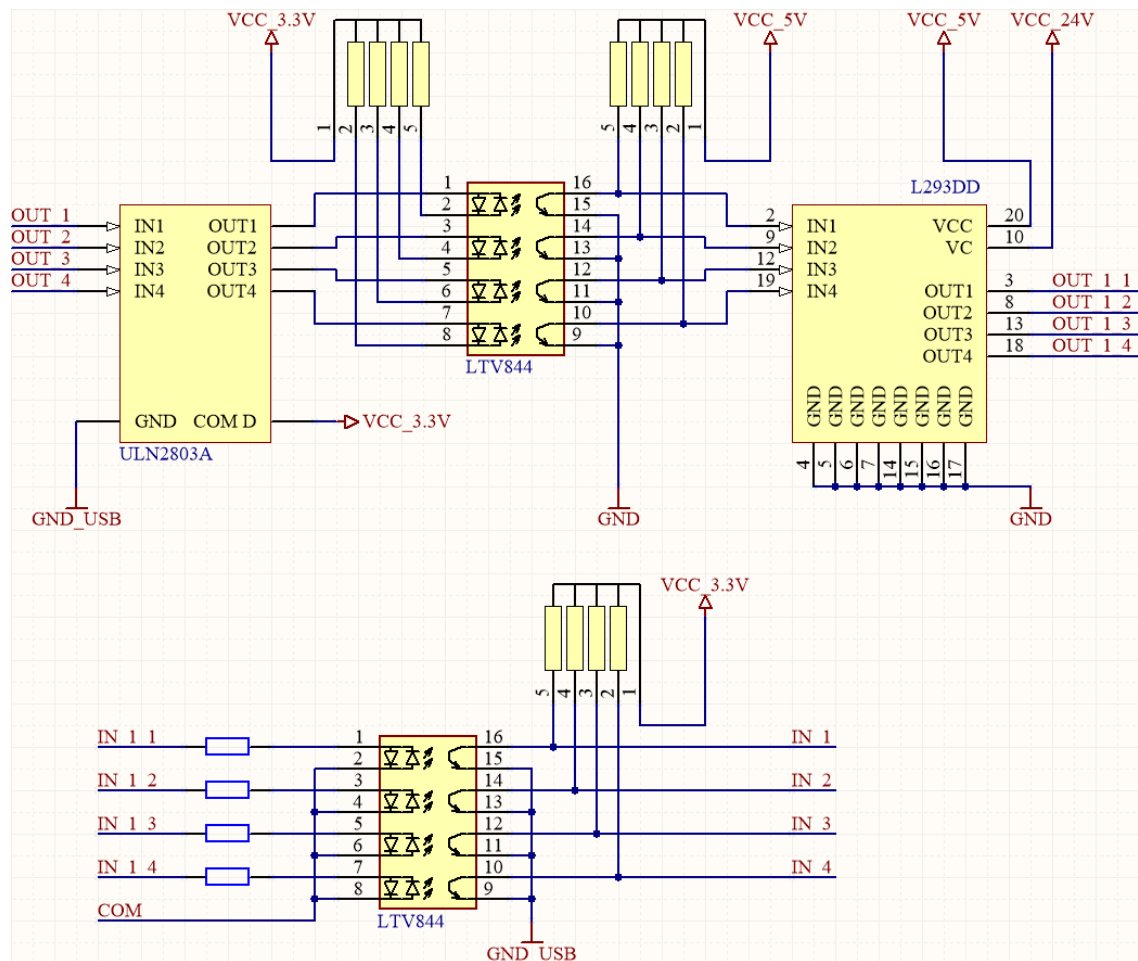
7.1.1. példa: A BORIS által küldött nyers utasítás byte-ok értelmezése és végrehajtása

A korábban szinkron módon működő adatküldés viszont az új megvalósításban már megszakítás vezéreltté lett téve [14] annak érdekében, hogy fő programnak ne kelljen megvárnia a kimenő byte-ok soros kiléptetését. A kommunikáció lebonyolításának teljes forráskódja az M20 mellékletben olvasható.

7.2. A kimenetek és a bemenetek felépítése

A tervezésnél a be- és kimenetek áramköreinek (7.2.1. ábra) a megtervezése jelentette a legnagyobb kihívást, ugyanis a megfelelő galvanikus leválasztás mellett fontos szempont volt, hogy a késleltetés lehető legalacsonyabb maradjon. A bemenetek

leválasztásához elegendő volt egy optocsatolót beiktatni a 24 V DC és a 3,3 V DC jelszintek közé, de a mikrovezérlő és az optocsatoló kimeneteinek alacsony teljesítménye miatt a kimenetek leválasztásához ugyanez a megoldás már nem lett volna elegendő. Ehelyett a mikrovezérlő kimenetei tranzisztorok segítségével kapcsolgatják a 3,3 V DC kimeneti jeleket, amik egy optocsatolón keresztül egy motorvezérlő kimeneteit kapcsolgatják. Az oszcilloszkópos mérések (M21 melléklet) eredményei alapján az elkészült be- és kimenetek átlagos késleltetése $\sim 3,5 \mu\text{s}$ és $\sim 60 \mu\text{s}$ között van.



7.2.1. ábra: A helyettesítő elektronika be- és kimeneteinek egyszerűsített kapcsolási rajza

Az eredeti I/O interfészhez képest egy újítás, hogy a hagyományos soros port helyett egy USB csatlakozó került az eszköz tokozására, ami egy USB-to-UART chip segítségével létesít soros kapcsolatot a számítógép és az eszköz mikrovezérlője között. Ez nem csak a soros porttal nem rendelkező számítógépek csatlakoztatását oldja meg, de az eszköz belső áramkörének a táplálását is.

8. Konklúzió

A feladatkiírásban megfogalmazott célokat sikeresen teljesítettem. A kiválasztott ipari folyamat emulációját, illetve a hozzá tartozó irányítást sikeresen megvalósítottam a feladat megoldásához biztosított HIL szimulátor és PLC segítségével.

Az emuláció megvalósításához rendelkezésre álló HIL szimulátort Görbedi Ákos kollégával közösen tanulmányoztuk át és közösen is kísérleteztünk vele. Az ebből szertett tapasztalatok révén pedig már egyedül kutattam tovább és dokumentáltam a FAB beépülőmodullal történő összetett vizualizációk megvalósítási lehetőségeit.

Az ipari folyamat emulálásának megvalósítása közben arra a meggyőződésre jutottam, hogy alapvetően a BORIS egy átgondolt és intuitív fejlesztői környezet, ráadásul bőséges eszközkínálattal rendelkezik, így, ha csak az erősségei alapján kellene megítélnem, akkor egy kiváló terméknek tartanám, amelyet nagyon jól lehetne alkalmazni az oktatásban, a kutatásban és a fejlesztésben is. Ezzel szemben a feladat megoldásához használt verzió sajnos igen súlyos hibákkal és hiányosságokkal rendelkezik a használhatóság és a stabilitás terén, amelyek közül a legkirívóbbak:

- A visszavonás (undo) és a megismétlés (redo) műveletek teljesen hiányoznak a szoftverből. Csak a mentés nélküli bezárással és visszanyitással lehet visszatérni egy korábbi állapothoz.
- A törlések nem kérnek megerősítést a felhasználótól, még az olyan blokkok esetén sem, amelyek összeköttetésben állnak más blokkokkal. Ez a visszavonás lehetősége nélkül felettébb felkavaró tud lenni.
- Előfordul, hogy az üzenetablakok, amelyek megkövetelik az elsőbbséget a felhasználótól, takarásban ugranak fel, meggátolva ezzel a felhasználót abban, hogy hozzájuk férjen. Ennek hatására az alkalmazás egyfajta blokkolt állapotába kerül.
- Minden külső fájlra abszolút elérési útvonalak formájában hivatkozik, ami miatt rendkívül körülményesen mozgatható a fájlrendszerben. Ezt a hiányosságot tovább súlyosbítja az, hogy a BORIS projektek megnyitásakor nincsenek ellenőrizve ezek a fájlhivatkozások, és ettől csak még nehezebb kinyomozni, ezeknek a hollétét.

Mindezek fényében és az emuláció megvalósításáért tett számos kerülőmegoldások hatására úgy vélem, hogy a BORIS a hozzá tartozó FAB beépülőmodullal együtt nem alkalmasak a kidolgozott ipari folyamathoz hasonló emulációk effektív megvalósítására. Ennek ellenére az I/O interfész megfelelően funkcionált, ezért mint különálló terméket teljes mértékben alkalmasnak találom arra, hogy akár az oktatásban, akár egy későbbi

irányítástechnikai versenyekre való felkészülésben használva legyen. Mindazonáltal fel kell ismerni azt a tényt is, hogy a BORIS nem egy általános célú programozási nyelv, mint például a C, hanem egy elsősorban rendszerek szimulációjára kifejlesztett speciális fejlesztői környezet, ezért nem is várható tőle, hogy minden feladat hatékonyan megoldható legyen benne. A rendeltetésszerű használatát viszont nem kíséreltük ki kellő mélységig ahhoz, hogy azzal kapcsolatban állás foglaljak.

Az feladatkiíráson túli céljaimat szintén sikerül teljesítenem, sőt még egy olyan segédprogramot is megvalósítottam a BORIS projektek hordozhatósági problémájának a megoldására, amely nem volt része sem a feladatkiírásnak, sem az azon túli feladatoknak.

A BORIS és az I/O interfész közötti kommunikáció feltárásával és az I/O interfész működtetéséhez fejlesztett függvénykönyvtáram segítségével megnyitottam a lehetőségét egy alternatív szimulációs szoftver fejlesztésének, amelyik képes lehet a BORIS kiváltására. Mindemellett a Qt Quick egy olyan újszerű felhasználási lehetőségét is kikísérleteztem és egy működő emulációval alá is támasztottam, amire korábban nem volt példa. A Qt Quick felhasználási területei ugyanis annak ellenére, hogy rendkívül szerteágazóak, ipari folyamatok emulálása nem tartozik közéjük. Ennek ellenére kifejezetten alkalmasnak találtam erre a célra, és potenciált látok a komponenseinek egy ilyen irányú bővítésében is.

Az I/O interfész helyettesítésére tett kísérleteim szintén sikeresek voltak, mindazonáltal a be- és kimeneti áramköreinek a megvalósításához, Hajdu József villamosmérnök segítségét kértem az elektronikában szerzett tágabb ismeretei és tapasztalatai miatt. Az ő segítségével a kísérleti helyettesítő elektronikából egy gyártható és kompakt termék született.

Irodalomjegyzék

- [1] Ingenieurbüro Dr. Kahlert, „WinFACT overview,” [Online]. Available: <https://www.kahlert.com/engl/winfact-uebersicht>.
- [2] Ingenieurbüro Dr. Kahlert, „Flexible Animation Builder,” [Online]. Available: <https://www.kahlert.com/engl/flexible-animation-builder>.
- [3] Omron Corporation, „CJ2M-CPU32,” [Online]. Available: <https://industrial.omron.hu/hu/products/CJ2M-CPU32>.
- [4] Omron Corporation, „CX-Programmer,” [Online]. Available: <https://industrial.omron.hu/hu/products/cx-programmer>.
- [5] Wikipedia, „IEC 61131-3,” [Online]. Available: https://en.wikipedia.org/wiki/IEC_61131-3.
- [6] Eltima IBC, „Serial Port Monitor,” [Online]. Available: <https://www.eltima.com/products/serial-port-monitor/>.
- [7] MikroElektronika d.o.o., „EasyPIC v7,” [Online]. Available: <https://www.mikroe.com/easypic-v7>.
- [8] MikroElektronika d.o.o., „mikroC PRO for PIC,” [Online]. Available: <https://www.mikroe.com/mikroc-pic>.
- [9] Qt Company, „Qt Creator,” [Online]. Available: https://wiki.qt.io/Qt_Creator/hu.
- [10] Wikipedia, „Altium Designer,” [Online]. Available: https://en.wikipedia.org/wiki/Altium_Designer.
- [11] IAR Systems AB, „IAR Embedded Workbench for Arm,” [Online]. Available: <https://www.iar.com/products/architectures/arm>.
- [12] Wikipedia, „Modulo operation,” [Online]. Available: https://en.wikipedia.org/wiki/Modulo_operation.
- [13] Wikipedia, „Path (computing),” [Online]. Available: [https://en.wikipedia.org/wiki/Path_\(computing\)](https://en.wikipedia.org/wiki/Path_(computing)).
- [14] ARM Keil, „STM32 USART (interrupt mode) Example,” [Online]. Available: <https://www.keil.com/download/docs/359.asp>.

Nyilatkozat

Alulírott Miklós Árpád, mérnök informatikus MSc. szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Tanszékcsoport Műszaki Informatika Tanszékén készítettem, mérnök informatikus MSc. diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Informatikai Tanszékcsoport könyvtárában, a helyben olvasható könyvek között helyezik el.

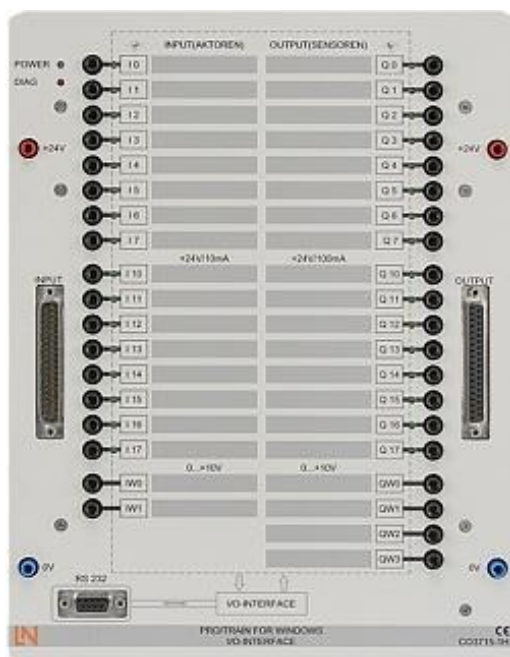
2021. május 10.

.....

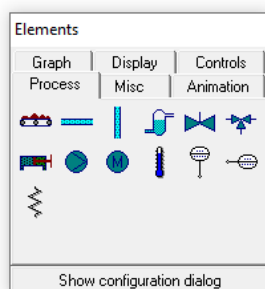
Aláírás

Mellékletek

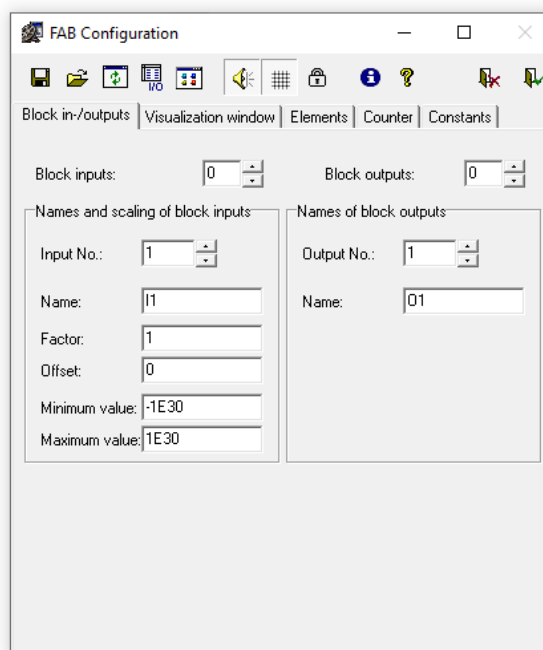
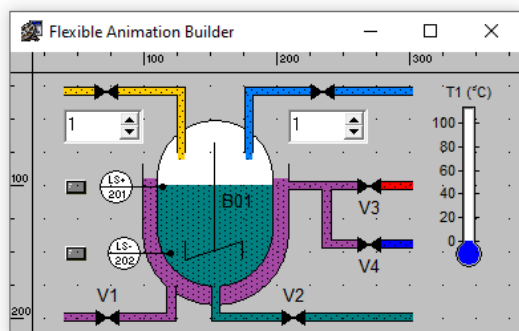
M1 A CO3715-1H típusjelzésű I/O interfész



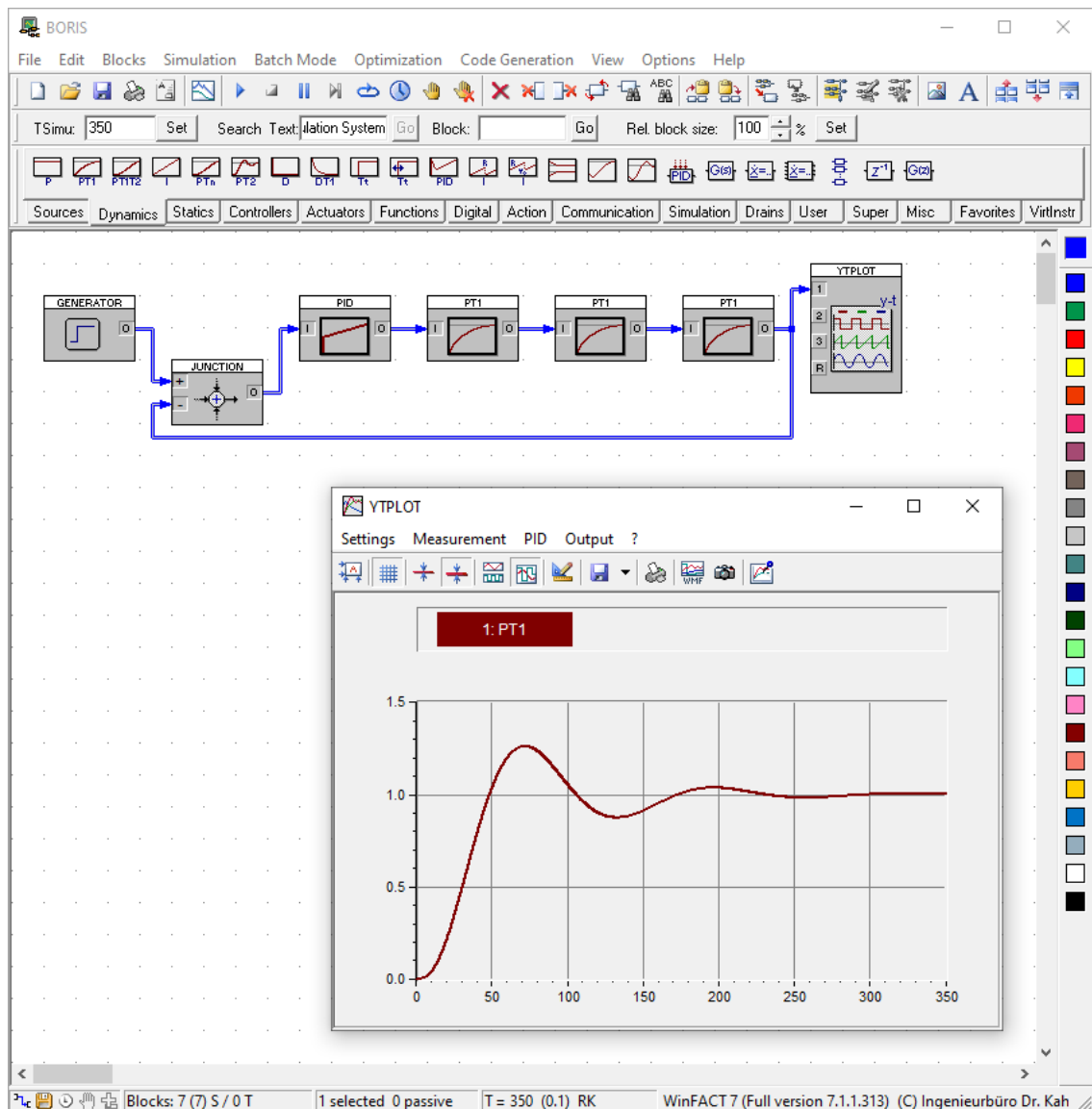
M2 A FAB kezelőfelülete



Nr.	Input	Output
1	0	0
2	0	0
3	0	0
4	0	0
5	0	0
6	0	0
7	0	0
	+0.01	+0.1
	+1	+10
	+100	+100
	-0.01	-0.1
	-1	-10
	-100	-100



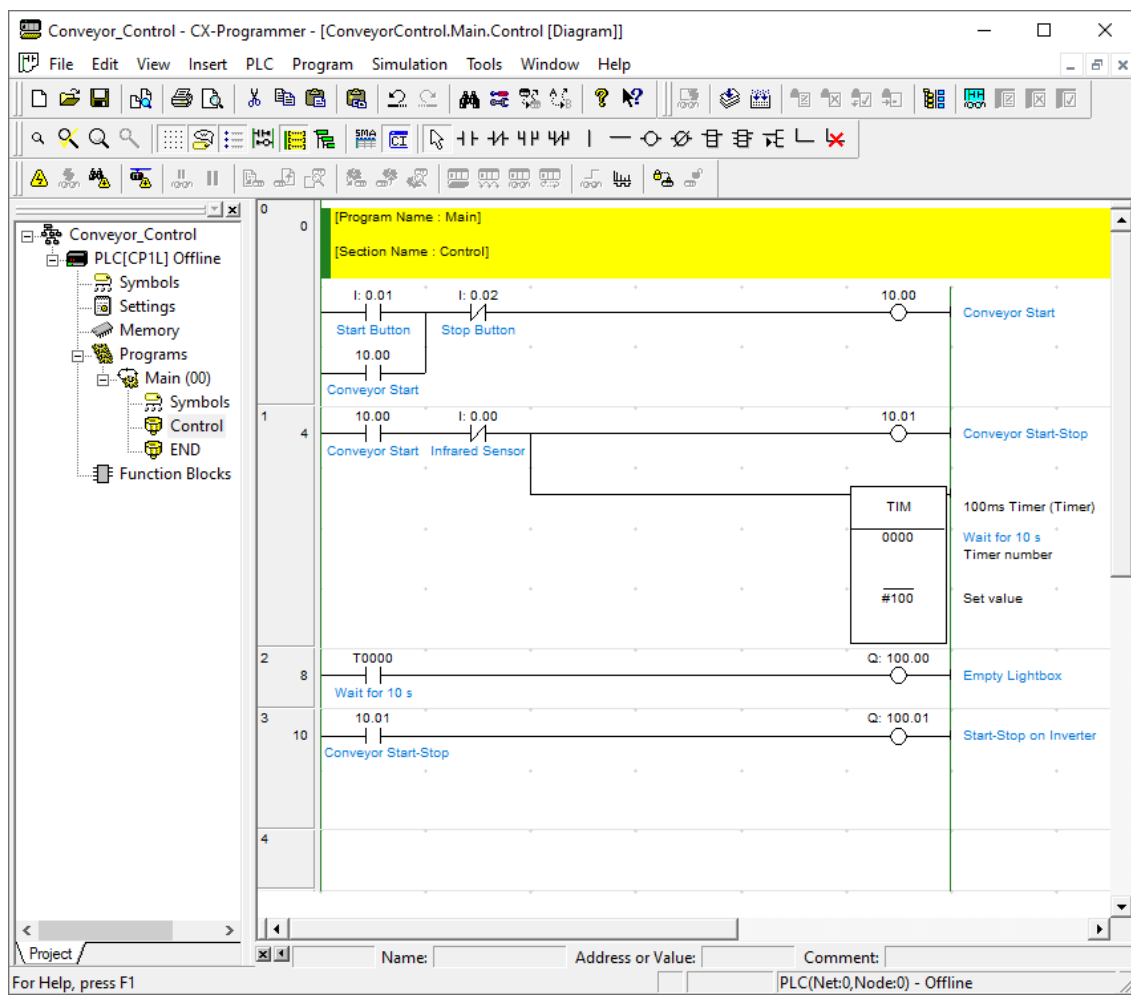
M3 A BORIS kezelőfelülete



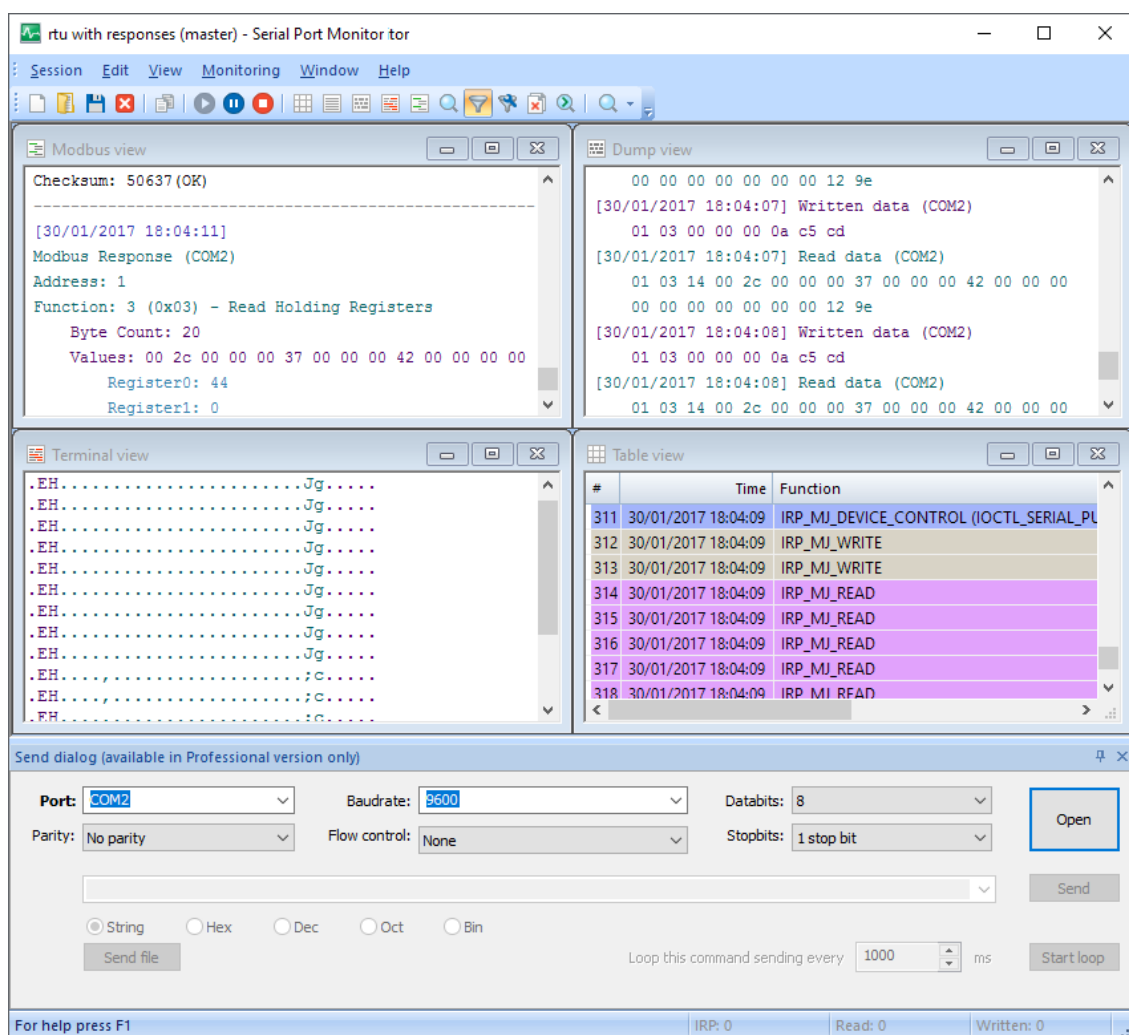
M4 A CJ2M-CPU32 típusú PLC



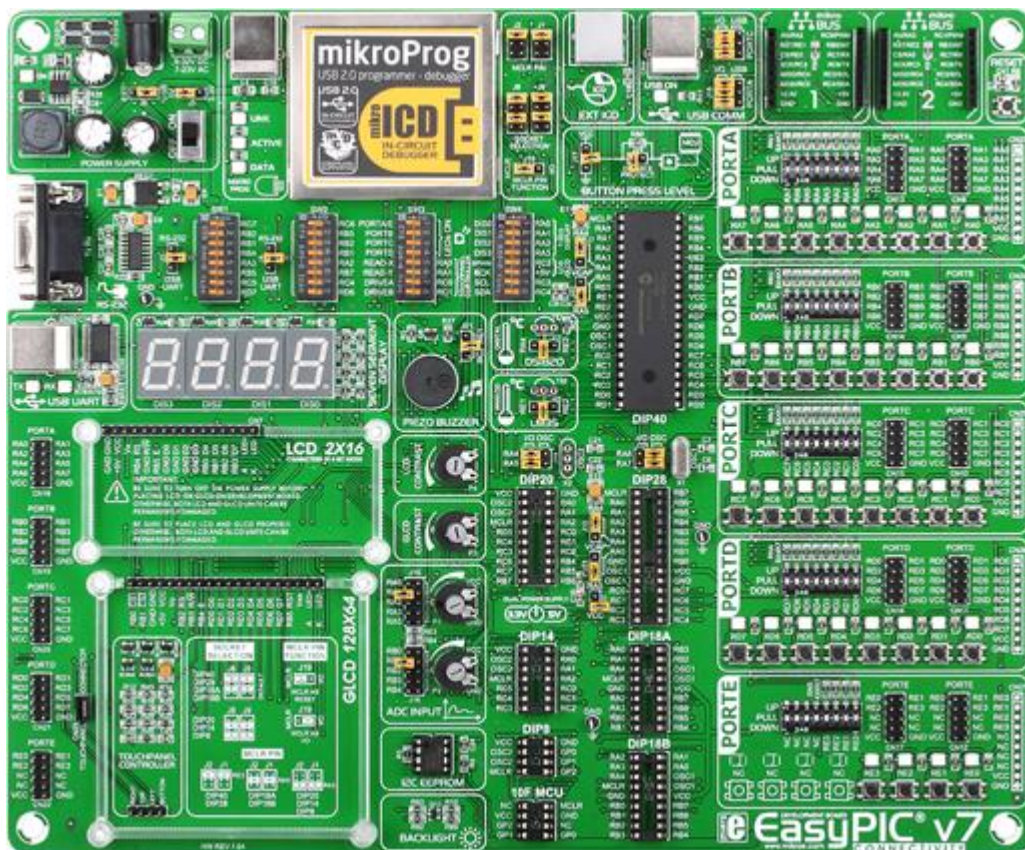
M5 A CX-Programmer kezelőfelülete



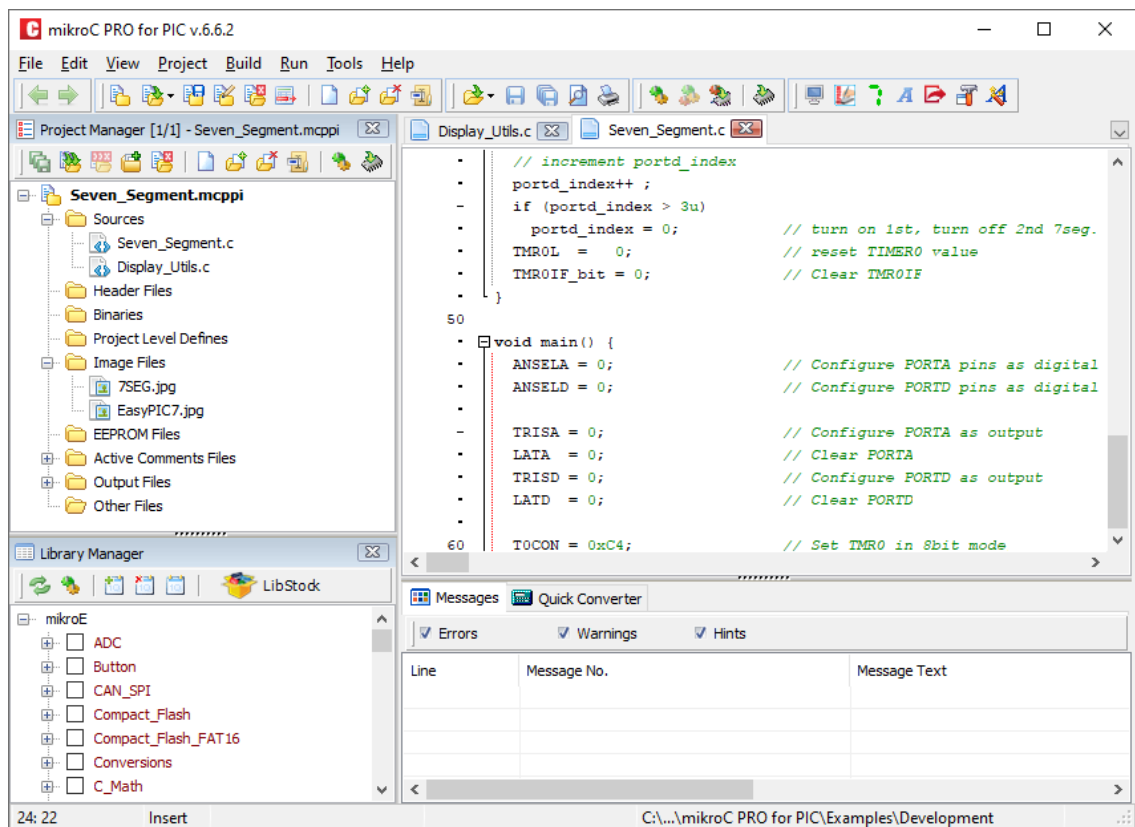
M6 A Serial Port Monitor kezelőfelülete



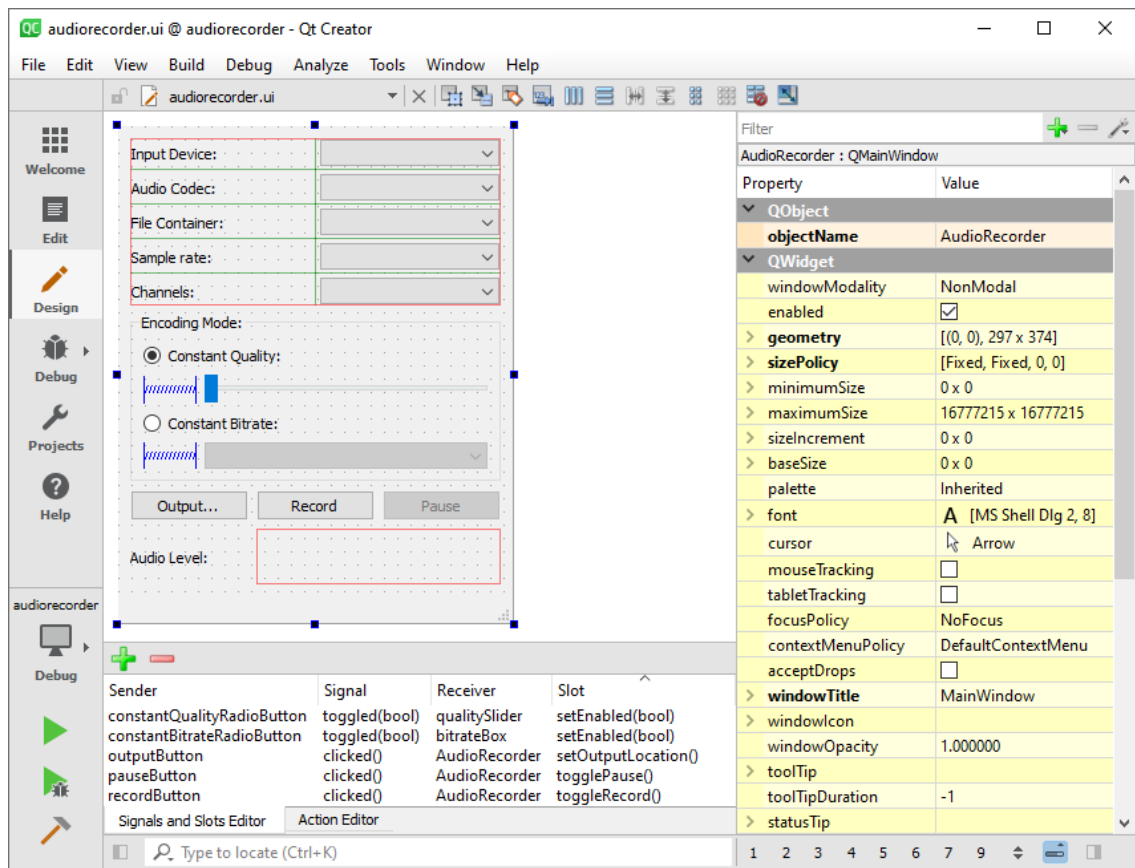
M7 Az EasyPIC v7 fejlesztőlap



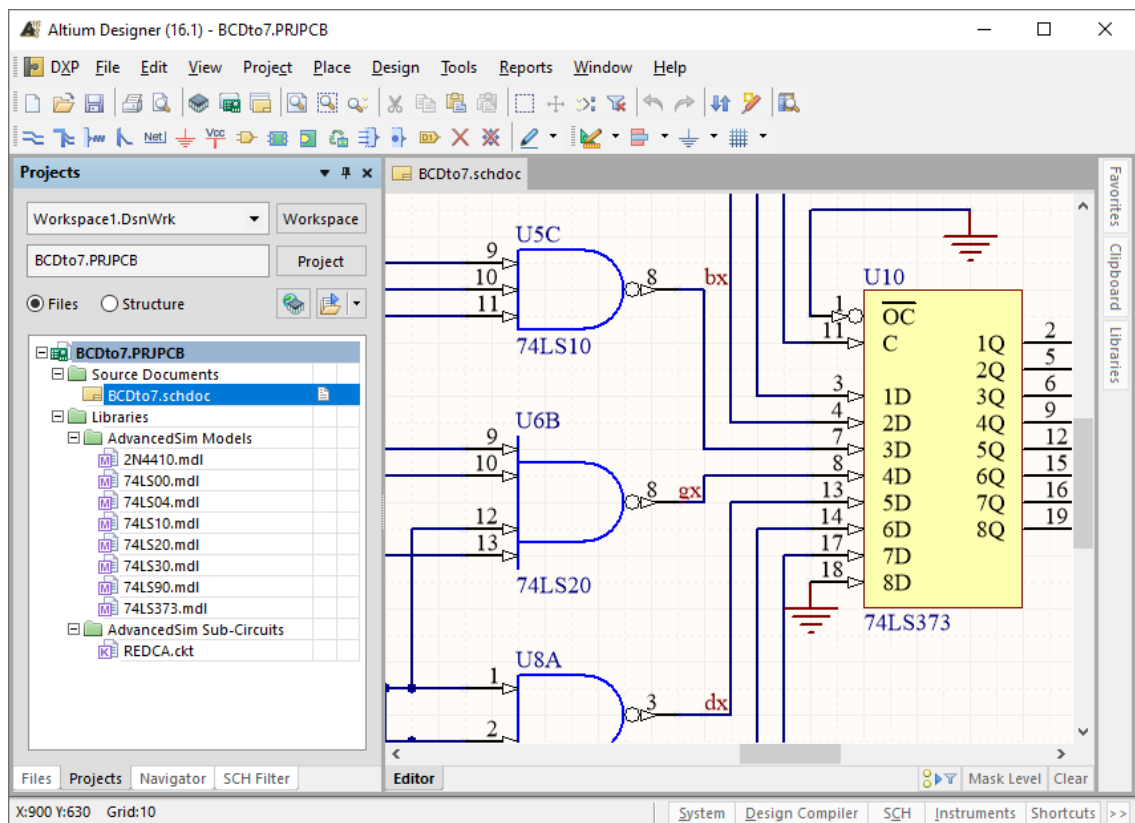
M8 A mikroC PRO for PIC kezelőfelülete



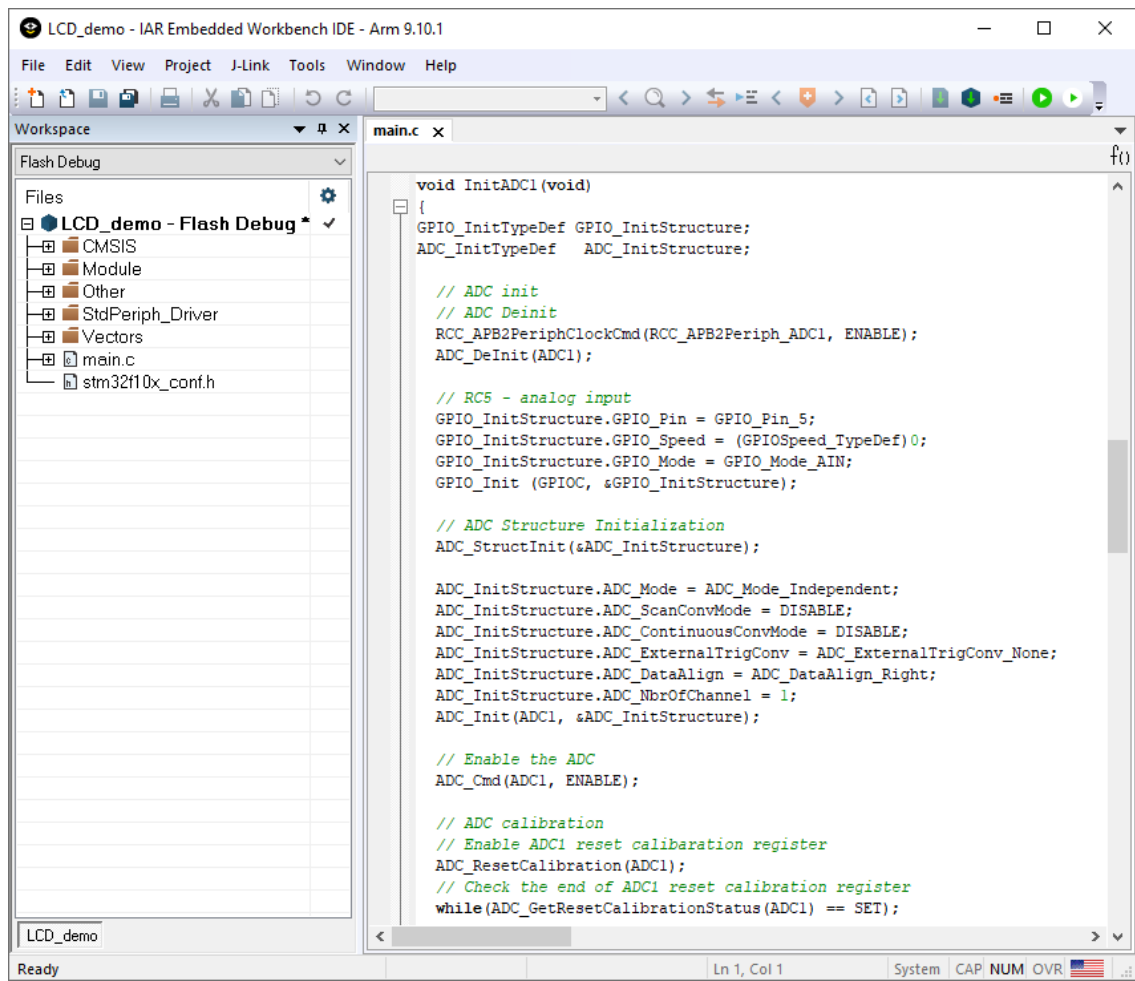
M9 A Qt Creator kezelőfelülete



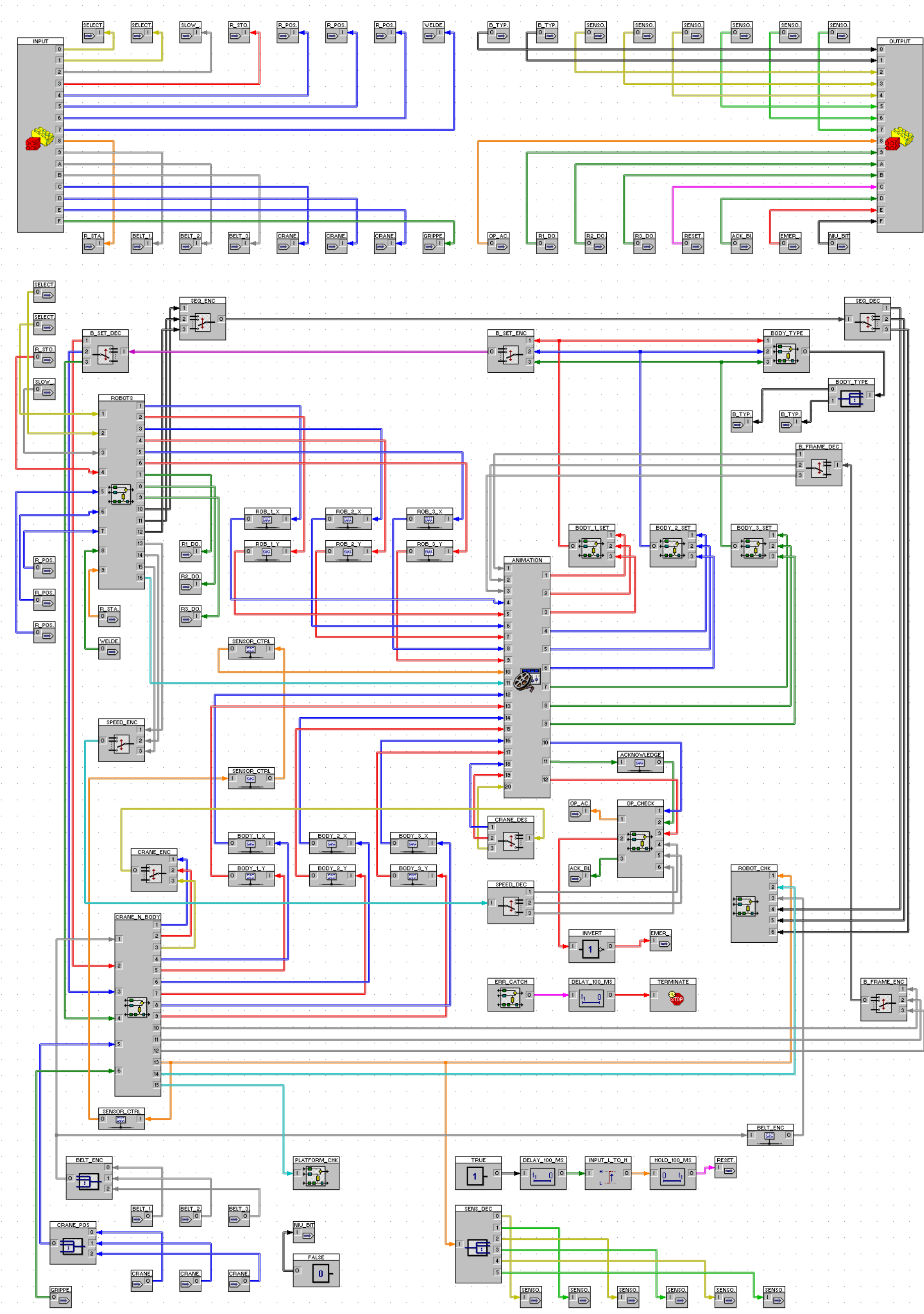
M10 Az Altium Designer kezelőfelülete



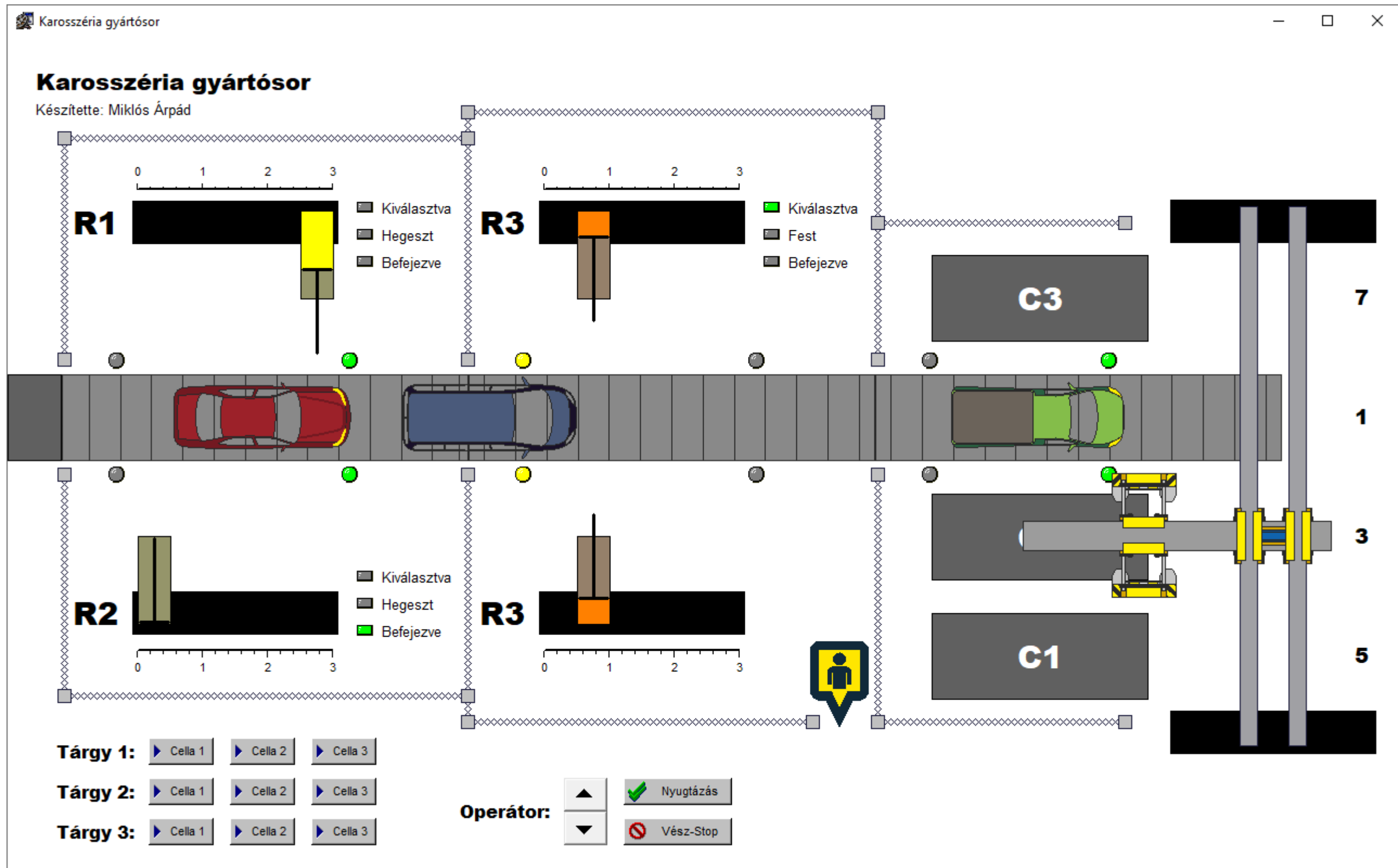
M11 Az IAR Embedded Workbench for Arm kezelőfelülete



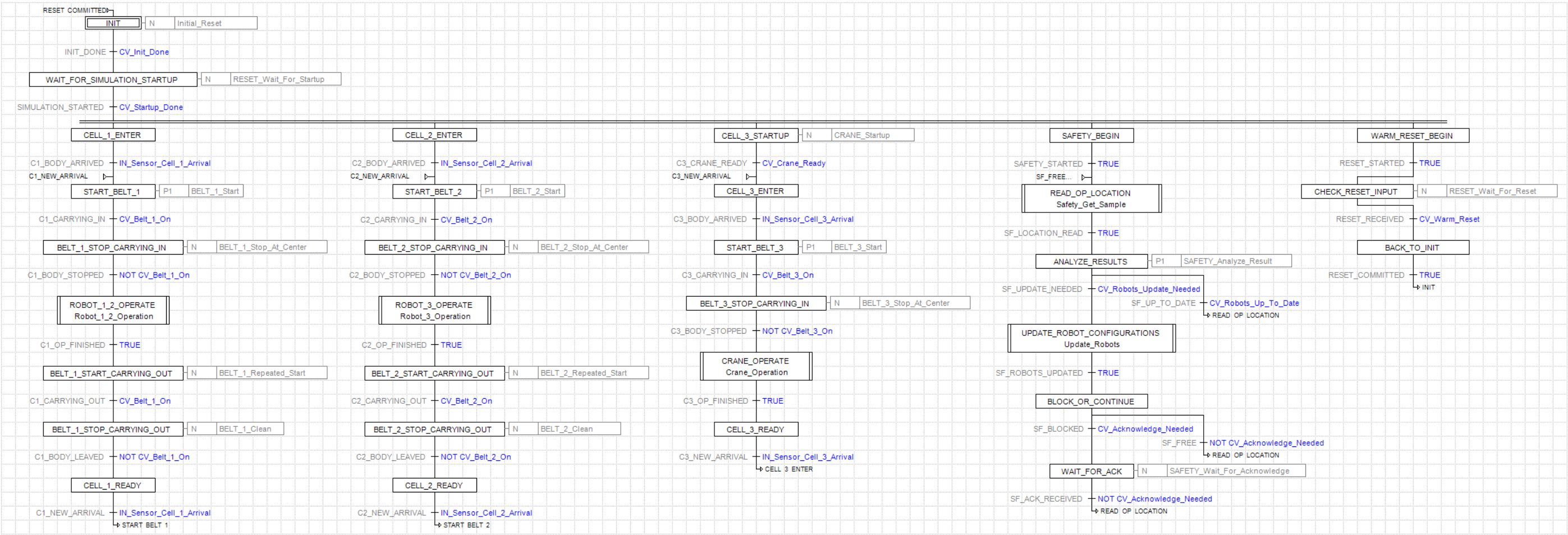
M12 A karosszéria gyártósor emulációjának legfelső szintje



M13 A karosszéria gyártósor vizualizációja



M14 A karosszéria gyártósor irányításának legfelső szintje



M15 A BORIS Corrector osztályának teljes forráskódja

```
#include <boris_corrector.h>
#include <cstdlib>
#include <iomanip>
#include <fstream>

using namespace diplomamunka;

bool diplomamunka::is_boris_file(const fs::path &p)
{
    static const std::set<std::string> supported_types
    {
        ".bmp", ".bsy", ".fab", ".sbl"
    };
    auto extension = util::to_lower(p.extension().string());
    return supported_types.count(extension) > 0;
}

fs::path diplomamunka::add_temp_prefix(const fs::path &p)
{
    auto temp = p;
    temp.replace_filename("~$" + p.filename().string());
    return temp;
}

boris_corrector::boris_corrector(const fs::path &working_dir) :
    cwd(working_dir) {}

std::set<fs::path> boris_corrector::search_for_files()
{
    fs::recursive_directory_range finder(cwd);
    std::set<fs::path> files;
    std::copy_if
    (
        finder.begin(), finder.end(),
        std::inserter(files, files.end()), is_boris_file
    );
    return files;
}

std::string boris_corrector::correct_paths(const std::string &s,
    const std::set<fs::path> &files)
{
    static const std::regex matcher
    {
        R"([a-z]:\\([^\\"/:*?"<>|\r\n]+\\)*)" +
        R"([^\\"/:*?"<>|\r\n]+\.(bmp|bsy|fab|sbl))",
        std::regex::ECMAScript | std::regex::icase |
        std::regex::nosubs | std::regex::optimize
    };
};
```

```

    return util::regex_replace(s, matcher,
        [this, &files](const std::string &match)
        {
            auto it = std::find_if(files.begin(), files.end(),
                [this, &match](const fs::path &file)
                {
                    return fs::ends_with(match, fs::relative(file, cwd));
                });

            return it != files.end() ? it->string() : match;
        });
}

int boris_corrector::execute()
{
    const auto logPercent = [](std::size_t p) -> std::ostream &
    {
        return std::cout << "[" << std::setw(3) << p << "%] ";
    };

    double percentage = 0.0;
    logPercent(percentage)
        << "Searching for BORIS-specific files..." << std::endl;

    try
    {
        const auto files = search_for_files();
        const double step = 100.0 / files.size();
        logPercent(step / 2.0)
            << files.size() << " files found. " << std::endl;

        for (const auto &file : files)
        {
            logPercent(percentage += step)
                << "Processing file: " << file.string() << "... ";

            if (correct_file(file, files))
                std::cout << "DONE" << std::endl;
            else
                std::cout << "FAILED" << std::endl;
        }
    }
    catch (std::exception &e)
    {
        logPercent(percentage)
            << "Error occurred: " << e.what() << std::endl;
        return EXIT_FAILURE;
    }

    logPercent(100) << "Correction finished." << std::endl;
    return EXIT_SUCCESS;
}

```

```

bool boris_corrector::correct_file(const fs::path &f,
    const std::set<fs::path> &files)
{
    const auto temp = add_temp_prefix(f);

    if (std::fstream in(f.string(), std::ios::in),
        out(temp.string(), std::ios::out); in && out)
    {
        for (std::string line; std::getline(in, line);)
            out << correct_paths(line, files) << std::endl;
    }
    else
    {
        return false;
    }

    fs::remove(f);
    fs::rename(temp, f);
    return true;
}

```


M16 Az I/O interfész működését utánzó prototípus programja

A prototípus megszakításkezelő és main függvényeinek forráskódja

```
#include <common.h>
#include <core/BorisProdigy.h>
#include <serialcomm/Serial.h>

/**
 * @brief A magas prioritású megszakításokat kiszolgáló rutin.
 *
 * Amennyiben a prioritáskezelés nincs engedélyezve, minden
 * megszakítás ide fut be.
 */
void interrupt() {
    /* Az UART fogadó pufferébe érkezett adat hatására bekövetkezett
    megszakítás kezelése. */
    if (RC1IF_bit && RC1IE_bit)
        Serial_handleInterrupt();
}

/**
 * @brief A program futásának kezdőpontja.
 */
void main() {
    /* Az alkalmazás inicializálása és futtatása. */
    BorisProdigy_initialize();
    BorisProdigy_run();
}
```

A prototípushoz tartozó soros port kezelésének forráskódja

```
#include <serialcomm/Serial.h>
#include <core/BorisProdigy.h>

/**
 * A beérkező adatokat tároló fogadó puffer deklarációja és
 * inicializálása.
 */
uint8_t receiveBuffer[10] = { 0 };

/**
 * Az elküldésre szánt adatokat tároló kimenő puffer deklarációja és
 * inicializálása.
 */
uint8_t transmitBuffer[20] = { 0 };

/**
 * A beérkező adatbyte-okat számláló változó deklarációja és
 * inicializálása.
 */
uint8_t receiveCounter = 0;
```

```

/*
 * Inicializálja a soros kommunikációs portot és beállítja az
 * adatátviteli sebességet a megadott értéknek megfelelően, majd
 * engedélyezi a megszakítást az adatbyte-ok érkezésére.
 */
void Serial_initialize(enum BaudRate baudrate) {
    /* A soros kommunikációs port inicializálása és a megadott
    adatátviteli sebesség beállítása. */
    switch (baudrate) {
        case BAUD_1200:
            UART1_Init(1200); break;
        case BAUD_2400:
            UART1_Init(2400); break;
        case BAUD_4800:
            UART1_Init(4800); break;
        case BAUD_9600:
            UART1_Init(9600); break;
        case BAUD_19200:
            UART1_Init(19200); break;
        case BAUD_38400:
            UART1_Init(38400); break;
        case BAUD_57600:
            UART1_Init(57600); break;
        case BAUD_115200:
            UART1_Init(115200); break;
    }

    /* A soros portra érkezett adatbyte-okkal járó megszakítások
    engedélyezése. */
    RC1IF_bit = false;
    RC1IE_bit = true;
}

/*
 * Kezeli a soros portra érkezett adatbyte-tal járó megszakítást és
 * törli a megszakítás jelzőbitjét.
 */
void Serial_handleInterrupt() {
    /* A fogadott byte kiolvasása a soros port pufferéből. */
    uint8_t next_byte = UART1_Read();

    /* A fogadott byte-nak megfelelő kérelem belehelyezése a kérelmek
    várakozási sorába. */
    switch (next_byte) {
        case 0x02: /* Get Device Type Request */
            BorisProdigy_setRequest(REQUEST_GET_DEVICE_TYPE, 0, 0);
            break;
        case 0xb9: /* Get Input Request */
            BorisProdigy_setRequest(REQUEST_GET_INPUT, 0, 0);
            break;
        case 0xba: /* Set Output Request */
            receiveCounter = 2;
            break;
    }
}

```

```

/* A kimenetet beállító új paraméterek fogadása, amennyiben egy
korábbi utasítás az új paraméterek érkezését jelezte. */
if (receiveCounter && next_byte != 0xba) {
    receiveBuffer[2 - receiveCounter] = next_byte;

    if (--receiveCounter == 0)
        BorisProdigy_setRequest(REQUEST_SET_OUTPUT, receiveBuffer[0],
        receiveBuffer[1]);
}

/* A lekezelt megszakítás jelzőbitjének törlése. */
RC1IF_bit = false;
}

/*
 * Kiküldi az átadott karakterláncot a soros kommunikációs porton
 * keresztül.
 */
void Serial_sendString(char* str) {
    Serial_sendData(str, strlen(str));
}

/*
 * Kiküldi az átadott adattömböt a soros kommunikációs porton
 * keresztül.
 */
void Serial_sendData(uint8_t* data_ptr, uint8_t size) {
    uint8_t i;

    /* A mutatott memóriaterület átmásolása a kimenő pufferbe és a
    kimenő puffer tartalmának kiküldése a soros kommunikációs porton
    keresztül. */
    memcpy(transmitBuffer, data_ptr, size);
    for (i = 0; i < size; i++)
        UART1_Write(transmitBuffer[i]);
}

```

A BORIS utasításait kiszolgáló központi logika forráskódja

```

#include <core/BorisProdigy.h>
#include <serialcomm/Serial.h>

/*
 * A kérelmek várakozási sorának deklarálása.
 */
volatile RequestQueue requestQueue;

/*
 * Az eszköz sorozatszámának deklarálása és inicializálása.
 */
uint8_t serialNumber[] = "S03715-1H.110";

```

```

/*
 * Inicializálja a mikrovezérlő I/O portjait, a perifériáit és az
 * alkalmazást a hozzá kapcsolódó összes modullal együtt.
 */
void BorisProdigy_initialize() {
    /* Az A/D és D/A átalakítók, és a komparátorok kikapcsolása. */
    ADCON0 = ADCON1 = ADCON2 = 0x00;
    CM1CON0 = CM2CON0 = VREFCON0 = VREFCON1 = VREFCON2 = 0x00;

    /* Minden I/O port standard jelváltozási sebességű digitális
    bemenetre konfigurálása, és a hozzájuk tartozó tárolók törlése. */
    ANSELA = ANSELB = ANSELC = ANSELD = ANSELE = 0x00;
    SLRCON = LATA = LATB = LATC = LATD = LATE = 0x00;
    TRISA = TRISB = TRISC = TRISD = TRISE = 0xFF;

    /* A B és C portok digitális kimenetre konfigurálása a LED-ek
    működtetéséhez. */
    TRISB = TRISD = 0x00;

    /* Az alkalmazás moduljainak inicializálása. */
    requestQueue.begin = requestQueue.end = 0;
    Serial_initialize(BAUD_9600);
}

/*
 * Eltárolja a megadott kérelmet a kérelmek várakozási sorának soron
 * következő rekeszébe.
 */
void BorisProdigy_setRequest(enum RequestType type, uint8_t data_h,
    uint8_t data_l) {
    /* A várakozási sor következő elemének felülírása a megadott kérelem
    paraméterekkel. */
    requestQueue.requests[requestQueue.end].type = type;
    requestQueue.requests[requestQueue.end].dataHigh = data_h;
    requestQueue.requests[requestQueue.end].dataLow = data_l;

    /* A sor utolsó elemére mutató számláló növelése. */
    requestQueue.end = (requestQueue.end + 1) % REQUEST_QUEUE_SIZE;

    /* A várakozási sorban levő következő kérelem kijelölése, amennyiben
    a várakozási sorban túlcsoordulás keletkezett. (A legrégebbi kérelem
    el fog veszni.) */
    if (requestQueue.end == requestQueue.begin)
        requestQueue.begin =
            (requestQueue.begin + 1) % REQUEST_QUEUE_SIZE;
}

```

```

/*
 * Feldolgozza a soron következő kérelmet a kérelmek várakozási
 * sorából, amennyiben a sor nem üres.
 */
void BorisProdigy_handleRequests() {
    /* A bemenet beolvasása után annak a jelenlegi állapotait tároló
    tömb. */
    uint8_t port_data[2] = { 0 };

    /* Azonnali visszatérés a függvényből, ha a várakozási sor üres. */
    if (requestQueue.begin == requestQueue.end) return;

    /* A soron következő kérelem feldolgozása. */
    switch (requestQueue.requests[requestQueue.begin].type) {
    case REQUEST_GET_DEVICE_TYPE:
        Serial_sendData(serialNumber, 14);
        break;
    case REQUEST_GET_INPUT:
        port_data[0] = PORTA;
        port_data[1] = 0 | (PORTC & 0x0F) | ((PORTE & 0x0F) << 4);
        Serial_sendData(port_data, 2);
        break;
    case REQUEST_SET_OUTPUT:
        LATB = requestQueue.requests[requestQueue.begin].dataHigh;
        LATD = requestQueue.requests[requestQueue.begin].dataLow;
        break;
    }

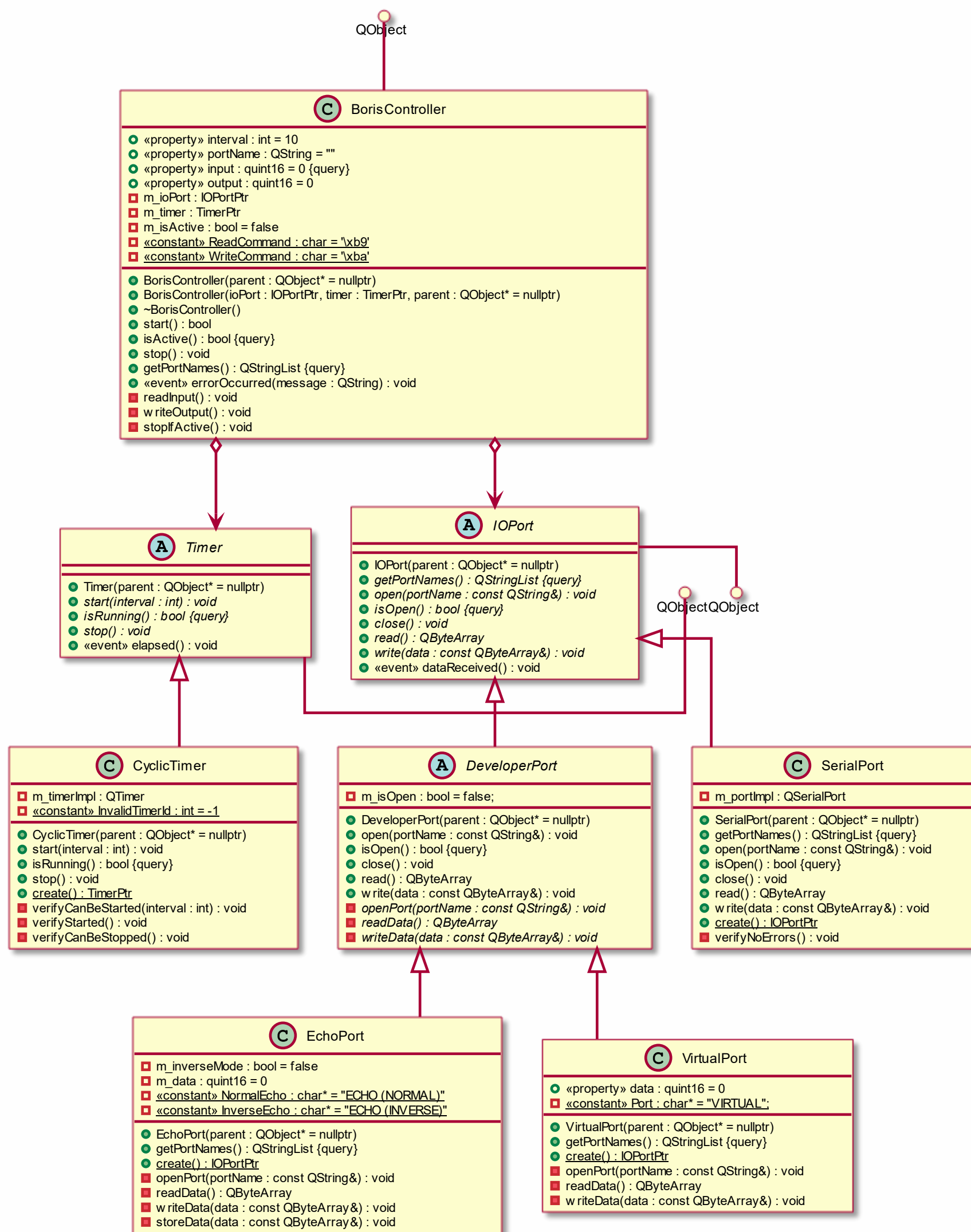
    /* A várakozási sorban levő következő kérelem kijelölése. */
    requestQueue.begin = (requestQueue.begin + 1) % REQUEST_QUEUE_SIZE;
}

/*
 * Engedélyezi a globális megszakításokat és végtelen ciklusban
 * végrehajtja a beékező kérélmeket.
 */
void BorisProdigy_run() {
    /* A megszakítások prioritáskezelésének tiltása, majd a periferiális
    és globális megszakítások engedélyezése. */
    IPEN_bit = false;
    PEIE_bit = GIE_bit = true;

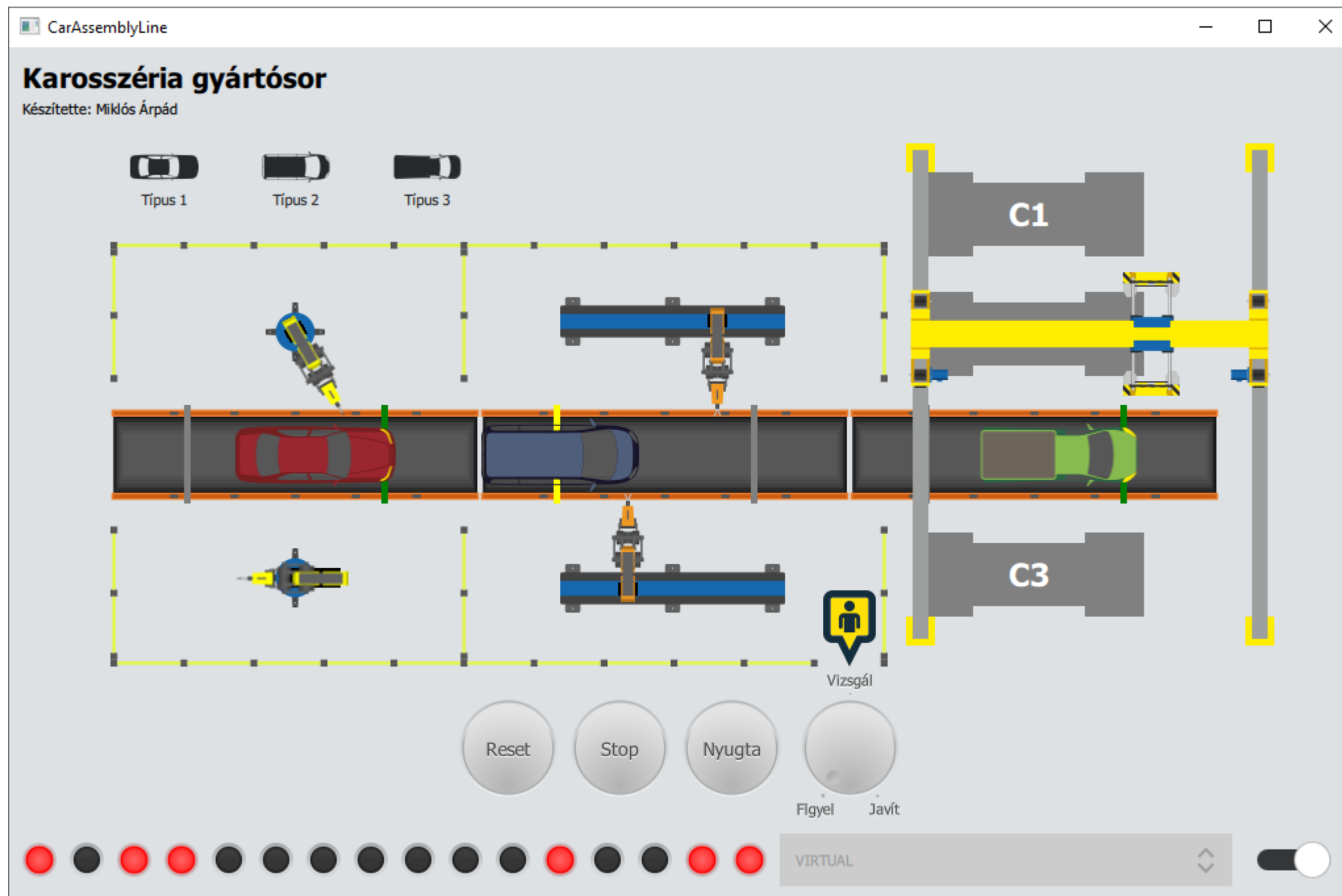
    /* A beérkezett kérélmek folyamatos feldolgozása és a megfelelő
    műveletek végrehajtása. */
    requestQueue.begin = requestQueue.end = 0;
    while (true)
        BorisProdigy_handleRequests();
}

```

Diplomamunka



M18 A karosszéria gyártósor alternatív vizualizációja



M19 A BORIS Controller integrálása az alternatív emulációba

Az integráció megvalósítása C++ oldalon

```
#include <BorisController.h>
#include <CyclicTimer.h>
#include <VirtualPort.h>

#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <QQmlContext>

using namespace Diplomamunka;

int main(int argc, char* argv[]) {
    QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
    QGuiApplication application(argc, argv);
    QQmlApplicationEngine engine;

#ifdef QT_DEBUG
    IOPortPtr ioPort = VirtualPort::create();
    BorisController boris(ioPort, CyclicTimer::create());

    engine.rootContext()->setContextProperty("debugPort",
        ioPort.get());
    engine.rootContext()->setContextProperty("boris", &boris);
#else
    BorisController boris;

    engine.rootContext()->setContextProperty("debugPort", nullptr);
    engine.rootContext()->setContextProperty("boris", &boris);
#endif
    engine.load(QUrl("qrc:/MainWindow.qml"));

    return application.exec();
}
```


Az integráció megvalósítása QML oldalon

```
import QtQuick 2.14
import QtQuick.Window 2.14

Window {
    id: window

    visible: true
    color: "#ddee4"
    minimumWidth: 1024
    minimumHeight: 648

    contentItem.transform: Scale {
        xScale: yScale
        yScale: {
            var widthRatio = width / minimumWidth
            var heightRatio = height / minimumHeight

            return Math.min(widthRatio, heightRatio)
        }
    }

    Column {
        spacing: 10

        ProjectTitle {
            id: projectTitle

            title: "Karosszéria gyártósor"
            author: "Miklós Árpád"
        }

        AssemblyLine {
            id: assemblyLine

            Binding {
                target: assemblyLine
                property: "input"
                value: boris ? boris.input : 0
            }

            Binding {
                target: boris
                property: "output"
                value: assemblyLine.output
            }
        }
    }
}
```

```

PortController {
    id: portController

    debugging: debugPort

    Component.onCompleted: portNames = boris.getPortNames()
    onDropDownOpened: portNames = boris.getPortNames()

    onConnectedChanged: {
        if (connected) {
            connected = boris.start()
        } else if (boris.isActive()) {
            boris.stop()
        }
    }

    Binding {
        target: boris
        property: "portName"
        value: portController.selectedPort
    }

    Binding {
        target: debugPort
        when: portController.debugging
        property: "data"
        value: portController.debugData
    }

    Connections {
        target: boris
        onErrorOccurred: portController.showError(message)
    }
}
}
}

```

M20 A helyettesítő elektronika kommunikációját lebonyolító program

```
#include "init.h"
#include "stm32f10x.h"
#include <stdint.h>
#include <stdio.h>
#include <string.h>

#define TBUF_SIZE 256
#define MAX_DATANUMBER 3

void UART_AdatKuld(char data);
void UART_StrKuld(volatile char* s);
unsigned int read_port_input(void);

struct buf_st {
    unsigned int in;      // Következő bekezezés indexe
    unsigned int out;     // Következő küldés indexe
    char buf[TBUF_SIZE]; // Puffer
};

static struct buf_st tbuf = {
    0,
    0,
};

static unsigned int tx_restart = 1;

unsigned char receiveddata[MAX_DATANUMBER + 1];
unsigned char datanumber;
unsigned int portout, portin;

int main() {
    tbuf.in = 0;
    tbuf.out = 0;
    tx_restart = 1;

    Clk_Init();

    GPIOB_Init();
    GPIOA_Init();
    GPIOD_Init();
    GPIOE_Init();

    USART2_Init();

    GPIO_PinRemapConfig(GPIO_Remap_SWJ_JTAGDisable, ENABLE);
```

```

while (1) {
    if (datanumber != 0) {
        switch (receiveddata[0]) {
            case 0x02:
                UART_StrKuld("SO3715-1H.110");
                datanumber = 0;
                break;
            case 0xb9:
                portin = read_port_input();
                UART_AdatKuld(portin >> 8);
                UART_AdatKuld(portin);
                datanumber = 0;
                break;
            case 0xba:
                if (datanumber == 3) {
                    portout = receiveddata[1];
                    portout = (portout << 8) + receiveddata[2];
                    GPIO_Write(GPIOD, (0xFFFF - portout));
                    datanumber = 0;
                }
                break;
            default:
                datanumber = 0;
                break;
        }
    }
}

unsigned int read_port_input(void) {
    unsigned int input, input_s;
    input_s = GPIO_ReadInputData(GPIOB);
    input = ((input_s >> 3) & 0x007F);
    input = input + ((input_s << 2) & 0xf000);
    input_s = GPIO_ReadInputData(GPIOA);
    input = input + ((input_s << 4) & 0x0f00);
    input_s = GPIO_ReadInputData(GPIOE);
    input = input + ((input_s << 7) & 0x0080);
    return (0xffff - input);
}

void UART_StrKuld(volatile char* s) {
    while (*s) {
        UART_AdatKuld(*s);
        *s++;
    }
    UART_AdatKuld(0x00);
}

```

```

void UART_AdatKuld(char data) {
    struct buf_st* p = &tbuf;

    // Adatok hozzáadása az átviteli pufferhez.
    p->buf[p->in & (TBUF_SIZE - 1)] = data;
    p->in++;

    // A TX megszakítás engedélyezése, ha éppen tiltva van
    if (tx_restart) {
        tx_restart = 0;
        USART_ITConfig(USART2, USART_IT_TXE, ENABLE);
    }
}

void USART2_IRQHandler(void) {
    struct buf_st* p;

    /* RXNE megszakítás-kezelő */
    if (USART_GetITStatus(USART2, USART_IT_RXNE) != RESET) {
        receiveddata[datanumber] = USART_ReceiveData(USART2);
        datanumber++;
    }

    /* TXE megszakítás-kezelő */
    if (USART_GetITStatus(USART2, USART_IT_TXE) != RESET) {

        USART2->SR &= ~USART_FLAG_TXE; // Megszakítás törlése
        p = &tbuf;

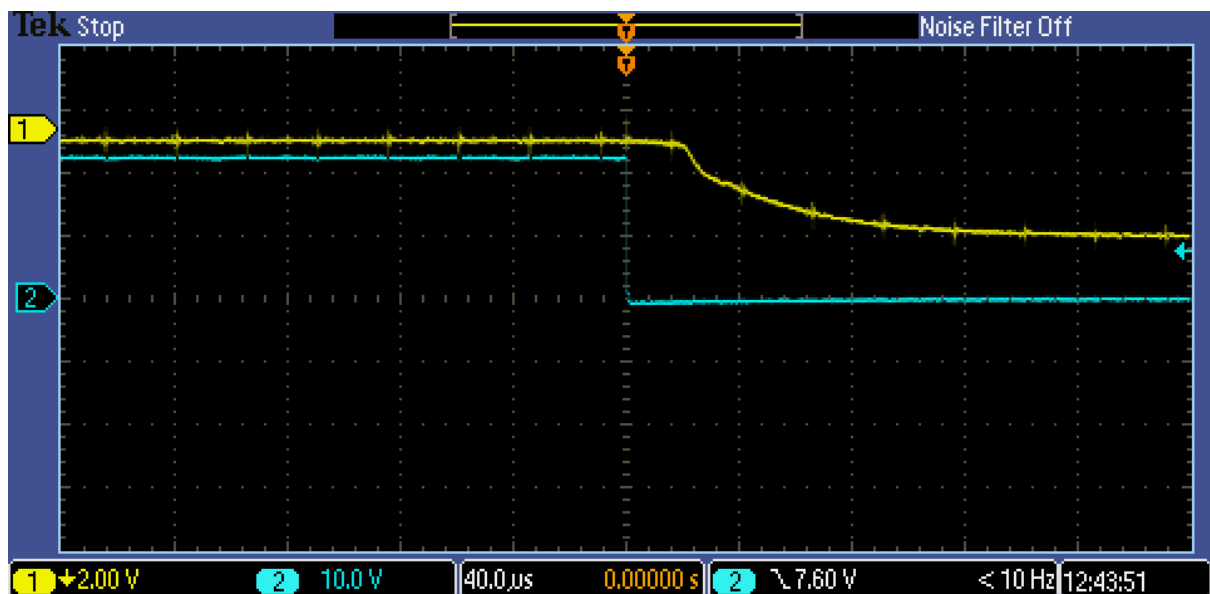
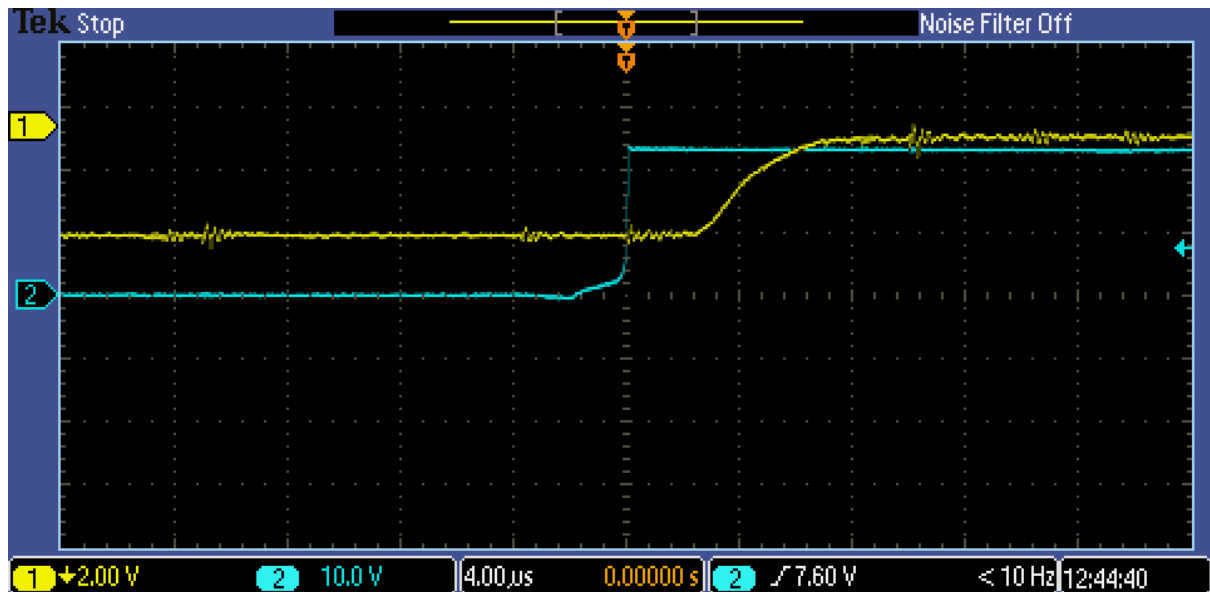
        if (p->in != p->out) {
            USART2->DR = (p->buf[p->out & (TBUF_SIZE - 1)] & 0xFF);
            p->out++;
            tx_restart = 0;
        }
        else {
            tx_restart = 1;
            // A TX megszakítás letiltása, ha nincs mit küldeni
            USART2->CR1 &= ~USART_FLAG_TXE;
        }
    }
}

```

M21 A helyettesítő elektronika oszcilloszkópos vizsgálatainak eredményei

- 1-es csatorna (sárga): a mikrovezérlő felőli oldal
- 2-es csatorna (kék): a felhasználó felőli oldal

Bemenet



Kimenet

