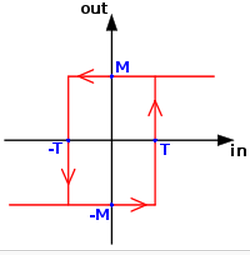
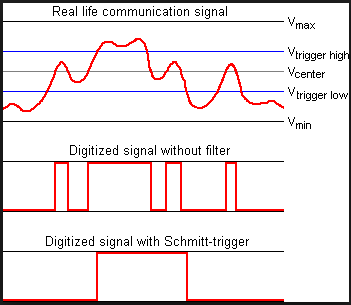
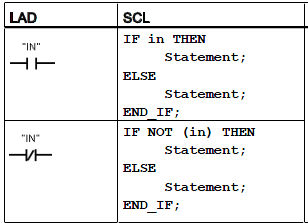
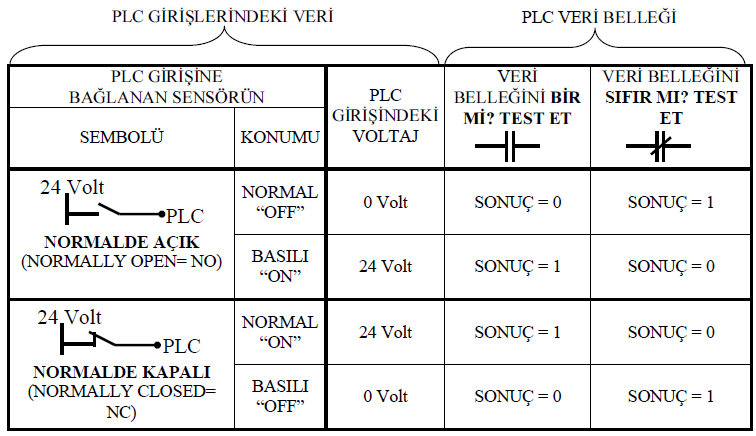
 

Schmitt Trigger

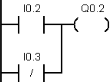
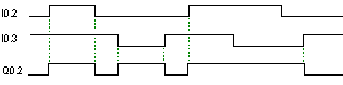
Giriş Lojik 1 mi ?  Giriş Lojik 0 mı ?  Test sonucunu “OUT” a yaz..

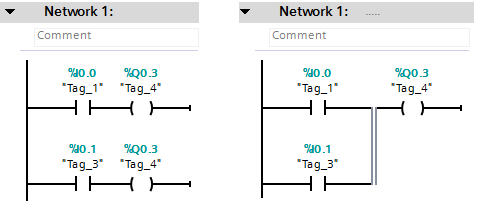
Örnek 1:

Örnek 2:

I0.2 ve I0.3!ün değili iki ayrı satır ile yazılır ise durum farklı olur.. Buradaki veya yapısı anlaşılmalı..

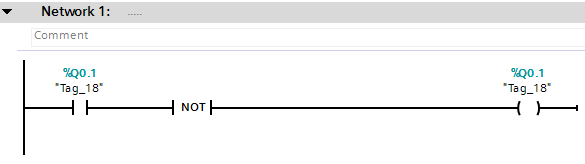


Sol taraftaki durumda Q0.3 çıkışı sadece I0.1 girişine bağlı.. I0.0’ın etkisi olmaz.. Sağtarafta ise I0.0 ve I0.1 veya kapısı ile bağlı…

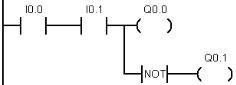
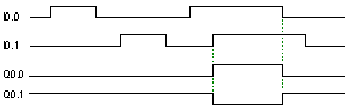
 “ NOT” (Değilleme) komutu enerji akışının yönünü lojik olarak değiller (0🡪1, 1🡪0).

  Test sonucunu tersleyip (0🡪1, 1🡪0) yaz..

Örnek 3:



Örnek 4:

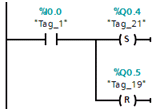
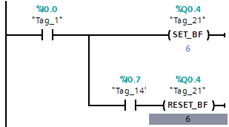
SET ve RESET , Set Bit Field ve Reset Bit Field komutları

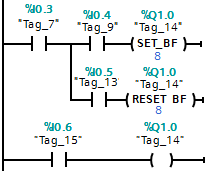
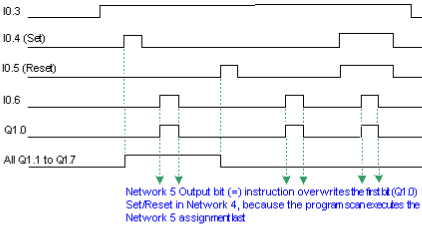
When S (Set) is activated, then the data value at the OUT address is set to 1. When S is not activated, OUT is not

changed. When R (Reset) is activated, then the data value at the OUT address is set to 0. When R is not activated, OUT is not changed.

Örnek 5:

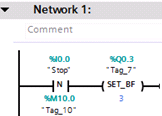
 

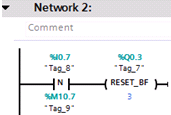
Örnek: (Q1.0 ın durmuna dikkat.. PIQ ile fiziksel çıkış arasındaki fark açısından.)

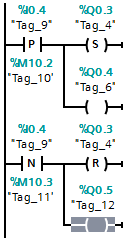
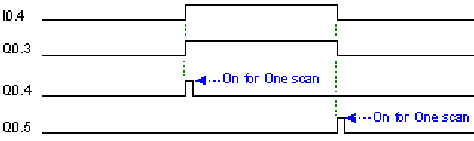
 

Yükselen ve Düşen Kenar Komutları

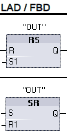
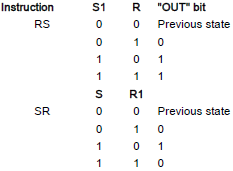
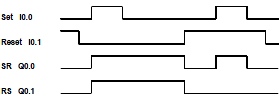




NOT: Aynı giriş için (örneğin I0.0) Yükselen ve Düşen kenar tetiklemelerine farklı yardımcı bitle-bayraklar tanımlanmalı.. Diğer türlü doğru çalışmaz..

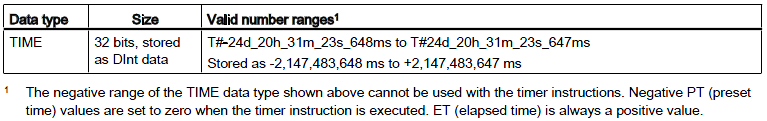
Set ve Reset Öncelikli (Baskın) Komutlar

Zamanlayıcılar.. (1 ms)







The number of timers that you can use in your user program is limited only by the amount of memory in the CPU. Each

timer **uses a 16 byte**

When a timer is started due to an edge change on the input of a TP, TON, TOF, or TONR instruction, the value of *the continuously-running internal CPU timer* is copied into the START member of the DB structure allocated for this timer instruction. This start value remains unchanged while the timer continues to run, and is used later each time the timer is

updated. Each time the timer is started, a new start value is loaded into the timer structure from the internal CPU timer.

**When a timer is updated,** the start value described above is subtracted from the current value of the internal CPU timer to determine the **elapsed time. The elapsed time is then compared with the preset to determine the state of the timer Q bit.** **Note that the elapsed time is clamped at the preset value (the timer does not continue to accumulate elapsed time after the preset is reached).**

**A timer update is performed when and only when:**

● A timer instruction (TP, TON, TOF, or TONR) is executed

● The "ELAPSED" member of the timer structure in DB is referenced directly by an instruction

● The "Q" member of the timer structure in DB is referenced directly by an instruction

**Timer programming**

The following consequences of timer operation should be considered when planning and creating your user program:

● You can have multiple updates of a timer in the same scan. The timer is updated each time the timer instruction (TP, TON, TOF, TONR) is executed and each time the ELAPSED or Q member of the timer structure is used as a parameter of another

executed instruction.

● You can have scans during which no update of a timer occurs. It is possible to start your timer in a function, and then cease to call that function again for one or more scans. If no other instructions are executed which reference the ELAPSED or Q members of the timer structure, then the timer will not be updated. A new update will not occur until either the timer instruction is executed again or some other instruction is executed using ELAPSED or Q from the timer structure as a parameter.

**● Although not typical, you can assign the same DB timer structure to multiple timer instructions. In general, to avoid unexpected interaction, you should only use one timer instruction (TP, TON, TOF, TONR) per DB timer structure.**

● **Self-resetting timers are useful to trigger actions that need to occur periodically. Typically, self-resetting timers are created by placing a normally-closed contact which references the timer bit in front of the timer instruction.** This timer network is typically located above one or more dependent networks that use the timer bit to trigger actions. When the timer expires (elapsed time reaches preset value), the timer bit is ON for one scan, allowing the dependent network logic controlled by the timer bit to execute. Upon the next execution of the timer network, the normally closed contact is OFF, thus resetting the timer and clearing the timer bit. The next scan, the normally closed contact is ON, thus restarting the timer. **When creating self-resetting timers such as this, do not use the "Q" member of the timer DB structure as the parameter for the normally-closed contact in front of the timer instruction. Instead, use the tag connected to the "Q" output of the timer instruction for this purpose.** The reason to avoid accessing the Q member of the timer DB structure is because this causes an update to the timer and if the timer is updated due to the normally closed contact, then the contact will reset the timer instruction immediately. The Q output of the timer instruction will not be ON for the one scan and the dependent networks will not execute.

**Time data retention after a RUN-STOP-RUN transition or a CPU power cycle**

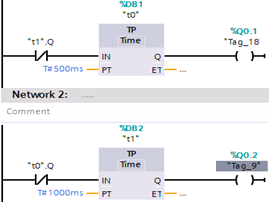
**If a run mode session is ended with stop mode or a CPU power cycle and a new run mode session is started, then the timer data stored in the previous run mode session is lost, unless the timer data structure is specified as retentive (TP, TON, TOF, and TONR timers).**

When you accept the defaults in the call options dialog after you place a timer instruction in the program editor, you are automatically assigned an instance DB which cannot be made retentive. **To make your timer data retentive, you must either use a global DB or a Multiinstance DB.**

**Assign a global DB to store timer data as retentive data ? ….**

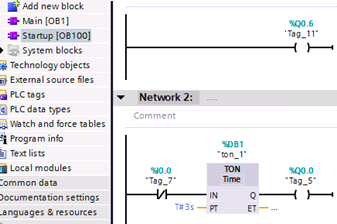
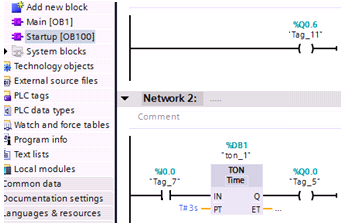
**Assign a multi-instance DB to store timer data as retentive data ? ....**

Örnek :

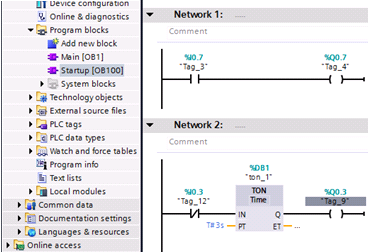


**Timer’ların güncellenmesi, 1. örnek**…

OB 100 1.durum 2.durum

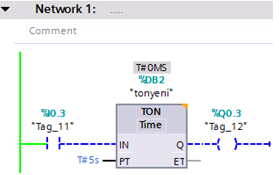
OB1

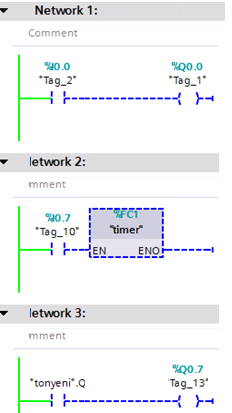


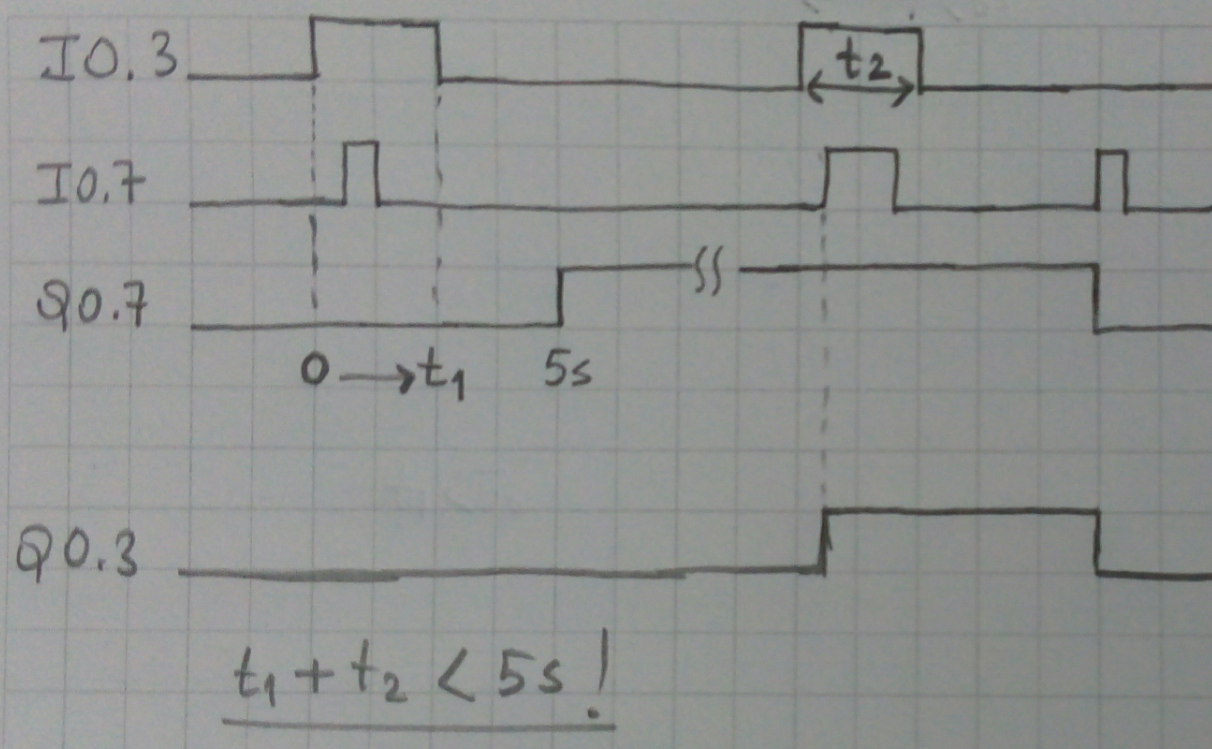
**Timer’ların güncellenmesi, 2. Örnek (Fonksiyon (Alt-Program), Zamanlama diyagramı ile birlikte..)**

OB1

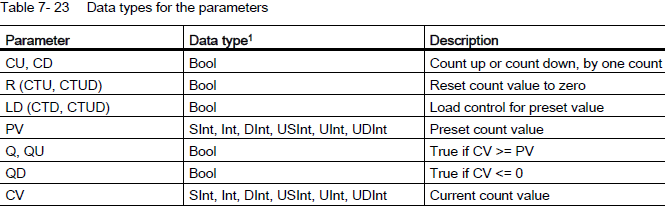
FC1\_ ”timer”

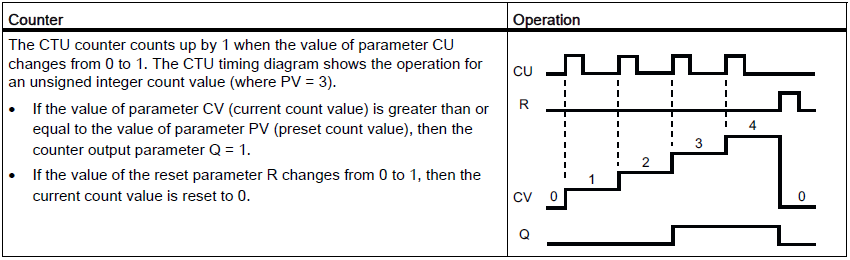
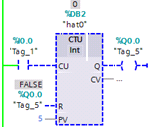


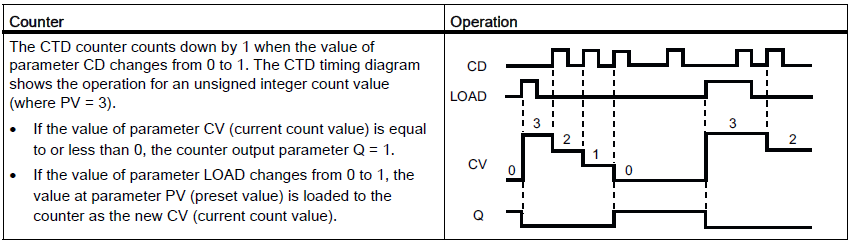


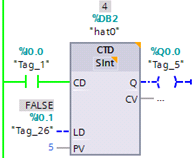
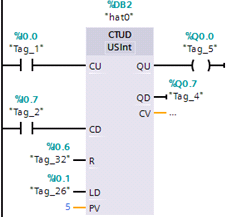


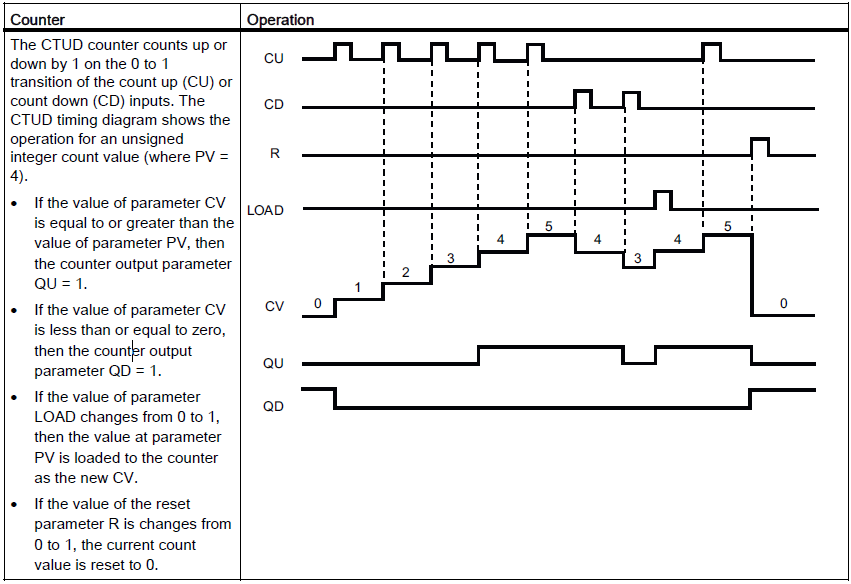
Sayıcılar







The number of counters that you can use in your user program is limited only by the amount of memory in the CPU. Counters use the following amount of memory:

● For **SInt or USInt** data types, the counter instruction uses **3 bytes**.

● For **Int or UInt** data types, the counter instruction uses **6 bytes**.

● For **DInt or UDInt** data types, the counter instruction uses **12 bytes**.

**These instructions use software counters whose maximum counting rate is limited by the execution rate of the OB in which they are placed. The OB that the instructions are placed in must be executed often enough to detect all transitions of the CU or CD inputs. For faster counting operations, see the CTRL\_HSC instruction**

**When you place counter instructions in an FB, you can select the multi-instance DB option, the counter structure names can be different with separate data structures, but the counter data is contained in a single DB and does not require a separate DB for each counter. This reduces the processing time and data storage necessary for the counters. There is no interaction between the counter data structures in the shared multi-instance DB**.

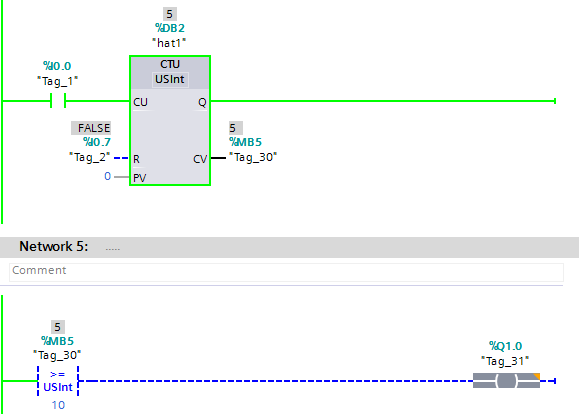
**Counter data retention after a RUN-STOP-RUN transition or a CPU power cycle**

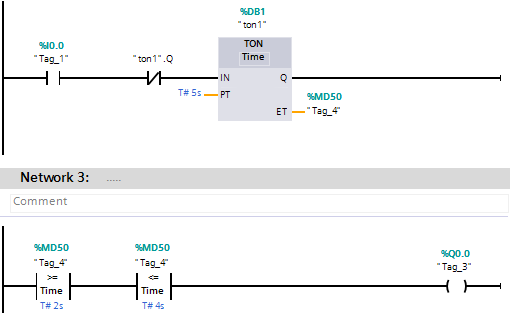
If a run mode session is ended with stop mode or a CPU power cycle and a new run mode session is started, then the counter data stored in the previous run mode session is lost, unless the counter data structure is specified as retentive (CTU, CTD, and CTUD counters).

When you accept the defaults in the call options dialog after you place a counter instruction in the program editor, you are automatically assigned an instance DB which cannot be made retentive. To make your counter data retentive, you must either use a global DB or a Multiinstance DB.

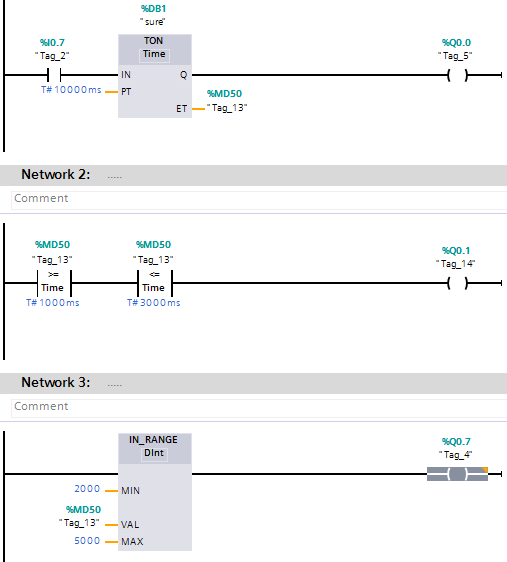
Karşılaştırma Komutları



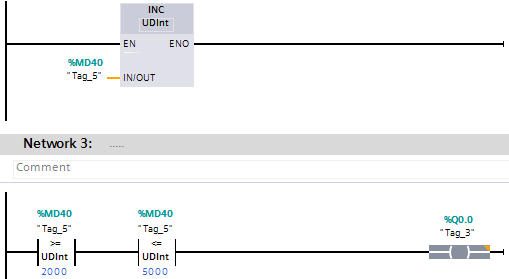




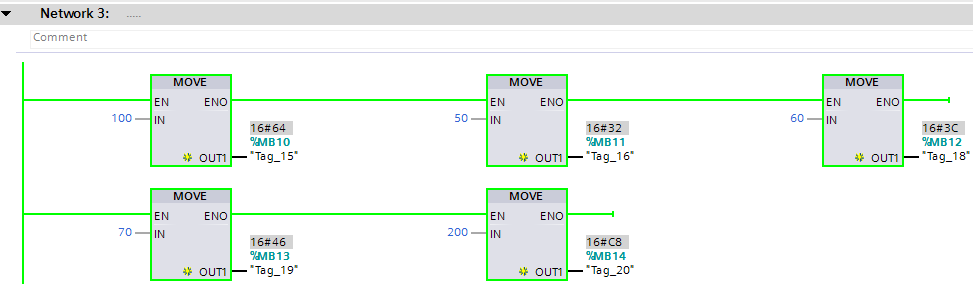
Karşılaştırma… IN RANGE OUT RANGE

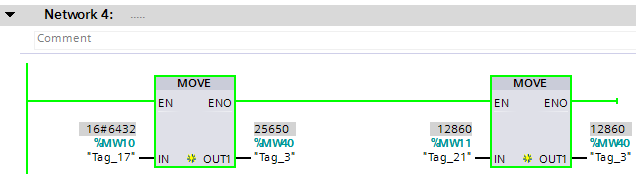


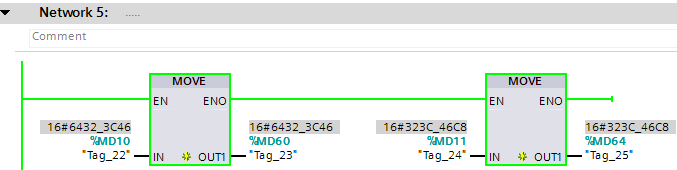
INC …



MOVE Komutu







Reel SAYI..

Sayıcılar Problem1

