Siemens gyártmányú PLC-k programozása
Önálló laboratórium beszámoló

Írta: Csábrádi Attila

Az önálló laboratórium feladata:

Ismerkedés a PLC (programozható logikai vezérlő) alapú rendszerekkel. Ezen belül a beszámoló a következő területeket érinti: A vezérlő architektúrális felépítése. A PLC-t vezérlő szoftver tulajdonságai. Olyan jellemzők hangsúlyozása, melyek megkülönböztetik a PLC-t a többi processzor alapú rendszertől. A létradiagramm (LAD) alapú programozás elvei és használata. Fontos függvények leírása. Esettanulmány keretein belül az állapot alapú vezérlés bemutatása-

Mi is tulajdonképp a PLC?

A mai világ a mikroprocesszorok világa. Elenyésző azoknak az alkalmazásoknak száma, ahol nem alkalmazzák ezen eszközöket. Egy átlagember ha számbaveszi az elektronikus "kütyüit", akkor rájöhet, hogy legalább 2-3 mikroprocesszor által vezérelt eszközzel rendelkezik: okostelefon, táblagép, notebook, okos TV. Tehát ezek az architektúrák nem kerülhetőek meg a 21. században.

Természetesen processzorok irányítják a személyi számítógépeket is, de ezek általános célterületű eszközök, ezeket nem tárgyaljuk. A fókuszunkba azon processzoros rendszereket tesszük, melyek egy specifikus feladat végrehajtására vannak tervezve, illetve felprogramozva. Ezek a feladatok nagyrészt valamilyen ipari folyamatok, amiről a mikroprocesszoros vezérlés adatokat gyűjt, és azok alapján beavatkozik a folyamatba. Konyhanyelven szólva, olyan "számítógépekkel" foglalkozunk, melyeknek nincs képernyője, se billenyűzete. Egyszerű megfogalmazás, de jól összefoglalja a lényeget.

Sokféle mikroprocesszoros architektúra létezik, mind rendelkezik előnyökkel és hátrányokkal. Annak érdekében, hogy rávilágítsunk a PLC-k létjogosultságára az architektúrák rendszerében, foglalkozzunk egy kicsit a "konkurenciával".

1. FPGA (programozható logikai mátrix)

Az FPGA kakukktojásnak is tekinthető a felsorolásban, mivel ebben a rendszerben nem található processzor (egy végrehajtó egység, mely az utasításokat végrehajtja). Ez az eszköz úgynevezett logikai cellákból épül fel. Egy logikai cella pár igazságtáblából (look-up-table: LUT), D-flip-flop-ból (Dff) épül fel. Egy LUT pár bit Karnaugh-tábla alapú kezelését tudja megvalósítani, melyeknek kimenete a Dff bementére van kötve. Egy logikai cella ez alapján csak egyszerű funkciókat tud ellátni, de sok cella megfelelő összekötésével komolyabb struktúrák is elérhetőek.

Az FPGA-k programozási nyelve a Verilog, mely egy hardverleíró nyelv. Szintaktikája hasonlít a C nyelvhez, olyan módosításokkal, hogy a digitális rendszerek elemeinek leírása egyértelmű legyen. Például a D-flip-flop megvalósítása:

```
always @ (posedge Clock) begin
        D <= Bemenet;
end
```

assign Kimenet = D;

A fenti kód egy D változó értékét állítja a bemenet értékére a rendszer órajelének felfutó éleire, majd ezt a változót a kimenethez rendeli. A fejlesztőkörnyezetben a kívánt rendszer leírását követően a Verilog-kód alapján szintetizálódik a digitális hálózat az FPGA-n belül a logikai cellák összekapcsolásával. Így az adott feladathoz alkalmazkodó vezérlést tudunk fejleszteni, mely bármikor újrakonfigurálható. Említésre méltó az is, hogy a Verilog-al akár mikroprocesszor is szintetizálható az FPGA-n belül.

Egy FPGA sebessége, illetve teljesítmény/ár aránya is kisebb a többi opcióhoz képest, amit ellensúlyoz az FPGA-s rendszerek rugalmassága.

2. Mikrokontroller

Ez a legnagyobb processzor-architektúra csoport, sok felhasználási területtel.

Az első típus a 80-as években fejlesztett Intel 8051-es processzor volt, mely óriási sikereket ért el, a mai napig alkalmazzák. Tipikus fejlesztőnyelve az Assembly, mellyel nagy sebességet lehet elérni egyszerűsége miatt. Bonyolultabb rendszerek programozása kényelmetlen ily módon, mert az Assembly-vel minden regisztert és bitet állítanunk kell a fejlesztés során, de cserébe a program teljes mértékben kézben van tartva, mert pontosan végigkövethető minden értékállítás.

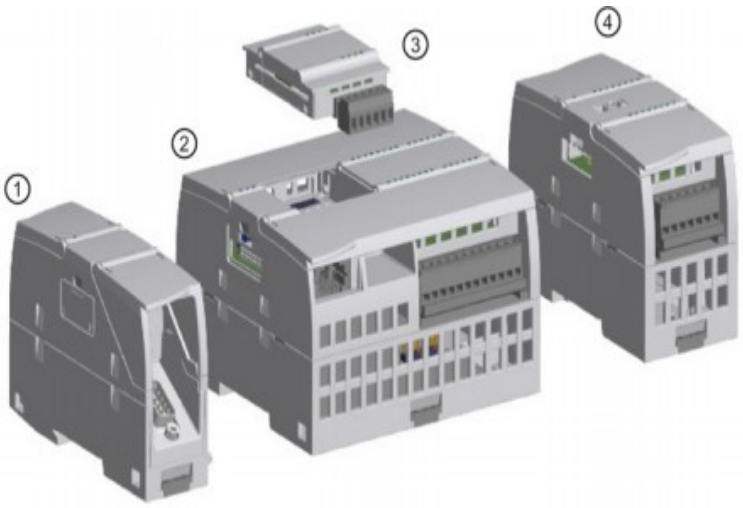
A 8051-es továbbfejlesztésének tekinthető az ARM processzor. Erre már magasabb szintű nyelvekkel szokás fejleszteni, tipikusan C-t használnak. Egy fejlesztőkártyához elérhető rengeteg előre létrehozott függvény, melyekkel bonyolultabb rendszerek kevés programsorral létrehozhatóak. A függvények elnevezése konzisztens a funkciójukkal, de a dokumentációk a függvény minden tulajdonságát tartalmazza. Persze így nincs teljesen kézben tartva a program futása, mivel egy egyszerűbb függvény is rengeteg regisztert állít a háttérben, ezek követése nem rentábilis.

Megemlítendő még a DPS (jelfeldolgozó) processzorok, melyek egyszerűbb eszköztárral rendelkeznek, de ezek a leggyorsabb architektúrák. Az egész rendszer felépítése alá van rendelve a bejövő jelek gyors fogadásának. Elsősorban spektrumanalízisre, szűrésre alkalmazható.

Látható, hogy mindegyik típusnál valamilyen speciális tudásra van szükség az adott rendszer képességeinek maximális kihasználásához. Az FPGA-nál tervezési tapasztalat szükséges (ne legyen latch; szinkronitáshiba stb.), a 8051-esnél az Assembly tudása mellett regiszterszintű tervezés szükségszerű, ARM-nél a C nyelv legalább középszintű elsajátítása kritikus (pointerek biztoskezű kezelése, paraméterátadások stb.), DPS-nél a spektrumanalízis mögött álló komplex matematika értése elvárható.

És most térjünk vissza a PLC-hez! Legfőbb előnye, hogy felprogramozása semmilyen különleges előzetes tudást nem igényel. Később láthatjuk majd, hogy a programozásnak több formáját alkalmazhatjuk, többek közt a szemléletes létradiagramot. Minimális tudással és tapasztalattal összetett rendszerek alkothatóak. A C nyelvhez képest jóval tömörebb nyelvezettel bír. Míg az ARM esetében egy egyszerű számláló használata is több, mint 50 sorból áll (számláló beállítása, indítása, hozzá tartozó megszakítás beállítása, megszakítás lekezelése), addig az PLC utasításlistás nyelvével akár egy paranccsal beállítható és indítható egy bonyolultabb számláló is. Az előre leprogramozott funkcióblokkok (C-beli függyvények PLC-s megfelelője) segítségével gyorsabb fejlesztés elérhető. Például egy bonyolult vezérlés beállítása (PID) szintén egy paranccsal megtehető, ha tudjuk a PID vezérlő paramétereit.

Foglalkozzunk egy kicsit a külső kialakításokkal! A mikrokontrollereket általában fejlesztőkártyára szerelve árusítják. Az alkalmazási területtől függően ehhez kiegészítő kártyát kell tervezni, illetve az alkatrészeket arra ráforrasztani. Ezzel szemben a PLC kimenetei nem nyomtatott áramkörök huzalozásán keresztül kapcsolódik a vezérlendő eszközhöz, hanem szabványosított jelvezetékekkel. További perifériák pedig könnyen a központi vezérlőhöz kapcsolhatóak a 'rack' segítségével. A képen látható módon az eszközök egy fémsínre erősíthetőek. A PLC-k masszív felépítésűek, külső házuk mechanikai, termikus, elektromágneses védelmet nyújtanak. Robosztusságuk miatt az ipari környezet közelében telepíthetőek, így minimalizálható a jelvezetékek hossza.



A PLC-k alapelvei

Ha a PLC-k programvégrehajtását akarjuk jellemezni, akkor a legfontosabb kulcsszó a CIKLIKUSSÁG. A programozott feladatok minden ciklus alatt végrehajtódnak. Egy ciklus alatt a következők történnek: kimenetek írása; bemenetek olvasása; a felhasználói program utasításainak végrehajtása; rendszerkarbantartás elvégzése.

A fent bevezetett ciklusra az angol terminológia a 'scan cycle' kifejezést alkalmazza. A továbbiakban scan-ként fogunk rá hivatkozni, a PLC-re pedig a CPU (central processing unit) kifejezést is fogjuk alkalmazni.

Egy ciklus során minden kommunikációs pont (analóg, digitális, kimenet, bemenet) leképeződik egy belső memóriaterületre, amit folyamatképnek nevezünk. Ez a folyamatkép tulajdonképp egy "pillanatfelvétel" az összes fizikai bemenetről és kimenetről, amelyekkel a vezérlő kapcsolatban van.

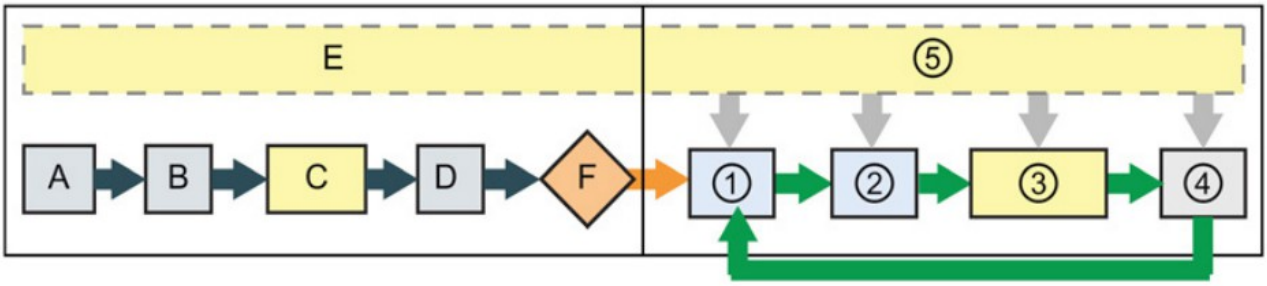
A PLC beolvassa a bemeneteket a felhasználói program végrehajtása előtt, és azok értékeit letárolja a folyamatkép bemeneti partíciójában (I-memória). Ezzel biztosítva van, hogy ezek az értékek konzisztensek (állandóak; a program elején elhelyezkedő utasítás ugyanazzal a bemeneti adatokkal dolgozik, mint a program végén elhelyezkedő utasítás) lesznek a felhasználói utasítások végrehajtása során.

A felhasználói program elvégzése után a CPU a program kimeneti eredményeit eltárolja a folyamatkép kimeneti partíciójában (Q-memória), tehát nem közvetlenül az egyes fizikai kimeneteket állítja. A scan egy külön részfeladata a kimeneti partíció alapján a fizikai kimenetek vezérlése.

A fent részletezett folyamat biztosítja, hogy a programlefutás során a bemeneti és kimeneti logikai értékek konzisztensek legyenek egy scan alatt. Így megakadályoztuk a fizikai kimenetek "ugrálását", mivel a program egy kimeneti értéket egy scan során többször is állíthat. A kimeneti értékek tehát csak jól meghatározható időpontokban vehetnek fel új értékeket.

A scan jobb megértésének érdekében vizsgáljuk meg a folyamatábráját!

A folyamatábra teljes megértéséhez tisztáznunk kell a megszakítás (interrupt, IT) fogalmát. Az IT a processzoros vezérléseknél gyakran használt tervezési eljárás. Az elvégzendő feladatoknak különböző fontosságot (prioritást) tulajdoníthatunk. Logikánk szerint a magasabb prioritású feladatokat hamarabb el kéne végeznünk. Egy program végrehajtását megszakíthatjuk egy IT-vel, hogy egy fontosabb feladatot elvégezhessünk azonnali hatállyal. Ha végrehajtottuk a megszakítás által előírt utasításokat, akkor folytathatjuk a program fő folyását. Például: Egy motor vezérlési ciklusa közben jelzést kaptunk, hogy a motor túlmelegszik. Azonnal leállunk a motor további vezérlésével és elvégezzük a szituációnak megfelelő utasításokat (hűtés bekapcsolása, fordulatszám csökkentése stb.).



A folyamatábra A-B-C-D-E-F betűkkel jelölt egységei nem a scan részei, ezek csak egyszer futnak le a tápfeszültség ráadása (esetleg reset) után, ezért ezt a szakaszt indításnak (startup) nevezzük.

A: Az I-memória törlése.
B: A Q-memória inicializálása. A kezdő érték beállításfüggő. Lehet zérus, az utolsó érték, vagy valamilyen más érték.
C: A nem-megmaradó memória és adatblokkok inicializálása. Megszakítások engedélyezése. Indulási programsorok végrehajtása.
D: A fizikai bemenetek állapotának másolása az I-memóriába.
E: A startup során elmentünk minden megszakítási kérelmet, amit majd a scan során szolgálunk ki.
F: A Q-memória kimásolásának engedélyezése a fizikai kimenetekre.

1: A Q-memória kiírása a fizikai kimenetekre.
2: A fizikai bemenetek állapotának másolása az I-memóriába.
3: A felhasználói program végrehajtása.
4: Rendszer-diagnosztika végrehajtása.
5: A scan bármely részfolyamata megszakítható. IT kérelem esetén a vezérlést átadjuk az IT-t kiszolgáló részprogramnak.

A programunk nem csak a folyamatképen keresztül tud kommunikálni a fizikai ki- és bemenetekkel (de ajánlott). Lehetséges a fizikai bemenetek azonnali beolvasása, de ezzel nem frissítjük az I-memóriát. Fizikai kimenetek azonnali írása frissíti a Q-memóriát.

A CPU a következő típusú kódblokkokat támogatja, hogy a programunknak jól átlátható struktúrát adjunk:

- Szervező blokk (organization block, OB): Megadja a program struktúráját.
- Funkció (FC) és funkcióblokk (FB): Tartalmazzák a programkódot, melyek a szükséges feladatot végrehajtják. Rendelkeznek bemeneti és kimeneti paraméterekkel is, így osztanak meg adatot a hívó blokkal. Az FB használ saját adatblokkot is, hogy eltárolja a változóit, melyeket más blokk is fel tud használni.
- Adatblokk (data block, DB): Adatot tárol, amit a többi blokk fel tud használni.

A scan minden részfolyamatát (kivéve a megszakításokat) rendszeresen kiszolgáljuk adott (szekvenciális) sorrendben. A megszakítások a prioritásuknak megfelelően lesznek végrehajtva, ha engedélyezve vannak. Egy megszakítási esemény során a CPU beolvassa a bemenetet, végrehajtja az IT-hez tartozó OB-t, írja a kimeneteket, amihez a hozzá rendelt folyamatképet (process image partition, PIP) használja.

A rendszer garantálja, hogy a scan végre lesz hajtva egy adott idő alatt, amit maximum ciklusidőnek neveznek. Ha nem így történik, akkor a CPU egy időzítéshibát generál.

A felhasználói program végrehajtása során minden programban elhelyezkedő OB sorra kerül, a hozzájuk tartozó FC-kel és FB-kel együtt. Az OB-k futási sorrendjét a számuk határozza meg, az alacsonyabb számúak előny élveznek.

A megszakítások a scan bármely részénél megjelenhetnek, megjelenésüket valamilyen esemény generálja. Ha az esemény megtörténik, a CPU megszakítja a scan-t, és meghívja azt az OB-t, amelyik az adott eseményhez van rendelve. Miután az OB végzett az esemény lekezelésével, a CPU folytatja a felhasználói program futását a megszakítás érvényre jutásának pontjától.

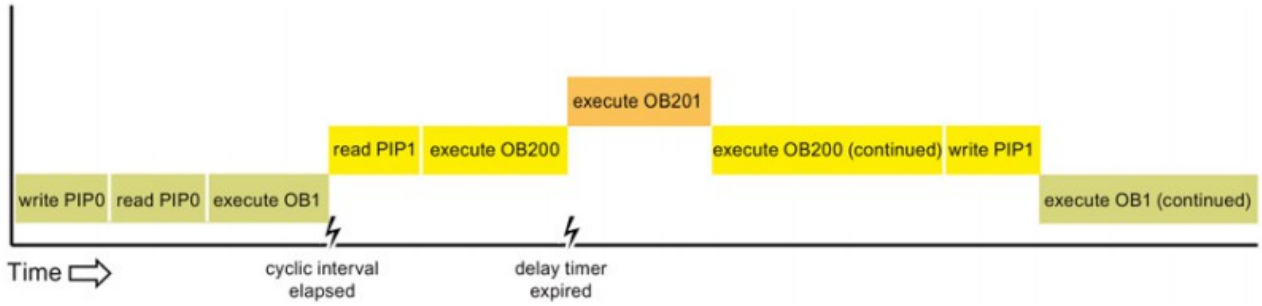
Az OB-kal tudjuk a program futását irányítani. A CPU egy adott eseménye okozza a szervező blokk futását. Az OB-k nem hívhatják egymást, és se FC, se FB nem hívhat OB-t. Csak egy megszakítási esemény indíthatja el egy OB végrehajtását. A CPU az OB-ket prioritásuk szerint kezeli, nagyobb prioritású OB-k hamarabb végrehajtódnak az alacsonyabbakhoz képest. A legalacsonyabb prioritásosztály az 1-es (ez a fő programfolyam prioritása), a legmagasabb a 26-os.

Beállítható, hogy az adott OB megszakítható legyen-e. A főprogram OB-k mindig megszakíthatóak, de az összes többi OB beállítható, hogy ne legyenek megszakíthatóak. Ha a megszakítható módot alkalmazzuk, akkor ha az OB végrehajtás alatt áll, és egy magasabb prioritású esemény lép érvénybe mielőtt az OB végezne, akkor az OB megszakad, hogy a magasabb prioritású eseményhez rendelt OB futhasson. Végeztével a megszakított OB folytatja futását. Ha egyszerre több esemény lép érvénybe, akkor a prioritásuk sorrendjében lesznek végrehajtva.

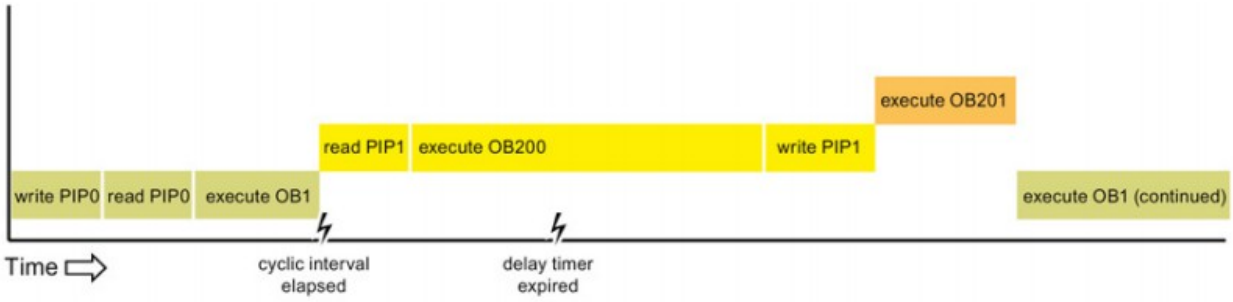
Nem megszakítható módot használva az OB a befejezéséig fut, függetlenül attól, hogy milyen prioritású esemény történt ez alatt.

Nézzünk ehhez egy illusztrációt!

Megszakítható mód



Nem megszakítható mód



Mindkét esetben megszakítási események egy ciklikus lefutású OB-t és egy késleltetett OB-t indítanak. A késleltetett OB-nek (OB201) nincs saját folyamatképe, és 4-es prioritással fut. A ciklikus OB (OB200) rendelkezik folyamatképpel (PIP) és 2-es prioritást kapott, ennek a megszakíthatósága különbözik a két ábrán. A főprogram (OB1) mindkét esetben megszakad, mivel azok mindig megszakíthatóak. Az első esetben a késleltetett OB érvényre jutásakor megszakítja az OB200-at, mivel nagyobb prioritású és az OB200 megszakítható. Az OB201 kiszolgálása után az OB200 futása folytatódik, majd annak vége után visszatér a vezérlés a főprogramhoz. A második esetben az OB200 a befejezéséig fut, csak utána kapja meg a késleltetett OB a vezérlést. A főprogram mindkét esetben ugyanakkor kapja vissza vezérlést. Nem megszakítható mód esetén a programfutás könnyebben nyomonkövethető, de a legmagasabb prioritású OB az érvényre jutásától jóval később tudott csak lefutni.

Minden CPU esemény rendelkezik egy alapértelmezett prioritással. Azonos prioritással rendelkező eseményeket a "hamarabb-jött-hamarabb-kiszolgálva" (first-in-first-out: FIFO) elv alapján ütemezi.

Néhány alapvető esemény prioritása:

- főprogram 1
- startup 1
- késleltetés 3
- ciklikus megszakítás 8
- hardveres megszakítás 18
- időzítéshiba 22-26

A PLC támogatja minden más programozási nyelveknél használatos változótípusokat, a következőkben kiemelünk párat.

Bit típusúak:

- Bool: 1 bit
- Byte: 8 bit
- Word: 16 bit
- Dword: 32 bit

Integer típusúak:

- USInt: előjel nélküli 8 bites integer
- Sint: előjeles 8 bites integer
- Uint: előjel nélküli 16 bites integer
- Int: előjeles 16 bites integer
- UDInt: előjel nélküli 32 bites integer
- Dint: előjeles 32 bites integer

Valós típusúak:

- Real: 32 bites lebegőpontos szám
- Lreal: 64 bites lebegőpontos szám

Idő típusúak:

- Date: 16 bites szám, 1990. jan. 1. óta eltelt napokat tárolja
- Time: 32 bites szám, milliszekundumok számát tárolja
- TOD (time of day): 32 bites szám, éjfél óta eltelt milliszekundumokat tárolja

Karakter típusúak:

- Char: 8 bites karakter
- String: 254 karaktert tárol

A STEP 7 alapelve a szimbolikus programozás. Szimbolikus nevet (tag) adhatunk címekhez és adatokhoz. De az adatstruktúrák teljes megértéséhez szükséges megismerni az abszolút címzést.

A CPU több módot kínál az adatok tárolására a programvégrehajtás során:

- Globális memória: A CPU rendelkezik előre kijelölt memóriaterületekkel, úgy mint I-memória, Q-memória, bit-memória (M). A bit-memória minden típusú kódblokk által elérhető korlátozás nélkül.
- Adatblokk (DB): A kódblokkok használják adatok tárolására. Az adat elérhetetlen lesz, amint a hozzá tartozó kódblokk futása befejeződik. Egy globális DB felhasználható minden kódblokk által, míg egy adott FB-hez rendelt DB az adott funkcióblokk paraméterei által strukturált.
- Ideiglenes memória: Ha egy kódblokk hívásra kerül, a CPU ideiglenes (local: L) allokál számára, amit használhat a futása alatt. A kódblokk végeztével ezt a területet újraallokálja más kódblokknak.

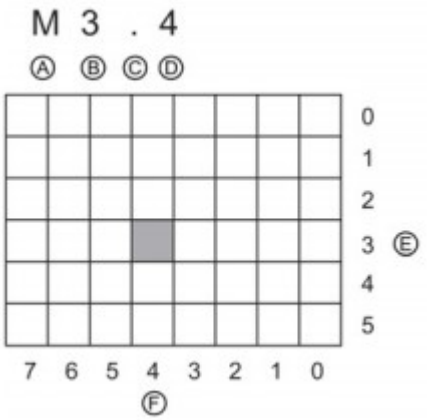
Minden memóriatípusnak egyedi címzése van. Ezen címekkel lehet elérni információkat a memóriában. Ha azonnali hozzáférést akarunk, akkor a címzéshez hozzáfűzzük a ':P' karaktereket. A közvetlen hozzáférés szabályait már fentebb részleteztük.

Az abszolút címzés három elemből áll:

- Memóriaterület (I, Q, M)
- Kért adat típusa (byte: B, szó: W)
- Az adat címe

Általában egy adott bit értékére vagyunk kíváncsiak. Ekkor nem a fenti szintaktikát használjuk. Megadjuk a memóriaterületet, a byte helyzetét és a bit helyzetét.

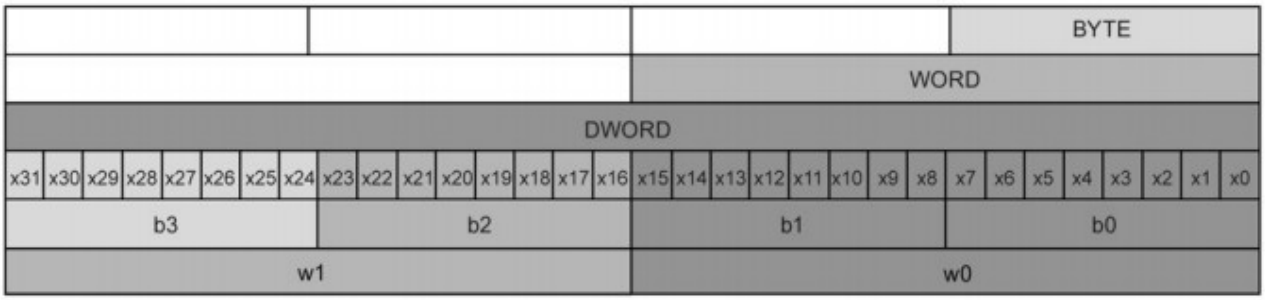
Nézzünk egy példát!



A kód alapján a bit-memória 3-ik byte-jának 4-ik bit-jét akarjuk elérni.

Ha a változókat és címeket elláttuk tag-ekkel, akkor előfordulhat, hogy a tag által hivatkozott adat egy részét (slice) szeretnénk elérni. Ez is lehetséges a következő módon: "<tag név>".x/b/w.n

Tehát megadjuk a kívánt tag-et, a hozzáférés méretét (x: bit, b: byte, w: szó), majd a kért adatnagyság sorszámát. A következő ábra illusztrálja egy tag ilyen módon történő címzését.



Látható, hogy adott hozzáférési kóddal milyen adatrészhez kapunk hozzáférést.

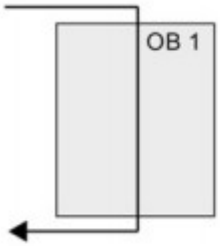
A PLC-programozás alapjai

Mikor egy felhasználói programot hozunk létre feladatok automatizálásának céljából, az utasításokat kódblokkokba szedjük (amik lehetnek: OB, FB, FC).

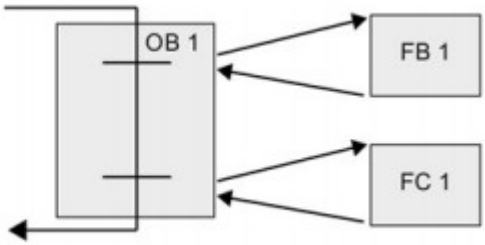
A kívánt vezérlés követelményei alapján a szoftvernek választhatunk lineáris- és moduláris struktúrát.

- Egy lineáris program sorrendben hajtja végre az utasításokat, egyiket a másik után. Tipikusan egy lineáris program az összes utasítást egy OB-ba teszi, amit aztán a CPU minden scan során végrehajt.
- Egy moduláris program adott kódblokkokat hív meg adott feladatok (task) elvégzésére. Egy moduláris struktúra létrehozásához az összetett automatizálási folyamatot kisebb taskokra bontjuk fel. Mindegyik kódblokk egy-egy részfolyamat vezérléséért felelős. Ebben a struktúrában a kívánt kódblokkot egy másik blokkból hívjuk.

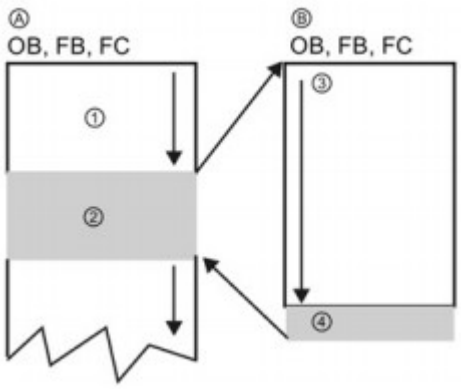
lineáris struktúra



moduláris struktúra



Az FB-ket és FC-ket úgy tervezzük, hogy egy-egy taskot elvégezzenek. Így modul-kódblokkokat alkottunk. Úgy építjük fel a szoftvert, hogy egy másik blokk hívja meg ezeket az újrafelhasználható modulokat. A hívó blokk átadja az eszköz-specifikus paramétereket a hívott blokknak. Mikor egy kódblokk egy másik kódblokkot hív, a CPU végrehajtja a hívott blokkban levő utasításokat. Miután végzett, a CPU a hívó blokk utasításainak végrehajtását folytatja.



Egy megszakítási eseményhez is rendelhetünk OB-t. Ha az esemény érvénybe lép, a CPU végrehajtja a hozzárendelt OB utasításait. Ezután a CPU folytatja a felhasználói program futtatását attól a ponttól, ahol az esemény érvénybe lépett, ami a scan bármely pontján megtörténhet.
A bal oldali ábrán látható egy tipikus megszakítás kiszolgálás. Az A főprogram fut (1), majd az esemény érvénybe lép (2), így az IT-hez rendelt OB fog futni, majd annak végeztével visszatérünk a főprogramhoz.

Feladatspecifikus kódblokkok létrehozásával egyszerűbben tervezhető és implementálható a felhasználói szoftver.

- Egy adott feladat elvégzésére újrafelhasználható blokkot hozhatunk létre, például egy motor vagy szivattyú vezérlése. Ezek a blokkok elmenthetőek egy könyvtárban, későbbi felhasználás céljából.
- Moduláris felépítésű programot könnyebb megérteni és kezelni, fejleszteni és javítani.
- A hibák keresése (debugging) is egyszerűsödik. Blokkonként haladhatunk a teszteléssel.

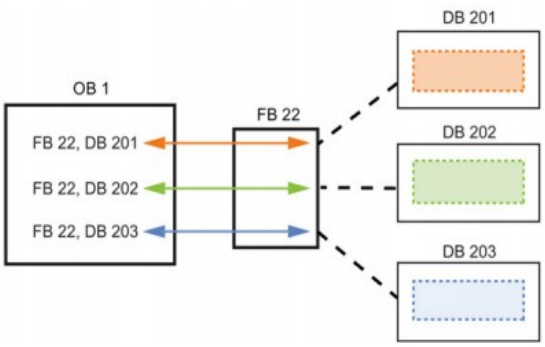
Következő lépésként ismerkedjünk meg a szoftverünk építőköveivel kicsit részletesebben!

Egy funkció (FC) szubrutinként működik. Az FC egy olyan kódblokk, mely tipikusan egy specifikus feladatot lát el az adott bemeneti értékekkel. Az FC az eredményeket a memóriában tárolja. Egy FC a program futása során többször hívható. Újrafelhasználásával egyszerűbben tudjuk kezelni a többször felmerülő feladatokat. Az FC-nek nincs hozzárendelt adatblokkja. Ideiglenes memóriát (L) használ a számításaihoz. Az ideiglenes adatok nem maradnak meg. Hogy az adatok megmaradjanak az FC végeztével is, a kimeneti értéket hozzá kell rendelni a globális memóriához, például a bit-memóriához (M) vagy egy globális adatblokkhoz.

Egy funkcióblokk (FB) olyan szubrutin, amely rendelkezik memóriával. Az FB lementi a bemeneti (IN), kimeneti (OUT) és be/kimeneti (IN_OUT) paramétereit a hozzárendelet adatblokkban (instance datablock). Ez az adatblokk az FB végezte után is tárolja az adatokat.

Általában az FB-kel olyan taskok vezérlését végezzük, amelyek nem végeznek egy scan alatt. Így a vezérlő paramétereket tároljuk, amikhez a következő scan is hozzá tud férni. Egy FB hívásakor a hozzá rendelt DB-ket is meghívjuk.

Az FB-hez rendelt adatblokkot hívási DB-nek is szokás nevezni, mivel az FB végeztével a hozzá rendelt DB az adott FB legutóbbi hívásának adatait tárolja. Egy általános felépítésű vezérlő FB-vel így több eszköz irányítása is lehetséges, mivel az FB többször hívható különböző DB-kel (egy eszköz, egy DB).



A mellékelt ábrán a főprogram OB-ja ugyanazt az FB-t hívja meg háromszor, mindig más adatblokkot használva. Így egy általános FB-vel vezérlünk három hasonló eszközt, például motort, úgy, hogy mindegyik motor státuszát egy-egy DB-ben tároljuk.

Az adatblokkokat arra használjuk, hogy a programot felépítő kódblokkok adatait tároljuk bennük. Az összes programblokk hozzá tud férni a globális adatblokkhoz, de a hívási adatblokkok FB-specifikus adatokat tartalmaznak.

A felhasználói program adatot tud tárolni a CPU erre kijelölt memóriaterületein, mint például az I-memóriába, Q-memóriában és bit-memóriában. A DB-ket gyors adatelérésre használjuk, mivel ekkor az adat magában a programban van tárolva.

A DB-ben tárolt adat nem törlődik mikor a blokk bezárul vagy a hozzá tartozó blokk futása befejeződik. Két fajta DB van:

- Globális adatblokk, az összes OB, FB, FC hozzáférhet az itt tárolt adatokhoz.
- Hívási adatblokk egy FB-hez tartozó adatokat tárolja. Részei az FB paraméterei (Input, Output, In_Out), és a statikus adatai. Az FB ideiglenes memóriája nem másolódik a hívási DB-be.

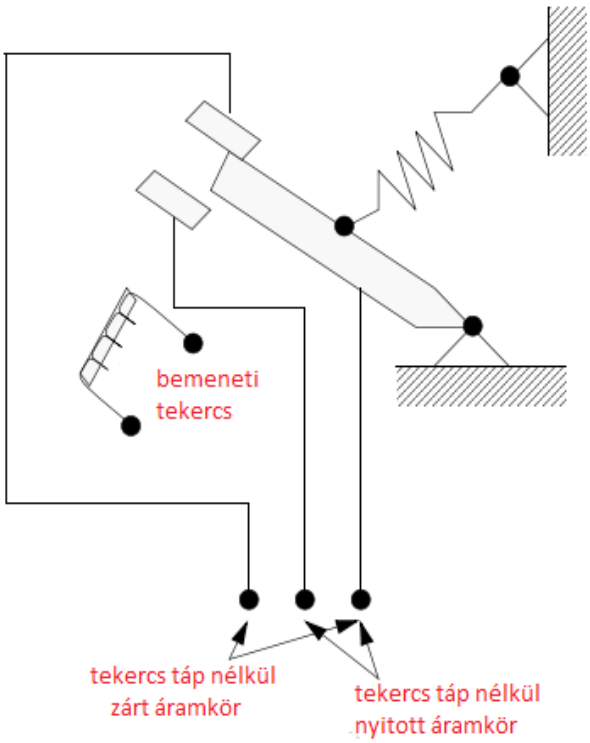
Habár a hívási DB egy FB-hez szorosan kapcsolódó adatokat tartalmazza, bármely blokk hozzá tud férni a hívási DB-khez.

Térjünk át a programozás gyakorlatiasabb részére! A STEP 7 három szabvány által szabályozott programozási nyelvet támogat:

- Létra-logika (LAD)
- Funkció-blokk diagram (FBD)
- Strukturált vezérlő nyelv (SCL)

Az FBD-vel a feladatainkat grafikusan oldjuk meg, a blokkok megfelelő összeköttetésével. Az SCL egy Pascal-szerű leíró nyelv. Mi a LAD programozással fogunk foglalkozni.

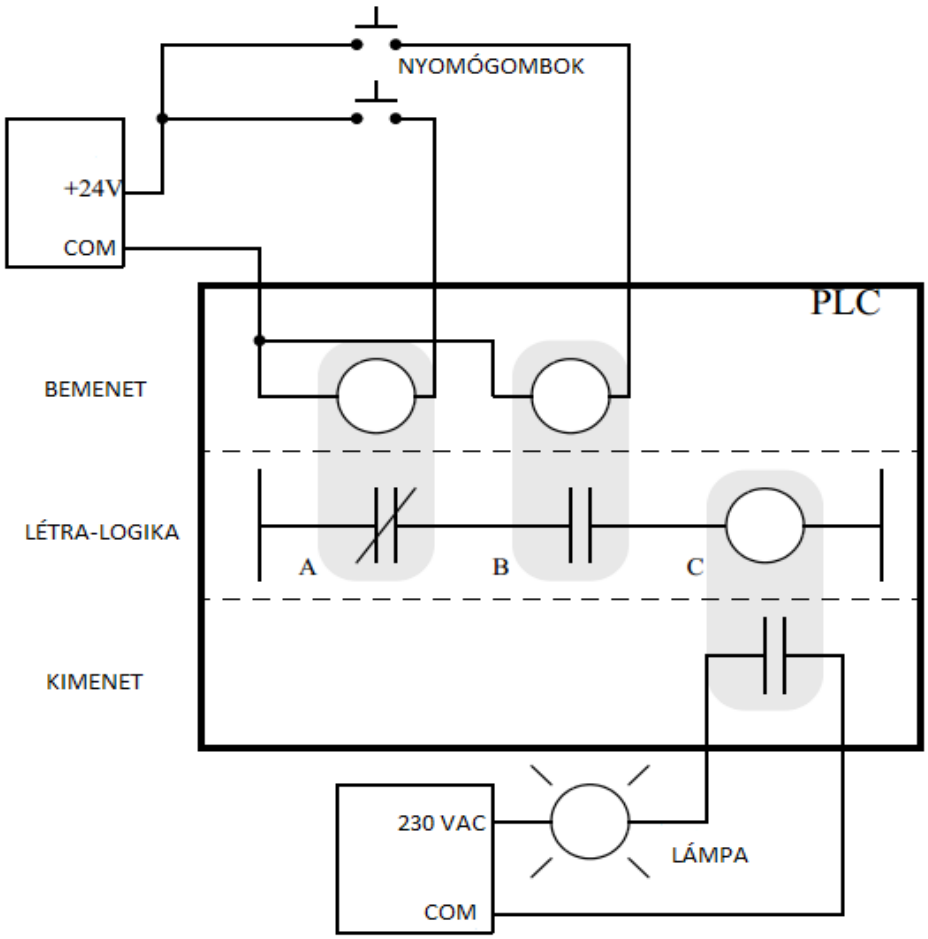
A létra-logika fejlesztése során figyelembe vették, hogy lemásolják az irányítástechnika hőskorának fő egységének, a relének a működését. Ez a döntés annak idején fontos stratégiai lépés volt, mert így megkönnyítették az átmenetet az elektromechanikus kapcsolók és a szoftveres vezérlés között. Természetesen a területen dolgozó mérnökök átképzése is leegyszerűsödött.



A modern irányító rendszerek csak a relék logikai tulajdonságait aknázzák ki.

Ha elektromos áram folyik a bemeneti tekercsen, akkor az mágneses teret generál. A tér erőt fejt ki az armatúrára, mely mozgásával vagy létrehoz, vagy megszakít kapcsolatot. Mikor lekapcsoljuk a tekercsről az áramforrást, akkor az armatúrát egy erő (rúgó vagy gravitációs) visszatéríti az eredeti pozíciójába. Azt a kapcsolatot, mely zár a tekercsre adott táp hatására, alapesetben nyitottnak nevezzük. A reléket arra használjuk, hogy egy adott áramkörrel vezérlünk egy másik áramkört, anélkül, hogy a két áramkör között galvanikus kapcsolat lenne. A létradiagramban a kapcsolókat két egymással párhuzamos, a vezetékezésre merőleges vonal -| |- reprezentálja, a tekercset egy kör -( )- jelzi.

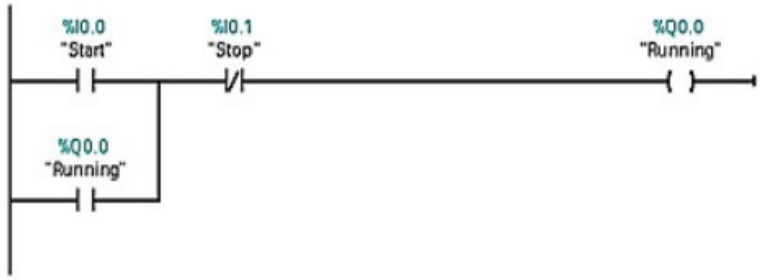
Nézzünk egy egyszerű példát!



A bemenetünk két tekercs, melyekre egy-egy nyomógombbal tudunk feszültséget kapcsolni. A szürke háttér jelzi, hogy melyik tekecs melyik kapcsolót vezérli. Az 'A' kapcsoló jelzése kicsit eltér a fentebb tárgyaltaktól. Ez azért van, mert ott a relé alapesetben zárt kimenetét alkalmaztuk, vagyis mikor a tekercsére feszültséget kapcsolunk, akkor lesz nyitva a kapcsoló. Tehát ha a felső nyomógombot lenyomjuk ('B' kapcsoló zár), az alsót pedig nem ('A' kapcsoló zár), akkor a létra-logikában ábrázolt áramkör szakadásmentes lesz, így a 'C' kapcsolóhoz tartozó relére feszültség kerül. A kimeneti kapcsoló zár, a lámpa világítani fog, mert zártuk az áramkört.

Tehát a programozási feladat adott: A PLC-ben lévő létra-logikát kell létrehoznunk. A rendszer teljesen digitálisnak is elképzelhető. Vegyük úgy, hogy a két nyomógomb a CPU bit-memóriájának egy-egy bitjének felel meg. Ha a bit 1, akkor a nyomógomb le van nyomva, egyébként pedig nincs. A 'C' kapcsolóhoz tartozó tekercset pedig vegyük a Q-memória egy bitjének! Ha a tekercs feszültség alatt van, akkor a kimenetet 1-re állítjuk, egyébként 0-ra.

Most a fent tárgyaltak alapján szerkesszünk alapvető logikai hálózatokat! Az ábrán egy egyszerű motorvezérlés logikája látható.



A hálózat kétféle elemből épül fel: kapcsolóból és tekercsből. A kapcsoló egy bit státuszát olvassa be, a tekercs pedig egy bit értékét állítja. A kapcsoló teszteli a bit bináris értékét. Alapesetben nyitott kapcsoló esetén, ha a bit 0, akkor "nem folyik rajta áram", ha 1, akkor "folyik rajta áram". Alapesetben zárt kapcsoló fordítva működik, ha a bit 0, akkor "folyik rajta áram".

Ha egy kimeneti bithez kapcsolót rendelünk (Running kimenetet egy kapcsoló olvassa), akkor az előző programlefutás eredménye befolyásolja a jelenlegi programlefutás eredményét.

A kapcsolók rendszere logikai ÉS és VAGY kapcsolatokat írnak le. Sorba kapcsolt kapcsolók ÉS logikát írnak le, párhuzamos kapcsolásuk VAGY logikának felel meg. Ez alapján elemezzük a példát!

Tegyük fel, hogy a motor nem üzemel (Running=0) és indítani szeretnénk (Start=1) és a leállítógomb nincs lenyomva (Stop=0). Ekkor a Start kapcsolója vezeti az áramot, a Stop kapcsolója szintén (alapesetben zár!!), így a motort elindítottuk (Running=1). Nézzük meg a következő scan-t úgy, hogy már nem adunk Start jelet! Ekkor a Running kapcsolón keresztül fog folyni az áram. Ha bármikor leállítanánk a rendszert (Stop=1), akkor a létrahálózatban szakadás lenne, sehogy se lehetne működtetni a motort (Running mindenképpen 0 lenne.)

Használható még invertált kimeneti tekercs is, melyet ugyanúgy ferde vonallal különböztetünk meg a normál kimeneti tekercstől, mint az alapesetben zárt kapcsolót jeleztük. Ez a tekercs akkor állítja 1-re a kimeneti bitet, ha nem folyik rajta áram.

A tekercseknek nem feltétlenül a létradiagram végén kell elhelyezkednie, akár a kapcsolók közé is tehetjük őket. A példában ha a tekercset a párhuzamosan kapcsolt kapcsolók elé helyezzük, akkor ugyanazt a működést érjük el, mivel ugyanazon feltételek mellett fog rajta áram folyni.

A programozást segítő fontosabb funkciók

Az előbbiekben megismerkedtünk a létra-programozás alapjaival, ezek után bármilyen logikai alapú vezérlést meg tudunk valósítani. De a hatékonyabb szoftverfejlesztés érdekében összetettebb rendszereket is érdemes tanulmányozni, mivel előre "gyártott" blokkok segítségével emberközelibbé tehetjük programunkat. A STEP7 számtalan ilyen funkciót támogat, a továbbiakban a fontosabbakat foglalkozunk.

Először nézzük meg a komparátor utasításokat! A komparátorok szintén kapcsolókként jelennek meg a létra-diagramban. De jelen esetben az, hogy a kapcsoló vezet vagy sem, nem közvetlenül a folyamatkép valamelyik bit-jétől függ. Ha a komparátorban deklarált logikai állítás igaz, akkor a kapcsoló vezet, különben nem. Összehasonlítás csak két ugyanolyan típusú adat közt lehetséges.
Például két adatot összehasonlító kapcsoló a következőképp néz ki:

--|Adat1 == Adat2 (Byte)|--

Tehát a kapcsolóban deklaráltuk a két komparálandó adatot, a vizsgált kapcsolatot (jelen esetben: Egyenlő-e a két adat?), az adatok típusát (byte). Ez a kapcsoló vezet, ha az Adat1 megegyezik Adat2-vel.

További kapcsolatok:

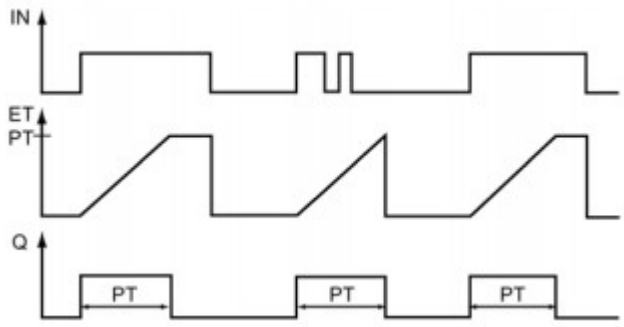
- <> nem egyenlőek
- >= nagyobb vagy egyenlő
- <= kisebb vagy egyenlő
- > nagyobb
- < kisebb

A STEP7 a következő időzítőket támogatja:

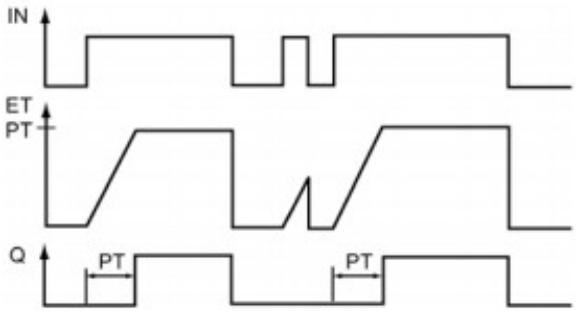
- A TP-időzítő egy pulzust generál, melynek hossza előre beállított
- A TON-időzítő kimenete aktív lesz az előre beállított késleltetés után
- A TOF-időzítő bekapcsolja a kimenetét, majd kikapcsolja az előre beállított késleltetés lejárta után
- A TONR-időzítő aktívvá teszi a kimenetét az előre beállított késleltetés után, az eltelt idő több időzítési szakaszból összeadódhat, reseteléssel állíthatjuk vissza az eltelt időt a kezdeti értékre

Az időzítők kimenetei kapcsolókkal olvashatóak, bemeneteik tekerccsel állíthatóak. Az időzítők működését jobban megérthetjük az idődiagramjuk elemzésével.

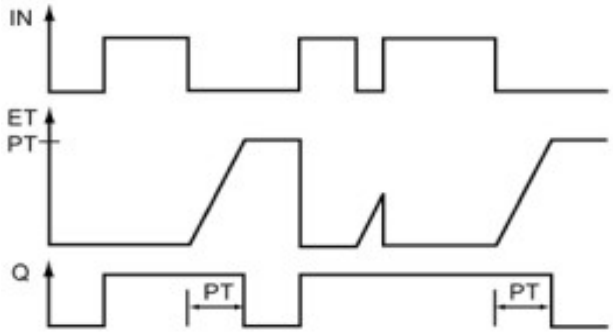
TP



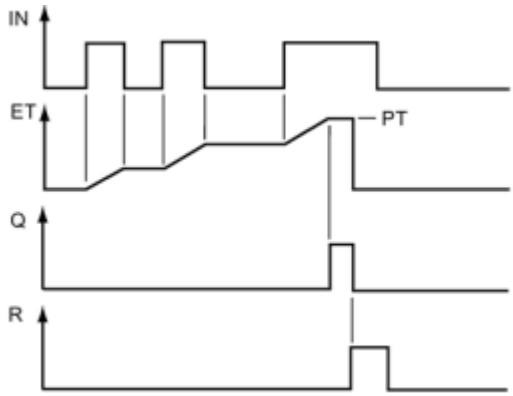
TON



TOF



TONR



Az idődiagramok elemzésével könnyen megtalálhatjuk az alkalmazásunkhoz legjobban illő időzítőt. Az IN a bemeneti logikai érték, az ET (elapsed time) az eltelt idő, a PT (preset time) az időzítés határa, a Q a kimenet, az R a reset bemenet.

Az ET és PT időzítésértékeket speciális időleíró adatformátumban kezeljük, melyekt a T# előtaggal jelezzük. Pl.: 200 ms: T#200ms; 2.3 sec: T#2s_300ms

Az időzítőket egy scan során többször is lekérdezhetjük. Az időzítő frissül, ha az időzítő utasítást végrehajtjuk, és ha az eltelt időt (ET) vagy a kimenetet (Q) paraméterként használjuk. Ez előnyös, ha a legfrissebb időzítési adatokra szükségünk van. Ha egy scan során konzisztens értékeket szeretnénk, akkor az időzítő-parancsokat azok előtt az utasítások előtt helyezzük el, amelyek felhasználják a kimeneteit, és a kimeneti értékekhez rendelt tag-eket kérdezzük le ahelyett, hogy közvetlenül az időzítő adatblokkjában tárolt adatokkal dolgoznánk.

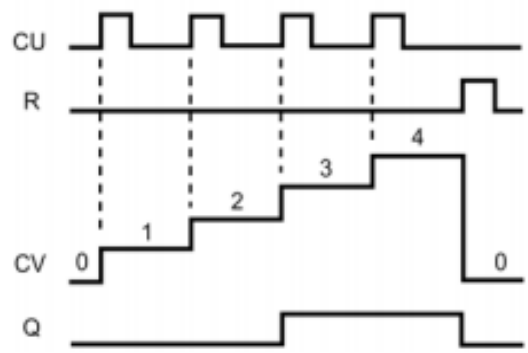
A magukat resetelő időzítők hasznosak, ha egyes taskokat időközönként kell elvégeznünk. Általában ezeket úgy hozzuk létre, hogy az időzítő bemenetére egy alapesetben zárt kapcsolót kapcsolunk, amelyhez az időzítő kimeneti bitje van rendelve. Ez a hálózat tipikusan azon programrészek előtt található, amelyek

felhasználják a kimenetet. Mikor az időzítés lejár (ET eléri a PT-t), a kimenet aktív lesz egy scan erejéig, így a kimenettől függő utasítások lefuthatnak. Az időzítő-hálózat következő futása során a bemenet nem lesz aktív, az időzítő resetelődött és a kimenet kikapcsolódott. A következő scan során így újraindul az időzítő. Fontos, hogy a bemeneti kapcsolónál a kimenetre egy tag-en keresztül hivatkozzunk, nem pedig közvetlenül az időzítő adatblokkjára. Ez azért szükséges, mert az alapesetben zárt kapcsoló közvetlen vezérlése esetén mindig frissül az időzítő és azonnal meg fog történni a resetelés, a kimeneten pedig nem fog megjelenni az aktív impulzus.

A számláló utasításokat valamilyen belső vagy külső események számlálására használjuk.

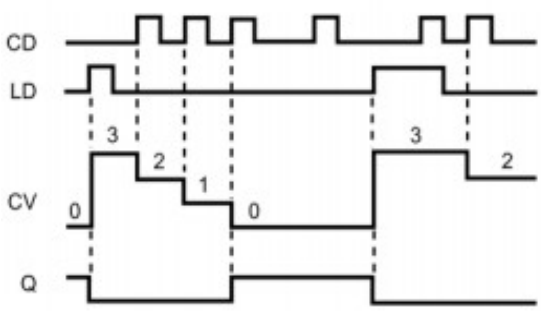
- A felfelé számláló (CTU) 1-el növekedik, mikor a bemenetén 0->1 váltás történik
- A lefelé számláló (CTD) 1-el csökken, mikor a bemenetén 0->1 váltás történik
- A fel- és lefelé számláló (CTUD) növekedik vagy csökken 1-el, attól függően, hogy a felfelé vagy lefelé számláló bemenetén történik 0->1 váltás

CTU számláló (PV=3)



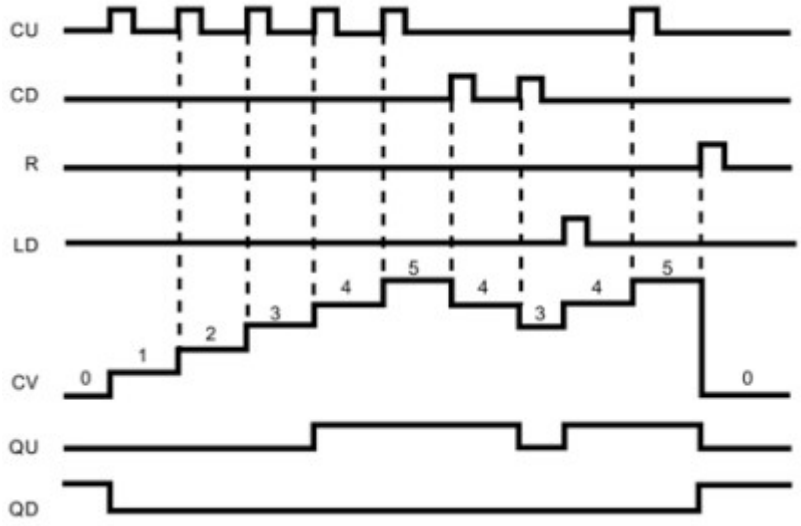
Ha a CV (számláló) paraméter nagyobb vagy egyenlő a PV (előre beállított határérték) paraméterhez képest, akkor a kimenet 1.
Az R (reset) bemeneten 0->1 váltás hatására a CV nullázódik, így a kimenet is.

CTD számláló (PV=3)



Ha a CV paraméter kisebb vagy egyenlő 0-hoz képest, akkor a kimenet 1.
Az LD bemeneten történő 0->1 váltás hatására a PV értéke betöltődik a CV-be.

CTUD számláló (PV=4)



Ha a CV nagyobb vagy egyenlő a PV-hez képest, akkor a QU kimenet 1 lesz. Ha a CV kisebb vagy egyenlő a 0-hoz képest, akkor a QD kimenet 1 lesz.
Az LD bemeneten történő 0->1 átmenet hatására a PV értéke betöltődik a CV-be.
Az R bemeneten történő 0->1 váltás hatására a CV értéke 0 lesz.

Így, hogy már megismerkedtünk bonyolultabb függvényekkel is, érdemes foglalkozni a LAD és az FBD programozások közötti kapcsolatról.

Az eddigiek során említve volt, hogy az FBD egy grafikus programozási módszer, melynek fő elemei a "dobozok", melyek egy-egy függvényt realizálnak. De ezeket a dobozokat (a továbbiakban látunk rá példát) használhatjuk LAD alkalmazása közben is, így átláthatóbbá tehetjük a munkánkat. Ezek alapján egy CTU számláló doboza rendelkezik két bemenettel (CU és R), valamint két kimenettel (Q és CV). A bemenetek kapcsolókkal vezérelhetőek, kimenetei (tipikusan a Q) felhasználhatóak más programrészek irányítására.

Nézzünk még további hasznos függvényeket!

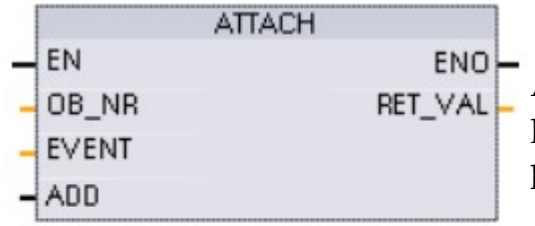
A digitális vezérlés fontos eleme a memória. Valamilyen folyamattal kapcsolatos vezérlési információt sokszor érdemes egy úgynevezett flip-flop-ban (FF) tárolni. A digitális technikában sokféle FF használatos, mi az SR-FF-t tárgyaljuk.

Az SR-FF 2 bemenettel (Set és Reset) és 1 kimenettel (Q) rendelkezik. Ha a Set=1 és Reset=0, akkor Q=1. Ha Set=0 és Reset=1, akkor Q=0. Ha mindkét bemenet alacsony, akkor a Q értéke nem változik, megegyezik az előző állapot Q értékével. De mi történik akkor,

ha mindkét bemenet egyszerre aktív? Ekkor a FF dominanciája dönt. Ha Set-domináns, akkor a kimenet 1 lesz, ellenkező esetben pedig 0. Az ábra mutatja a flip-flop FBD során használt alakját. Ennek is bemeneteit vezérelhetjük, kimenetét olvashatjuk. Az S1-bemenetjelzés alapján tudjuk, hogy Set-domináns eszközről van szó.

Előfordulhat olyan vezérlési szituáció, mikor nem egy jel szintjére vagyunk kíváncsiak, hanem a jel változására. Erre szolgál a P_TRIG függvény. A kimenete magas lesz, ha a bemeneten pozitív (0->1) jelváltás történt. A kimenet csak egy programciklusig tartja az értékét. Természetesen a negatív jelváltást is ellenőrizhetjük, erre az N_TRIG függvényt használhatjuk.

Végül ismerkedjünk meg a megszakításokat kezelő függvényekkel! Először a hardveres megszakítással foglalkozunk.



A hardveres IT két vezérlőfüggvénye az ATTACH és DETACH. Az előbbi engedélyezi a kiszolgáló rutin futását az adott esemény hatására, az utóbbi tiltja azt.

Az OB_NR bemenet egy OB azonosítót vár. Ezzel adható meg, hogy melyik OB-t akarjuk beállítani IT-rutinnak.

Az EVENT bemenettel adható meg, hogy melyik hardveres megszakítási eseményt akarjuk az OB-hez kötni.

ADD=0 esetén törlődnek az eddigi megszakítási események, felülíródnak az új által. Az ADD=1 beállítással az előző megszakítási események listáját nem töröljük, hanem bővítjük azt az EVENT bemeneten megadott eseménnyel.

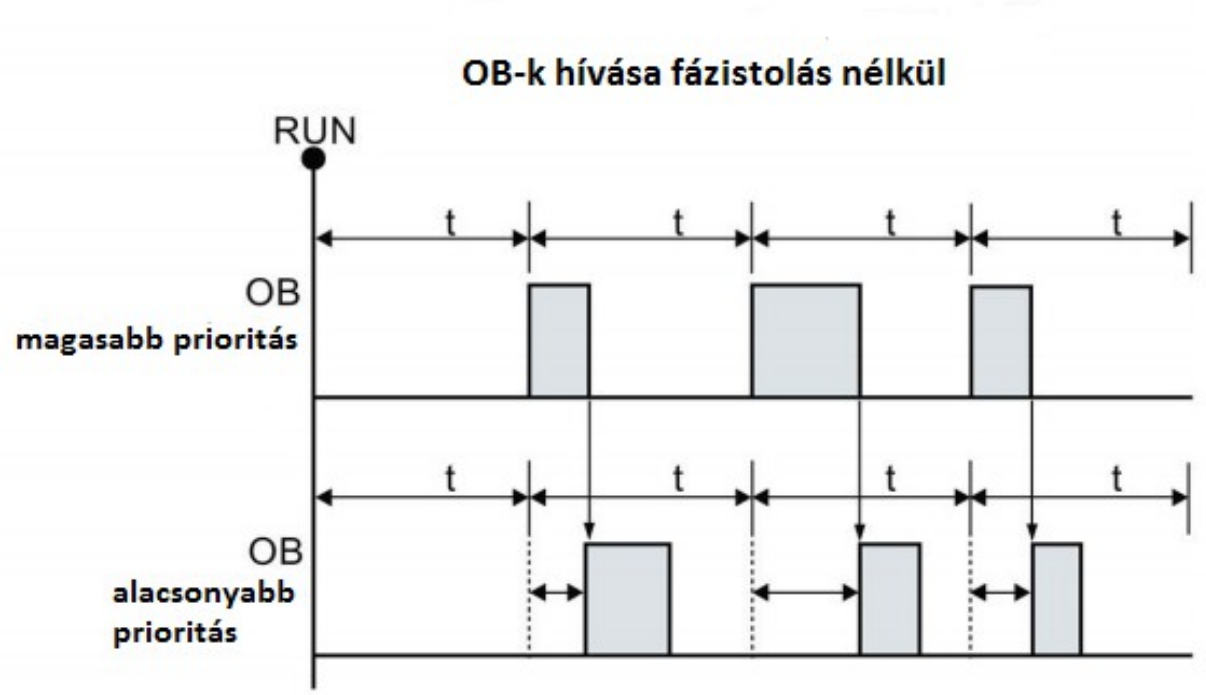
A RET_VAL kimenettel ellenőrizhető a hozzárendelés státusza. Például a RET_VAL értéke 8090, ha nem létező OB-hoz akarunk rendelni eseményt.

A másik gyakran használt IT-típus a ciklikus megszakítás, melynek beállítófüggvénye a SET_CINT. Eme függvénnyel beállítható, hogy az adott OB bizonyos időközönként lefusson, megszakítva ezzel a főprogram menetét.

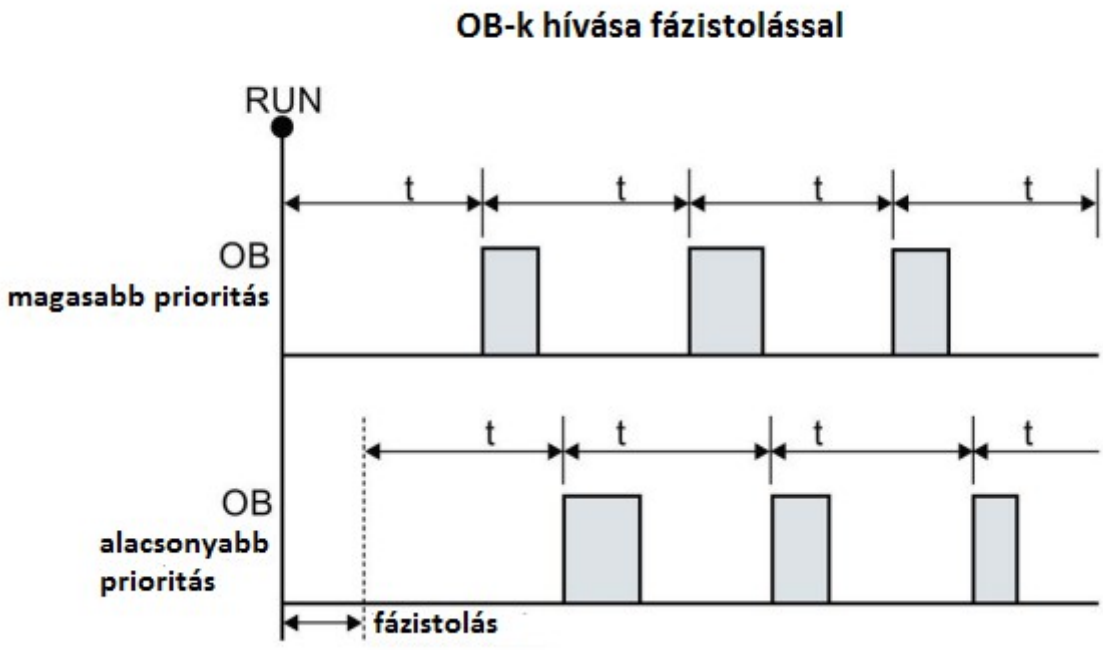
Az OB_NR és RET_VAL értelmezése megegyezik az előbb tárgyaltakkal. A CYCLE és PHASE bemenetek az időintervallum és a fázistolás értékét várják mikroszekundumban megadva. Ha CYCLE=100, akkor az OB_NR által referált programrész megszakítja a programfutást 100 us-ként. A megszakítást kiszolgáló rutin lefut, majd a vezérlés

folytatja a főprogramot a megszakítás pontjától. CYCLE=0 esetén az IT nem aktív.

A fázistolás egy olyan késleltetés, mely a beállított időintervallum előtt érvényesül. A fázistolással kisebb prioritású OB-k futását tudjuk vezérelni. Nézzük meg egy példán keresztül a fázistolás jelentőségét!



Ha alacsonyabb és magasabb prioritású OB-kat ugyanazzal a 't' időintervallummal hívnak, akkor az alacsonyabb prioritású OB csak akkor fut le, mikor a másik végzett. Tehát az alacsonyabb prioritású OB végrehajtásának kezdőpontja a magasabb prioritású OB futási idejétől függ.



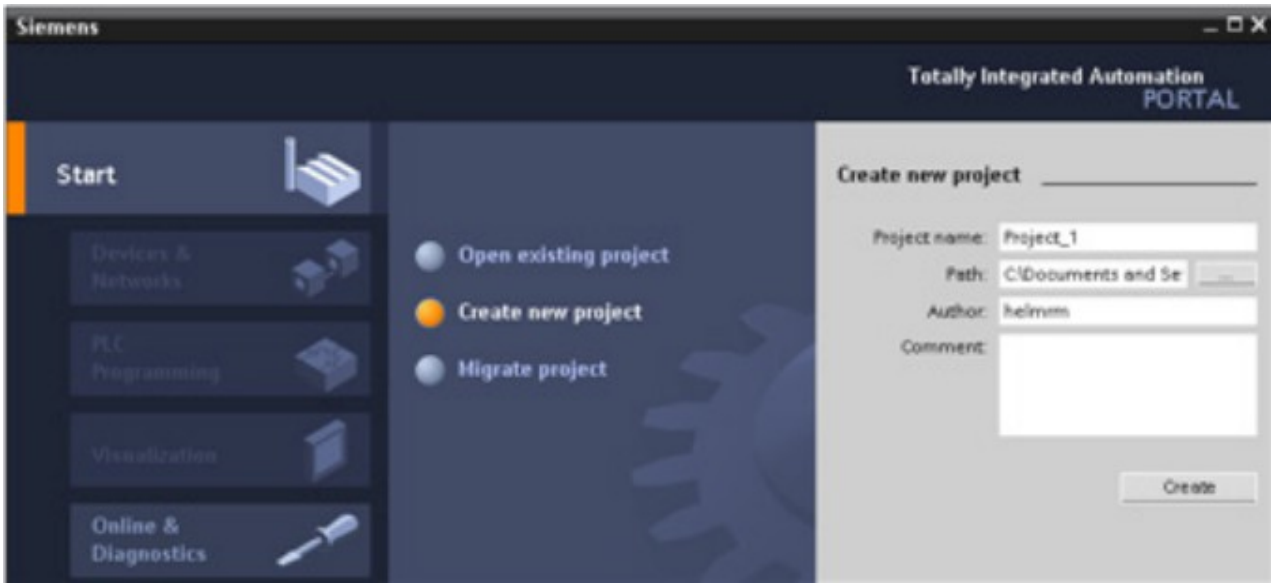
Ha el akarjuk érni, hogy az alacsonyabb prioritású OB meghatározott időpontokban kezdjen el futni, akkor olyan fázistolást kell beállítani, amely nagyobb a magasabb proritású OB futási idejénél.

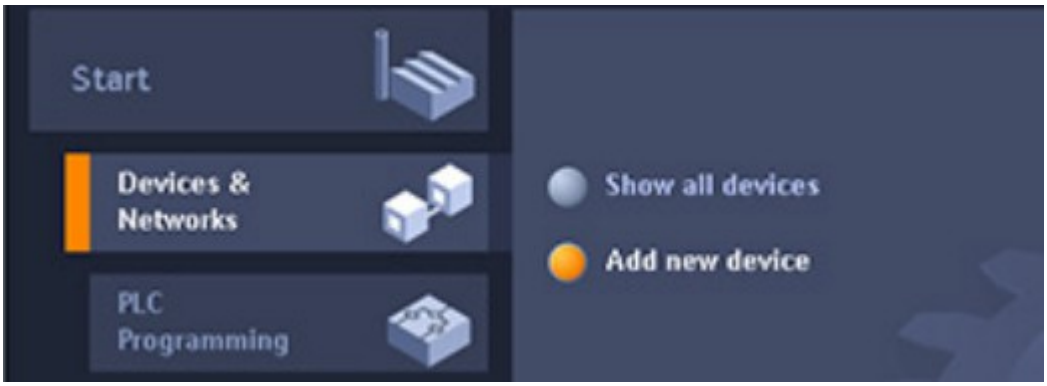
Így az OB-k futása nem függ egymástól, könnyebb követni a program futását.

Bevezető a STEP7 fejlesztőkörnyezet használatába

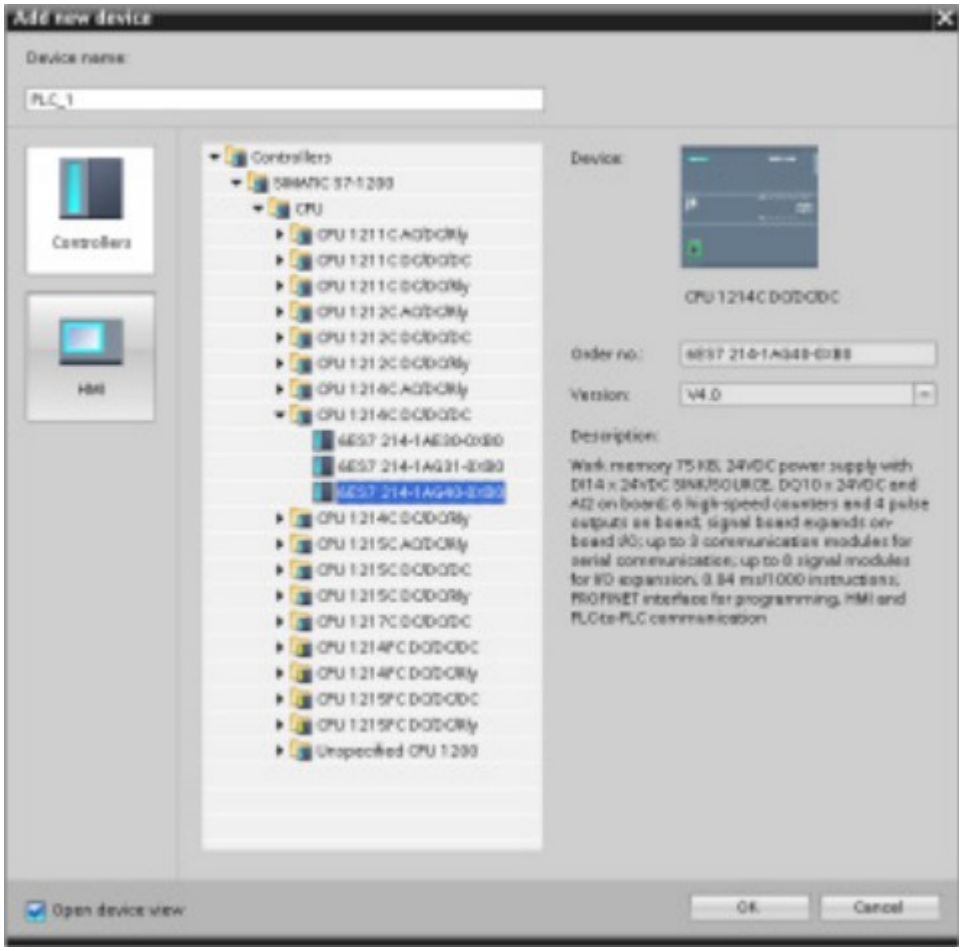
Új projekt létrehozása

A Start portálban kattints a "Create new project" gombra. Add meg a projekt nevét, majd kattints a "Create" gombra.



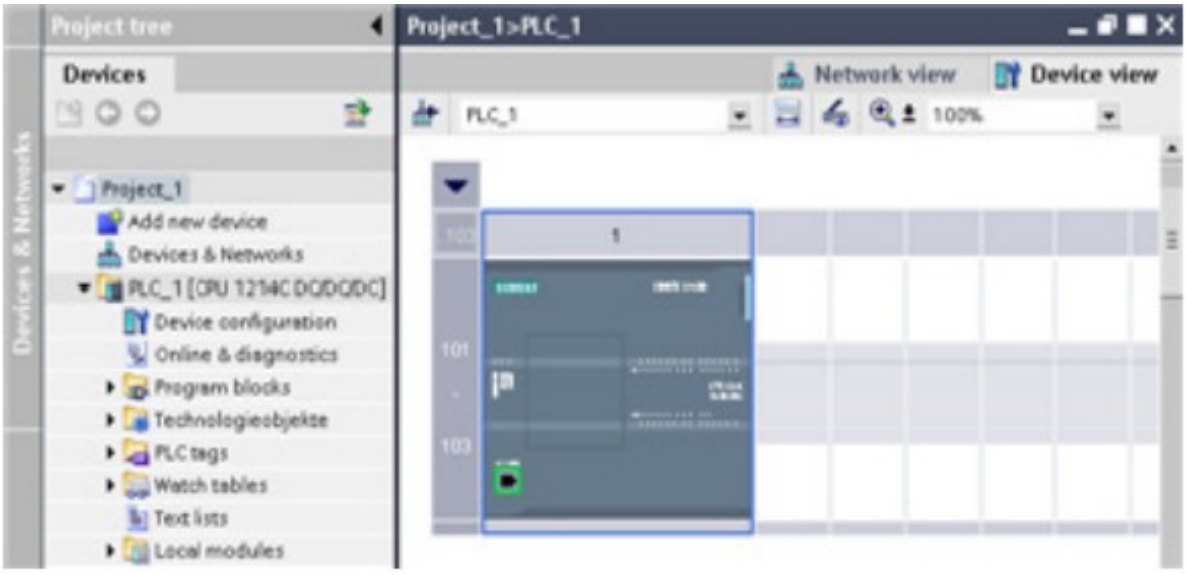


Miután létrejött a projekt, válaszd ki a Devices & Networks portált. Kattints az "Add new device" gombra.



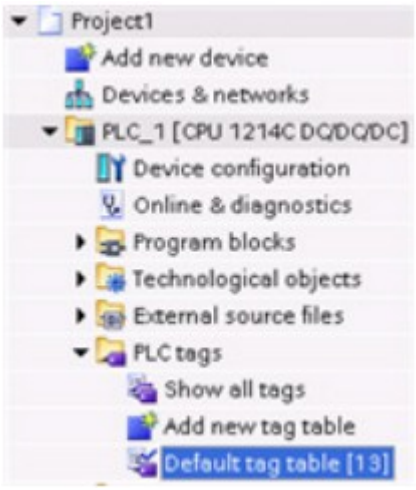
Most egy processzort kell hozzáadnunk a projekthez. Az "Add new device" ablakban válasszunk ki egy SIMATIC CPU-t. Kattintsunk az "Add"-ra.

Ha a bal alsó sarokban lévő "Open device view" opció meg van jelölve, akkor a Projekt nézet jelenik meg.



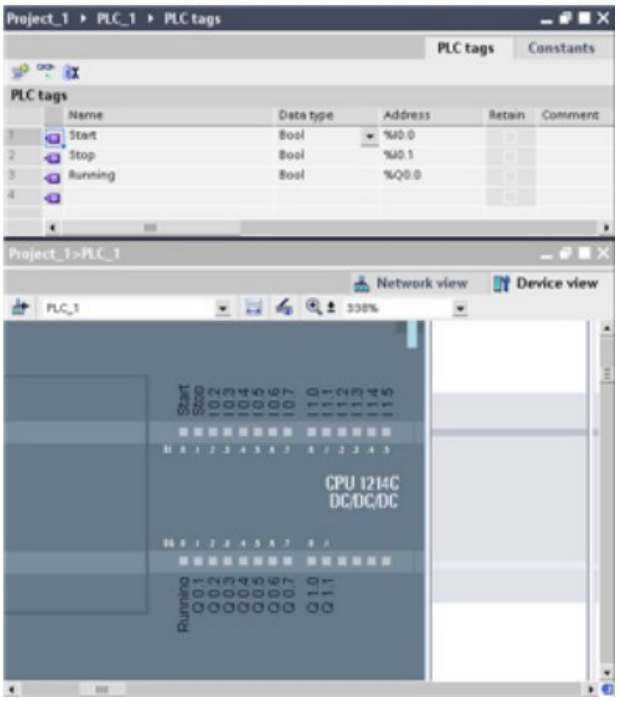
A "Device view" ablakban megjelenik a kiválasztott CPU.

Tag-ek létrehozása



A PLC tag-ek a címek, be- és kimenetek szimbolikus nevei. Egy PLC tag létrehozása után a STEP7 lementi azt a tag table-be. A projekt összes szerkesztője hozzáfér ehhez a táblázathoz.

A Device editor-ban nyissuk meg a tag table-t.



A készüléket mutató ablakban a CPU-t nagyítsuk annyira, hogy az I/O pontjait könnyen ki tudjuk választani. Innen húzhatjuk a bemeneteket és kimeneteket a tag táblázatba.

Húzzuk be a I0.0 és I0.1 bemeneteket és a Q0.0 kimenetet, majd nevezzük el ezeket a következőképp: Start, Stop és Running.

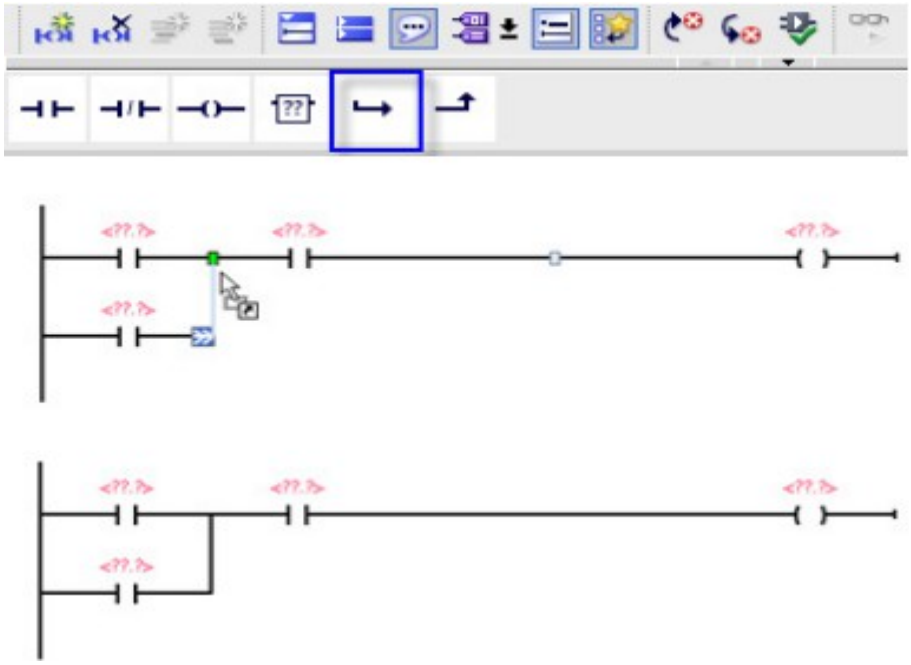
Innentől ezekkel a nevekkel hivatkozhatunk ezekre a pontokra.

Egy egyszerű hálózat létrehozása

Első lépésként meg kell nyitnunk a programszerkesztőt. Ehhez az eddig is használt projektfában nyissuk meg a "Program blocks" mappát, majd abban duplakattintás a "Main [OB1]"-re. Ez a főprogram.

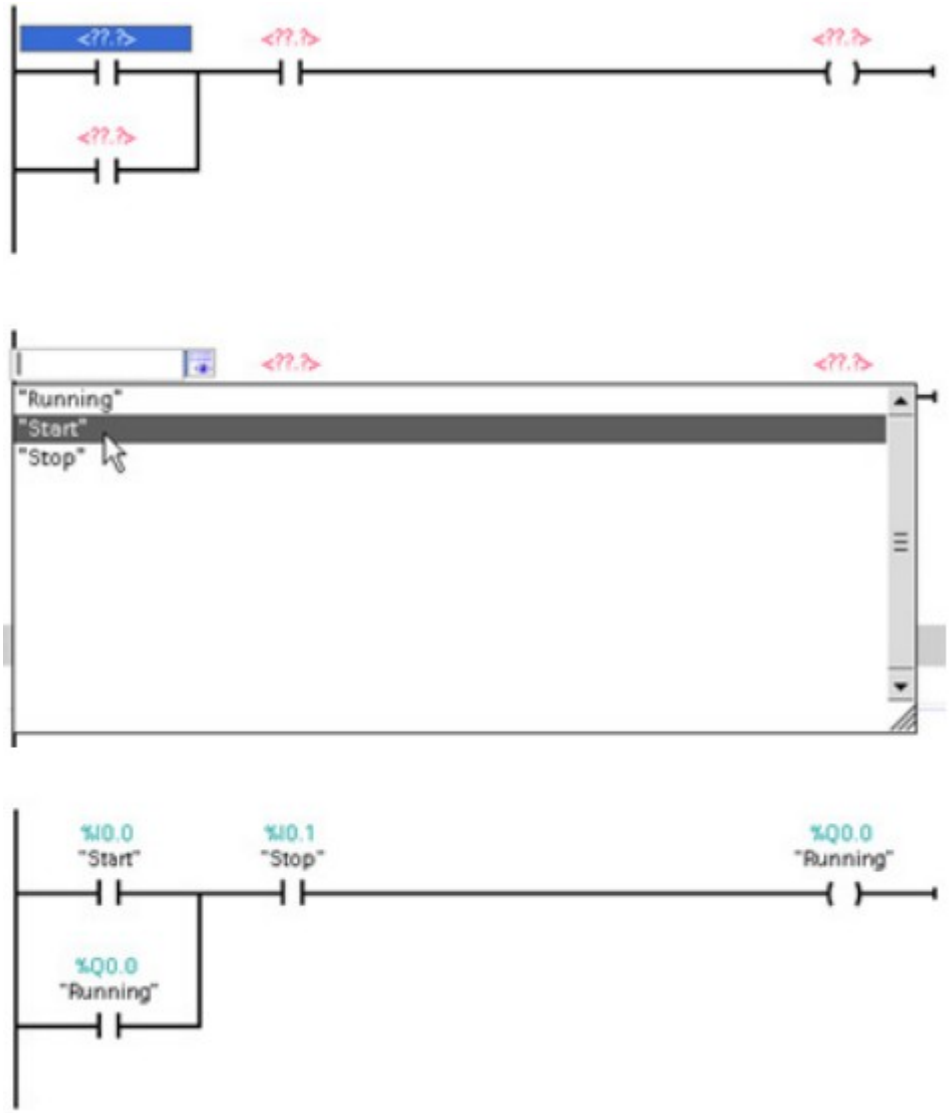
Az eszköztárban kattintsunk az alapesetben nyitott kapcsoló jelére, majd adjunk hozzá a hálózathoz kettőt. Végül a kimeneti tekercs jelének kiválasztásával adjunk a hálózathoz egy tekercset is.

Következő lépésként hozzunk létre még egy ágat!



Kattintsunk a bal oldali sínre. Jelöljük ki az ágnyitás opciót és kezdjünk meg egy új ágat. Adjunk ehhez az ághoz még egy alapesetben nyitott kapcsolót. A duplanyilat húzzuk oda, ahova csatlakozni szeretnénk (első ág két kapcsolója közé)

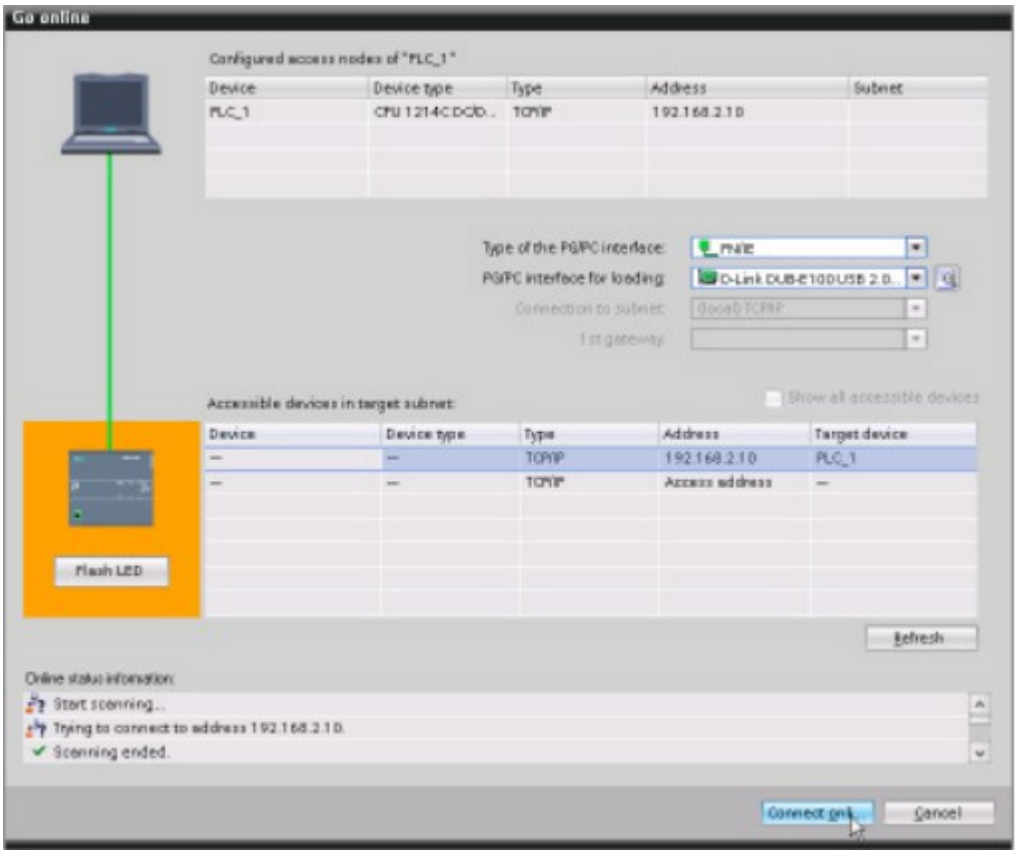
A PLC tag-ek felhasználása



Kattintsunk az első alapesetben nyitott kapcsoló alapértelmezett címére, ami <??.?>. Utána kiválaszthatunk egy tag-et a tag táblázatból. Rendeljük a tag-eket a hálózat elemeihez a fent látható módon.

Így létrehoztuk a már tárgyalt motorvezérlő programot.

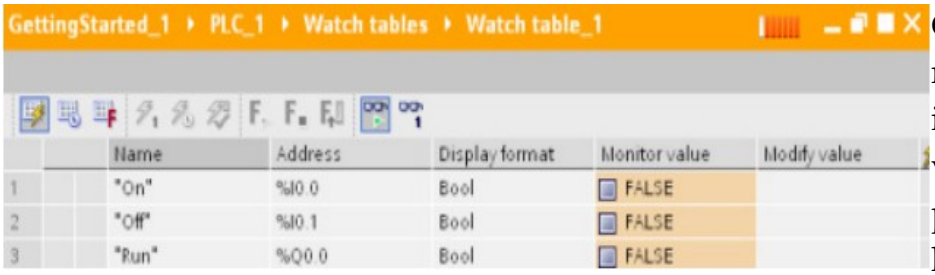
Csatlakozás



Online kapcsolatot kell létesíteni a felprogramozó eszköz és a CPU között, hogy a programunkat le tudjuk tölteni. Kapcsolat felvételéhez egyszerűen a "Go online" gombra kell kattintani.

Ha ez az első csatlakozás, akkor ki kell választani a PG/PC interfész típusát, majd a specifikus interfészt a "Go online" ablakban.

Az eszközt körülvevő narancssárga színű keret jelzi, hogy a csatlakozás sikerrel járt.



Csatlakozás esetén a munkaterület fejlécei is narancssárgára váltanak. A projektfában követhető a létrehozott és a feltöltött projekt összehasonlítása. A zöld pontok azt jelentik, hogy ezek szinkronizáltak, és mindkettő ugyanazt a konfigurációt tartalmazza.

A "Monitor all" opció segítségével a tag-ek értéke követhető.

Egy esettanulmány

A jegyzet korábbi részei alapján már nem jelenthet gondot egyszerű kombinációs hálózatok tervezése. De általában a vezérlendő környezet ettől jóval bonyolultabb programot kíván. Természetesen ezeket a feladatokat is meg lehet oldani úgy, hogy az alapszintű kombinációs logikák összekapcsolásából alkotjuk meg a nagyobb projektet, de ehhez tapasztalat szükséges. Nem említve, hogy ekkor a szoftver átláthatósága, ezzel együtt javíthatósága sem lesz ideális.

Több módszer áll rendelkezésre, amikkel egy bonyolultabb problémát kisebb részfeladatokra tudunk felbontani. Az esettanulmány keretein belül az állapot alapú tervezéssel fogunk megismerkedni, mivel ez a paradigma áll legközelebb az emberi gondolkozáshoz, illetve így a feladatok döntő többsége megoldható.

A módszer alapja, hogy a vezérlési problémát állapotokra bontjuk fel. Ezután deklaráljuk, hogy milyen események hatására válthatunk állapotot. Ezt a felbontást az állapotgráfban illusztrálhatjuk a rendszer jó átláthatóságáért.

A módszer során mindegyik állapothoz rendelünk egy SR-tárolót. A tároló 1-be vezérlésével jelezzük, hogy az adott állapotnál tartunk, továbblépéskor visszaállítjuk 0-ba. A program követhetősége miatt szükséges startup esetén ezeket a tárolókat inicializálni. Ez akkor jelentős, mikor a tárolók adatai remanens memóriaterületen vannak tárolva, így kezdeti értékük nem tudott. Ilyenkor az első állapotnak deklarált tárolót 1-be állítjuk, a többit 0-ba indulás esetén. Általában a PLC-k rendelkeznek egy olyan rendszerregiszterrel, mely jelzi ez első scan-t indítás után.

Fontos, hogy mindig egy állapot legyen aktív. Ez úgy oldható meg, hogy a tárolókat az előző állapot és az állapotváltás feltétele állítja 1-be. 0-ba a következő állapot állítja.

A kimeneteket azon állapotok OR kapcsolatával vezéreljük, amelyhez hozzárendeltük őket.

Az alapok ismertetése után nézzük a feladatot!

A feladatunk egy egyszerűsített gyalogos-közlekedési lámpa rendszer fejlesztése. 2 irányban irányítjuk a gyalogosforgalmat, észak-dél és nyugat-kelet irányban. Ez 2 forgalmi lámpát jelent, így összesen 6 izzót kapcsolgatunk. Ezek:

- É-D irány: L1 piros, L2 sárga, L3 zöld
- K-NY irány: L4 piros, L5 sárga, L6 zöld

A rendszerben található 2 nyomógomb (S1 és S2) is, 1-1 mindkét lámpánál. Ha a pirosat jelző iránynál megnyomják a gombot, akkor a másik, zöldet jelző iránynál azonnal váltunk sárgára, majd 4 sec leteltével pirosra és a másiknál pirosról zöldre. Ugyanez a folyamat játszódik le ismét, ha a most pirosat kapó

iránynál megnyomják a gombot. Érdemes felvenni az állapotokat, a hozzájuk tartozó kimeneteket, valamint azon eseményeket, amelyek hatására állapotváltásnak történnie kell.

Állapot leírása	Állapot száma	L1	L2	L3	L4	L5	L6	Átmenet oka
K-NY zöld	1	1	0	0	0	0	1	4-es állapotban 4 sec letelt
K-NY sárga	2	1	0	0	0	1	0	1-es állapotban S1-et megnyomták
É-D zöld	3	0	0	1	1	0	0	2-es állapotban 4 sec letelt
É-D sárga	4	0	1	0	1	0	0	3-as állapotban S2-őt megnyomták

Fontos megjegyezni, hogy az állapotok felvétele nem szabályokhoz kötött, máshogy is létrehozhattunk volna állapotokat. Azzal is önkényesen jártunk el, hogy az indulás utáni első állapotnak a K-NY zöldet tettük. Ha a specifikáció nem korlátozza, akkor bármi lehet a startup utáni állapot.

Állapot alapú tervezés esetén a szoftver 3 részből áll:

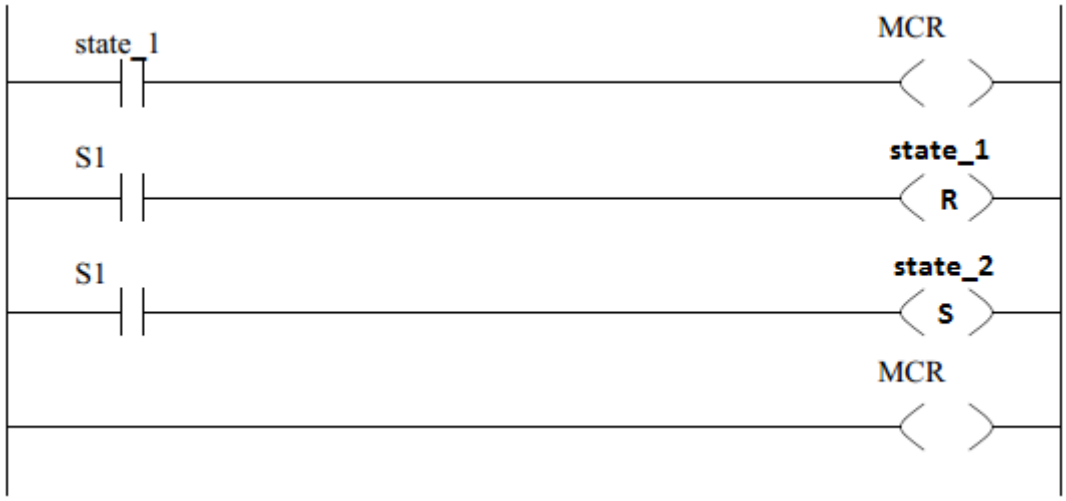
- Állapotokhoz rendelt SR-tárolók inicializálása induláskor
- Tárolók irányítása
- Kimenetek irányítása

Inicializálás

Az MCR tekercsek működése: Az általuk közrefogott programrészlet csak akkor fut le, ha a pár első tekercsére 1 kerül, jelen esetben, ha most indul a CPU (STARTUP=1). MCR tekercset között csak tárolók állíthatóak. Ez egy Allen Bradley specifikus parancs. Ha a mi fejlesztőrendszerünk nem ismeri eme tekercset, akkor az állapotváltásoknál a STARTUP bitet is ellenőrizni kell.

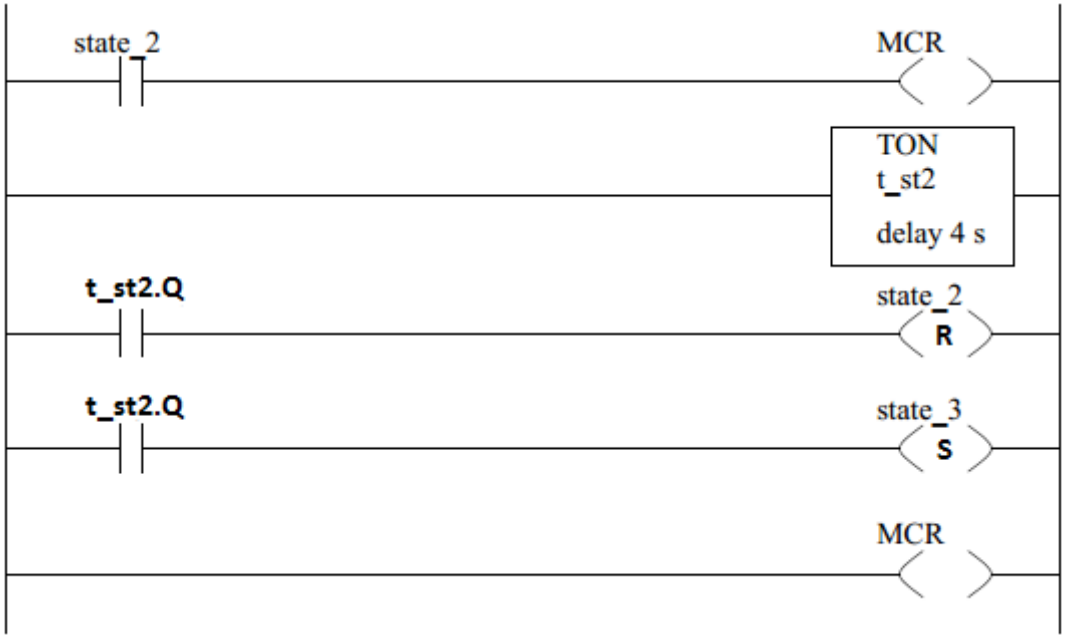
Induláskor ez a programrészlet egyszer fut le, beállítva az 1-es állapotot kezdő állapotnak.

State_1 állapot



Ha 1-es állapotban vagyunk és megnyomták az S1-et, akkor a 2-es állapot tárolóját aktiváljuk.

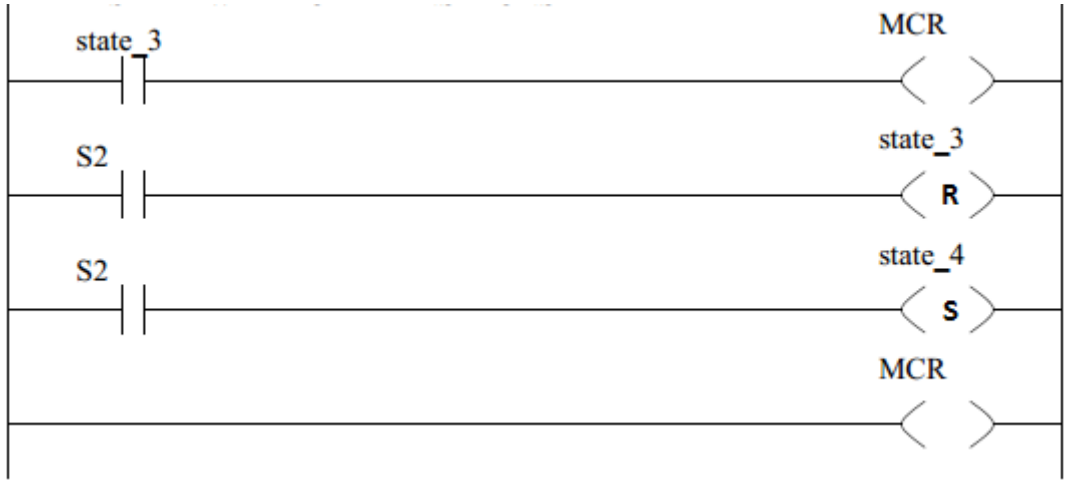
State_2 állapot



Az állapot aktiválódása után indítjuk a TON számlálót. Amikor lejár, akkor lépünk a következő állapotba.

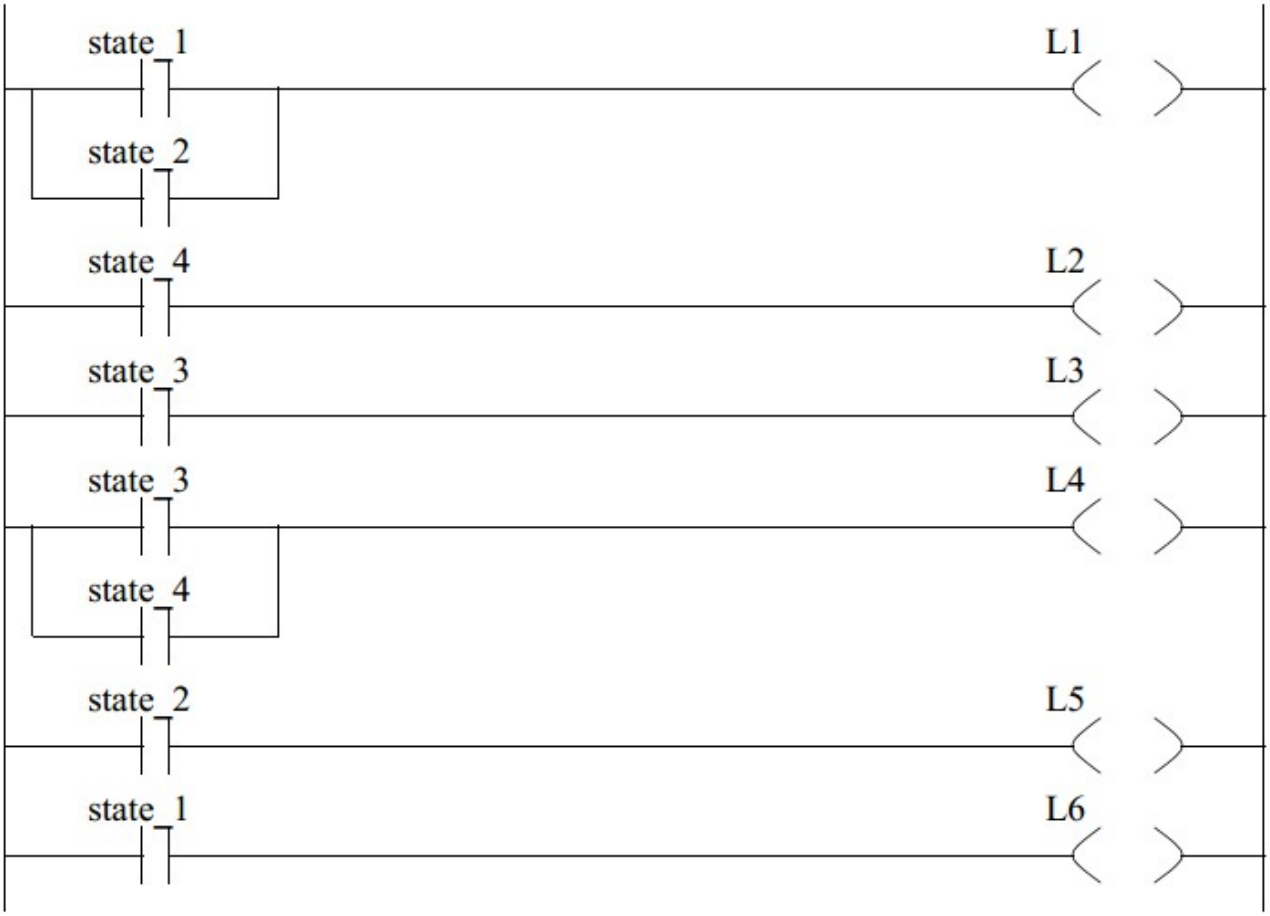
Ezek alapján a következő két állapot vezérlése magától értetődő.

State_3 állapot



State_4 állapot

Kimenet



A kimenetek vezérlésénél roppant hasznos a részletezett állapottáblázat felvétele. Csak ki kell olvasni, hogy az egyes izzók mely állapotokban aktívak.

FONTOS!! Az állapotokat rendeljük OR kapcsolatban a kimeneti tekercsekhez, nem pedig fordítva!