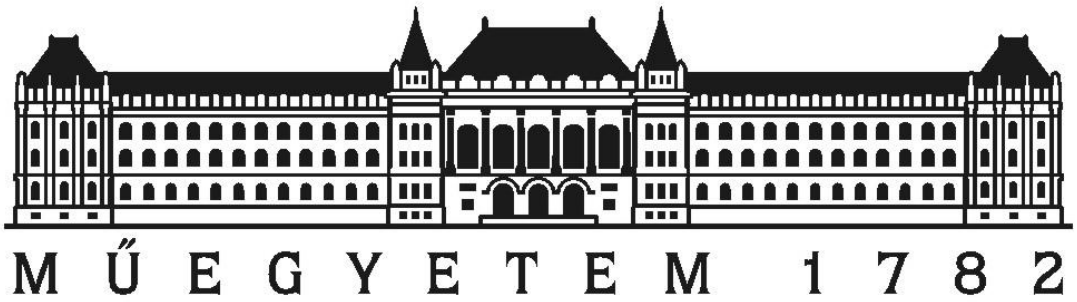


ESP8266 alapú lakásautomatizálási rendszer fejlesztése

Virtuális PLC-szerű vezérlő

Készítette: Seres Zsombor, Neptun: A93C8G

Konzulens: Kovács Viktor

Tartalom

1. Motiváció:

Célkitűzésem, egy olyan virtuális PLC-szerű vezérlő fejlesztése, amely firmware frissítés nélkül lehetővé teszi az ESP újra konfigurálását. A projekt magja egy virtuális processzor, amely egy .txt fájlt értelmez és hajt végre soronként az előre definiált (ill. bővíthető) utasításkészlete segítségével. A munkám során szem előtt tartottam, hogy az utasítások könnyen értelmezhetőek legyenek, ennek köszönhetően, aki C programozásban esetleg nem annyira jártas, egy egyszerű szövegfájl segítségével is elkészítheti egyszerűbb lakásautomatizálási rendszer logikáját. A projekt egyelőre asztali környezetben futtatható, de úgy írtam a kódot, hogy beágyazott rendszerre is könnyen átültethető legyen. (ezt később kifejtem) Ezen felül igyekeztem, ahol lehetett minél generikusabban programozni annak érdekében, hogy az advanced user-ek forráskód szintjét is bővíthessék a szoftvert.

A vezérlő eddig megvalósított funkciói:

- -bool, integer változók kezelése
- -címkék kezelése, sima és feltételes ugró utasítás
- -logikai, aritmetikai utasítások
- -periféria kezelés: GPIO és Example Timer

2. A szoftver felépítése

2.1 Generic_Container ősosztály:

Mielőtt rátérnék a projekt fő elemére a processzorra, ismertetem a Generic_Container osztályt. Ez azért lényeges, mivel a lentebb látható felépítési ábrán, az összes …_Container nevű objektum ennek a class-nak a leszármazottja. Ez az osztály ahogy a neve is sugallja generikus, a példányosításkor/örökléskör megadott típusból/osztályból tartalmaz elemeket. Tagváltozói az alábbiak:

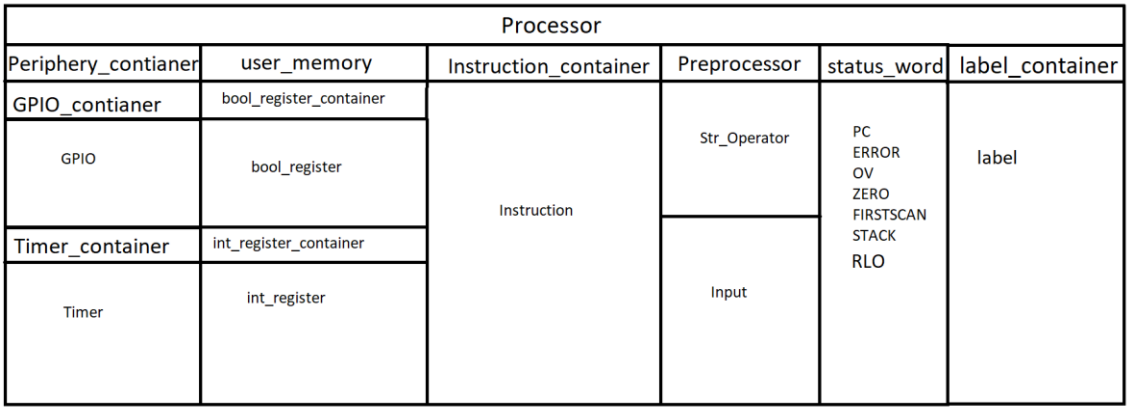
```
int elem_num = 0;
int MAX_NUM = DEF_MAX_SIZE;
T* element;
```

A MAX_NUM az a maximális elemszám, ami lehet a Container-be. A konstruktorában ennyi elem foglalódik le fixen a memóriában . Az elem_num a hasznos (nem memóriaszemét) elemek számát mutatja. Az element pedig egy pointer, aminek típusát a T template mondja meg. Az osztály tagfüggvényei segítségével, hozzáadhatunk új elemet, index segítségével adott elem értékét kaphatjuk vissza.

Felmerülhet a kérdés, hogy miért nem a c++ std::vectorát használom. Ennek oka, hogy az dinamikusan nyújtózkodó tömböt használ.

2.2 Processor főosztály

Ahogy fentebb is írtam projektem fő eleme a processzor osztály, ő az, aki tartalmazza az összes többi vezérlő elemet és ő instruálja azokat. Az alábbi ábra szemlélteti a felépítését: (Nem volt célom, minden tagváltozó, tagfüggvény feltűntetése, az ábra csupán a megértést segíti)



Röviden jellemezném az ábrán feltűntetett elemeket:

2.2.1. status_word: A processzor állapotát mutatja:

```
class status_word
{
public:
	unsigned int PC;
	bool RLO;
	bool ERROR;
	bool OV;
	bool ZERO;
	bool FIRSTSCAN;
	unsigned int STACK;//arra hivatott, hogy ha fv-t hívunk akkor tudjunk visszatérni
};
```

PC (program-counter) az éppen feldolgozandó sor számát mutatja

RLO: a logikai műveletek eredménye tárolódik benne

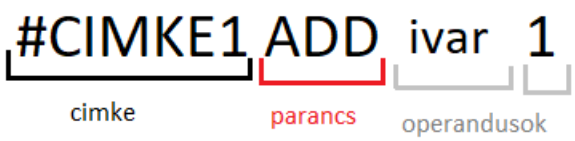
ERROR: ha a program futása során bárhol hiba történik, igaz értéket vesz fel. Ha bármelyik ciklusban értéke true-ba vált, akkor a következő ciklus már nem fut le, a program futása megszakad.

FIRSTSCAN: Azt mutatja, hogy az első ciklusban vagyunk-e. Azért van jelentősége, mert a SETUP és CREATE utasítások csak az első CPU ciklusban futnak le

A többi változót egyelőre nem használom.

2.2.2. Preprocessor:

Feladata a PC által mutatott sort átalakítani Line típusú class-á, amelyet már a Processor könnyen kezel. Itt térnék ki, arra, hogy a .txt fájlban milyen is lehet egy sor felépítése:



Az elemeket whitespace karaktereknek kell elválasztania, a sorrend kötött. A sornak lehet címkéje, de ez nem kötelező, a címkét „#” karakterrel jelölöm. Minden sornak kell lennie értelmezhető parancsának. Ezután következnek az operandusok. Megjegyezném, hogy ugyanaz a parancs hívható többféle paraméterszámmal, ekkor lehet, hogy eltérő működést fog mutatni (lsd. JMP) vagy az is lehet, hogy ugyanazt a műveletet hajtódik végre többször (erre példát mutatok a 6. pontban).

A Preprocessor 2 interface-t implementál. Az Input, a fájl menedzselésért felelős, míg az Str_Operator string->Line átalakításhoz szükséges string daraboló függvényeket tartalmazza. A Preprocessor-on belül van egy currentLine tagváltozó, amely egy Line osztály példánya. A convert során ennek a mezői töltődnek ki, a processor ezzel a dolgozik tovább. A Line tagváltozói:

```
bool have_label = false;//van-e címkéje az adott sornak
string label;
string command;
string* parameter_pointer;
unsigned int parameter_num = 0;
const unsigned int MAXPARAMETERNUM = MAX_PARAMETER_NUM;
```

2.2.3. Label_Container:

Címkék gyűjteménye. A Label osztály egy példánya, tartalmazza a címke nevét, illetve, hogy melyik sorhoz tartozik.

2.2.4. Instruction_Container:

Azoknak a parancsoknak a halmaza, amelyek nem hoznak létre változót, és nem inicializálnak perifériát sem. Ezek a user_memory illetve a Periphery_Container hatáskörébe tartoznak. Instruction-ök „tömbje”, az Instruction class felépítése:

```
class Instruction {
protected:

	string instruction_type;//na ez azért kell, hogy híváskor tudjuk, melyik típusú
inter_containernek kell szólni
	void	(*fv_pointer)(Line*	line,Memory*	user_memory,status_word*
sw,Inter_Container* inter_container, Label_Container* label_container);
	string name;
```

Minden Instruction-nek van neve, amin keresztül hívható, illetve egy függvénypointer-e, amely megmutatja, hogy Call esetében, milyen C-s metódust kell meghívni. Az intstruction_type megmondja, hogy milyen típusú inter_container-t kell alkalmazni, az adott parancs futtatása során Az intercontainer feladata, hogy átmenetileg parsolja, C-s nyelvre a .txt fájlban felsorolt operandusokat, így azokkal , a hívott függvény már tud dolgozni.

2.2.5. User_memory

Memory class példánya, az ő szerep köre változók létrehozása, módosítása, átadása másik vezérlő elemnek. 2 nagy container-e a bool illetve int_register_container. Ők int_register-ek és bool_register-ek gyűjteményei. „”_register classok, a register generikus osztály leszármazottjai, melynek tagváltozói:

```
string name;
string type;
U value;
```

Minden registernek van neve, típusa, illetve értéke.

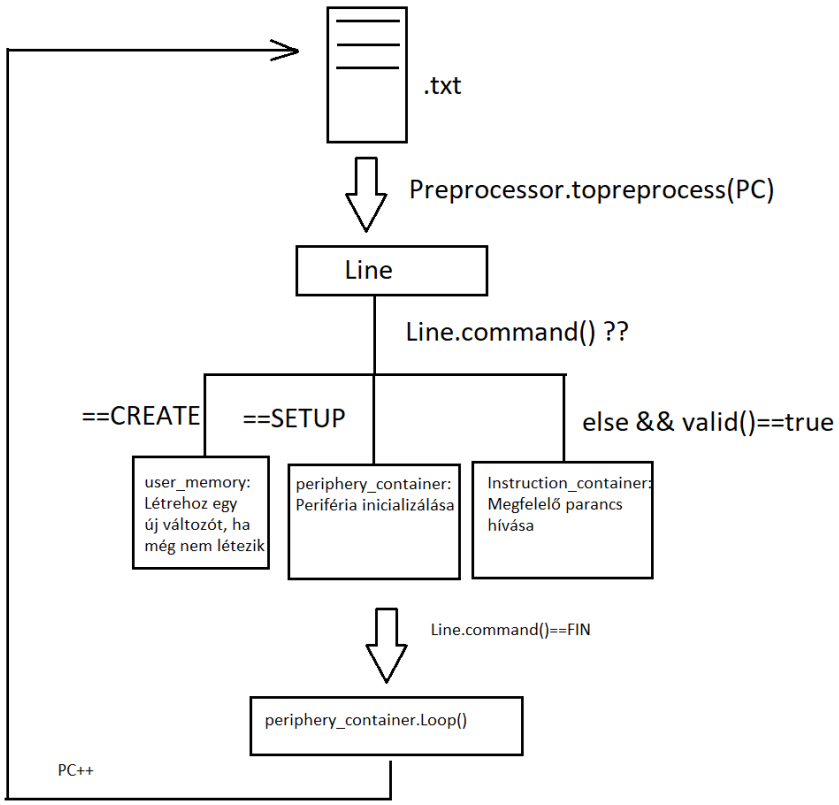
2.2.6. Periphery_container

Rendes és virtuális perifériák menedzselésére szolgáló osztály. GPIO és Timer Containerei vannak, (ezekről lásd később), az ő feladat a Periphery Loop futtatása, amely frissíti a perifériák állapotát.

3. A program működése:

A végtelen ciklus indulása előtt meg kell hívni a theProcessor.CPU_Init() függvény-t. Funkciója, hogy alaphelyzetbe állítsa a status_word-öt, illetve végig olvassa a text file-t és az alapján kitöltsea Label_Container-t. END_PROGRAM utasításig olvas. A címkék előre kiolvasása azért, szükséges, mert a programban előfordulhat előre ugrás is.

A theProcessor.CPU_Loop() fv egészen a FIN utasításig végre hajtja a sorokat. A FIN utasítás a PC-t 0-ba állítja, azaz egy új ciklus kezdetét jelzi. Az alábbi ábra egyetlen sor feldolgozását mutatja:



A megfelelő sor-t a Preprcoessor átalakítja Line-á. Attól függően, hogy Line.Command() változó micsoda, eldönti a processzor, hogy melyik vezérlő elemnek kell szólnia. Az ábrán is látható, hogy a Periféria ciklus, csak akkor fut le, ha FIN utasításhoz értünk, azaz egy CPU ciklus végéhez . Jogos a felvetés, hogy a Periféria Loop gyakrabban, esetleg minden sor után lefuthatna. Ez az igény, például pontosabb Timereknél merülhet fel. Mivel nekem nincs szükségem gyakoribb periféria frissítésre, elegendő számomra az előbbi megoldás.

4. Utasításkészlet

Parancs név:	**Típus:**	**Eredmény helye:**	**Operandusok száma:**	**Funkció:**
CREATE	Változó létrehozó	user_memory	2	Létrehoz a memóriában egy új változót, ha még nem létezett
SETUP	Periféria inicializáló	periphery_contaiener	2k+1, k!=0	Perifériákat inicializál vagy módosít
FIN	Program szervezési	-	0	PC-t nullába állítja
END_PROGRAM	Program szervezési	-	0	Megmondja, hogy a CPU_Init() meddig olvassa a fájlt
JMP	Program szervezési	-	1 vagy 2	ha 1 operandust kap, akkor az operandusba megadott címkére ugrik feltétlenül. ha 2-t akkor csak akkor ugrik, ha a második bool operandusa true
AND	Logikai	RLO	>=1	ha 1 operandussal hívjuk, akkor az RLO-n és az operanduson elvégzi a logikai és műveletet
OR	Logikai	RLO	>=1	u.a, mint az and, csak or logikai műveltet végzi el
NOR	Logikai	RLO	>=1	u.a, mint az and, csak nor logikai műveltet végzi el
NAND	Logikai	RLO	>=1	u.a, mint az and, csak nand logikai műveletet végzi el
XOR	Logikai	RLO	>=1	u.a, mint az and, csak xor logikai műveletet végzi el
NEG	Logikai	RLO	=1	Az első operandust negálja.

Parancs név:	**Típus:**	**Eredmény helye:**	**Operandusok száma:**	**Funkció:**
RLO	Debug	-	0	RLO értékét kiírja a konzolra
EQU	Logikai	RLO	2	2 intről megmondja, hogy egyenlőek-e.
ADD	aritmetikai	első operandus	>=1	ha egyetlen operandussal hívjuk, akkor az adott operandus értékét kiírja a konzolra. Több operandus esetén összeadja az összes operandus értékét.
SUB	aritmetikai	első operandus	>=1	u.a, mint ADD, csak kivonás műveletet hajtja végre
MUL	aritmetikai	első operandus	>=1	u., mint ADD, csak a szorzás műveletét hajtja végre
=	-	első operandus	2	Az első operandusba átmásolja a második operandus értékét.

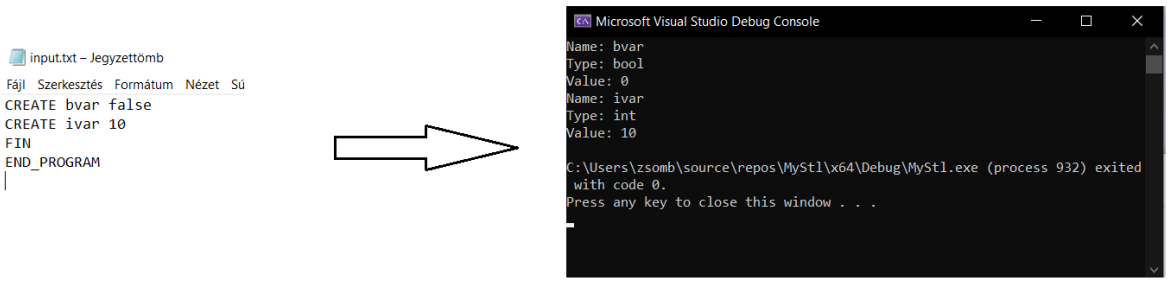
5.Változó létrehozása:

Változó létrehozására a CREATE paranccsal van lehetőség, a változó típusát nevének első karaktere azonosítja:

- i-integer
- b-bool

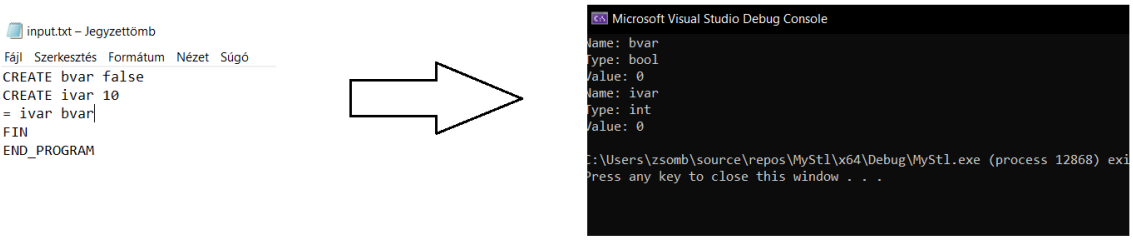
Ebből derül ki a memória számára, hogy az int_register_containerjéhez kell hozzáadni új elemet vagy a bool_register_containerjéhez. A program a literálok kezelése miatt nem engedélyez számmal kezdődő változó nevet. Kötelező valamilyen kezdeti értéket megadni. Azonos típusú változóból nem létezhet ugyanolyan nevű.

Példa: (A memória tartalmát kiprintelem)



Itt szeretném megemlíteni a konverzió lehetőségét. Mivel eddig csak int és bool típusú változóm van ez annyira nem égető dolog, de mégis biztosítottam rá lehetőséget. Átjárást a két típus között az „=”operátor biztosít:

Példa:



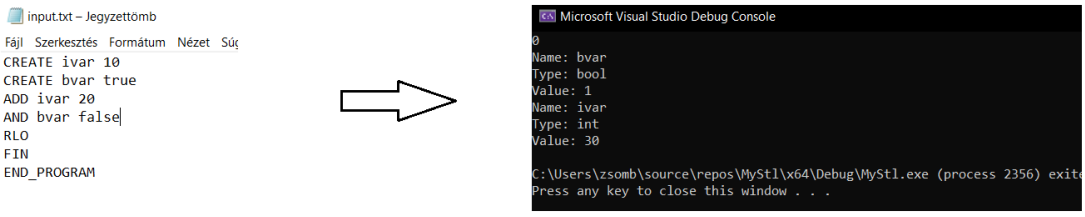
Itt látható, hogy a 2 változó létrehozása után az „=” operátor hatására bvar konvertálódik 0-vá és felülírja ivar változót.

6. Instruction hívása:

Az Instruction class-t röviden már jellemeztem, itt példát szeretnék mutatni pár feature-re:

6.1. Hívás literálokkal:

Példa:

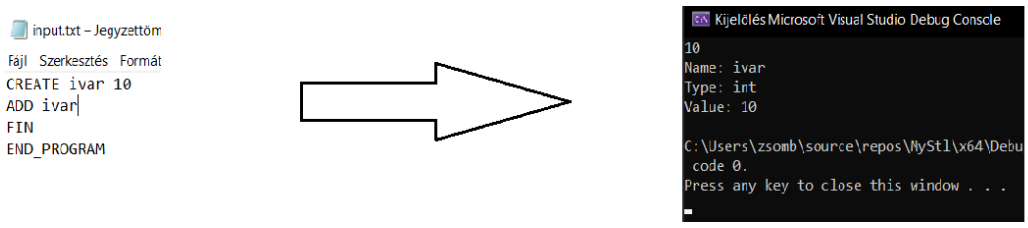


Az aritmetikai és logikai műveltek paraméterei is lehetnek literálok. A console-ban látszik, hogy az összeadás és az és az AND művelet is a megfelelő módon működik.

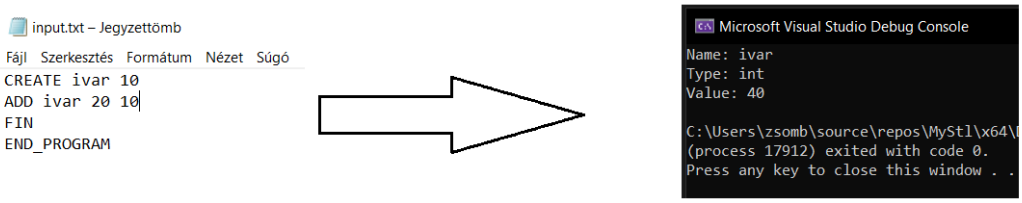
6.2. Különböző paraméterszám kezelése:

Példa:

1 operandus:



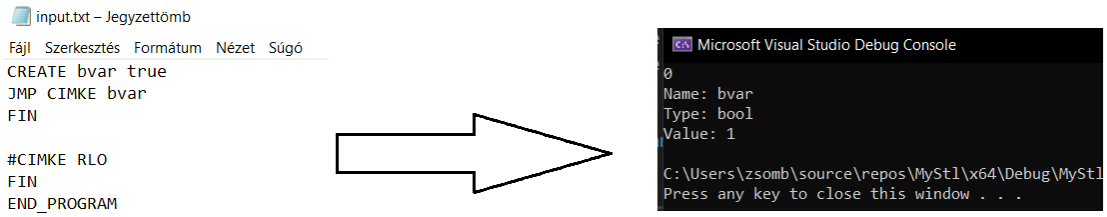
több operandus:



1 operandus hívása esetén, ivar változó értéke nem változik, viszont értéke kiíródik a console-re. Több operandus esetén pedig ciklikusan meghívódik az alap függvény, ezért az összes tagot összeadja és értékét elmenti ivar-ba.

6.3. If megvalósítása:

Feltételes ugró utasítással módunkban áll egyszerű if megvalósítása:



A .txt alapján látható, hogy csak abban az esetben írjuk ki RLO értékét, ha ugrunk, mivel bvar változó értéke true, ezért ezt meg is tesszük.

7. Perifériák:

Egyelőre GPIO-t és egy virtuális Timer típust valósítottam meg, ezek elveire és mintájára építve, könnyen bővítheti a felhasználó a perifériák listáját.

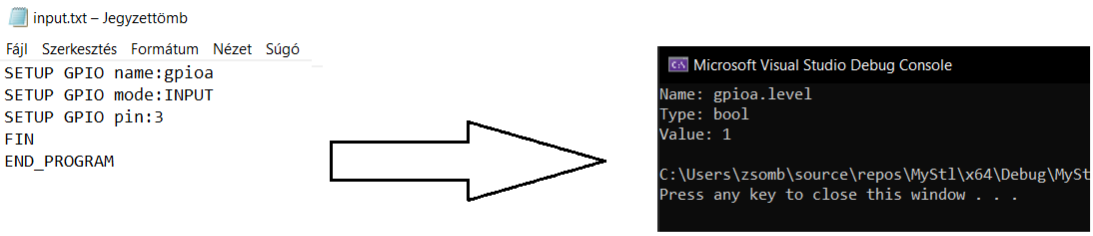
7.1 Perifériák inicializálása:

Perifériákat a SETUP kulcsszó segítségével inicializálhatunk fel. 1. operandus ként meg kell mondanunk milyen típusú perifériát szeretnénk létrehozni a programban. Egy adott Periféria tulajdonságát kulcs érték párok segítségével adhatjuk meg. Nem kötelező minden kulcsot egy sorban kitölteni, megtehető ez több sorban. Mindig a legutoljára megadott nevű periféria tulajdonságát módosítjuk.

Amikor kitöltöttük, a periféria szükséges tulajdonságait (ez nem feltétlenül az összes), a memóriába beleíródnak a változó [periférianeve].[alakban] alakban. Emiatt a már meglévő utasítások használhatóak a periféria változóin is.

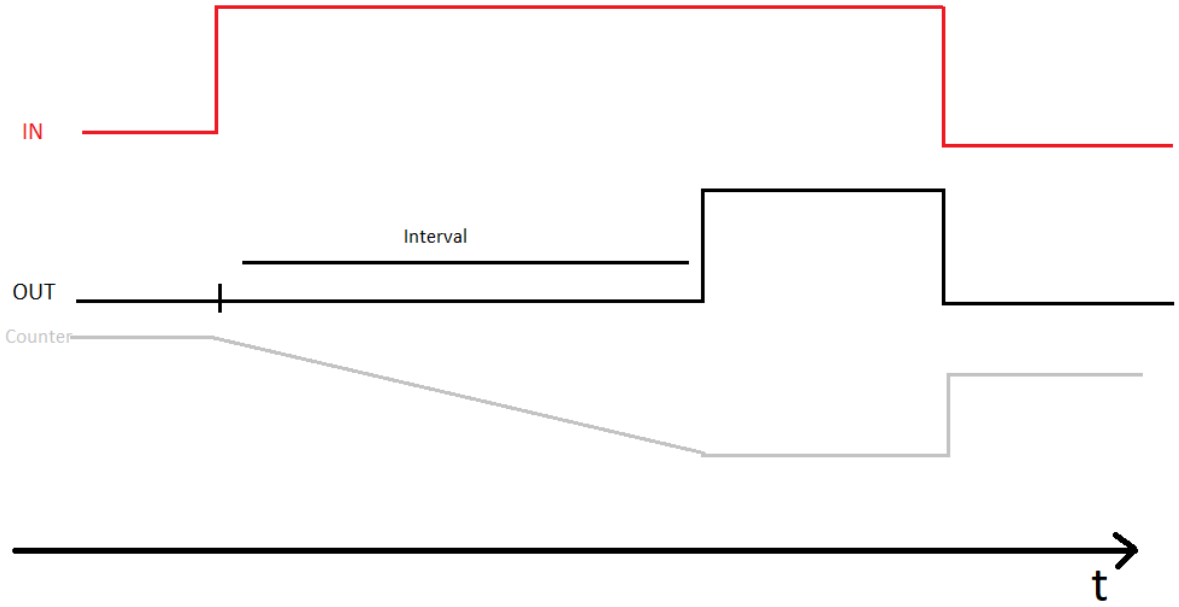
Lehetőség van dedikált, perifériákhoz kötött parancsok létrehozására, ilyen lehet pl GPIO_Read, GPIO_Write. Ezek csak akkor íródnak be az instruction_container-be ha már van legalább egy érvényes típus az adott GPIO-ból. A regisztrálás logikáját kidolgoztam, de még nem implementáltam dedikált utasításokat.

Példa:



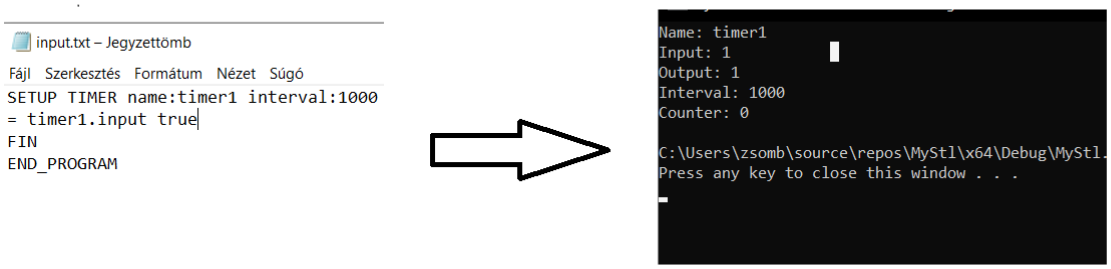
7.2 Example Timer

Példaként egy virtuális On Delay típusú Timer-t valósítottam meg, pontossága ms-s. Működését az alábbi időfüggvények jól szemléltetik:



Látható, hogy ha az IN bemenetre logikai magas értéket kapcsolunk, akkor a belső számláló Intervall-ról elkezd csökkeni, egészen addig amíg a 0-t el nem éri. Ha a counter=0, abban az esetben az OUTPUT is magas értékű. Amikor IN alacsony logikai szinten van, a Counter vissza áll Interval értékre, számolni csak akkor fog, mikor ismét logikai magas jelenik meg a kimeneten.

Példa:



Fel inicializáltam a Timert, elindítiottam a timer1.input true értékbe állításával.CPU_Loop-ot elég sokszor meghívva a Counter eléri a 0 értéket és ennek megfelelően az OUTPUT is 0-ba áll.

8.Tovább fejlesztési lehetőségek:

- Megvalósítás ESP-n
- Több változó típus felvétele
- Utasításkészlet bővítése
- Error Class
- Perifériák bővítése
- Kliens alkalmazás

Kijelentem, hogy én Seres Zsombor, az önálló laboratóriumi feladatot önállóan, a megengedett segédforrások segítségével és a tanszéki konzulenssel együttműködve oldottam meg.
Seres Zsombor
2021.05.18.