

Basics on the SFC Language

1 SFC In general

Even though *Sequential Function Chart* (SFC) is called a programming language, it is actually not one. It is designed as an aid to making structured programs, particularly in the control of operations that have a sequential nature. The language is flexible and can be used at several levels: From the top level, where SFC can be used to describe the main states in a process, to a lower detailed level for code events within the main states.

In other words, SFC can be used to partition (divide up) control-tasks in the same way as top-down design of programs. It is a big advantage to be able to use a state diagram that has been worked out beforehand as a jumping-off point. For all major programming tasks—at any rate, those of the sequential type—this is a natural way of proceeding. In that way, we get a good overview of the sequence, the states, and the conditions for transitions between the states.

A sequence in SFC consists of three main elements: *steps*, *transitions*, and *actions*:

- Steps are most often related to the individual states or phases that are to be controlled.
- Transitions contain conditions that must be satisfied in order for the control to proceed from one state to the next.
- Actions are associated with the individual steps and define events and instructions that are to be performed in the individual process phases.

Figure shows an example of a function chart in SFC. In general, function charts can be described as follows:

- The start of the program consists of a special type of step with double side edges¹ that are called the initiation step. This step is activated automatically when the PLC is set to Run mode. It is also to this step the program usually returns to, either by means of programmed returns or after the program sequence has been completed.
- The other blocks (with single side edges) are steps. In Figure 1, these blocks are given identifiers such as Stir and Drain. These often represent a particular state or phase in the process that is to be controlled.
- One or more instructions/actions are performed in association with each step. These can be actions associated with outputs or changes in internal variables.
- Small horizontal lines are entered between steps. These mark the transitions. These determine when and where the PLC will continue carrying out the code. When the transition is complete, the step before the transition is deactivated and the step after the transition is activated.
- Branching can be used to create alternative or parallel sequences.
- It is also possible to jump between steps that are not directly associated with each other.

How does it work?

When the PLC is set in Run mode, the initiation step Ready is activated. When the operator presses the start button ($\text{Start} = \text{TRUE}$), the PLC begins the sequence by deactivating the Ready step and activating the Fill step. This step remains active until the tank is full (level sensor *Tank full* sends a logically high signal). Then Fill is deactivated and Stir is activated. This step is to remain active for a certain time. This is indicated by the next transition, $\text{Stir.T} > t\#30s$. This comparison is satisfied (TRUE) when a built-in timer has reached a value of 30 seconds. Then the Stir step is deactivated and the Drain step is activated. When the tank is empty (sensor signal *Tank empty* becomes TRUE), Drain is deactivated and the initiation step Ready is activated again. The program is now ready for a new run. It is this neat and elegant way of

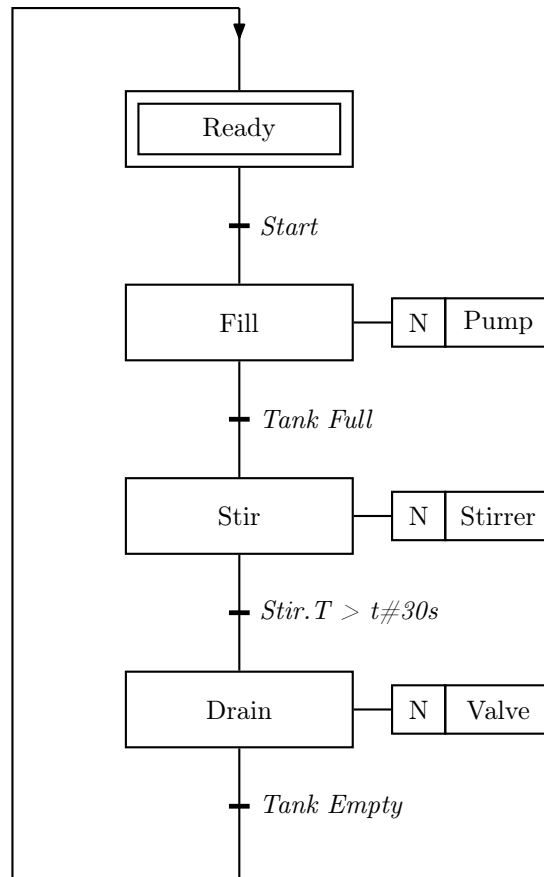


Figure 1: Example of SFC

programming sequences that should make SFC a clear first choice in many PLC applications:

- Steps that show the phases and states in sequential processes.
- Transitions that define only the conditions that must be satisfied in order for the sequence to continue from one step to a succeeding step in the sequence. Transitions are (often simple) Boolean conditions.
- Actions that couple control of outputs to the individual phases in the process

1.1 SFC Basics

Graphic Symbols

As with the symbols in the other graphical languages, the standard does not impose any requirements for the symbols in SFC to have any particular graphic format, but rather specifies a semigraphic format that utilizes only text characters. The individual manufacturers naturally enough choose a fully graphic format, but the forms of the symbols in the standard must be followed even though the types of lines and colors may vary. Figure shows an overview of the symbols in SFC as they can look in a fully graphic format.







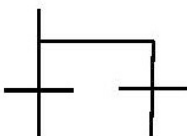
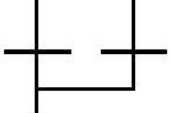
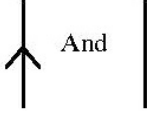
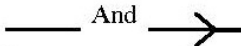

Steps	Initiation step (one of these)		
	Ordinary step		
Transitions	Between two steps		
Parallel sequences	Parallel branching (AND divergence)		
	Parallel convergence (AND convergence)		
Alternative sequences	OR divergence		
	OR convergence		
Connecting lines	Up and down		
	Left and right		
Jumps	Jump to StepX		

Figure 2: SFC Graphics Symbols

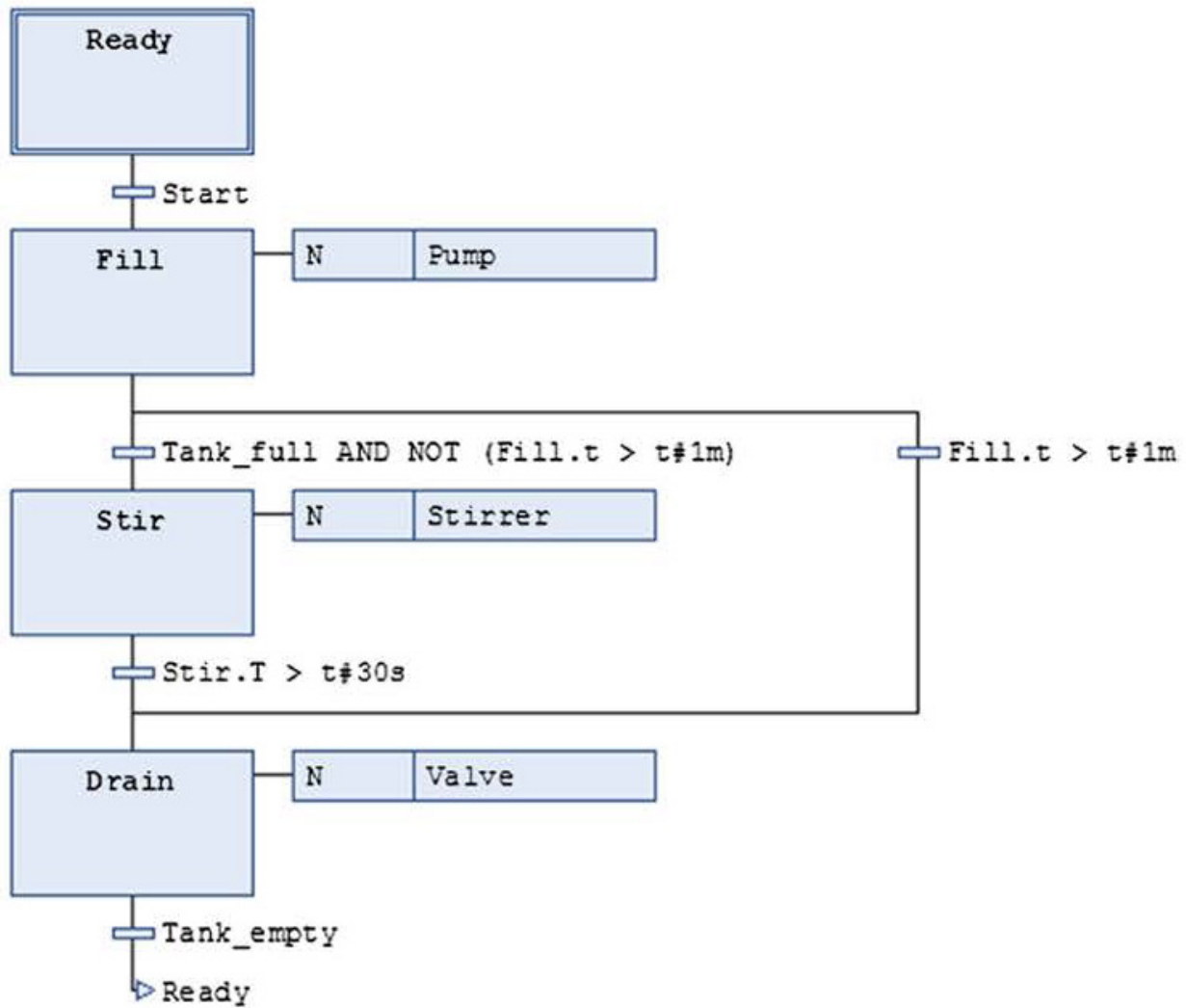


Figure 3: SFC with alternative branches

Alternative Branches

In the example in Figure 1, there is only one possible route in the sequence, but a SFC can contain many alternative branches. Let us therefore expand the example by adding the following: If the pump has been running for more than 1 minute without the *Tank full* giving a high signal, the sequence should jump to Drain. See Figure 3 This addition constitutes an alternative sequence, where the sequences first split up (a so-called OR divergence), to rejoin again farther down (OR convergence). After the divergence (after the Fill step), there follows one transition for each alternative branch. It is extremely important that these transitions be mutually exclusive. This means that the conditions in these transitions must be such that only one of the transitions can be satisfied at any time, thus the word OR. If such transitions are not mutually exclusive, two or more alternative branches can be activated simultaneously and that is regarded as an error. In Figure 3, the condition AND NOT Fill.T>t#1m prevents the two branches from being activated simultaneously. The alternative branching in this example does not contain any step. This is because the purpose of this particular alternative branch is just to jump over a step. Similarly, alternative branching can be used to create a loop that performs one or several steps a desired number of times. Note this alternative way of jumping to another step by using the jump symbol (see Figure 2), where you enter the name of the step to which you want to jump, in this case, Ready.

Parallel Branches

It is also possible to activate parallel branches in SFC, that is, branches that will be executed in parallel with each other. This branching does not need to contain an equal number of steps, but the branches are

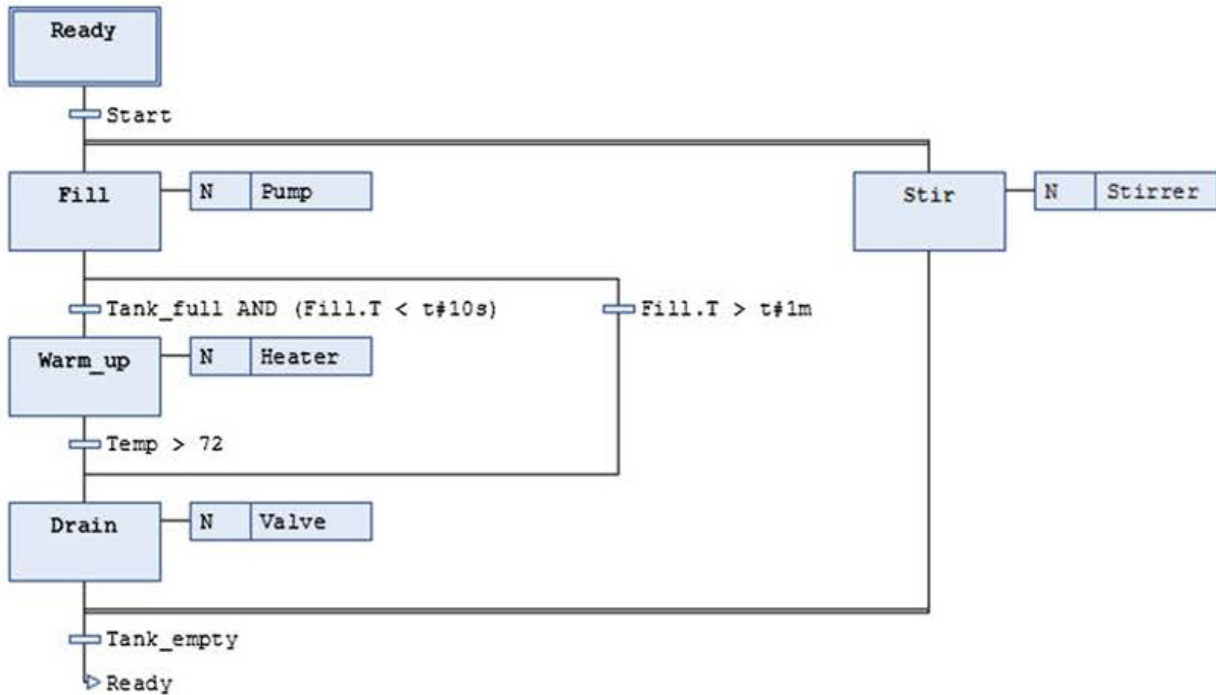


Figure 4: SFC with alternative branches

activated simultaneously and are terminated simultaneously. For this reason, there is a common transition condition before the place where the sequences diverge and one common transition condition immediately after the place where the sequences converge. be as shown in Figure 4. Note that the parallel sequences are activated by one and the same transition (**Start**) and that they converge before a common transition (**Tank empty**). Let us alter the SFC diagram in Figure 4 by making the following modification: We would now like to have the product mixture (the batch) warmed up after the tank is filled. Furthermore, we want the stirring to take place in parallel with filling, heating, and emptying. The result can be as shown in Figure 4. Note that the parallel sequences are activated by one and the same transition (**Start**) and that they converge before a common transition (**Tank empty**). Also note that parallel branches are executed independently of each other before they reach the convergence point. In order for parallel sequences to be deactivated, the following conditions must be satisfied:

- All the sequences must have reached their last step. (Both **Drain** and **Stir** must be active in Figure 4.)
- The transition after the convergence must be satisfied. (**Tank empty** must be TRUE in Figure 4.)

This means that the transition after the convergence is not evaluated by the PLC before all parallel branches are in their last step

Steps

We have seen that each individual step, including the initiation step, is assigned an identifier (a symbolic name). This name must be unique within the POU in question. This means also that the name is local within the POU so that a step in another POU can have the same name. A step is either active or inactive, but several steps can be active simultaneously. It is not possible to place two steps consecutively without a transition in between. One or more actions can be associated with each individual step. These actions can be programmed in any of the languages in the IEC 61131-3 standard.

Each individual step that is used in the sequence is assigned two variable: **Step_name.X** and **Step_name.T**. These variables can be used in programming transitions and actions:

- **Step_name.X** is a Boolean variable that is TRUE when the step is active. When the step is inactive, the state is FALSE. This variable can be used to perform actions when the associated step is activated or deactivated or continually as long as the step is active. The variable can also be used in transitions.

Note: It is not possible to manipulate the state of this variable. In that case, it would result in an error message from the compiler.

- **Step_name.T** is a variable of the data type TIME. All steps in SFC, including the initial step, have built-in timers. As soon as a step is activated, the built-in timer starts and runs until the step is deactivated. In other words, the variable contains the elapsed time for the step in question. With Codesys, when the step is deactivated, the timer is reset to $t\#0s$.

As mentioned previously, the variables **Step_name.X** and **Step_name.T** can be accessed and used in programming actions and transitions, but they cannot be manipulated.

Transitions

The conditions of the transition determine when the previous step is deactivated and the following step activated. In order for a transition to be tested at all by the PLC, all of the steps directly above the transition must be active. When the transition is satisfied, the previous step is deactivated first, before the following step is immediately activated. Two transitions cannot be placed adjacently without a step in between.

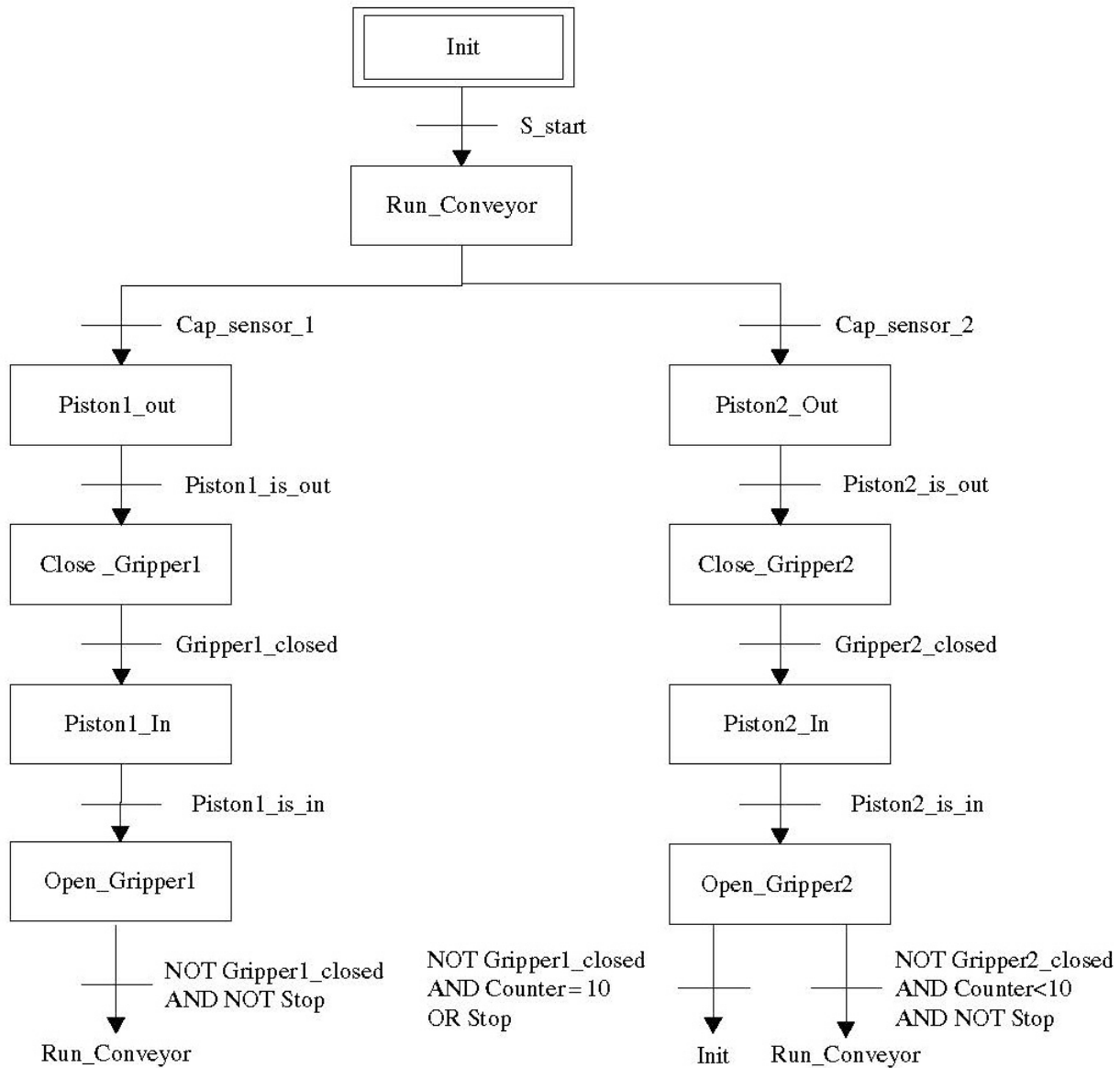


Figure 5: SFC for automatic packaging facility

Figure 5 shows the sequence for an automatic packaging facility. All of the transitions are simple Boolean variables or Boolean expressions. The result of testing a transition is therefore either TRUE or FALSE. A transition can be programmed by exploiting one of the IEC 61131-3 languages.

Actions

One or more actions (instructions) can be associated with each separate step. If no actions are associated with a step, the step is either a delay step or a step that functions to converge alternative branches. There are naturally rules for how actions and instructions are presented in the function chart. In the IEC standard, an individual action is presented as a rectangular box that is associated to the step in question.

See Figure 6, this direct link to the step makes it easy to see where and how the actions are activated or initiated. The first field in the rectangle always contains a qualifier. This is a character, or possibly two characters, that identifies the type of action that is to be performed. If the qualifier is a time type, a time is also shown in the TIME format in this field. The next field contains the action or name of the action that is to be performed. Most often, this is a single action or instruction that typically changes the state of a Boolean variable. In such cases, the action is the name of a Boolean variable. If there are other types

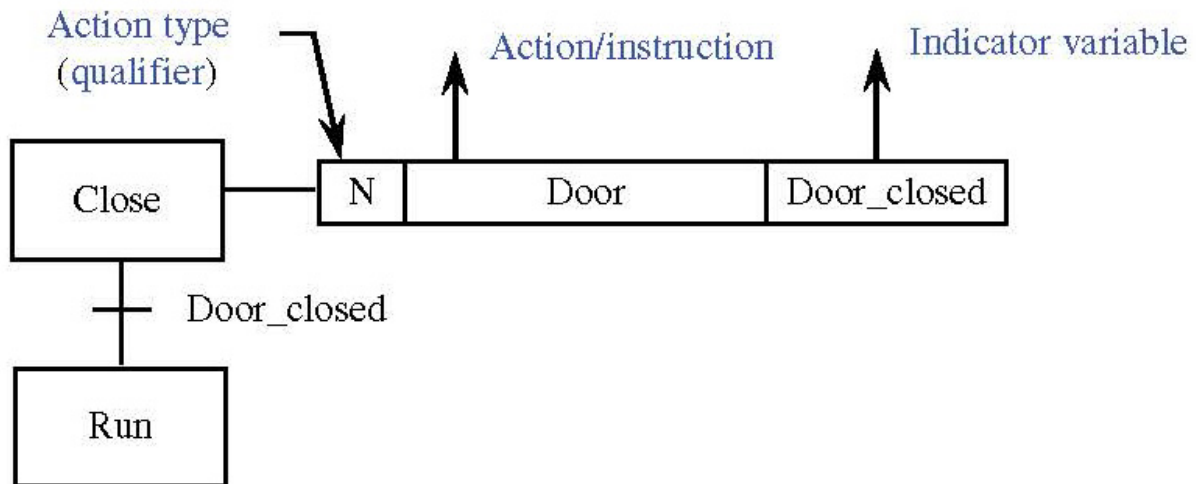


Figure 6: Example of action

of instructions or more instructions that are to be performed, a user-selected (but permitted) name is given in the action field. This name points to a named program code that contains a set of instructions. In both cases, the qualifier specifies which action is to be executed. The third and last field can contain an indicator variable, if needed. Such indicator variables are not a requirement in the standard, and it is therefore up to the manufacturer to implement the capability if he wishes to provide it. Usually the indicator variable will be a variable that is changed as the result of the action that is being performed. Often, the indicator variable will be used as a condition in the next transition as in the figure.

Qualifier	Type	Description
N	Non-stored	Action that is performed as long as the associated step is active
S	Set (Stored)	Stored action. Performed until it is reset
R	Reset	Deactivates a stored action
P	Pulse	A pulse action that is performed once each time the step is active. (See Section 12.5.2)
L	Time Limited	Time-limited action. Stops after a given time or when the step is deactivated
D	Time Delayed	Time-delayed action. Starts after a given time if the step is still active
SD	Stored and time Delayed	Stored and time-delayed action. The action is set active after a given time, even though the step deactivates before that. The action continues until it is reset
DS	Time Delayed and Stored	Time-delayed and stored action. If this step is still active after the specified time, the action will start. It will run until it is reset
SL	Stored and time Limited	Stored and time limited. The action starts when the step becomes active and will continue during the given period or until it is reset
P1	Pulse—rising edge	A pulse action that is executed only once when the step is <i>activated</i>
P0	Pulse—falling edge	A pulse action that is executed only once when the step is <i>deactivated</i>

S—stored; D—delayed.

Figure 7: Action Types/Qualifiers

Action Types/Qualifiers

To make it easier to control how and when actions are executed, the standard specifies a set of action types (see Figure 7). The first three types in the table, N, S, and R, are the ones most used, along with actions of type P (or P1, and P0 if these are implemented). An N was given in the example earlier. The letter N refers to an action of the Non-stored type and indicates an action that is to be performed continually as long as the associated step is active. This choice is also the default. The other action types, all of which deal with time, are needed only in exceptional cases. One of the reasons for this is that the built-in timers in the steps are sufficient for most cases of time management of the sequence. Furthermore, it is usually a good design technique to create the function chart so that you avoid actions that need to be activated and deactivated, “independent” of the process in the sequence. This increases the programmer’s control and thereby reduces the risk of unforeseen events.

Action Control

It is important to be clear that the standard defines that all actions are executed one extra time after the action is deactivated. This means that the actions are performed at least twice. This does not apply to actions of type P1 and P0. These are executed only once. Such an extra execution is illustrated in Figure 8 for a type N action. This rather strange requirement for an extra execution is luckily not an absolute requirement, because it sometime creates trouble. When it comes to the time-related action types L, D, SD, DS, and SL, use of this will require that the associated delay or duration (of the TIME type) be stated. To what extent this can be done in the qualifier field or elsewhere depends upon the implementation. In the same Figure 8 it can be noticed an action type SD. Here, the time is stated in the qualifier field together with the action type. The figure also shows the functional principle for an action of the SD type. Note that all action types that are stored must be reset at another place in the sequence by using the qualifier R and giving the same action name. This also applies to action types S, SD, DS, and SL. The possibilities of activating actions in different ways are also significant for how we build up the sequence.

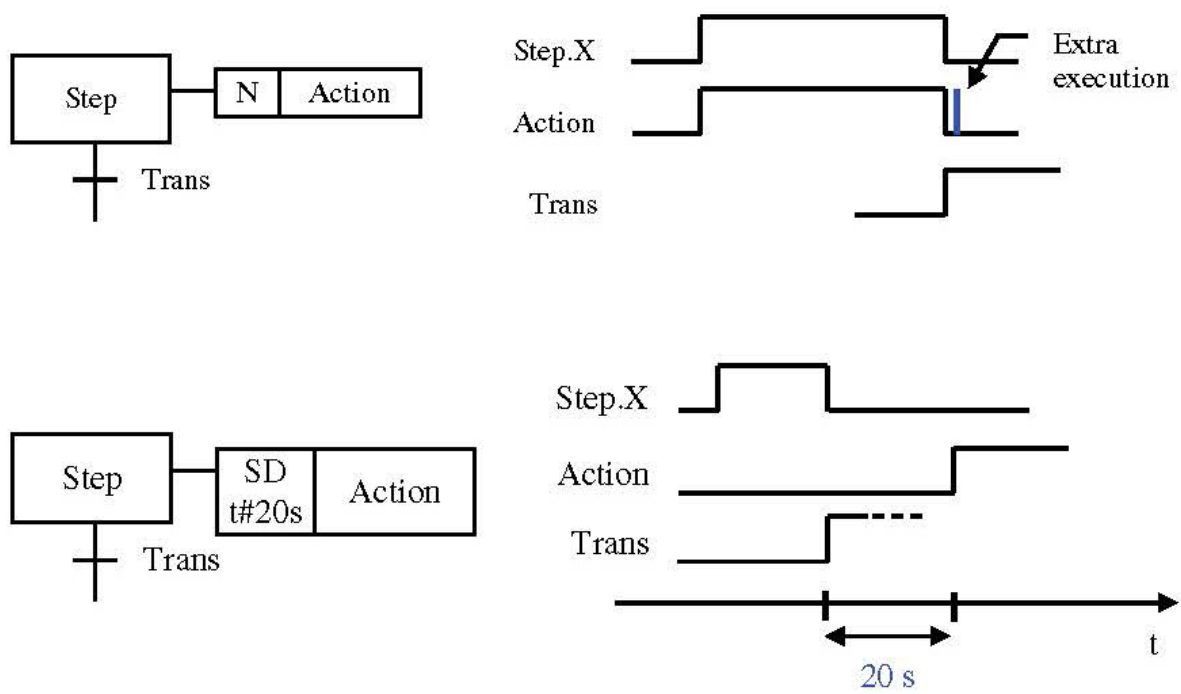


Figure 8: Extra execution

Variable	Type	Functional action
SFCInit	BOOL	When the flag is set TRUE, all steps are deactivated and the initiation step is activated. All steps, actions, and other flags are reset. Nothing is processed again until the flag is set back to FALSE
SFCReset	BOOL	Corresponds to SFCInit except that the initiation step is processed (the flag can therefore be reset in the initiation step)
Pause_SFC	BOOL	As long as this variable is TRUE, all execution of the diagram will stop. Execution of actions also pauses so that the state of outputs, for instance, freezes
SFCTrans	BOOL	The flag becomes TRUE as soon as a transition has been performed
SFCCurrentStep	STRING	This variable stores the name of the step as active at any time. If several steps are active (parallel sequences), the name of the step farthest to the right is registered

Figure 9: Some Codesys variables used to Diagram execution control

Control of Diagram Execution

Manufacturers who implement SFC as one of the languages also implement some special objects (flags) that can be used as a kind of external control of SFC execution. Such objects are not implemented in the standard, and it is therefore not possible to define them in general. However, it is possible to say something general about them. No matter which of the development tools you use, where SFC is one of the languages, it is probable that the following objects (flags) are defined (although possibly with different names): **SFCInit**, **SFCReset**, and **SFCPause**. CODESYS defines these flags plus a few others for control of time, error management, and information. Some of these are shown in Figure 9, but there could be many others, so check the documentation.

1.2 Good Design techniques

Good structure and design of the SFC chart is naturally important. One thing is that the program should be comprehensible and logical. Another thing that is even more important is that the program has to be capable of being run and that possible conflicts are eliminated. In particular, improper use of alternative and parallel sequences is something that can quickly create problems. We will try to explain this by means of two examples.

Figure 10 shows a sequence where a classic error has been committed. An alternative sequence diverges within a parallel sequence. Since the transitions that activate the branching in an alternative sequence must always be mutually exclusive, Step_C and Step_D can never be active simultaneously. Therefore, the parallel sequence can never be closed and Step_F and the subsequent step will never be reached. A simple way to fix the problem in this example is to introduce a “step” that converges the alternative branches before the AND convergence. In the example above, the design error was catastrophic in the sense that the processing comes to a halt. Faulty design can also, under certain circumstances, function for a while and then suddenly generate a serious error. See the next example.

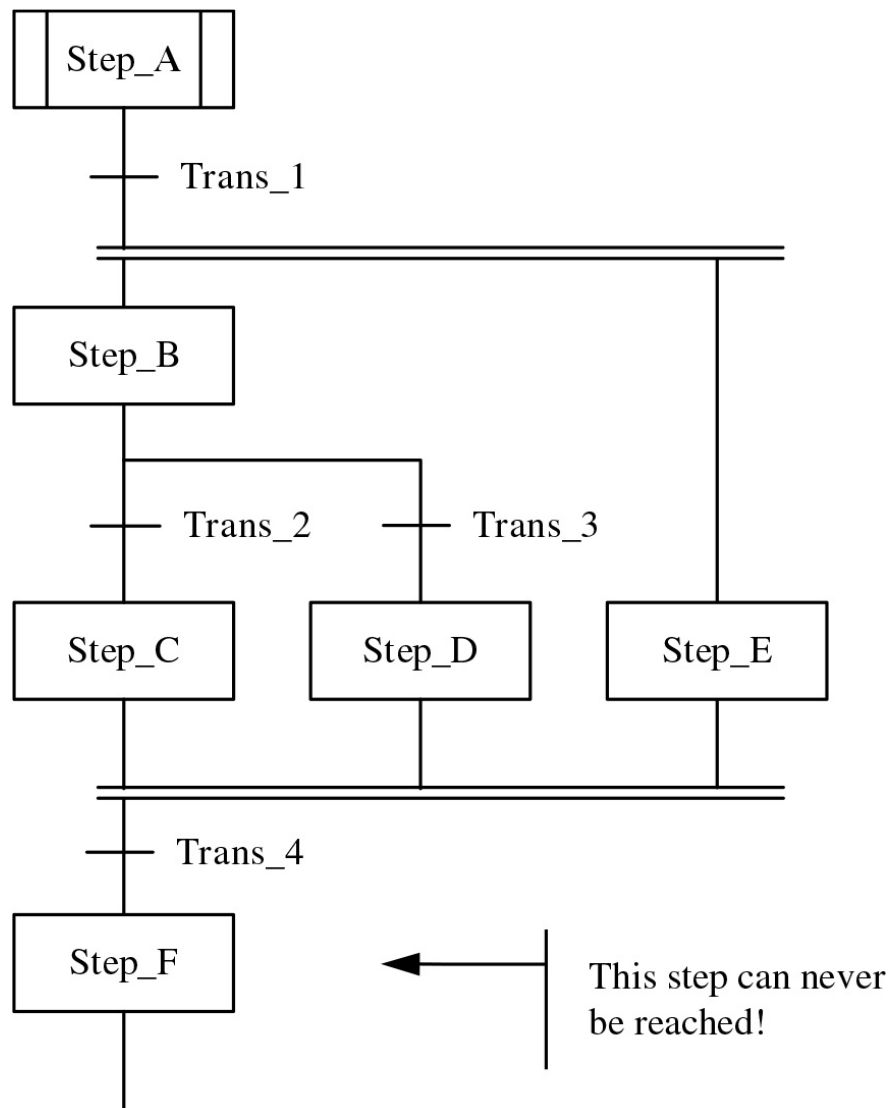


Figure 10: Wrong SFC design

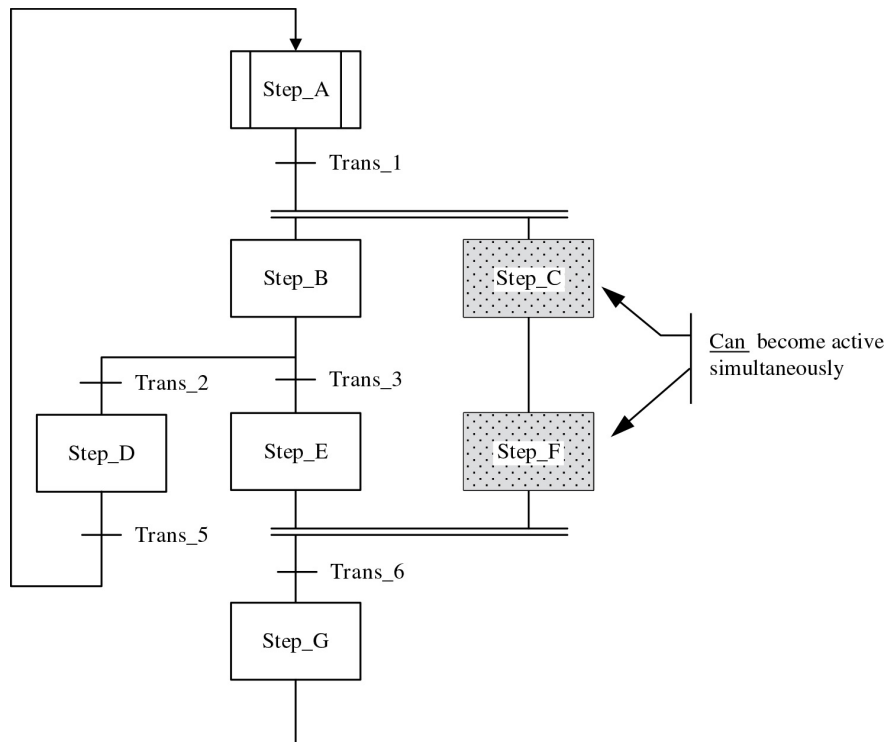


Figure 11: Risky SFC design

In the diagram in Figure 11, there is a branching out of the parallel sequence, and this can create unforeseen problems. The following can happen here (assume that Step_A is active):

- Trans_1 is satisfied, which will activate Step_B and Step_C.
- Then assume that Trans_2 is satisfied in addition to Trans_4. This means that Step_D and Step_F are activated.
- Now, when Trans_5 and Trans_1 are satisfied, Step_C becomes activated again. Thereby two steps (C and F) in the same branch are active simultaneously, something that is not permitted under the standard.

What consequences this will cause depends upon how the manufacturer has implemented SFC. In the worst case, the PLC will go into an Error mode

A cookbook

Possible recipes for working on designing a good sequence in SFC. It is in no way applicable to all possible problems, but perhaps it can be of some help.

1. Think through the sequence(s) and particularly into how many steps the sequence ought to be divided. If you have done proper preliminary work, you will have one or more flowcharts or state diagrams as a starting point. It can often be a good idea to split up the problem into several sequences. When it is a large process that is to be controlled (large with respect to complexity), it is probably a good idea to use macro-steps. Then, for example, you will have an overall sequence with several subsequences, one for each macro-step. It is also possible to call up another sequence via an action.
2. Also evaluate whether alternative or parallel sequences are needed:
 - You need an alternative branch if there are several possible paths through the sequence, and only one of them should be chosen. Then each transition to each branch has its own condition.
 - You need a parallel branch if there are two or more branches to be traversed simultaneously.

Then there will be a common transition to all the branches.

3. Construct the sequence(s).
4. Program the transitions. These usually consist of simple Boolean tests or time conditions. Remember to use mutually exclusive conditions for alternative sequences.
5. Program the actions. Individual Boolean actions are stated directly in the SFC diagram, but other actions are programmed as objects under the POU and are called up from SFC.
6. There will also probably be a requirement to manage other external events that may take place such as power failure or activation of an emergency stop. Here, implicit SFC variables such as SFCPause and SFCInit can be useful.