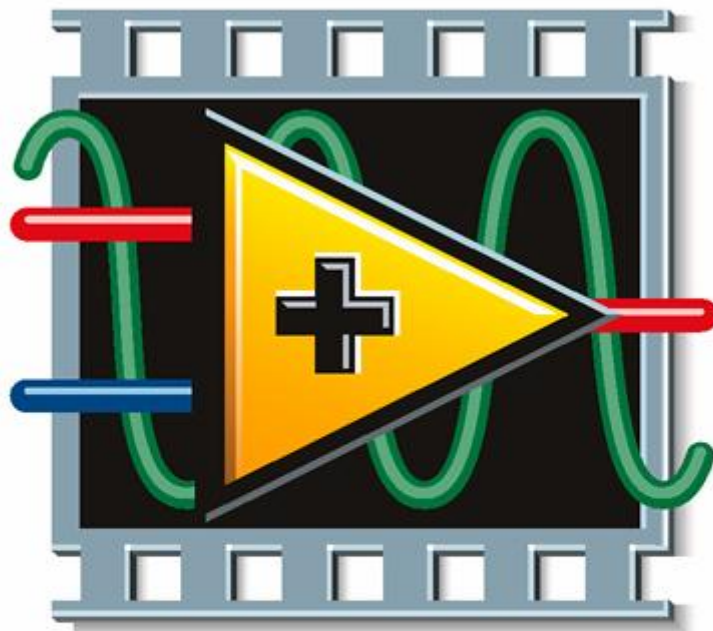


LabVIEW segédlet



NATIONAL INSTRUMENTS
LabVIEW™

Készítette:

Friedl Gergely

Egyetemi tanársegéd


A segédletről

A segédlet a LabVIEW grafikus programozói környezet alapjait mutatja be. Célja a Szabályozási rendszerek című tárgy hallgatói számára egy online elérhető, a tárgy gyakorlatainak tematikájához jól használható oktatási segédlet biztosítása. A segédletben bár említésre kerülnek az egyes részek neveinek magyar megfelelői is, mivel a program angol nyelvű, a könnyebben követhető és értelmezhető jegyzet kialakítása érdekében minden esetben az eredeti angol megfelelő szerepel a dolgozatban. A segédletet igyekszem folyamatosan bővíteni. Amennyiben bármilyen észrevétel vagy javaslat merül fel a művel kapcsolatban, esetleg hiba található benne, azt kérem a friedl@maxwell.sze.hu e-mail címre küldeni.

A LabVIEW-ről általában

A LabVIEW szoftver a **National Instruments** terméke, melynek első verziója 1986-ban jelent meg, és azóta a világ egyik vezető szoftvere mérésautomatizálás terén. A szoftver neve a **L**aboratory **V**irtual **I**nstrument **E**ngineering **W**orkbench rövidítéséből ered. Ahogy arra a nevéből is lehet következtetni, LabVIEW környezetben egy úgynevezett virtuális műszer (*Virtual Instrument*, VI) megvalósítására van lehetőség grafikus programozási környezetben, de természetesen általános célú programok fejlesztésére is felhasználható. A grafikus programozási környezet annyit jelent, hogy a programozás során nem szöveges kód készül, hanem különböző függvényeket/utasításokat reprezentáló elemek összekapcsolásával épül fel a program. LabVIEW környezetben folyamatvezérelt, adatfolyam elvű programozásra van lehetőség, a program végrehajtási sorrendjét az utasítások kapcsolódási rendszere határozza meg.

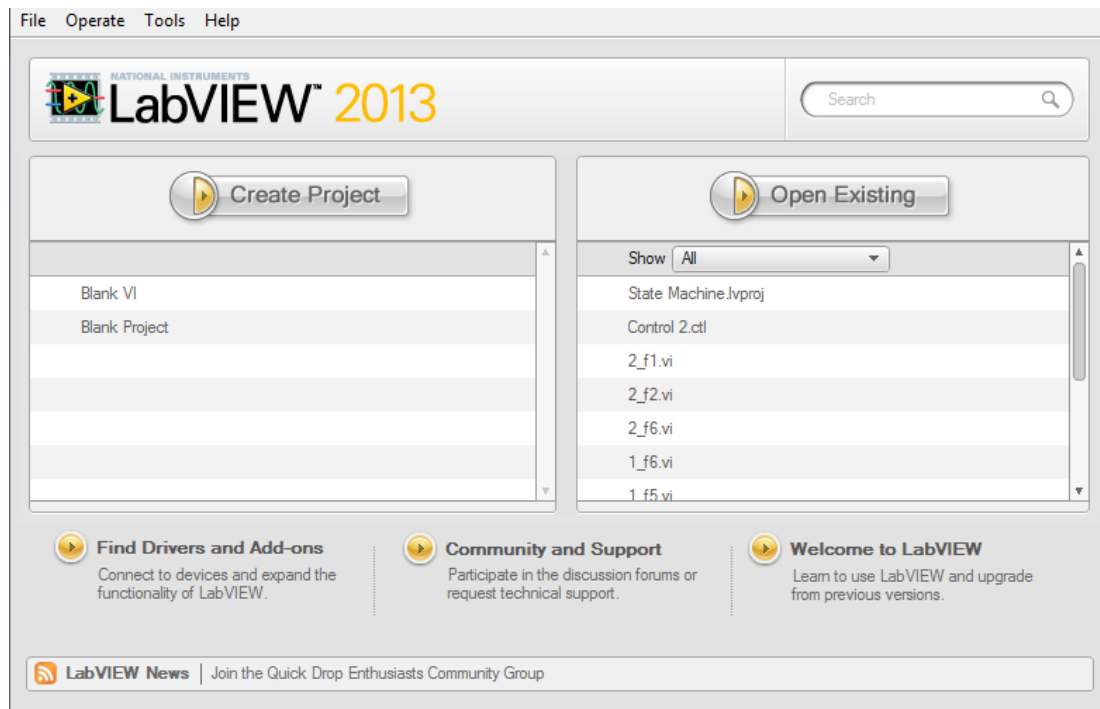
Alapvető információk

LabVIEW elindítása az ikonjára  duplán kattintva lehetséges. Az indítás után megjelenik a program kezdőfelülete (lásd: 1. ábra). A kezdőfelület felső részén egy menü található. A *File* menüben szerepel tulajdonképpen minden, ami a számunkra érdekes részt, a kezdőképernyő közepén is található opciókat tartalmazza. Az *Operate* menü távoli irányításról szól, míg a *Tools* menün keresztül a csatlakoztatott hardverekkel lehet például kommunikálni, illetve azok tesztelése is lehetséges itt. A kezdőfelület alján kiegészítők és *tutorial*-ok érhetőek el. A kezdőfelület közepén található az a rész, amely számunkra szükséges lesz a félév folyamán. Itt az alapvető opciók közül van lehetősége választani a programozónak. Ezek:

- *Create Project*: Új projekt készítése:
 - *Blank VI*: új program készítése.
 - Lehetőség van különböző beépített programtervezési módszerek sablonjainak megnyitására is: *State Machine*, *Finite Measurement...*
- *Open Existing*: Korábban elkészített projekt megnyitása.

Valamely opció kiválasztása után megjelenik a programozói felület, amely már korábban *Virtual Instrument* néven lett megemlítve. A VI két részből áll:

- *Front Panel*: ez a virtuális műszer kezelőfelülete, ide helyezhetők el a különböző vezérlő és kijelző egységek, valamint design elemek;
- *Block Diagram*: itt zajlik a grafikus programozás, ez a virtuális műszer belseje.

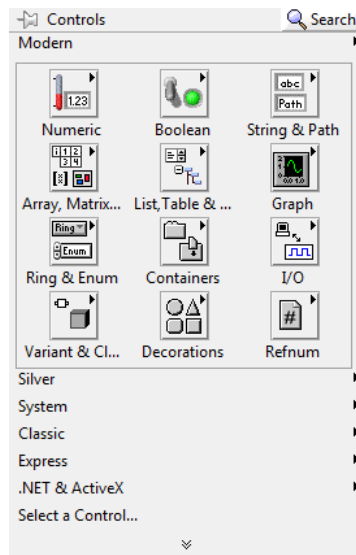


1. ábra: A LabVIEW kezdőfelülete.

Ennél a pontnál érdemes megjegyezni pár hasznos billentyűkombinációt és ezek hatását:

- Ctrl+T: *Front Panel* és *Block Diagram* egymás mellé helyezése, kitöltve a képernyőt.
- Ctrl+E: váltás a *Front Panel* és a *Block Diagram* között.
- Ctrl+N: új üres VI megnyitása.
- Ctrl+Z: utolsó módosítás visszavonása.
- Elemre kattintva Ctrl+C majd Ctrl+V hatására a kurzor helyére másolja kijelölt elemet
 - *Local variable*-t másolva létrejön egy új kontroll/indikátor is.
 - Ciklust/struktúrát/szekvenciát másolva a bennük található elemek is másolásra kerülnek.
 - Több elem kijelölése egérrel a Shift billentyű nyomva tartása mellett lehetséges.

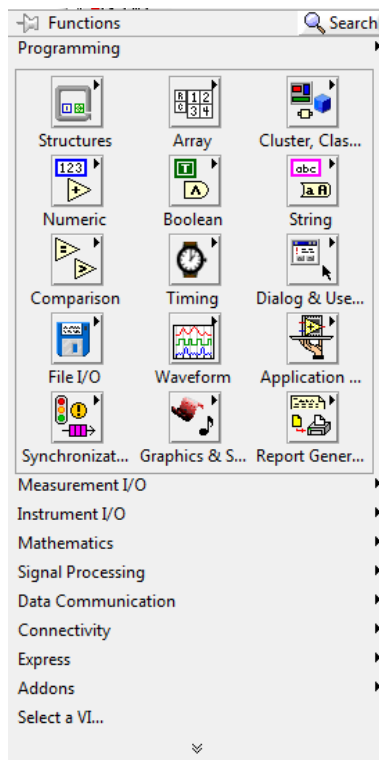
A *Front Panel*-en elhelyezhető vezérlő és kijelző elemeket (illetve szokás még kontrolloknak és indikátoroknak nevezni őket) az úgynevezett *Controls Palette*-n belül lehet böngészni. A *Controls Palette* a *Front Panel*-re jobb egérgombbal kattintva, illetve a *View* → *Controls Palette* elérési útvonalon érhető el. Itt lehetőség van minden adattípusnak megfelelő kontroll, illetve indikátor egység böngészésére célnak megfelelően. Amennyiben a paletta valamely eleme további alegységeket rejt, arra az ikonjának jobb felső sarkában található ► jel hívja fel a figyelmünket. A kurzort az ikon fölé helyezve automatikusan megnyílik egy legördülő ablak, mely ezeket az alegységeket tartalmazza. A kontrollok és indikátorok 4 különböző designban érhetőek el, melyek közül ízlésünk szerint választhatunk: *modern*, *silver*, *system* és *classic*.



2. ábra: A Controls Palette.

Az elhelyezett kontrollok, illetve indikátorok programozói felülete megjelenik a *Block Diagram*-on. Értelmszerűen a kontrollok kimenettel/kimenetekkel, míg az indikátorok bemenettel/bemenetekkel rendelkeznek.

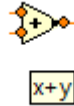
A bemenetek és a kimenetek között valamilyen kapcsolatot hozunk létre a programozás során. Ezt a kapcsolatot a *Block Diagram*-on, a programozói felületen kell megvalósítani különböző függvények és ciklusok segítségével. Ezeket rendszerezve a *Functions Palette*-n lehet egyszerűen elérni. A *Functions Palette* a *Block Diagram*-ra jobb egérgombbal kattintva, illetve a *View* → *Functions Palette* elérési útvonalon érhető el. Itt adattípusnak vagy műveleti jellegnek megfelelően rendszerezve találhatóak a különböző feladatokat ellátó grafikus egységek.



3. ábra: A Functions Palette.

Lehetőség van még egy módon elemeket elhelyezni a *Front Panel*-en, vagy a *Block Diagram*-on, ez pedig az úgynevezett *Quickdrop*.

Az elemeket a termináljaikon keresztül lehet összekötni, adattípusnak megfelelően. A terminálok adattípusát a színük jelöli. Egy elem csak akkor ad ki a kimenetére értéket, ha az összes bemenetére érkezett érték.



4. ábra: Terminálok jelölése.

A félév során a LabVIEW műveleteik csak töredékével tudunk megismerkedni, viszont az alapműveletek ismerete a beszámolómérésen elvárt. Kisarkított példa ugyan, de például amennyiben félév során csak az összeadás műveletet alkalmazzuk, a kivonás műveletének helyes alkalmazása is számonkérhető.

A LabVIEW használata során feltűnhet, hogy a kurzor kinézete különböző pozíciókban automatikusan változik az ellátandó feladatnak megfelelően. Ez azért van, mert az úgynevezett *Tools Palette*-n ez be is van állítva automatikusnak. A *Tools Palette* a *Front Panel*-ről és a *Block Diagram*-ról egyaránt elérhető a *View* → *Tools Palette* elérési útvonalon keresztül, illetve a shift billentyű nyomva tartása mellett jobb egérgomb lenyomása segítségével.





5. ábra: A Tools Palette.


A *Tools Palette* az alábbi kurzorfunkciókat biztosítja:


- : *Automatic Tool Selection*, automata kurzor funkció.
- : *Operate value*, működtető mód, kontrollok értékének megváltoztatására szolgál.
- : *Position/size/select*, objektumok kiválasztására, áthelyezésére, méretének megváltoztatására használható.
- : *Edit Text*, szöveg, illetve címke elhelyezését teszi lehetővé. Szöveg elhelyezésére lehetőség van még dupla kattintással bármely üres területen.
- : *Connect Wire*, huzalozó, elemek összekötésére szolgál a *Block Diagram*-on.
- : *Object Shortcut Menu*, az elemek saját menüjének megnyitására szolgál. Erre lehetőség van egyszerűbben is, elegendő az elemre jobb egérgombbal kattintani.
- : *Scroll Window*, görgető üzemmód, a gördítősáv használata nélkül változtatható az ablak látható pozíciója.
- : *Set/Clear Breakpoint*, töréspont elhelyezésére/törlésére szolgál.
- : *Probe Data*, mérési pont (*Probe*) elhelyezésére szolgál, mely segítségével a program futása közben követhető egy huzalon aktuálisan továbbított adat a *Block*










Diagram-on (indikátor elhelyezésével csak a *Front Panel*-en lenne látható ez az érték).



-  : *Get Color*, a *Front Panel* egy tetszőleges elemének a színének a másolása.
-  : *Set Color*, lehetőség van a *Front Panel* szinte bármely elemének bármely része színének beállítására, illetve a *Block Diagram*-on a háttérszín és a ciklusok háttérszínének beállítására. Megjegyzés: az ikon bal oldali négyzetének színe a *Front Panel*-ről, míg a jobb oldali négyzetének a színe a *Block Diagram*-ról másolt színt mutatja.


A fenti funkciók közül érdemes az automatikus kurzor funkció választót bekapcsolva hagyni, ekkor a működtető, a kiválasztó és a huzalozó módok között automatikusan választ a LabVIEW, így gyors és kényelmes marad a programozás.

A különböző kontrollok, indikátorok, és függvények elhelyezése után könnyen megeshet, hogy a programkód átláthatatlanná válik. Ennek megelőzése érdekében a LabVIEW különböző rendezési opciókat tartalmaz. Ilyen lehetőség az középre, balra, illetve jobbra igazítás, egyenletes távolságra elhelyezés, vagy azonos méretre szabás. 

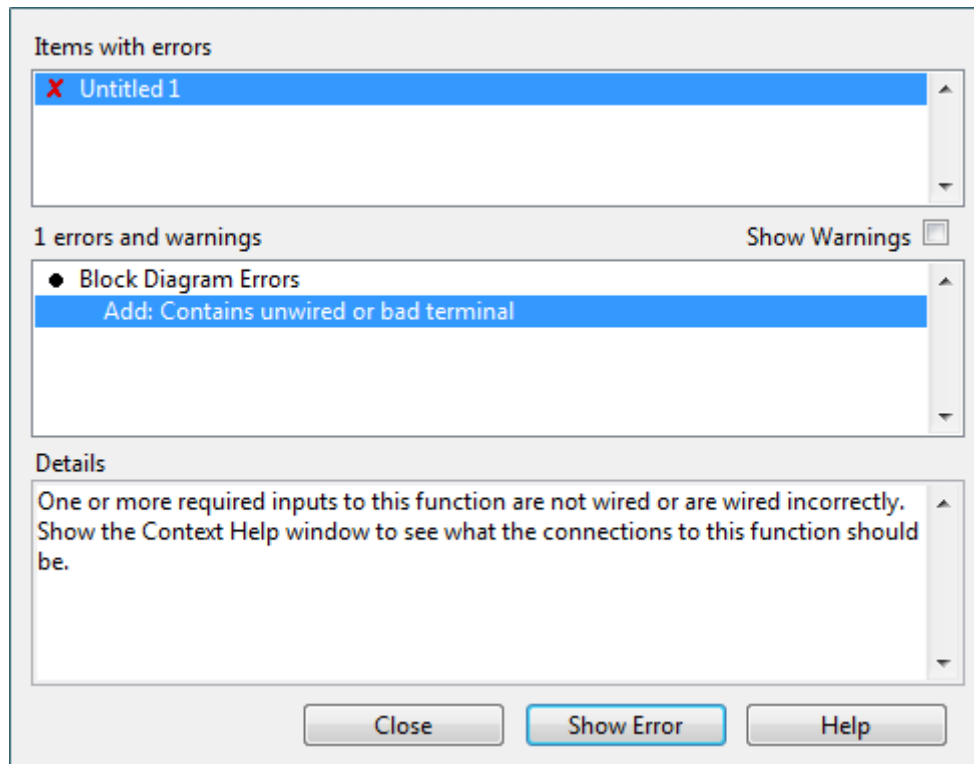
Az elkészített programot már csak le kell futtatni. A LabVIEW különböző futtatási módokat (*Run Mode*) biztosít különböző igényeknek megfelelően. Ezek ikoncsoportja  a *Front Panel*-en és a *Block Diagram*-on is megtalálhatók a bal felső sarokban. A különböző futási módok, és az aktuális módban hozzájuk tartozó ikonok kinézete az alábbi:

- Egyszeri futtatás
 - Futás közben 
 - Hiba esetén 
- Folyamatos, ismétlődő futtatás
 - Futás közben 
 - Hiba esetén 
- Szüneteltetés
 - Szüneteltetés közben 
- Leállítás
 - Futás közben 
 - Szüneteltetés közben 
 - Leállítás 
 - Futás közben 

Még gyakorlott programozók is hibákat ejthetnek egy kód elkészítése során, mely hiba forrását természetesen meg kell találni, és ki kell javítani. Ezt a folyamatot egyszerűen csak *debug*-olásnak hívjuk. Szerencsére LabVIEW környezetben számos hasznos lehetőség van a hibák felderítésére és kiküszöbölésére. Amennyiben a program elindítására szolgáló nyomógomb tört nyíl  formában látható, akkor olyan probléma van a programmal, mely miatt a program egyszeri lefutása sem biztosított. Ilyen eset akkor állhat elő például, ha egy elem egyik működéséhez elengedhetetlen bemenete nem lett bekötve sehova sem, vagy például ha egy kimenet és egy bemenet adattípusa egy vezetéken összeegyeztethetetlen. (Megjegyzés: ezt az összeegyeztethetlenséget a LabVIEW a következőként jelöli: .)



Ilyen esetekben a tört nyílra  kattintva megjelenik az *Error list* felugró ablak (lásd: 5. ábra), amely az ilyen jellegű hibák listáját tartalmazza. Egy listán szereplő elemre és a *Show Error* opcióra kattintva, vagy a listaelemre duplán kattintva a program


megjeleníti a *Block Diagram*-on az adott hiba helyét. Az 5. ábrán látható hiba például egy szabadon hagyott összeadás művelet eredménye.

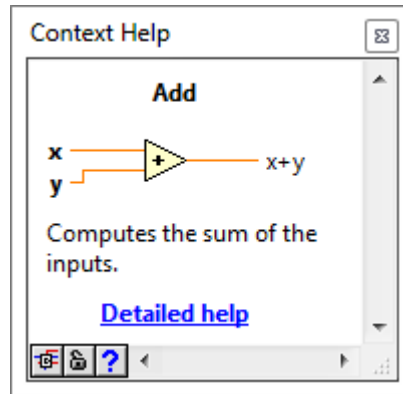


6. ábra: Az *Error list*.

Előfordulhat viszont olyan eset is, amikor a program hiba nélkül lefut, az eredmény mégsem az, amit a programozó várna a programtól. Ez akkor következik be, ha a kijelölt feladat elvégzéséhez a kód nem megfelelő. Ezt a hibát a LabVIEW természetesen nem tudja megtalálni, ugyanis az nem fogja tudni, valójában mit is szeretünk volna megvalósítani, ezeket a programozónak kell felkutatni. Természetesen az ilyen hibák felderítésére is több különböző módszerrel és eszközzel segítséget nyújt számunkra a program. Ezek a módszerek az alábbiak:

- *Highlight Execution*:
 - *Front Panel*-en futtatás előtt és közben is a  ikonra kattintva. 
 - A program futása során az adatfolyam végigkövethetővé válik, mivel a futási sebesség ember számára is követhetőre lassul, és a huzalokon továbbhaladó adat értékét folyamatosan közli velünk a program.
 - Nagyobb programok esetén érdemes csak akkor elindítani, ha a program futását a hiba vélt helyén egy *Breakpoint*-tal szüneteltetjük.
- *Breakpoint*, töréspont:
 - A program ehhez a ponthoz érve szünetelteti a futását.
 - Bármely elemre jobb egérgombbal kattintva: *Breakpoint* → *Set breakpoint*.
 - A *Breakpoint*-ként beállított elem piros színnel van jelölve a *Block Diagram*-on.
 - *Breakpoint*-ok törlésére a *View* → *Breakpoint Manager* elérési úton, vagy a
- *Probe*
 - Elérése a *Tools Palette*-ről lehetséges.
 - Bármely huzalra elhelyezve futás közben kiíródik a huzal éppen keresztülhaladó érték a felugró *Probe Watch Window*-on.

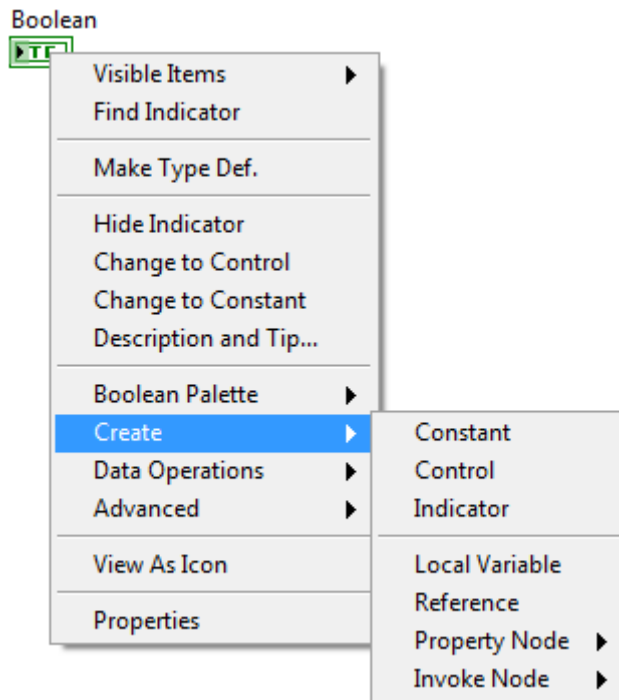
Egy program készítése során előfordulhat, hogy a programozó nem feltétlen tudja eldönteni, valójában egy adott elem pontosan hogyan is működik, illetve melyik bemenete milyen funkciókat lát el. Erre szolgál a LabVIEW speciális segédeszköze, a *Context help*, mely a *Front Panel*-en és a *Block Diagram*-on egyaránt elérhető. Az ablak jobb felső sarkában a sárga kérdőjelre  kattintva, majd a kurzort ráhúzva egy elemre bővebb információ tudható meg róla. A 6. ábra például az összeadás művelet *Context help*-jét mutatja.



7. ábra: A *Context help*.

A *Controls Palette* és a *Functions Palette* minden eleme rendelkezik saját menüvel (lásd: 8. ábra). Itt egy elemet teljes mértékben tesztre lehet szabni. Lehetőség van például az elemekhez címkét rendelni, egy elemet ki lehet cserélni egy másikra, vagy megtekinthetők az alapvető tulajdonságaik, stb. A fentieknél viszont vannak lényegesen hasznosabb menüpontok is. Ezek közül néhány, mely a félév során hasznunkra válhat:

- *Data Operations* (adat műveletek): lehetőség van alapértelmezett érték beállítására.
- *Representation* (ábrázolás): numerikus kontroll, illetve indikátor számbábrázolási módjának beállítása itt lehetséges.
- *Local Variable* (helyi változó): egy indikátor által jelzett érték a programon belül bárholnan beállítható, illetve egy kontroll több helyen is kimenetként szolgálhat. Segítségével esztétikusabbá is tehető a program, mivel rengeteg huzalozástól megszabadíthat.
- *Property Node* (tulajdonság csomópont): program futása közben bármely kontroll és indikátor legtöbb tulajdonsága beállítható, kijelezhető. Használatával alapértelmezett érték, pozíció, méret, stb. beállítására van lehetőség.



8. ábra: Egy logikai típusú indikátor saját menüje.

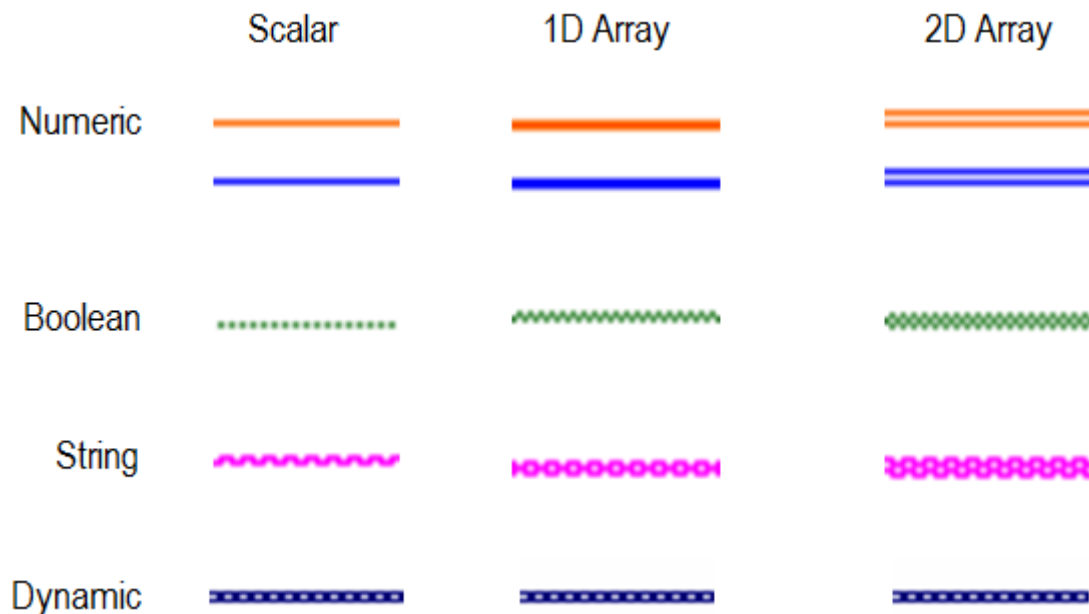
Adattípusok

LabVIEW környezetben különböző adattípusok kezelésére van lehetőség. A típusokat a 9. ábra foglalja össze.

Control	Indicator		
		Single-precision, floating-point numeric	Egyszeres pontosságú lebegőpontos (-∞ +∞)
		Double-precision, floating-point numeric	Dupla pontosságú lebegőpontos (-∞ +∞)
		Extended-precision, floating-point numeric	Kiterjesztett pontosságú lebegőpontos (-∞ +∞)
		8-bit signed integer numeric	Előjeles valós (integer) (-128 +127)
		16-bit signed integer numeric	Előjeles valós (integer) (-32768 +32767)
		32-bit signed integer numeric	Előjeles valós (integer) (-2147483648 +2147483647)
		8-bit unsigned integer numeric	Valós (integer) (0 +255)
		16-bit unsigned integer numeric	Valós (integer) (0 +65535)
		32-bit unsigned integer numeric	Valós (integer) (0 +4294967295)
		Enumerated type	Felsorolás típus
		Boolean	Igaz vagy hamis érték
		String	Szöveges változó, karakter tömbhöz hasonló
		Array	Tömb (a szín a tömb típusának függvénye)
		Cluster	Klaszter (lehet más színű is)
		Path	Elérési út (fájlhoz vagy mappához)
		Waveform	Analóg jel
		Digital waveform	Digitális jel

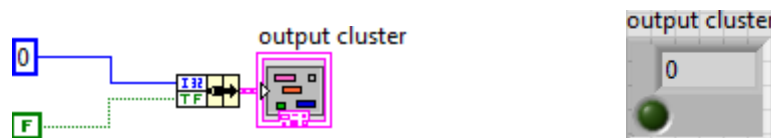
9. ábra: Adattípusok LabVIEW környezetben.

A huzaltípusok az általuk szállított információ adattípusának megfelelőek, a színük a típus, a kinézetük pedig attól függ, hogy skalár érték vagy tömb a szállított adat. A huzalok kinézete a különböző esetekben a 10. ábrán látható.



10. ábra: A huzalok kinézete adattípustól függően.

Az egyes grafikus egységek kimenetei semmilyen esetben sem köthetők össze. Két azonos típusú huzal által szállított információ egy huzalra helyezhető, ehhez a két huzal tartalmából tömböt kell képezni. Ezt az *Functions Palette* → *Array* → *Build Array* függvénnyel tehetjük meg. Különböző adattípusú huzalokat csak *cluster*-ként lehet összekötni, ezt a *Functions Palette* → *Cluster* → *Bundle* függvénnyel lehet megvalósítani.

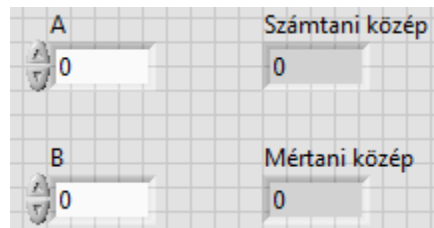


11. ábra: *Cluster* készítése a *Block Diagram*-on , és kinézete a *Front Panel*-en.

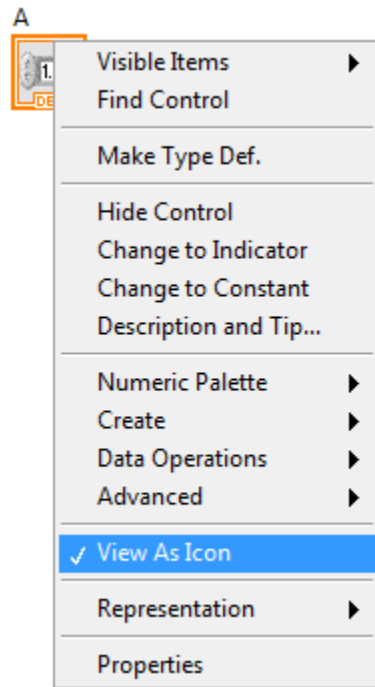
SubVI készítése

A program készítése során ügyeljünk a rendezettségre és átláthatóságra, ugyanis bonyolultabb problémák megoldása során a program meglehetősen nagy méretű lehet, és egyszerűen nem fér be a *Block Diagram* ablakába. Ezzel nincs is gond, mivel a *Block Diagram* méretét nem korlátozza az ablak mérete, a csúszkákkal a program tetszőleges részére navigálhatunk. Ez viszont jelentősen lelassíthatja a program készítését, LabVIEW-ban pedig nincs lehetőség közelítésre / távolításra (nincs ZOOM). Lehetőség van viszont úgynevezett *SubVI*-ok készítésére, mely segítségével a program egy része egyetlen grafikus elemmel helyettesíthető. Ez különösen hasznos, ha egy műveletet a programon belül többször kell elvégezni. Ennek bemutatása egy egyszerű, két szám számítani és mértani közepét számító programon keresztül történik. Először is *Front Panel*-en két numerikus kontroll, valamint két numerikus indikátor elhelyezésére van szükség (lásd: 12. ábra). A két kontroll a két bekért számot, a két indikátor pedig a mértani, illetve a számtani közepet fogja reprezentálni. Elhelyezésük után megjelenik a programozható grafikus felületük a *Block Diagram*-on. A kontrollok és indikátorok

ikonjának mérete csökkenthető. Ehhez a kontrollra/indikátorra kell jobb egérgombbal kattintani, majd a *View as icon* opciót kell kikapcsolni (lásd: 13. ábra).

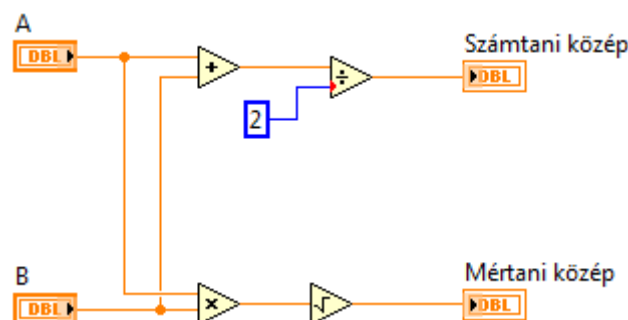


12. ábra: Az elhelyezett kontrollok és indikátorok a *Front Panel*-en.



13. ábra: Egy kontroll ikonként való megtekintésének kikapcsolása.



Legyen a két bemenet *A* illetve *B*. A számtani közép az $\frac{(A+B)}{2}$, míg a mértani közép a $\sqrt{A \cdot B}$ összefüggés segítségével számítható. A fenti egyszerű program megvalósításához szükség van egy összeadás műveletre, egy osztás műveletre, egy numerikus konstans értékre, egy szorzás műveletre, valamint egy gyökvonás műveletre. Ezek mind megtalálhatók a *Functions Palette* → *Numeric* elérési útvonalon. Az elkészült program a 14. ábrán látható.





14. ábra: A számtani és mértani közép számítására képes program.

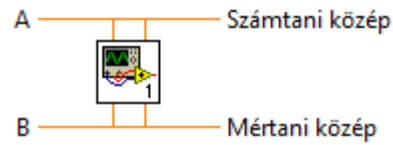
Ebből a programból fogunk most egy *SubVI*-t képezni. Gyakorlatban természetesen sokkal bonyolultabb és terjedelmesebb feladatmegvalósítást szokás *SubVI*-ként leképezni, itt most az

elv ismertetése a cél. A *Front Panel* jobb felső sarkában található az úgynevezett

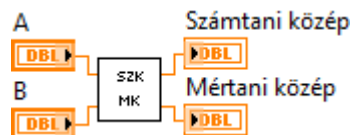
Icon Connector  illetve maga a még el nem készített *SubVI* ikonja . A kurzort az *Icon Connector*-ra helyezve az automatikusan átváltozik huzalozó módba. Ez annyit jelent, hogy az *Icon Connector* termináljait össze lehet kötni a program kontrolljaival/indikátoraival. Bár az sem lenne probléma, ha felhasználatlanul hagynánk terminálokat, érdemes olyan mintázatu terminálkiosztást választani, ami a megvalósítandó *SubVI*-hoz a legjobban illik. A fenti programnak két bemenete és két kimenete van. Az *Icon Connector*-ra jobb egérgombbal kattintva a *Pattern* legördülő menüben lehetőség van a terminálkiosztás megváltoztatására. A

jelleghöz legjobban illő, 2x2-es mintázatot  érdemes választani a *SubVI* elkészítéséhez. A terminálokhoz ezután kontrollt vagy indikátort kell rendelni. Egy egyszerűen úgy történí, hogy ki kell választani egy terminált az *Icon Connector*-on, arra bal egérgombbal kattintani, majd bal egérgombbal kell kattintani ezután az adott terminálhoz rendelni kívánt kontrollt vagy indikátort. Mivel a LabVIEW összes függvénye „balról jobbra halad”, érdemes a kontrollokat a bal oldali, míg az indikátorokat a jobb oldali terminálokhoz csatlakoztatni. Ez ebben az esetben azért is fontos, ugyanis ha így cselekszünk, akkor normál vízszintes

elrendezésben lesznek az elkészült *SubVI* termináljai , míg ellenkező esetben viszont függőleges terminálkiosztással készül el a *SubVI*



és a *Context help*-ben is butaság lesz látható. Az *Icon Connector*-on minden terminál színe elváltozik, amelyhez már valami rendelve lett. Az elkészülő *SubVI* ikonját szerkeszteni lehet az ikonra jobb egérgombbal kattintva az *Edit Icon* opciót választva. Itt több különböző rajzeszköz segítségével kedvünkre szerkeszthetjük az általunk készített grafikus egység kinézetét. Ezt elvégezve el lehet menteni az elkészült *VI*-t. Egy új, üres *VI*-t nyitva (Ctrl+N) le is lehet tesztelni az elkészült *SubVI*-t. Az új, már *SubVI*-t használó program a 15. ábrán látható.



15. ábra: Program *SubVI* használatával egyszerűsítve.

Kontrollok, indikátorok:

Már többször szó esett a kontrollokról, valamint az indikátorokról, azok különböző tulajdonságaikról. Ebben az alfejezetben listába szedve megtalálhatók a főbb típusok, s azok pár fontos és hasznos tulajdonsága. Minden kontrollhoz tartozik azonos típusú indikátor. Minden elhelyezett kontroll/indikátor átalakítható indikátorrá/kontrollá az elemre jobb egérgombbal kattintva a *Change to Indicator/Control* opció segítségével. Ha csak magunk helyezünk el a *Front Panel*-en indikátort, az nem feltétlenül lesz azonos azzal az adattípussal. Erre példa, hogy az alapértelmezetten elhelyezett numerikus indikátor 64 biten ábrázolt valós *double* érték, viszont ha a bemenetere komplex szám kerül, az indikátor csak a valós részét jeleníti meg. Viszont ha a kontrollok, és a hozzájuk tartozó műveletek elhelyezése és huzalozása után a kijelezni kívánt kimenetre jobb egérgombbal kattintva a *Create* → *Indicator* elérési útvonal szerint helyezük el az indikátort, akkor a kimenet

számábrázolásának megfelelő indikátor jelenik meg. A következőkben az egyes kontrollok és indikátorok kerülnek felsorolásra, a legfontosabb tulajdonságaikat megemlítve.

- Numerikus:

- Kontrollok, vezérlők:

- *Numeric control*: számérték beállítására van lehetőség.



- A tizedes tört egész és tört részét tizedesvesszővel kell elválasztani.

- *Slide*: csúszka.



- *Property Node* segítségével, vagy közvetlenül, az értékre dupla kattintással állíthatóak a határai.
 - Többféle elrendezésű és kinézetű választható.

- *Knob*: tekerőgomb.



- *Property Node* segítségével, vagy közvetlenül, az értékre dupla kattintással állíthatóak a határai.

- Indikátorok, kijelzők:

- Minden numerikus kontrollhoz tartozik azonos típusú és jellegű indikátor.

- Logikai:

- Kontrollok, nyomógombok:

- Kizárólag két állapot.
 - Különböző egyállású/kétállású módok beállítására van lehetőség az elem menüjén belül a *Mechanical Action* fül alatt.
 - Kétállású kapcsolók:

- *Switch When Pressed*: állapotváltás megnyomáskor.



- *Switch When Released*: állapotváltás felengedéskor.



- *Switch Until Released*: állapotváltás a tartás idejére.



- Egyállású kapcsolók:

- *Latch When Pressed*: állapotváltás megnyomáskor, majd állapot tartása kiolvasásig.






- *Latch When Released*: állapotváltás felengedéskor, majd állapot tartása kiolvasásig.



- *Latch Until Released*: állapotváltás megnyomáskor, majd állapot tartása a felengedés utáni kiolvasásig.

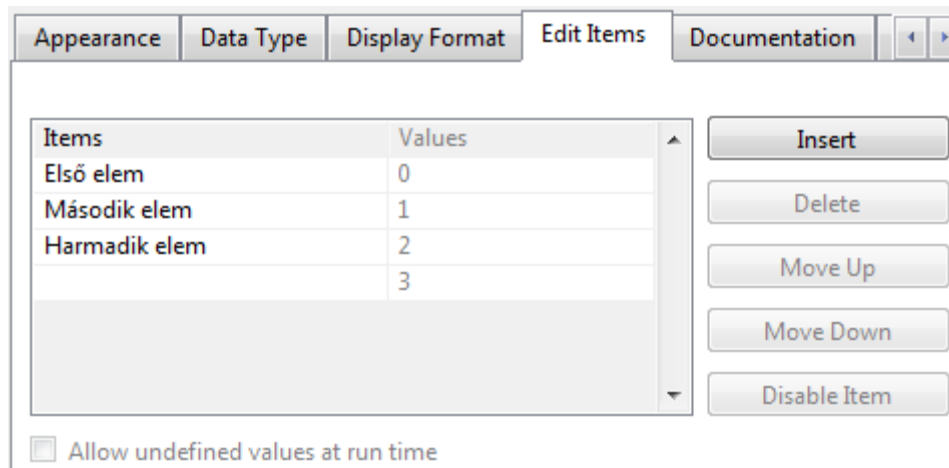


- Speciális kapcsológombok:

- *OK* , *Stop* , *Cancel*  gombok.
 - Funkciójukat tekintve tökéletesen megegyeznek, mind egyállású, *Latch When Released* módban üzemelő kapcsoló.
 - A gombok felirata változtatható az elemek saját menüjén belül a *Properties* opciót választva.

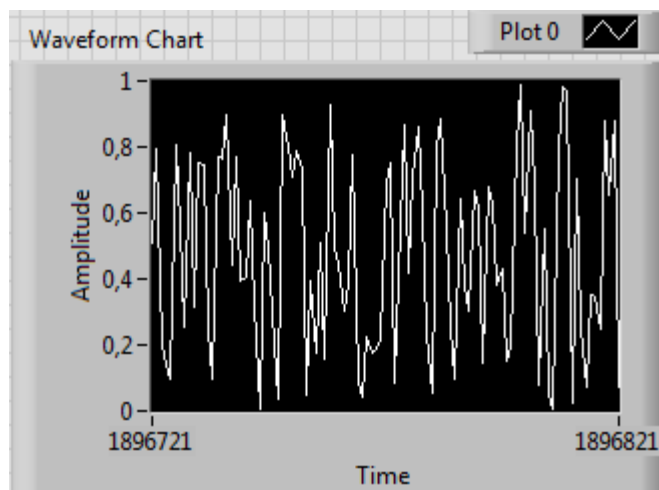
- Szöveges:

- A *Controls Palette*-n található a *String & Path* menüpont alatt.
- Kontrollal szövegbevitelre, indikátorral kiíratásra van lehetőség.
- A *Combo Box* egy szöveges felsorolási lehetőséget biztosít, lista szerkesztése a saját *Properties* menüjén belül az *Edit Items* fül alatt lehetséges, majd program futása közben egy legördülő menüből választható ki a kimenetre adni kívánt listaelem.
- Ide tartozik még a *File Path*, tehát az egyes fájlok elérési útvonala.
- Felsorolás:
 - *Enumerated Control* (ez a *Controls Palette*-n csak *Enum*-ként található meg), illetve *Ring Control* tartozik ide.
 - Számozott listák készítésére van lehetőség.
 - A kimeneten a tartalom, és a sorszáma is megjelenik.
 - Kimenetén *Create Indicator* parancs által létrehozott indikátoron a tartalma íródik ki, numerikus indikátorra kötés esetén pedig a sorszáma.
 - Szövegesen megkülönböztetett állapotok hozhatók létre, többállapotú, felhasználó által irányított *Case* struktúra használatához ideális.



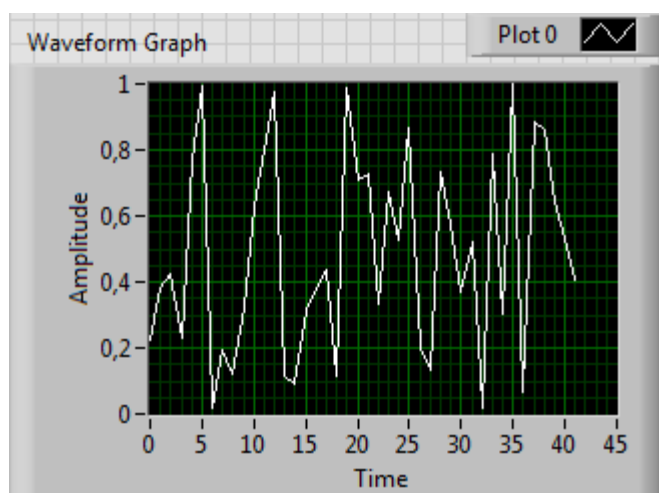
16. ábra: Az *Enum Control* elemeinek beállítása.

- *Waveform chart*
 - Egymás után érkező numerikus adatok egy virtuális kijelzőn történő megjelenítésére van lehetőség.
 - Két egymás után érkező értéket egyenes vonallal köt össze, így az adathalmaz folytonosnak látszik.







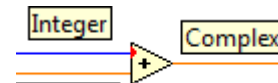
17. ábra: A *Waveform Chart* kinézete.

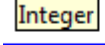
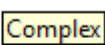
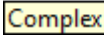




- *Waveform graph*
 - Egy numerikus tömb értékeinek ábrázolását teszi lehetővé virtuális kijelzőn.






18. ábra: A *Waveform Graph* kinézete.

A programok felépítése során számtalanszor szükséges különböző matematikai műveletek használata. A sokszor elegendő a 4 alpművelet használata, viszont bonyolultabb feladatok esetén elkerülhetetlen egyéb műveletek használata is. A matematikai műveletek a *Functions Palette* → *Mathematics* útvonalon érhetők el. Az alpműveletek (összeadás , kivonás , szorzás , osztás ) két bemenettel és egy kimenettel rendelkeznek. A kimenetük




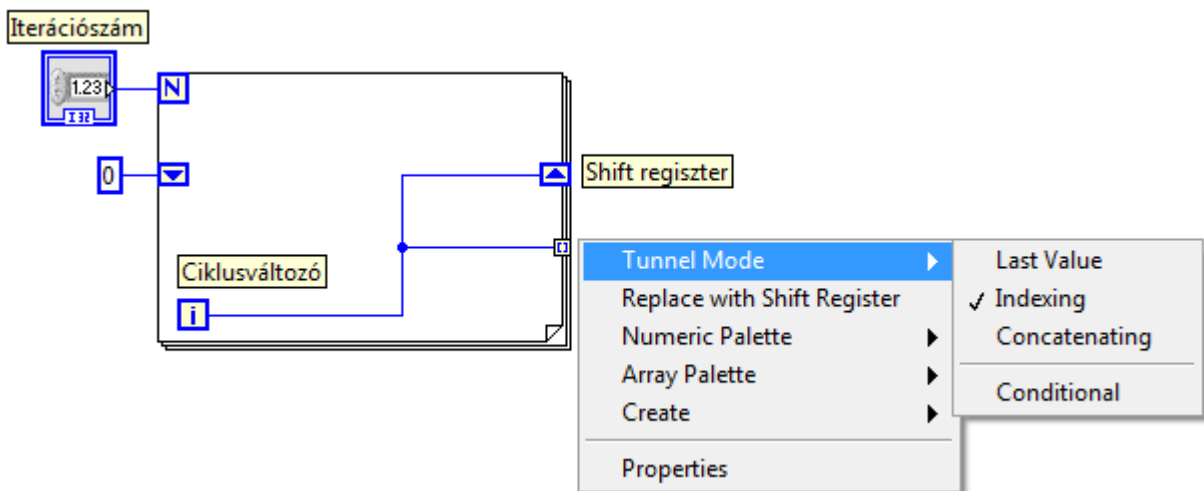
számbábrázolási módját a bemenetek határozzák meg, például:  +  = . Az adattípusbeli eltérést a LabVIEW a bemeneti terminálon piros színnel jelöli. Gyakran szükség lehet egyéb műveletek használatára is, mint a maradékos osztás , eggyel növelés  /csökkentés  (*increment/decrement*), vagy például abszolút érték képzés , stb. A *Mathematics* legördülő menűn belül a félév során csak az *Elementary* család ismerete szükséges. Itt trigonometriai, exponenciális és logaritmikus operátorok gyűjteménye található. A matematikai és fizikai állandókat a *Functions Palette* → *Numeric* → *Math & Scientific Constants* útvonalon keresztül lehet megtalálni.

String műveletek:


- String felépítés részekből , string bontása/egy részletének kinyerése 
- Numerikus - string konverziók (pl: egy számítás eredményének kiírása szövegbe fűzve) 
- Keresés stringben

A következőkben a különböző struktúrák és ciklusok tárgyalása következik, pár alapvető tulajdonságuk ismertetésével. Ezek a *Functions Palette*-n a *Structures* menüpont alatt található.

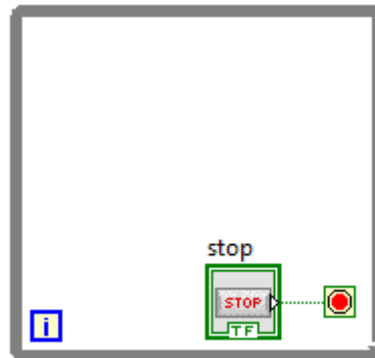
- *For* ciklus :
 - $i = 0$ -tól $N-1$ -ig fut, a ciklus így N -szer fut le.
 - ha N nem egész, egészre kerekített értékével számol a program.
 - N -et kötelező megadni!
 - Megadható feltétel terminál: *STOP* gomb.
 - A ciklusnak adható név (könnyebb azonosítás): Jobb klikk → *Visible items* → *Subdiagram Label*.
 - *Shift register* helyezhető el a ciklus szélére jobb egérgombbal kattintva a menüből az *Add Shift Register* opció választásával.
 - Ha nem csak az előző ciklusbeli értékre vagyunk kíváncsi, akkor a *shift registert* „megnyújtva” korábbi állapotok nyerhetők ki (pl: fibonacci sor).
 - Tömbkészítés automatikus indexeléssel lehetséges.
 - A ciklus során valamely változó értéket a ciklus rajzjelének széléhez huzalozva az érték a ciklus lefutása után kinyerhető.
 - Amennyiben minden egyes iterációban felvett érték érdekes, a *Tunnel*-re (alagút) jobb egérgombbal kattintva beállítható a *Tunnel Mode*.




19. ábra: *For* ciklus kinézete shift regiszterrel, valamint a *Tunnel Mode* beállítása.

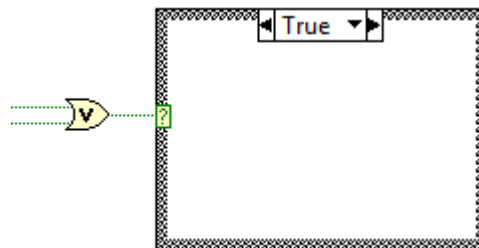
- *While* ciklus :
 - Egy feltétel beteljesedéséig fut.
 - Beállítható, hogy a leállási feltétel igaz, vagy hamis állapotra állítja le a ciklust.

- Van saját ciklusváltozója, mint a *For* ciklusnak, és ugyanúgy 0 kezdeti értékkel.
- *Shift Register*, valamint *Tunnel Mode* kezelése a *For* ciklusnál tárgyaltakkal megegyező módon történik.






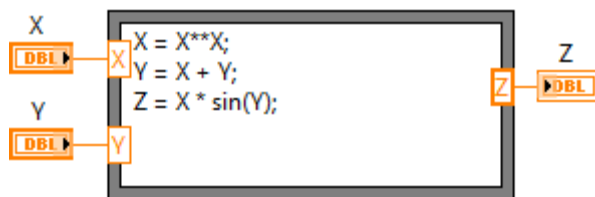
20. ábra: A *While* ciklus kinézete.

- *Case* struktúra :
 - Feltételek, elágazások kezelése.
 - Más programnyelvek *if* struktúrájához hasonló működés.
 - Alapvetően két állapot: igaz-hamis
 - Lehetőség van több állapot megkülönböztetésére. Ezek számmal, vagy *String*-gel különböztethetők meg (pl: *Enumerated Control* segítségével).



21. ábra: A *Case* struktúra kinézete.

- *Stacked sequence*  / *Flat sequence* :
 - Műveletek sorba rendezésére szolgálnak.
 - *Frame*-eket lehet egymás után építeni.
 - Lehetőség van egy előző *frame*-ben kiértékelt eredmények egy későbbi *frame*-ben történő használatára.
 - Segítségükkel a program jól strukturálható.
 - Használatuk főképp olyan feladatok megoldásakor szükséges, amikor a cél nem a program mielőbbi lefutása, hanem az egyes állapotok előre meghatározott ideig tartása. (pl: jelzőlámpa)
- *Formula node* :
 - Egyenletek, összefüggések implementálása írott kóddal.
 - Változó be- és kimenetek definiálhatók.
 - A numerikus műveleti egységek helyettesíthetők vele.
 - A sorokat pontosvessző zárja.
 - Beépített függvények is meghívhatók itt. (pl: szinusz, koszinusz, lásd: *Context help*)




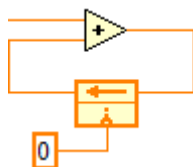
22. ábra: Egy tetszőleges művelet *Formula Node* segítségével.



Összehasonlító műveletek:

- *Functions Palette* → *Comparison* útvonalon érhetők el.
- Különböző események megkülönböztetésére szolgál (pl: más művelet lehet szükséges, ha egy érték 0-10 között van, és más, ha 10-20 között).
- *Select*: igaz és hamis bemenet esetén más-más ág értékét engedi tovább, helyettesíthet egy *Case* struktúrát egyszerűbb esetben.

Visszacsatolás:



- *Feedback Node* :
 - A *Functions Palette* → *Structures* → *Feedback Node* útvonalon érhető el, de automatikusan is létrejön, ha egy kimenetet egy, az adatfolyam korábbi bemenetére kötünk.






- Egyszerű visszacsatolás, a programon belül szinte bárhol elhelyezhető.
- A kimeneti értéket visszajuttatja a bemenetre, hogy az egy következő ciklusban újra felhasználható legyen.
- Színe a számábrázolási módnak és az adattípusnak megfelelően alakul.
- Az *Initializer Terminal* segítségével a *Feedback Node* kezdeti értékének beállítására van lehetőség. A terminált üresen hagyva mindig a legutolsó értéket tekinti alapértelmezettnek. Ez egy program többszöri lefuttatása esetén gondot jelent, mivel a korábbi futtatás eredménye lesz az alapértelmezett, így a program futása nem várt kimenetet eredményezhet.
- *Shift Register*  :
 - Ciklusokban iterációnkénti visszacsatolásra ad lehetőséget.
 - Létrehozása a ciklus peremére jobb egérgombbal kattintással, majd az *Add Shift Register* opció választásával történik.
 - A *Shift Register* alagutat (*Tunnel*) képez a ciklus és a külvilág között, információ be- és kicsatolására van lehetőség.
 - Színe a számábrázolási módnak és az adattípusnak megfelelően alakul.
 - A bal oldali részének bemeneti terminálján kezdeti érték beállítására van lehetőség.

Időzítés:

- A program futásának várakoztatását teszi lehetővé tetszőleges helyen.
- Fontos, ha a feladat jellege nem a gyorsaságot követeli meg, hanem egy adott állapot tartását bizonyos ideig (pl: jelzőlámpa)

- A *Get Date/Time String*  illetve a *Get Date/Time in Seconds*  függvény segítségével az aktuális dátum és időpont megjelenítésére van lehetőség. Ez például egy mérési jegyzőkönyv időbélyegének elhelyezésekor kifejezetten hasznos lehet.


Dialógus:

- Felhasználói beavatkozást igénylő feladatok esetétén dialógus ablak elhelyezésére van lehetőség.
- *One button dialog*: egy gombos dialógus, felhasználó engedélyezi a program további futását 
- *Two button dialog*: két gombos dialógus, igen - nem 
- *Tree button dialog*: három gombos dialógus, igen - nem - mégsem 



PÉLDÁK:

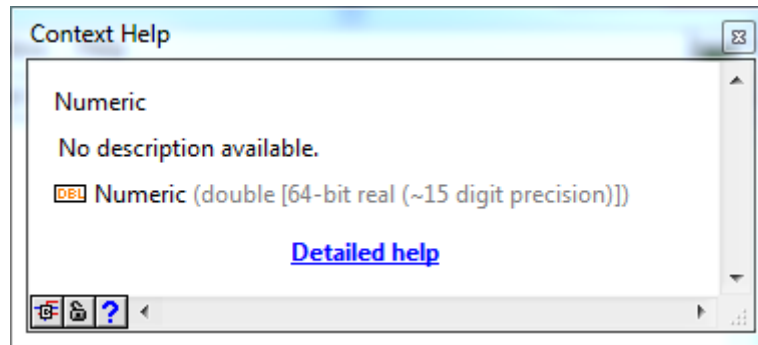
0. Ismerkedés a LabVIEW programmal

0.0. LabVIEW elindítása

Indítsuk el a LabVIEW programot az asztalon található ikonjára  duplán kattintva! Ekkor megjelenik az 1. ábrán látható kezdőfelület. Itt a bal oldali listából válasszuk ki a *Blank VI* lehetőséget! Megjelenik a *Front Panel* valamint a *Block Diagram*. Üssük le a Ctrl+T billentyűkombinációt az ablakok egymás mellé rendezéséhez! Alapértelmezetten a *Tools Palette*-n az automatikus kurzor üzemmód van beállítva, illetve alapértelmezetten megnyílik a *Context help*.

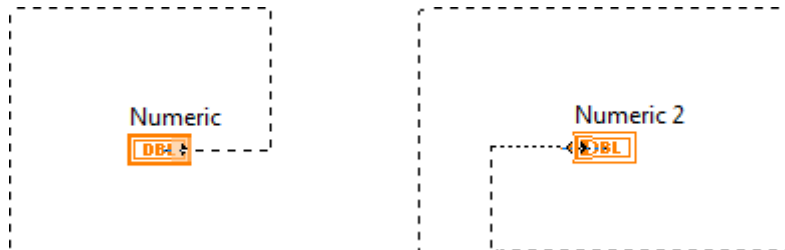
0.1. Alapvető LabVIEW funkciók megismerése

A LabVIEW legalapvetőbb építőelemei a numerikus kontrollok és indikátorok. Precizitástól függően tetszőleges számértéket adhatunk be a programnak, és írathatunk ki, természetesen az éppen aktuális számábrázolás határain belül. Helyezzünk el a *Front Panel*-en egy numerikus kontrollt és egy numerikus indikátort! Látható, hogy mindkét elem programozható felülete megjelent a *Block Diagram*-on. Kapcsoljuk ki a programozható felület ikonként történő megjelenítését (jobb egérgomb → *View as Icon* kikapcsolása)! Az elemek méretének csökkentése a program könnyebb átláthatóságát segíti elő. Az elemek egymáshoz képesti elhelyezkedése a *Front Panel*-en és a *Block Diagram*-on is beállítható, rendezhető. A két elhelyezett elemet kijelölve, majd az ablak tetején található három, rendezésre szolgáló legördülő ablak  közül a bal oldalt legördítve, és a felső sor valamely elemét választva a két elem egymáshoz képest vízszintes elhelyezkedésű lesz. A *Context help*-et bekapcsolva a sárga kérdőjellel , és a kurzort az elem felé helyezve látható, hogy alapértelmezetten 64 biten ábrázolt valós szám bevitelére és kiírására alkalmas elemeket lehet elhelyezni.





23. ábra: Az elhelyezett numerikus kontroll *Context help* képernyője.

Huzalozzuk össze a két elemet a *Block Diagram*-on. Vigyük a kurzort a kontroll termináljához (ez a kontroll jobb oldalán található)! Ekkor a kurzor megváltozik, huzalozó üzemmódba vált automatikusan. Kattintsunk a kimenetre, majd kattintsunk az indikátor bemeneti termináljára (ez pedig az indikátor bal oldalán található)! A LabVIEW ekkor összeköti a két elemet, a 10. ábrán láthatóaknak megfelelően egy vékony, narancssárga huzallal. A huzal csak vízszintesen és függőlegesen haladhat, illetve tetszőleges számú derékszögű kanyarból állhat. Amennyiben egyszerűen csak a kezdő-, illetve végpontra kattintunk huzalozáskor, a LabVIEW automatikusan generál egy optimális útvonalat. Az optimális útvonalak alkalmazása viszont nem feltétlenül a legegztétikusabb, legáttekinthetőbb megoldást eredményezi. Huzalozás közben a végpontra kattintás előtt a kattintási pontokat a LabVIEW fix, huzalnak érintendő pontnak tekinti. A huzalok keresztezhetik egymást. Huzalozás közben a tervezett huzalmenet átmenetileg szaggatott vonalként látható.



24. ábra: Huzalozás tetszőleges útvonalon.

Adjunk meg a *Front Panel*-en a kontrollnak egy tetszőleges, nem egész számértéket! A tizedes elválasztó a vessző karakter. Indítsuk el a programot folyamatos futtatási üzemmódban ! Látható, hogy a *Front Panel*-en az indikátoron megjelenik a megadott érték. Folyamatos futás közben az újonnan megadott értékek az Enter billentyű leütése után azonnal megjelennek az indikátoron. A kontroll fel/le nyilaival egyesével növelhető, illetve csökkenthető a kontroll értéke. Ha az indikátor számábrázolását megváltoztatjuk a saját menüje *Representation* legördülő ablaka alatt bármely „I” betűvel kezdődő *Integer* típusra, az indikátoron megjelenő érték a kontroll értékének egészre kerekítésének eredménye. Hasonlóan, ha a kontroll komplex számábrázolásra van beállítva, az indikátoron csak akkor jelenik meg a komplex szám valós részétől eltérő érték, ha az is komplex számábrázolási módra van állítva. Különböző számábrázolású elemek összekötésekor az indikátor terminálján egy piros színű jelölés látható. Ez a különböző számábrázolás közötti konverziót jelöli. Most kapcsoljuk be a *Highlight Execution* adatfolyam követési futtatási módot a *Block Diagram*-on a  ikonnal, majd ismét futtassuk le a programot! A program futása ekkor lassítva látható, minden huzalon látható, ahogy jelképesen végighalad rajt az adat. Ez a mód gyakran segítségünkre lehet esetleges elvi hibák felderítésekor.


1. Numerikus műveletek, ciklusok, struktúrák


1.1. Hatványozás

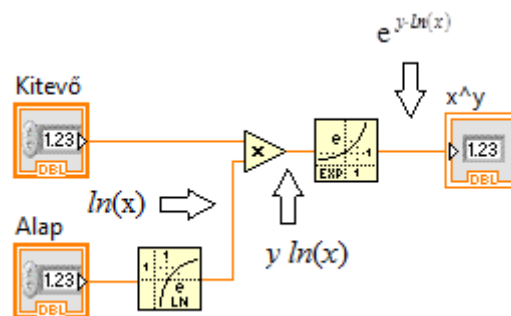
A LabVIEW függvényei között természetesen szerepel megoldás hatványozás egyszerű elvégzésére, mely a *Functions Palette* → *Mathematics* → *Elementary* → *Exponential* → x^y útvonalon érhető el. Most a hangsúly a LabVIEW egyes részeinek megismerésén van egy egyszerű feladaton keresztül. A feladat megoldásához szükséges az alábbi összefüggés ismerete:

$$x^y = (e^{\ln(x)})^y = e^{y \cdot \ln(x)}.$$

A feladat megoldásához szükség van tehát exponenciális és e alapú logaritmus számítására, valamint egy szorzásra. Egy elem a bemenetére érkező adatokon hajtja végre az általa reprezentált függvény kiértékelését, majd küldi az eredményt tovább a kimenetére. Jelen

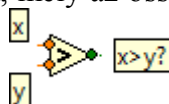
esetben az „Alap” nevű numerikus kontroll értékét egy e alapú logaritmust képző egység  bemenetére kell kötni (*Block Diagram* → *Mathematics* → *Elementary* → *Exponential* → *ln*), kimenetét pedig a „Kitevő” értékével együtt egy szorzó egység bemenetére, majd ennek




kimenetét kell ezután egy exponenciális képző egység  bemenetére kötni (*Block Diagram* → *Mathematics* → *Elementary* → *Exponential* → *exp*). Az exponenciális képző kimenetére ezután egy numerikus indikátort kell elhelyezni, és ezzel a program készen is van. Az elkészült programot a következő ábra mutatja.



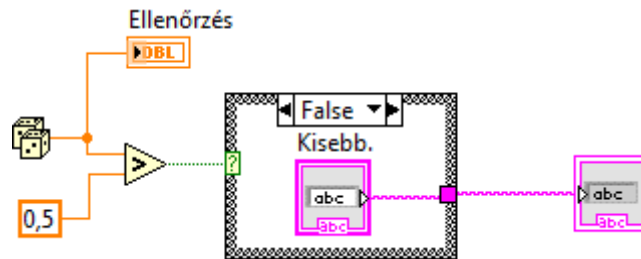
1.2. Case struktúra alkalmazása

A következő programnak egy egyszerű döntést kell meghoznia, mégpedig azt, hogy egy véletlenszerű 0 és 1 közötti szám nagyobb-e 0,5-nél, vagy kisebb. Ez a feladat tehát két jól megkülönböztethető állapotot feltételez (most egyelőre tekintsünk el az esetleges egyenlőségtől). A feladat megvalósításához szükség lesz egy véletlen szám generátorra, egy numerikus konstans értékre, egy összehasonlító operátorra, egy *String* indikátorra, két *String* kontrollra, egy *Case* struktúrára, és egy numerikus indikátorra a kimenet ellenőrzésére. Az összehasonlító operátorok a *Functions Palette* → *Comparison* menüpont alatt találhatóak. Ezek kimenete egy logikai érték, mely az összehasonlítás eredményét (igaz/hamis) jelöli. A feladat



megoldásához a *Greater?*  művelet szükséges. Ennek kimenete vezérli a *Case* struktúrát. Helyezzünk el egy véletlen szám generátort , mely a *Functions Palette* → *Numeric* menü alatt található! Ez a véletlen szám generátor 0 és 1 közötti álvéletlen számot generál. Huzalozzuk össze egy olyan elrendezést, mely megállapítja, hogy a véletlen szám nagyobb-e mint 0,5! Ezután helyezzünk el egy *Case* struktúrát, és az összehasonlító művelet kimenetét kössük a struktúra bemenetére ! A két külön állapot esetében ugyanazon a *String*

indikátoron kell az eredmény kiírni. A *Case* struktúra két állapota közül a tetején található nyilakkal lehet váltani, és az adott állapot esetén szükséges műveleteket pedig a struktúrán belül kell elhelyezni. Helyezzünk el egy *String* kontrollt az igaz és a hamis állapotba is! Ehhez a *Front Panel*-en kell létrehozni a két kontrollt, majd azok ikonját a *Block Diagram*-on a struktúra állapotaiba kell „behúzni”. Az egyik kontrollt a *Case* struktúra széléhez húzva létrejön egy *Tunnel*, ahol az adott feltétel eredménye kivezethető. Amennyiben a *Case* struktúra másik állapotában elhelyezett kontrollt is rákötjük ugyanerre a *Tunnel*-ra, úgy ugyanazon a kimeneten a feltétel teljesülésének megfelelő eredmény vihető rá, ugyanarra a kimenetre. Amennyiben nem sikerült a fentiek alapján a program elkészítése, a kész program a 25. ábrán látható. Mivel a véletlen szám generátor egy *double* precizitású lebegőpontos számot ad kimenetére, a 0,5-el való egyenlőségre az esély elhanyagolható.



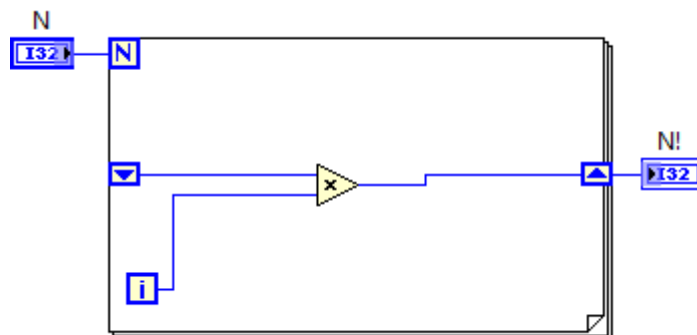
25. ábra: Egy egyszerű példa a *Case* struktúra használatára.

1.3. Faktoriális képzés *For* ciklus segítségével


A következő feladat megoldása segítségével bővebb betekintést nyerhetünk mind a *For* ciklus, mind pedig a *Shift Register* helyes alkalmazásába. Egy N tetszőleges nemnegatív egész szám faktoriálisa

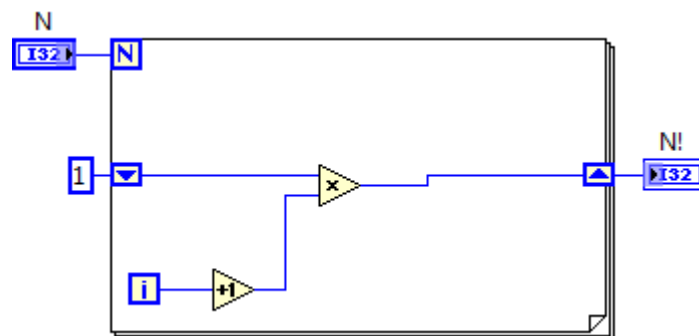
$$N! = \prod_{k=1}^N k = 1 \cdot 2 \cdot \dots \cdot (N-1) \cdot N$$

összefüggés alapján képezhető, illetve definíció szerint $0! = 1$. Ez annyit jelent, hogy a programot úgy kell szervezni, hogy kell egy változó, amely ciklusonként 1-gyel növekszik, és ezt a változót minden ciklusban meg kell szorozni az előző ciklus eredményével, mely az eddigi számok szorzata volt. A feladat megoldásához mindenképpen szükséges visszacsatolás alkalmazása. Ennek megvalósítása lehetséges egyrészt *Feedback Node*-dal, másrészt *Shift Register*-rel. A feladat megoldása *Shift Register* használata segítségével esztétikusabb, és átláthatóbb programot eredményez. Készítsük el a 26. ábrán látható programot! Adjunk meg N értékének egy viszonylag kis értéket, melynek faktoriálisát ismerjük (például $5! = 120$)!





26. ábra: Faktoriális képzés, első próba.

A program első ránézésre jónak tűnik, viszont a futtatás után a kimeneten 0 érték jelenik meg. Most futtassuk le a programot úgy, hogy bekapcsoljuk a *Highlight Execution*  módot! Látható, hogy az első iteráció futása során mind az i ciklusváltozó, mind pedig a *Shift Register* 0 értéket adott kimenetére, így annak szorzata is 0, ezt az értéket pedig visszacsatolva bármely számmal megszorozva szintén 0 értéket kapunk. A *Shift Register* bemeneti termináljára huzalozzunk egy numerikus konstans 1 értékkel. A ciklusváltozó kezdeti értékét nem lehet megváltoztatni, viszont nagyon egyszerűen, 1-et hozzáadva elérhető, hogy a ciklus ettől az értéktől induljon (ez az *Add* és az *Increment* függvény segítségével is egyszerűen megtehető). Meg kell-e változtatni ekkor N értékét, hogy a kimeneten ténylegesen $N!$ értéke jelenjen meg? Természetesen nem, bár a ciklusváltozó N mínusz 1-ig fut, a *For* ciklus N -szer értékelődik ki, a ciklusváltozó első értéke 1, az utolsó pedig $N-1+1=N$ lesz, tehát a program elvileg nem tartalmazhat már hibát. A kész program a 27. ábrán látható, ezt készítsük is el!



27. ábra: Faktoriális képzésére képes program.

Növeljük meg $N!$ indikátor vízszintes méretét a *Front Panel*-en a nagyobb számok megjelenítése érdekében, majd futtassuk le a programot folyamatos futtatás üzemmódban ! Látható, hogy a program működőképes, és helyes eredményt ad. Növeljük N értékét egyesével! Az tapasztalható, hogy $14!$ kisebb, mint $13!$, $17!$ értéke pedig negatív. A programozásban jártasak már rájöhettek, hogy a 32 bites előjeles *Integer* számábrázolás határát elértük. (Megjegyzés: a 32 bites előjeles *Integer* számábrázolás $-2,147,483,648$ -tól $2,147,483,647$ -ig terjedő számokat képes ábrázolni.) Állítsuk át az $N!$ indikátor számábrázolását előjel nélküli 64 bites *Integer*-re, majd futtassuk le a programot újra! Meglepve tapasztalhatjuk, hogy semmi változás nem történt. Vegyük észre, hogy $N!$ indikátor bemeneti terminálja piros színű, ez pedig a számábrázolások közötti konverziót jelenti.

Indítsuk el a *Context help*-et , majd a helyezzük a kurzort a huzalra! A huzalon áthaladó szám továbbra is előjeles 32 bites *Integer* szám. A huzalok által szállított érték számábrázolását az adatfolyamban korábban szereplő elemek határozzák meg. Kis keresgélés után megtalálható a hiba oka, ami nem más, mint a numerikus konstans érték, mely alapértelmezetten előjeles 32 bites *Integer* szám. Ezt is átállítva előjel nélküli 64 bites számábrázolású konstansra, a program újabb lefuttatása után már a várt eredmény látható a kimeneten. Természetesen a faktoriális képzés tulajdonságaiból adódóan túl nagy szám faktoriálisa így sem számítható ki, N értéke s legutolsó beállításokkal maximálisan 21 lehet. Nagyobb N értékek esetén közelítő eredmény nyerhető ki, ha *Integer* helyett *Double* számábrázolást használunk.

Érdekességként: *Extended Precision*-t beállítva (szükség szerint automatikusan változik, hogy hány bites a számábrázolás) a konstansnak és az indikátornak maximálisan 1754 faktoriálisának közelítő értéke számítható, mely értéke $1,97926 \cdot 10^{4930}$. Összehasonlításképpen, az ismert univerzum részecskéinek száma a becslések szerint 10^{72} és 10^{87} nagyságrendek közé esik. Természetesen ennek gyakorlati jelentősége nincs, a cél a

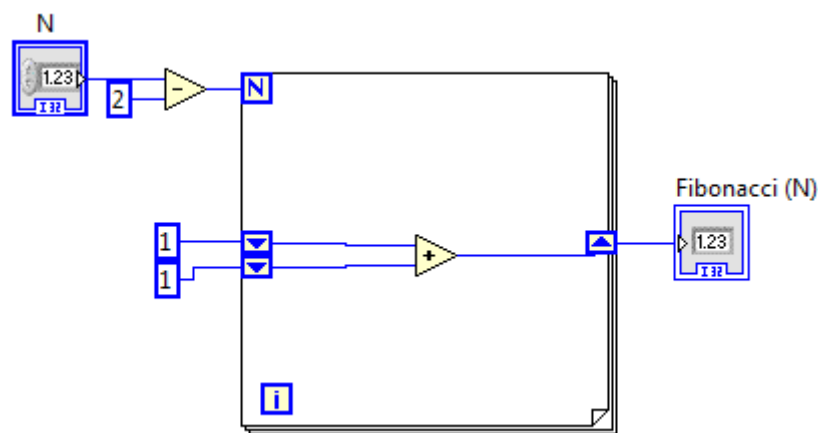
számábrázolási módok közötti különbségek bemutatása volt, gyakorlatban *Integer* számokat általában ciklushatár meghatározására, kezdeti érték beállítására használnak, az eredmény pedig legtöbb esetben egy *Double* skalár, a mérések eredménye pedig egy *Double* tömb.

1.4. Fibonacci sorozat N -edik elemének meghatározása

A Fibonacci sorozat bármelyik eleme az alábbi rekurzív összefüggés segítségével határozható meg:

$$x_i = x_{i-1} + x_{i-2}, \text{ ahol } x_1 = x_2 = 1.$$

A megoldásához szükség van tehát egy előző, és az azt megelőző iteráció eredményére. Erre lehetőség van a *Shift Register* bal oldalán található részét megnyújtásával, vagy jobb egérgombbal rákattintva az *Add Element* opciót választva. Mivel a Fibonacci sorozat első két értékének ismeretében bármelyik eleme kiszámítható, nem szükséges a ciklusváltozó alkalmazása. A feladat megoldása a 28. ábrán látható.



28. ábra: A Fibonacci sorozat N -edik értékének meghatározása.

Hogyan működik ez a program? Miért van szükség arra, hogy az N értékéből levonjunk 2-t? A *Shift Register* kezdeti értékei a Fibonacci sorozat első két eleme, a felső a második, az alsó pedig az első. Ha megfigyeltük, látható volt, hogy amikor a *Shift Register*-nek új elemet hoztunk létre, azt alulra helyezte el a program, tehát mindig a legalsó számít a legrégebbi értéknek. Ha nem vonnánk ki 2-t N értékéből akkor nézzük meg, mi történne. Legyen $N=1$. Ekkor a ciklus egyszer fog lefutni, a kimeneten $1+1=2$ fog megjelenni, amely a Fibonacci sorozat harmadik eleme, tehát $N=n$ esetén a kimeneten az $n+2$ -ik elem látható.

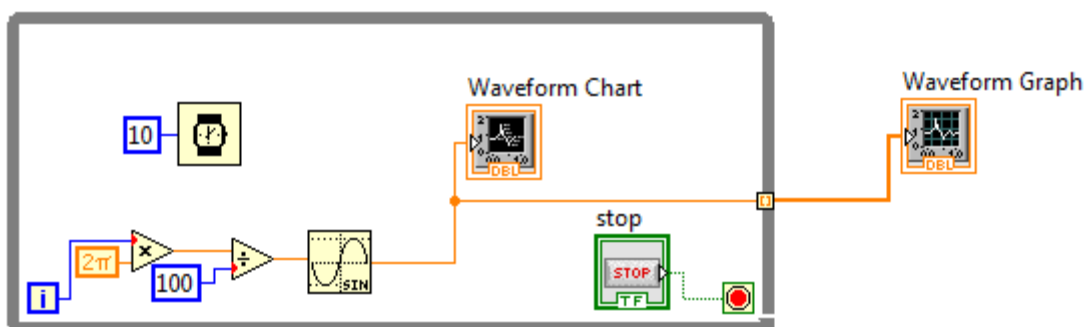
1.5. Szinusz függvény kirajzolása felhasználói beavatkozásig.

Ebben a feladatban a *while* ciklussal, illetve a különböző grafikus kijelzőkkel ismerkedhetünk meg. A program semmi másra nem lesz hivatott, minthogy egy szinusz függvényt kirajzoljon egészen addig, ameddig a programot le nem állítjuk. Használjunk *while* ciklust! A ciklusváltozónak a növekedését érdemes kihasználni, annak valamely konstansszorosának a szinuszát kell majd kiszámolnia a programnak. Érdemes odafigyelni a program készítése során, mivel rosszul megválasztott értékek esetén a kimenet elveszítheti szinuszos jellegét.

A program megvalósításához szükség lesz egy *while* ciklusra, egy *stop* nyomógombra, néhány numerikus műveletre, egy szinusz képző elemre, egy időzítő elemre, illetve egy *waveform chart*-ra és egy *waveform graph*-ra. Pár megfontolandó lépés:

- A ciklusváltozó értékét szorozzuk meg 2π értékével, ugyanis 2π az pont egy teljes periódus, így ha egy periódust 5 értékre szeretnénk felosztani (tehát 5 értékünk legyen periódusonként), akkor elegendő 5-tel elosztani utána ezt az értéket.
- Érdeemes egy periódust annyi részre felosztani, hogy a szinusz alakja már bőven felismerhető legyen.
- Ahhoz, hogy a *waveform chart* működését jobban megismerjük, érdemes egy időzítő elemet elhelyezni valamekkora késleltetéssel, hogy láthatóvá váljon, hogy minden egyes érték azonnal kiírásra kerül a kijelzőn.



Helyezzünk el egy *while* ciklust a *Block Diagram*-on! A ciklusváltozót szorozzuk meg 2π -vel! A 2π érték, mint konstans, megtalálható a *Functions Palette* \rightarrow *Numeric* \rightarrow *Math & Scientific Constants* útvonalon keresztül. Ezt az értéket osszuk el annyival, ahány részre szeretnénk osztani a szinusz hullámot, majd helyezzünk el egy szinusz számítására alkalmas elemet! A szinusz függvény a *Functions Palette* \rightarrow *Mathematics* \rightarrow *Elementary* \rightarrow *Trigonometric* útvonalon keresztül érhető el. A program futásának lassítása érdekében érdemes elhelyezni egy időzítő elemet. Erre most szolgáljon a *Wait (ms)* nevű parancs, mely a *Timing* fül alatt található a *Functions Palette*-n. Helyezzünk el egy *waveform chart*-ot és egy *waveform graph*-ot a *Front Panel*-en a *Controls Palette* \rightarrow *Graph* menüpontja alá! Az elsőt helyezzük a *while* cikluson belülrre, a másodikat pedig kívülrre! Erre a megkülönböztetésre azért van szükség, mert a *waveform chart* képes egymás után érkező skalár értékek megjelenítésére egymás után, a *waveform graph* viszont csak egy tömb/array elemeit rajzolja ki. A tömb nem más, mint azonos típusú elemek indexelt sorozata, egy tömb *i*-edik elemére a sorszámával/indexével lehet hivatkozni. A szinusz művelet kimenetét huzalozzuk rá mindkét kijelző bemenetére! A *waveform graph* ezt a lépést elutasítja. Ez annak köszönhető, hogy a cikluson kívülrre huzalozott érték egy *Tunnel*-en keresztülhalad a *graph* bemenetére, és a *Tunnel* nincs indexelő módba kapcsolva, ami annyit jelent, hogy csak a legutolsó iteráció eredménye jut a kimenetre. Ez ugye azért problémás, mert ekkor csak egy értéket tudnánk kinyerni a ciklusból, nem az egész szinusz hullámot. Az automatikus indexelés bekapcsolása a *Tunnel*-en jobb egérgombbal kattintva a *Tunnel Mode* \rightarrow *Indexing* parancs segítségével kapcsolható be. Ennek köszönhetően minden érték, mely a *Tunnel*-ig eljut egy tömböt fog képezni. A program még nincs kész, ugyanis amíg a *while* ciklus leállási feltétele nincs beállítva, addig a program nem futtatható. Helyezzünk el egy *stop* gombot a *Front Panel*-en! A *stop* gomb megtalálható a *Controls Palette* \rightarrow *Boolean* menüpont alatt. A kész program a 29. ábrán található.

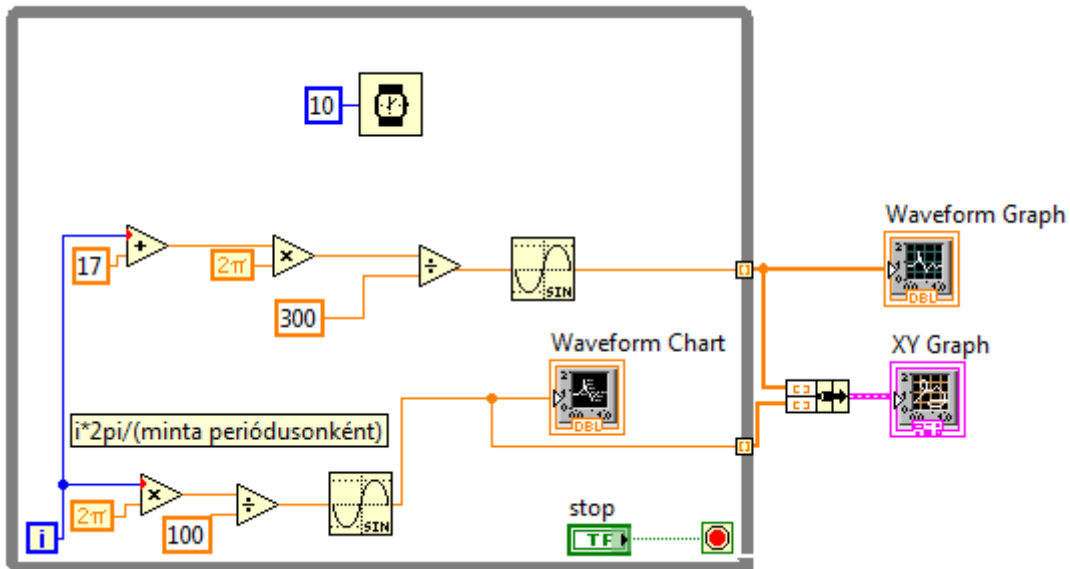


29. ábra: Egyszerű szinuszos jel generálása, és kijelzése.

A *Graph* menüben található még egy, számunkra hasznos virtuális kijelző, ez az úgynevezett *XY Graph*. Itt az *x* és az *y* tengelyen felvett helyet is beállíthatjuk. Ez különböző méréseknél elengedhetetlen lehet, például egy dióda feszültség-áram karakterisztikája vagy mágneses hiszterézis mérése esetén elengedhetetlen. Ehhez már két tömbre van szükség, egy *x* irányú,

és egy y irányú kitérést tartalmazó tömbre, melyet a korábban tárgyalt módon lehet létrehozni. A két tömböt egyetlen *Cluster*-ba kell elhelyezni. A *Cluster* különböző típusú elemek logikai csoportja, de természetesen azonos típusúakat is el lehet csoportosítani *Cluster*-ként. Két

tömböt egy *Cluster*-ba a *Bundle*  vagy a *Bundle by Name*  függvénnyel helyezhetünk. Ezek a *Functions Palette* → *Cluster, Class & Variant* menüpont alatt található. A 29. ábrán látható program az *XY Graph* kijelzővel kiegészítve a 30. ábrán látható.



30. ábra: Két különböző frekvenciájú és fázisú szinuszos jel *Lissajous* görbéjének kirajzolása.

1.6. Nullára redukált másodfokú polinom gyökeinek meghatározása

A következő példa egy egyszerű gyakorló példa. Egy olyan program elkészítése a cél, amely gombnyomásra egy tetszőleges nullára redukált másodfokú polinom gyökeit kiírja a kimenetre. Ezek a gyökök komplex számok is lehetnek. Egy másodfokú, x -től függő, nullára redukált polinom általános felírása az alábbi:

$$ax^2 + bx + c = 0.$$

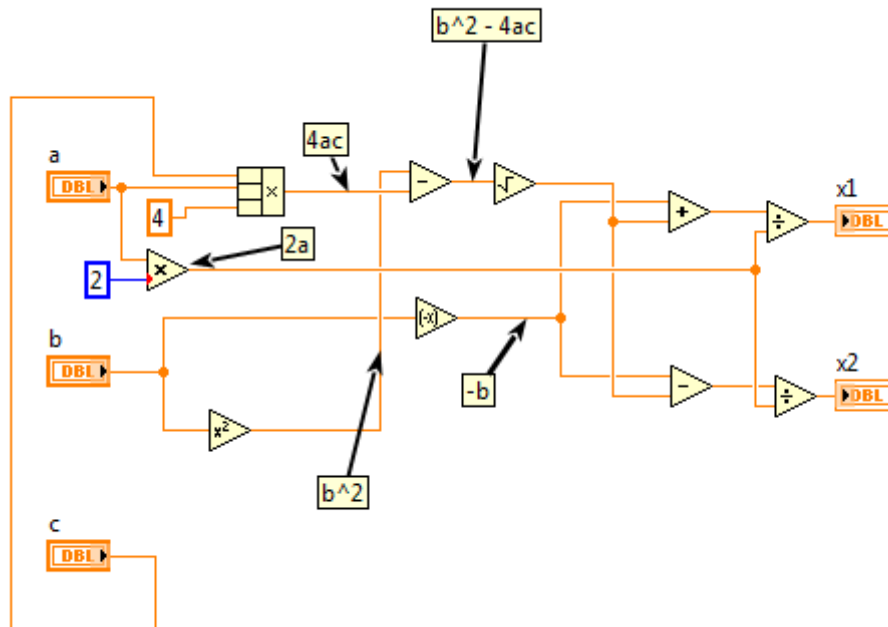
Legyenek a , b , és c valós számok. A gyökhelyeket az alábbi jól ismert összefüggés segítségével határozhatjuk meg:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

A $b^2 - 4ac$ kifejezés az úgynevezett diszkrimináns. A feladat megoldása során több fontos tényezőt kell figyelembe venni:

- Ha $a = 0$, akkor egy megoldás van: $x_1 = -\frac{c}{b}$.
- Ha a diszkrimináns értéke nagyobb 0-nál, akkor két valós megoldás van.
- Ha a diszkrimináns értéke megegyezik 0-val, akkor egy kétszeres gyöke van a kifejezésnek.
- Ha a diszkrimináns értéke kisebb 0-nál, akkor két komplex megoldás van, melyek egymás konjugáltjai.
- Ha $a = b = 0$, akkor az egyenletnek nincs értelme. ($c = 0$???)

Jól látható, hogy több *Case* struktúrára lesz szükség. Először legyen a bemenet három numerikus kontroll, a kimenet pedig két numerikus indikátor, majd később kicsit bonyolultabb, de felhasználóbarátabb programot készítünk. Első esetben csupán készítünk el egy programot, mely kiszámítja a két gyökhelyet, egyelőre ne foglalkozzunk a fent felsorolt esetekkel. A kész program a 31. ábrán látható, a honlapon található feladatok közül ez az egyes számú verzió.



31. ábra: Másodfokú polinom gyökhelyeinek meghatározása.

A fenti programmal rengeteg probléma van, nem ismeri fel, ha a értéke 0, és nem tudja kiszámítani a komplex gyököket. Komplex gyökök kiszámítására csak akkor van lehetőség, ha a legalább egy bemeneti számértéknek és az indikátoroknak adattípusát komplexre állítjuk. A különböző eseteket érdemes lenne szétválasztani *Case* struktúrák segítségével. A bővített, precízebb programok kódjait az olvasó a segédletben már nem találja meg, mivel rengeteg különböző esetet kellene egyszerre bemutatni, és a kód maga a mérete miatt egyszerűen nem férne el. Amennyiben lehetőség van rá, javaslom a segédlet és a példaprogramok együttes tanulmányozását. A kettes számú példaprogram már képes elkülöníteni az első, illetve a másodfokú polinomokat, elsőfokú polinom esetén is jó megoldást ad, viszont a program még mindig csak annyira képes, hogy egyszeri lefuttatás esetén az előre begépelt értékekre érvényes gyökhelyeket kiírja, és a komplex gyököket továbbra is csak vegyesen lehet megjeleníteni. A hármas számú program teljes egészében egy *While* ciklusba van helyezve, felhasználói beavatkozásig fut (STOP gomb megnyomásáig). Található benne egy „Számolj!” nevű egyállású logikai kapcsoló. A program csak akkor számítja ki az aktuális bemeneti paraméterek mellett gyökhelyek értékét, ha a felhasználó ezzel a paranccsal erre utasítást ad. A program most már képes valós bemeneti paraméterértékek mellett is komplex eredményt szolgáltatni, mivel az adatfolyam közben az adat típusa egy valós \rightarrow komplex konverzió esik át **DBL**. Ez a konverziós művelet a *Numeric* \rightarrow *Conversion* menüpont alatt található. Így viszont ha az eredmény valós, abban az esetben is komplex értéként látszik 0i képzetes résszel. A négyes számú program tanulmányozását főképp haladóbb hallgatóknak ajánlom. A program képes minden esetet megkülönböztetni, és az esetnek megfelelő szöveges üzenetként

az eredményt megjeleníteni. Tetszőleges valós bemeneti számhármassal esetén egy mondatban összefoglalva látható az eredmény. Itt több *String* művelet is alkalmazásra került.

1.7. d

2. Szekvenciák, helyi változók, több állapotú struktúra



2.1. Flat Sequence/Stacked Sequence alkalmazása

A különböző szekvenciák alkalmazása elősegítheti a program átláthatóságát, helyet lehet velük megtakarítani a *Block Diagram*-on. A *Flat* és a *Stacked Sequence* szinte minden funkciójában megegyezik. Egyetlen különbség van csupán a kettő között: a *Flat Sequence* alkalmazásakor minden *Frame* kiértékelése után lehetőség van az eredmény kinyerésére, míg *Stacked Sequence* esetén csupán az összes *Frame* kiértékelése után lehet csak adatot kinyerni valamelyik *Frame*-ből.


2.1.1. Stacked Sequence használata

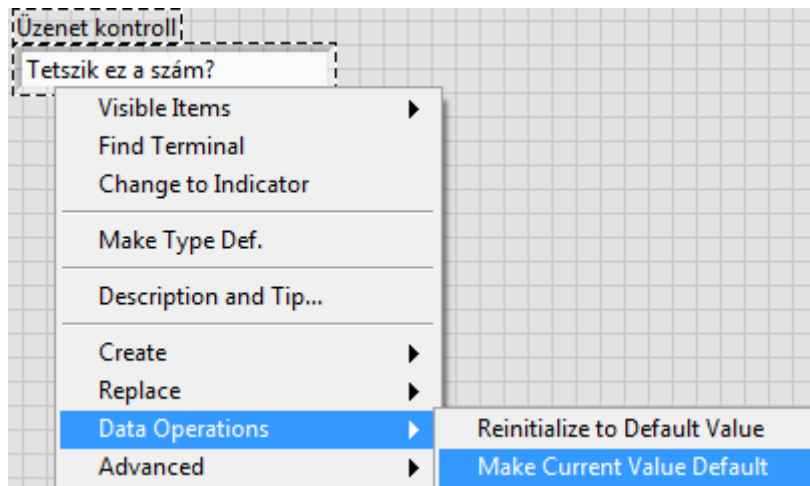
Stacked Sequence alkalmazása főképp helykihasználtság optimalizálására, egy folyamat főbb lépéseinek egyértelmű elkülönítésére szolgál. Egy új *Frame* létrehozására a szekvencia szélére jobb egérgombbal kattintva, majd az *Add Frame After* vagy az *Add Frame Before* opciót választva van lehetőség. Az egyszerűség kedvéért a feladat legyen a következő: generáljon a program az első *Frame*-ben egy random számot, melyet a második *Frame*-ben kiír egy numerikus indikátorra, majd a harmadik *Frame*-ben megkérdezi, hogy ez a szám számomra megfelelő-e. A program megfelelő feltételig történő futását egy *While* ciklus biztosítja. A ciklusba helyezzünk el egy *Stacked Sequence*-t, és adjunk hozzá két *Frame*-et! A *Frame* hozzáadása a szekvencia szélére jobb egérgombbal kattintva, majd az *Add Frame After* opciót kiválasztva lehetséges. A szekvencia használatának egy fontos tulajdonsága, hogy közvetlenül adat kicsatolására csak akkor van lehetőség, ha a teljes szekvencia minden képkockája kiértékelődött. Ez annyit jelent, hogy amennyiben egy értéket huzalozással kicsatolunk, akkor nem az aktuális *Frame*-en belül található műveletek kiértékelése után, hanem minden egyes *Frame* végrehajtása után jut csak a kimenetre az érték. Ez azzal jár, hogy a kicsatolással létrejövő *Tunnel* bemenetére más *Frame*-ek kimenetét nem lehet huzalozni. Ez tulajdonképpen logikus is, hiszen egy teljes szekvencia lefutása után nem lehetne egyértelműen eldönteni, melyik érték jusson el az „alagút végére”. *Stacked Sequence*-t használva több esetben is előfordulhat, hogy egy értéket egy későbbi *Frame*-ben fel



szeretnénk használni. Ilyen érték csatolásra *Frame Local*  használatával van lehetőség, melynek létrehozása a szekvencia szélére jobb egérgombbal kattintva, majd *Add Frame Local* opció segítségével történik. Helyezzünk el az első *Frame*-be egy véletlen szám generátort , melynek értékét írassuk ki a második *Frame*-ben! A harmadik *Frame*-ben kommunikáljon a program a felhasználóval, és például kérdezze meg, hogy tetszik-e a felhasználónak a véletlenszerűen generált szám. (Igen, a feladat nem feltételez túl sok fantáziát.) Erre a

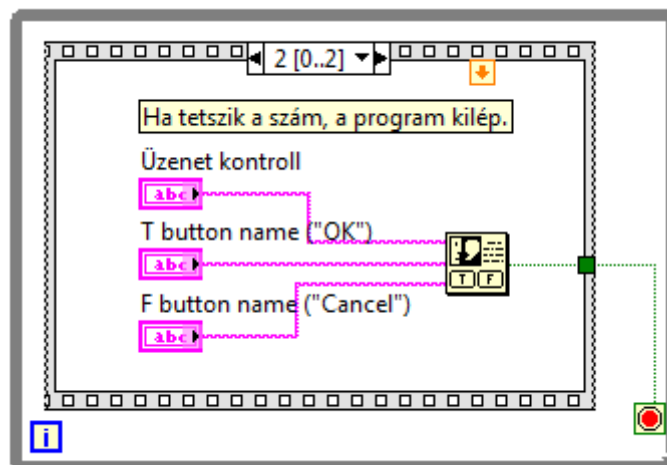


kommunikációra kiválóan alkalmas a *Two Button Dialog*  függvény, mely egy felugró üzenetet közöl a felhasználóval, az pedig egy igen-nem döntéssel eltávolítható. A dialógus különböző szövegezései egyenként egyéni beállíthatók az egyes bemenetein. Helyezzünk el a dialógus három szöveges bemenetére egy-egy *String* kontrollt, és írjunk a saját szájzünk szerinti üzeneteket és választási lehetőségeket! Amennyiben elmentjük a programot, és kilépünk a *LabVIEW*-ből a program következő indításakor ezek a kontrollok üresek lesznek. Ez azért történik, mert nem lett beállítva a kontroll alapértelmezett értéke. Az érték alapértelmezését a kontroll saját menüjén belül a 32. ábrán látható módon lehet elvégezni.



32. ábra: Kontroll értékének alapértelmezése.

A dialógus függvény az általunk választott logikai értéket adja kimenetére, így ha azt szeretnénk, hogy a program addig fusson, amíg egy számunkra tetsző számot nem generál a program, akkor egyszerűen csak a dialógus kimenetét kell ráhuzalozni a *While* ciklus leállító termináljára. A kész program a *Stacked Sequence* harmadik *Frame*-jével a 33. ábrán látható.

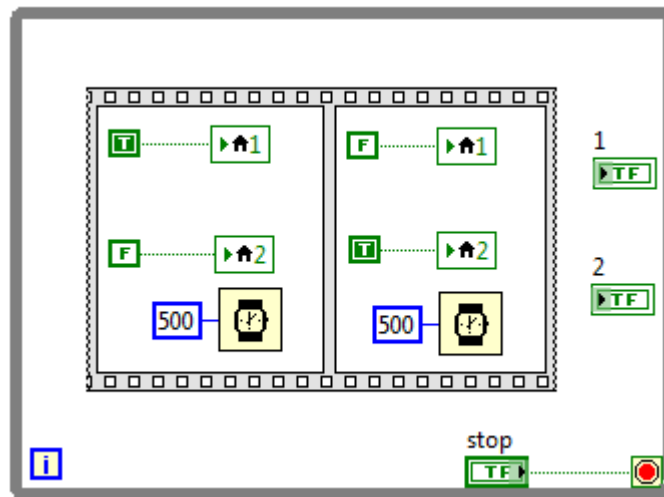


33. ábra: A *Stacked Sequence* és a dialógus függvény alkalmazásának demonstrálása.

2.1.2. *Flat Sequence* használata

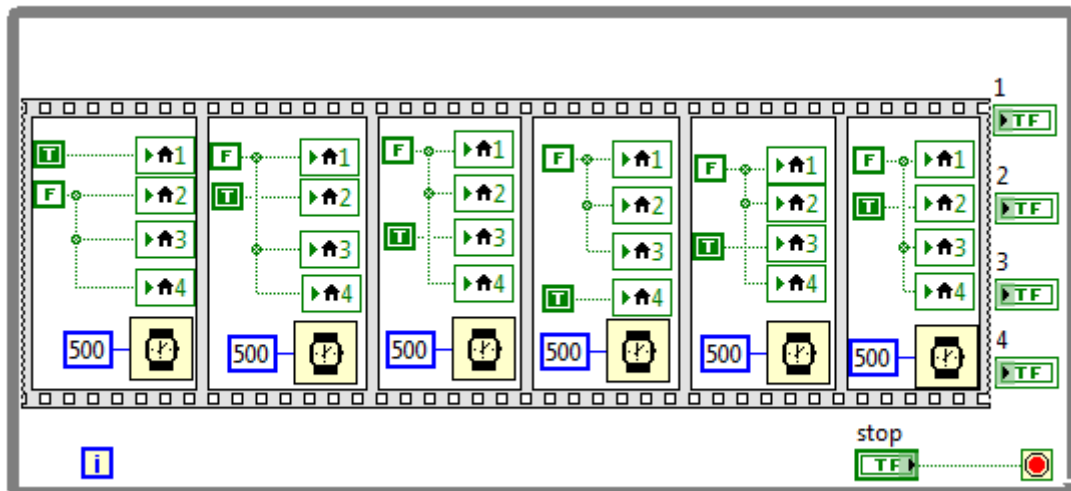
A *Flat Sequence* előnye a *Stacked Sequence*-el szemben az, hogy egy *Frame* végrehajtása után az eredmény azonnal kicsatolható a szekvenciából, nem szükséges az összes többi *Frame* feladatainak kiértékelése, hátránya viszont a rosszabb helykihasználás. Alkalmazása egy egyszerű programon keresztül történik, egy két LEDből álló villogó LabVIEW program bemutatása következik. Mivel a villogás folyamatos, ezért a program alapja egy felhasználói beavatkozásig futó *While* ciklus. A villogásban két LED játszik szerepet, tehát két állapotot, két képkockát, két *Frame*-et kell elkülöníteni. A *Flat Sequence* a *Structures* menüpont alatt érhető el. Egy új *Frame* létrehozása a szekvencia szélére jobb egérgombbal kattintva, majd az *Add Frame After* opció kiválasztásával lehetséges. A *Flat Sequence* biztosítja a különböző lépések végrehajtását szigorúan egymás után. Mivel két állapotban, két helyről kell vezérelni a LEDeket (itt közös alagút használatára nincs lehetőség), úgynevezett helyi változók/*Local Variable* használatára van szükség. A helyi változók segítségével az egyes kontrollok bárhol, és több helyről fejthetnek ki hatást, az indikátorok pedig bárhol, és

több helyről vezérelhetők. Helyi változót egy kontrollhoz vagy indikátorhoz az elemre jobb egérgombbal kattintva, majd *Create* → *Local Variable* opció segítségével lehet létrehozni, illetve a *Functions Palette*-n a struktúrák között, melyet elhelyezve, majd rákattintva kiválaszthatjuk, mely kontrollnak/indikátornak legyen a helyi változója. Mindkét LEDnek el kell helyezni mindkét *Frame*-ben egy helyi változót, hogy minden lépésben egyértelműen be legyen állítva a LED állapota. Ahhoz, hogy a villogás látható legyen, a program futását le kell lassítani. Ez a lassítás időzítő elemek segítségével történhet, méghozzá úgy, hogy mindkét *Frame*-ben megtörténjen a program kimeneti állapotának tartása, mivel így biztosítani lehet, hogy mindkét állapot ugyanannyi ideig tartson. A kész program a 33. ábrán látható.



34. ábra: Kétállapotú villogó programja.

Egy kicsit bővebb program segítségével már futófényt is készíthetünk, akár oda-vissza futással is (lásd: 34. ábra).




35. ábra: Oda-vissza futófény programja.

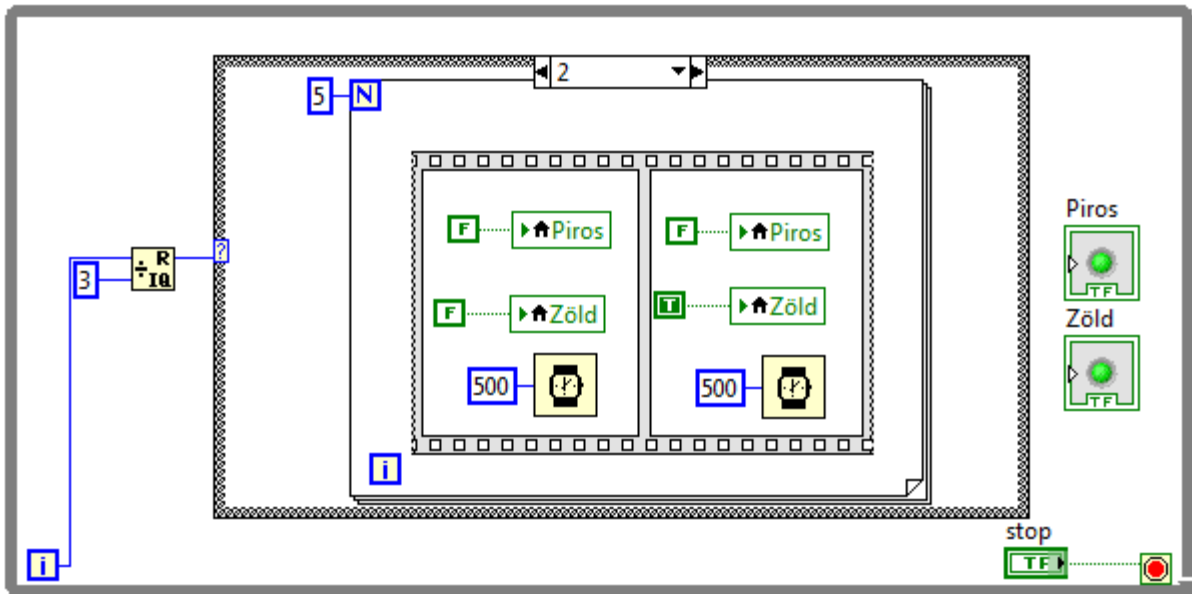
2.2. Több állapotú *Case* struktúra használata

Ahogy az korábban említésre került, *Case* struktúra segítségével több állapot megkülönböztetésére is lehetőség van, a program több irányba ágazhat itt el. Lehetőség van szöveges változók szerint, numerikus értékek szerint, de akár listaelemek (*Enum control*)

szerint is állapotok megkülönböztetésére. A most következő példa egy gyalogátkelőhely lámpájának virtuális elkészítése. A lámpa 3 különböző állapotban lehet:

1. A piros lámpa világít;
2. A zöld lámpa világít;
3. A zöld lámpa villog.

Az ehhez szükséges program valójában pofonegyszerű, és az eddig megszerzett tudás szinte teljes mértékben elegendő is az elkészítéséhez. Hogy a program felhasználói beavatkozásig fusson, helyezünk el egy *While* ciklust, és ebbe dolgozzunk ezután! Mivel 3 állapot megkülönböztetésére van szükség, használjuk ki most az *i* ciklusváltozó folyamatos növekedését, és a maradékos osztás műveletét! Amennyiben az *i* értékét 3-mal osztjuk maradékosan, a maradék értéke 0, 1 illetve 2 lehet, ez 3 különböző állapot megkülönböztetését teszi lehetővé. Helyezzünk el egy *Case* struktúrát, és a bemeneteként funkcionáló  terminálba huzalozzuk a maradékos osztás eredményét! Ekkor a terminál kék színre vált (feltéve, hogy az osztás nevezőjeként szolgáló numerikus konstans *integer* típusú). A *Case* struktúra tetején található *Selector Label* most nem *True / False* állapotokat mutat, hanem 0, *Default*, illetve 1. Amennyiben nem látható egyik állapot mellett sem, hogy *Default*, úgy azt nekünk kell megadnunk. Ennek megadása úgy történik, hogy az alapértelmezett állapotban a *Selector Label*-re jobb egérgombbal kattintva a *Make This The Default Case* opciót választjuk. Erre azért van szükség, mert az állapotválasztó olyan típusú, hogy elvileg végtelen értéket felvehet, speciel a mostani program a 2 állapotot sem ismeri, mivel ezt nem specifikáltuk, így ha a *Case* struktúra bemeneti termináljára a maradékos osztás eredményeként egy 2-es érték érkezne, úgy a 0, *Default* ág kerülne kiértékelésre. Hozzuk létre a 2 nevű állapotot! Ehhez kattintsunk jobb egérgombbal a *Selector Label*-re, majd válasszuk az *Add Case After* opciót. A LabVIEW automatikusan a 2 nevű állapotot hozza létre, bár ez nem azt jelenti, hogy tudná milyen állapotok jöhetnek szóba a bemeneten, ez most a szerencsének köszönhető, viszont listaelemmel vezérelve a *Case* struktúrát a LabVIEW képes az összes lehetséges állapotot automatikusan létrehozni (ehhez adott esetben a *Selector Label*-en az *Add Case for Every Value* opciót kell választani). Legyen a 0, *Default* nevű állapot a fenti felsorolás 1-es állapota, az 1 nevű a 2-es, míg a 2 nevű állapot a 3-as állapot. Helyezzünk el két LED-et, majd ezeket a *Block Diagramon* vigyük a *While* cikluson belülre! A két LED színét változtassuk meg pirosra, illetve zöldre! Ez az elemre jobb egérgombbal kattintva, majd a *Properties* opciót választva lehetséges. Itt a legelső ablakon belül beállítható az is, hogy be-, illetve kikapcsolt állapotban milyen színű legyen a LED. A LED-eket mindhárom állapotban helyi változók segítségével vezéreljük. A fent említett állapotoknak megfelelően hozzuk létre a 0, *Default* és az 1 állapotokat, mindkét esetben várakozzon valamennyit a program, a harmadik állapotban pedig építsük meg az előző fejezetben található villogót, csak most *While* ciklus helyett alkalmazzunk *For* ciklust, s adjunk meg egy *N* küszöböt, ahányszor szeretnénk, hogy a villogás bekövetkezzen. A kész program a 2 állapotot látva a 36. ábrán látható.

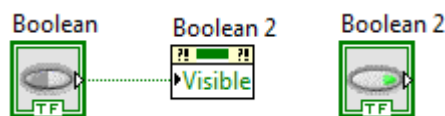


36. ábra: Gyalogátkelőhely lámpájának szimulációjára szolgáló program.

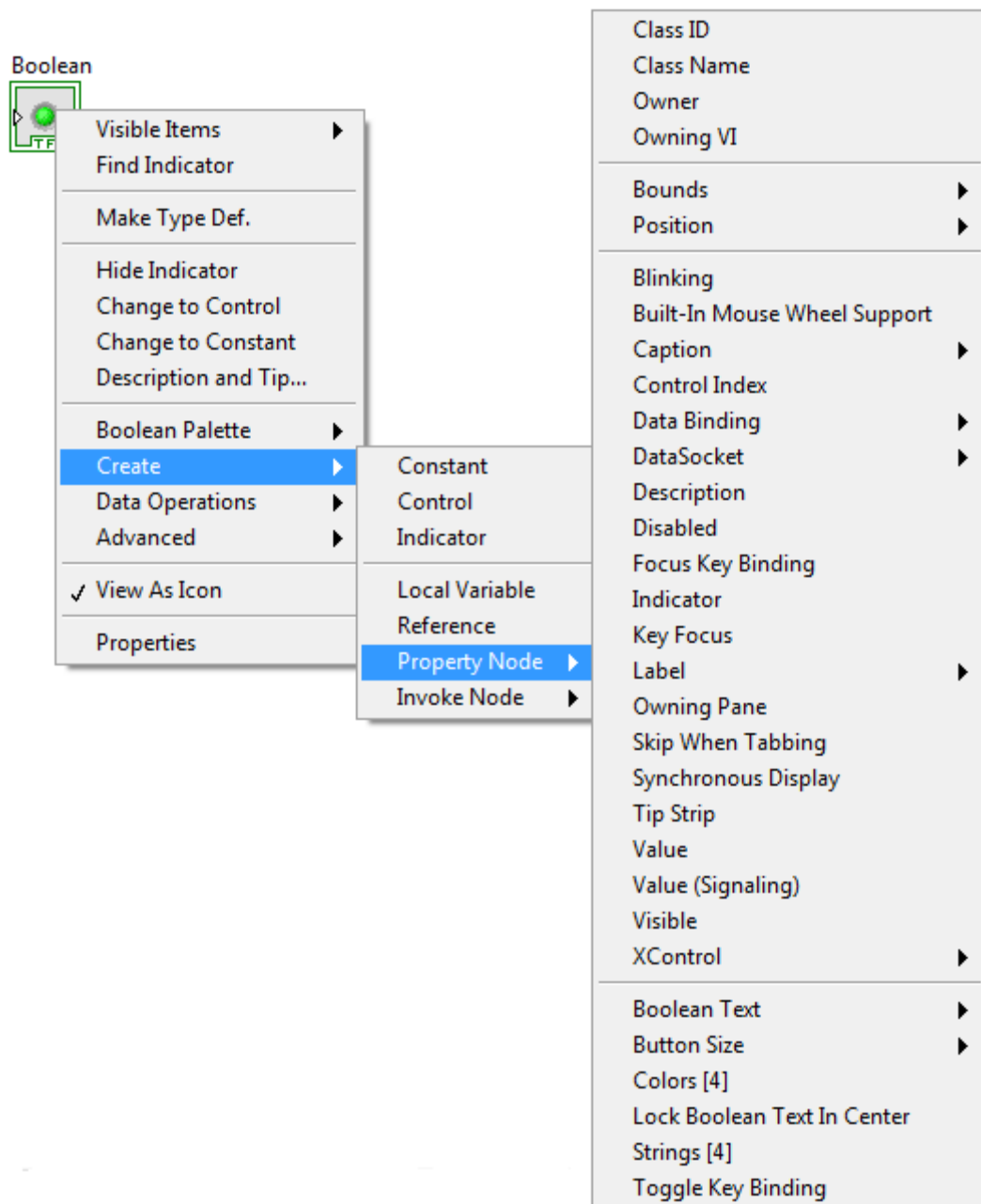
3. Tulajdonság csomópont (*Property Node*), állapotgép (*State Machine*)

3.1. *Property Node*-ok alkalmazása

A tulajdonság csomópontok (*Property Node*-ok) segítségével az egyes kontrollok és indikátorok különböző tulajdonságait változtathatjuk meg akár a program futása közben is. Segítségükkel megváltoztatható például egy LED színe, törölhető egy *Waveform Chart* kijelzője, vagy akár mozgathatjuk is az egyes építőelemeket a *Front Panel*-en. A *Property Node*-ok bemutatása ebben a fejezetben közel sem teljes, de nincs is szükség arra, hogy teljes legyen, alkalmazásuk bár nagyon széleskörű lehet, a tárgy keretein belül csak a megismerésük és alap szintű megértésük a cél 1-2 példán keresztül. Az első egyszerű példa megmutatja, hogy akár el is tűntethetünk egyes elemeket a *Front Panel*ről látszólag (az elemek továbbra is megtalálhatóak lesznek, csak láthatatlanul). Hozunk létre két darab logikai kontroll elemet, és helyezük el az egyik számára egy olyan *Property Node*-ot, amely a láthatóságért felel (lásd: 37. ábra)! Ezt az elemre jobb egérgombbal kattintva a *Create* → *Property Node* → *Visible* lehetőséget választva lehet megtenni (a *Property Node*-ok listája a 38. ábrán látható). A *Property Node* jelenleg kimenettel rendelkezik csak, ez annyit jelent, hogy most az adott tulajdonság értékének a kiolvasására van lehetőség, azonban a csomópontra jobb egérgombbal kattintva, majd a *Change All to Write* opciót választva beállíthatóvá válik ez az érték. Huzalozzuk ennek a csomópontnak a bemenetére a másik logikai kontroll kimenetét, majd indítsuk el a programot folyamatos futtatási módban. A kettes számú logikai kontroll eltűnik a *Front Panel*-en, mivel a láthatóságát vezérlő tulajdonság csomópontja hamis értéket kapott a bemenetére. A kontroll értékét változtatva a másik kontroll megjelenik, majd újabb változtatás esetén ismét eltűnik.



37. ábra: Az 1-es számú kapcsoló eltünteti a 2-es számú kapcsolót.

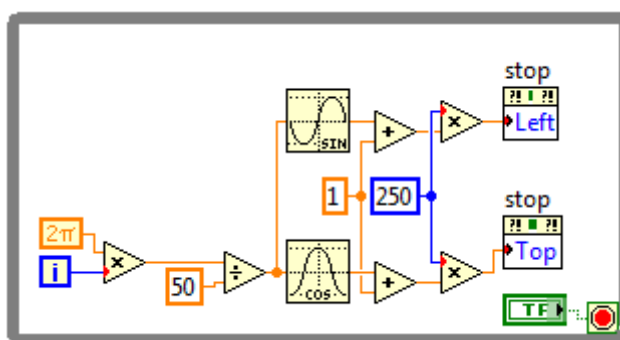


38. ábra: Egy egyszerű LED számára elhelyezhető *Property Node*-ok.

Mint az látható a 38. ábrán, szinte minden tulajdonsága beállítható egy-egy kontrollnak, vagy indikátornak. A következő egyszerű példában egy *Stop* gomb pozícióját fogjuk beállítani. Helyezzünk el egy *While* ciklust, és annak kilépési feltételeként szolgáló *Stop* gombot! A *Stop* gomb számára helyezzünk el a pozícióját meghatározó *Property Node*-ot, vagy *Node*-okat. Azért van megkülönböztetve a két eset, mert lehetőség van külön a horizontális, és a vertikális pozíciók megadására (*Position* → *Left* vagy *Top*), viszont lehetőség van egy tömb segítségével egyszerre mindkét érték megadására (*Position* → *All*). A program futása során bármikor beállítható egy adott elem pozíciója, így egy ciklus segítségével akár mozgatható is. A feladat legyen a következő: a *Stop* gomb haladjon körpályán! Ennek eléréséhez az szükséges, hogy a horizontális és a vertikális pozíció azonos frekvencián változzon, az egyik szinuszos, a másik pedig koszinuszos függvény szerint. A pozíciót a *Front Panel* képernyőjének tetejétől, és bal oldalától való távolságban kell megadni. Alapvetően a bal



felső pontja a *Front Panel*-nek a (0, 0) koordinátájú pont. Ezt a pontot az alábbi marker jelöli. Mivel a szinusz, illetve a koszinusz függvény (-1, 1) tartományban vesz fel értékeket, ha azt szeretnénk, hogy a bal és felső határt ne lépje túl a pozíció, a függvény kimenetéhez hozzá kell adni 1-et, így (0, 2) határok között fog mozogni ez az érték. Egységnyi elmozdulást viszont szemmel alig lehet észrevenni, mivel a *Front Panel* háttérében a rács egyetlen eleme 10 egység oldalhosszal rendelkezik, tehát a (0, 2) tartományú kimenetet érdemes még megszorozni körülbelül 200-al, így már rendkívül látványos mozgást is meg lehet valósítani. A nyomógomb pontos pozíciójának meghatározásához még annak ismerete is szükséges, hogy a nyomógomb bal felső sarkát illeszti az általunk megadott pozícióba a *Property Node*. Komplexebb, pontosabb pozicionáláshoz figyelembe vehetjük a *Stop* gomb horizontális és vertikális kiterjedését is. A nyomógombot körpályán mozgó program a 39. ábrán látható.



39. ábra: Pozicionálás *Property Node* segítségével.

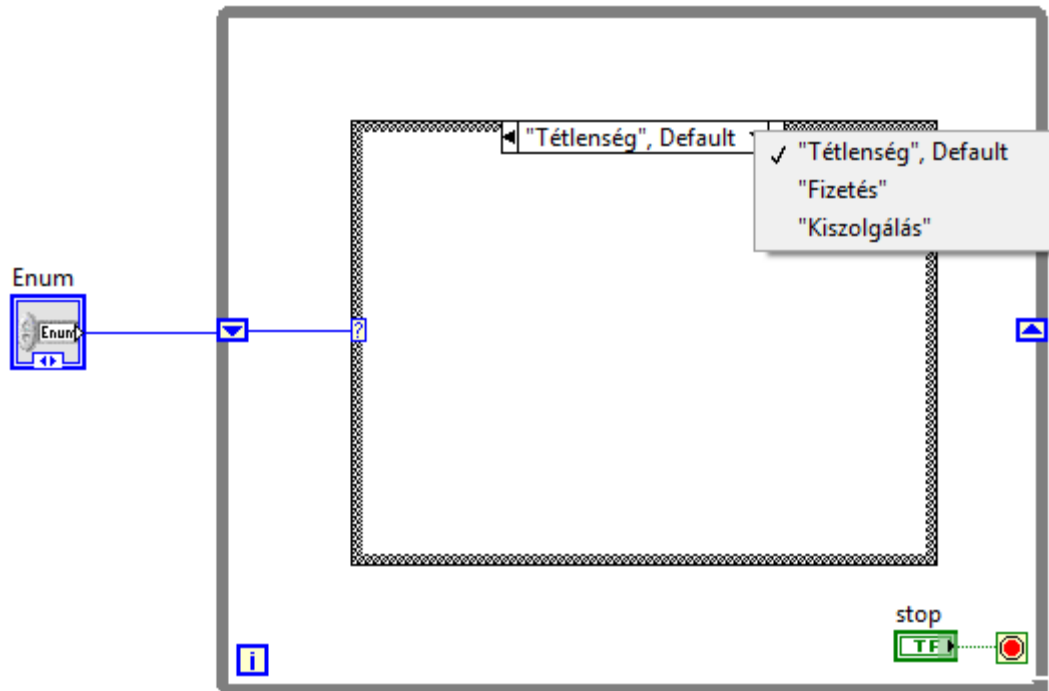
3.2. Állapotgép / State Machine

A *State Machine*, magyarul állapotgép egyfajta modell, melyet gyakran alkalmaznak programfejlesztésre. Az állapotgép véges állapotból felépülő folyamatot reprezentál, az egyes állapotok közötti átmenetek lehetőségével. Tekintsünk erre egy egyszerűbb példát (melyet később meg is valósítunk): egy italautomata. A működését egyszerűsítsük le három jól elkülöníthető fázisra:

- **Tétlenség:** a program várakozik addig, amíg a felhasználó ki nem választja a számára szükséges italt;
- **Fizetés:** a program addig tartózkodik ebben az állapotban, amíg a felhasználó megfelelő mennyiségű pénzt nem dobott a gépbe;
- **Kiszolgálás:** a gép visszajárót ad, és kiadja a választott italt.

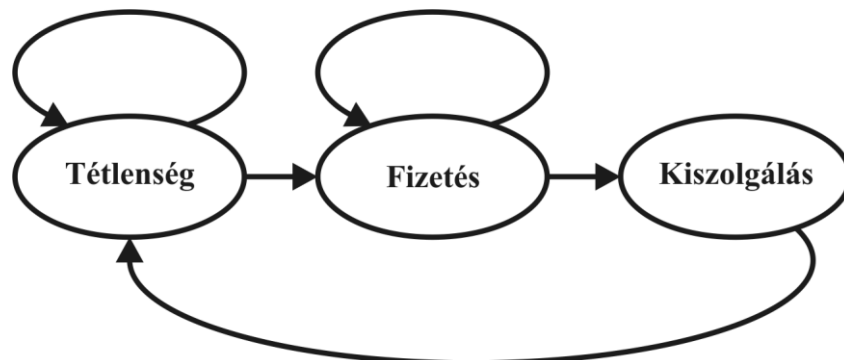
Állapotgépet sokféleképpen lehet megvalósítani LabVIEW környezetben, ezek közül is rengeteg készen elérhető az interneten böngészve. Ezek közül most talán a legegyszerűbb lehetőség bemutatása következik. A program 3 állapotot különböztet meg, most viszont az állapotok megkülönböztetése nem maradékos osztással, vagy egyéb hasonló módon fog történni, mivel minden egyes állapotban el kell azt dönteni, hogy vajon tovább lehet-e lépni egy másik állapotba, vagy sem. A program jobb átláthatóságáért *Enum Control* segítségével lesznek megkülönböztetve. Az *Enum Control* listája szerkeszthető a saját menüjén belül. Az *Enum Control* a *Controls Palette* → *Ring & Enum* menüpont alatt található. Az elemre jobb egérgombbal kattintva a *Properties* menüpont választásával felugró ablakon belül az *Edit Items* fül alatt hozunk létre egy listát a fent említett állapotok neveiből! Az egyes állapotokat a későbbiekben ez a lista fogja meghatározni. A *State Machine* vázát egy *Shift Register*-rel ellátott *While* ciklus, és egy benne elhelyezett *Case* struktúra alkotja (lásd: 40.

ábra). Huzalozzuk az *Enum Control*-t a *Shift Register*-be a cikluson kívül, ezzel kezdeti állapotot is megadva, tehát válasszuk ki az *Enum Control* listájából a **Tétlenség** állapotot a nyilak segítségével!



40. ábra: Egyszerű állapotgép váza.

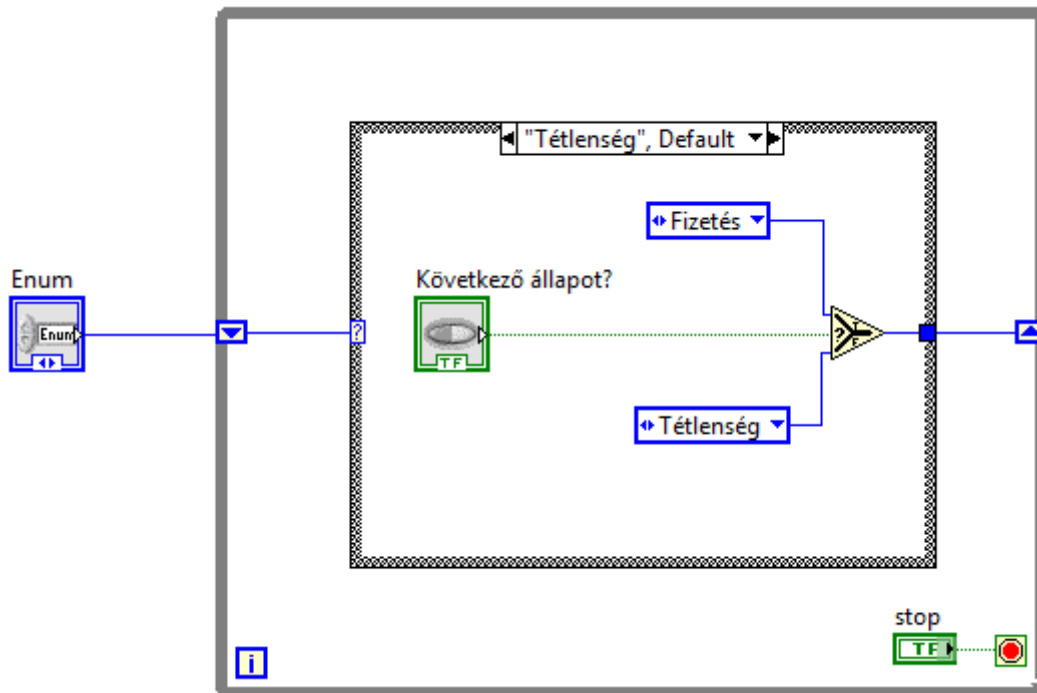
A rendszer állapotátmenetei is legyenek a lehető legegyszerűbbek, **Tétlenség** állapotból csak önmagába és a **Fizetés** állapotba, **Fizetés** állapotból csak önmagába és a **Kiszolgálás** állapotba, **Kiszolgálás** állapotból pedig csak a **Tétlenség** állapotba lehessen jutni (lásd: 41. ábra).



41. ábra: A megvalósítandó állapotátmenetek.

Minden egyes iterációban az adott állapotban (tehát a *Case* struktúrán belül) el kell dönteni, hogy mely állapotba jusson a program, s ezt az értéket a *Shift Register*-en keresztül a következő iterációnak át kell adni. Természetesen a *Shift Register*-nek csak ugyanolyan típusú adatot adhatunk, mint amilyen típusú adattal a kezdeti értéket megadtuk. Ehhez *Enum Constant* értéket kell létrehozni, de mivel az *Enum* egy számozott lista, melynek *integer* a típusa, *integer* számokat is elfogad a *LabVIEW*. Ez annyit jelent, hogy nem számít, hogy a *Shift Register*-be konstans 1-es értéket huzalozunk, vagy egy *Enum Constant* értéket **Fizetés** állapotba állítva, az eredmény ugyanaz. Itt szögezném le, hogy a listaelemek 0-s sorszámmal kezdődnek. *Enum Constant* értéket úgy lehet elhelyezni, hogy a már meglévő kontrollra jobb

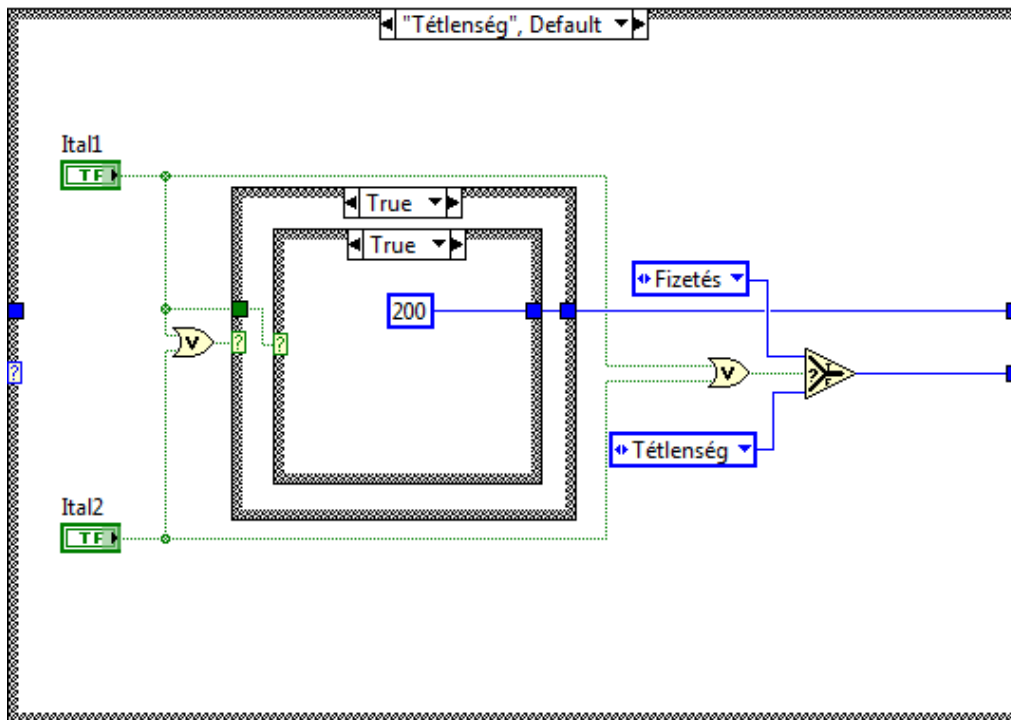
egérgombbal kattintva a kontroll számára egy konstans értéket hozunk létre. Ez egy teljesen független konstans lesz listaelemekkel, melyek értékét a konstansra rákattintva és a kívánt elemet kiválasztva lehet beállítani. Hozunk létre minden állapotban az állapotátmeneteknek megfelelően a 42. ábrán a *Case* struktúrában láthatókat!



42. ábra: Az állapotváltás, és annak feltétele.

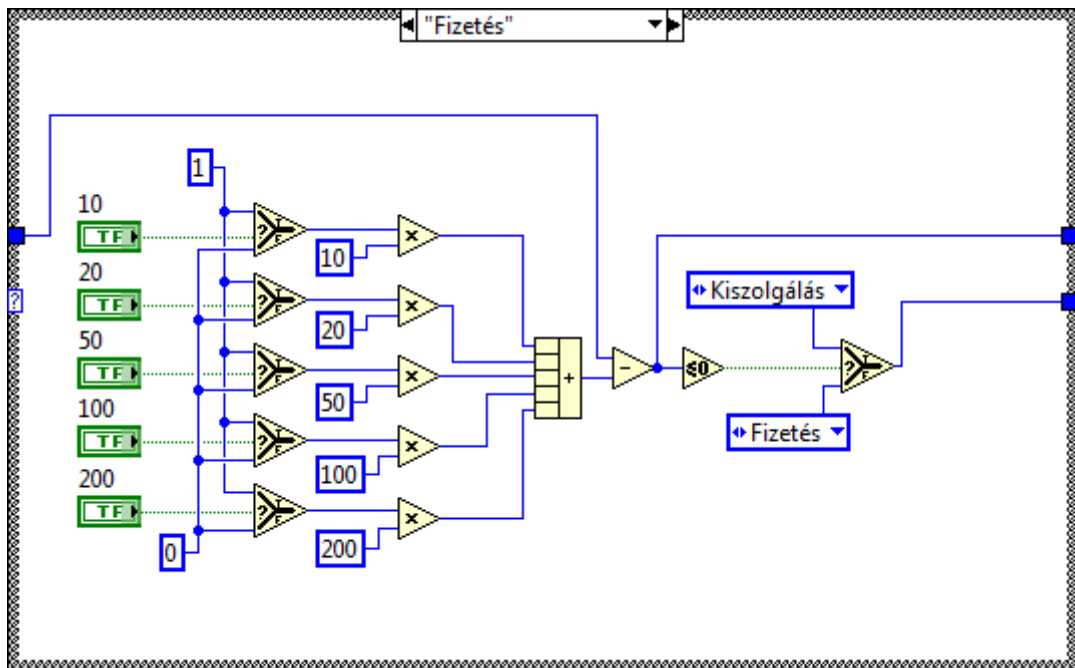
Az ábra azt mutatja, hogyha a „Következő állapot?” nevű feltétel teljesül, akkor a *Select* függvény a **Fizetés** állapotba állított konstans értékét engedi a kimenetre, s jut vissza a *Shift Register* által a következő iterációban a *Case* struktúra állapotválasztó részére. Természetesen amíg a feltétel nem teljesül, a *Select* függvény a **Tétlenség** értéket juttatja tovább, tehát a gép marad az aktuális állapotában. Ez a feltétel például úgy teljesülhet, hogy az automata felhasználója választ egy italt. Hasonlóan a **Fizetés** állapotban például addig marad a gép, amíg a kellő összeget nem dobta a gépbe a felhasználó. Ezek alapján készen van, egy alapvető állapotgép, viszont még semmilyen használható funkciója nincs. Lássuk el az italautomata állapotait alapvető funkciók ellátására képes programrészletekkel! A megvalósítandó program állapotai az alábbi funkciókkal rendelkezzen:

- **Tétlenség:** a gép kétféle itallal tud szolgálni. Az egyik 200 a másik pedig 300 forintba kerül. Amikor a felhasználó kiválaszt egy italt, a gép átlép a **Fizetés** fázisába. A gépnek meg kell jegyeznie, hogy melyik tétel lett kiválasztva, vagy legalább annyit, hogy az mennyibe került. Ez természetesen egy másik *Shift Register* segítségével egyszerűen megvalósítható. A két nyomógomb, melyek segítségével az ital kiválasztása történik, egy-egy egyállású logikai kontroll legyen, mivel ha visszatér a gép a **Tétlenség** állapotba, nem szabad újra ugyanazt az italt kiadnia. Ennek megvalósításának egy lehetséges verzióját (csak az állapotot) mutatja a 43. ábra. A belső *Case* struktúra másik ága 300-as konstans értéket takar, míg a külső 0-s értéket juttat a kimenetre, amely értéknek természetesen nincs jelentősége, mivel úgyis ebben az állapotban marad a program.



43. ábra: A **Tétlenség** állapot egy lehetséges megvalósítása.

- Fizetés:** A vásárlónak legalább annyi pénzt kell bedobnia a gépbe, mint amennyi a kiválasztott ital ára. A gép fogadjon el 10, 20, 50, 100 és 200 forintos pénzért! A pénzért bedobását reprezentálja egy nyomógomb lenyomása! Az egyes pénzért számára külön-külön nyomógombot kell létrehozni, és ezek a nyomógombok szintén egyállású kapcsolók legyenek, hogy egy gombnyomásra egy pénzért csak egyszer lehessen bedobni. A program addig marad ebben az állapotban, amíg ki nem lett fizetve az ital. Az ital ára – az előző pontban ismertettek szerint – vissza van csatolva, így az ebben az állapotban is elérhető. Vonjuk ki minden egyes gombnyomásnál a bedobott pénzért értéket, az eredményt pedig csatoljuk vissza! Mivel egy érték visszacsatolás már létezik, nem szükséges új *Shift Register* létrehozása, a kivonás eredményét rá lehet huzalozni ugyanarra a *Shift Register*-re, az állapot kilépési feltétele pedig lehet az, hogy a kivonás eredménye legyen kisebb vagy egyenlő, mint 0. A 44. ábrán látható programban ennek megvalósítása úgy történik, hogy amennyiben valamelyik nyomógombot lenyomja a felhasználó, a hozzá tartozó értéket 1-el, míg minden más értéket 0-val szoroz meg (egyszerre két nyomógomb lenyomása nem lehetséges, mivel a program futása nincs késleltetve ennél a pontnál). A szorzások eredményeinek összege levonásra kerül az ital árából / a még hátralevő összegből. A következő állapotba lépést ismét egy *Select* függvény bemenetére jutó feltétel határozza meg. Az ital ára és a bedobott pénzmennyiség közötti különbséget pedig az így megvalósított program visszacsatolja, ezáltal megvan a visszajáró értéke is.



44. ábra: A **Fizetés** állapot egy lehetséges implementálása.

- **Kiszolgálás:** e lépésben az italt kiszolgáltatta a gép a felhasználónak, és kiadja a visszajáró értékét. Italkiadó mechanizmus programozására e modell esetében nincs szükségünk, a honlapon található programban egy egyszerű „csapolás” animációt mutat a program a szemléltetés kedvéért. A program kiírja a visszajáró értékét, várakoztatja pár másodpercig ezt az állapotot, majd visszalép a **Tétlenség** állapotba.

Természetesen a programot sokkal több funkcióval is el lehetne látni: többfajta ital, többféle elfogadott érme / bankjegy, megadható egy érmekészlet, és ez alapján a visszajáróként kiadott különböző értékű pénzermék számát is meg lehetne határozni... Azok a hallgatók, akik LabVIEW környezetben szeretnék modellezni egy folyamatot TDK dolgozat, Szakdolgozat vagy Diplomamunka keretein belül, majd hogyanem elengedhetetlen az állapotgépek működésének és megvalósításának ismerete.