



Automation with PLC

Laboratory Guide

Microcontroller Laboratory Exercises

BMEVIAUAC08

Kovács Gábor

gkovacs@iit.bme.hu

Budapest University of Technology and Economics

Department of Control Engineering and Information Technology

2022

Table of contents

1	Introduction.....	3
2	Ladder Diagram	4
2.1	Boolean logic	4
2.2	Most important functions and function blocks.....	6
2.3	State machine-based control	9
3	Connected Components Workbench	13
3.1	Creating a new project	13
3.2	Program organization units	16
3.3	Variables	16
3.4	Ladder Diagram editor.....	18
3.5	Downloading the project to the PLC	21
3.6	Automated generation of project documentation.....	24
4	Measurement tasks	25

1 Introduction

The aim of the laboratory practice is to put basics of PLC programming into practice by controlling a physical process, namely a model of a manufacturing line.

The PLC used during the lab is an AB Micro 830 2080-LC30-24QBB PLC (Figure 1), a model of the compact controller range of Rockwell (Allen-Bradley), which is designated for stand-alone execution of simple automation tasks. As a compact device, its housing contains a 24V DC power supply, the CPU, and the IO modules. Micro 830 PLCs can be equipped with three expansion modules, providing further analog or digital IOs and other functions.



Figure 1 - Micro 830 PLC

As indicated by its model number, the PLC 2080-LC30-24QBB is equipped with 24 IO lines, out of which 14 are digital inputs and 10 are digital outputs, both using 24V voltage levels and being isolated from the CPU. Inputs are divided to two groups, each of which can be used also in sink and source configurations, while the transistor outputs are source ones. Digital states of input and output lines are reported by LEDs located at the front of the PLC.

Program memory of the CPU can store up to 10 000 instructions, size of the data memory is 20kB. The PLC is equipped with USB and RS232/RS485 serial interfaces, the former serves as the programming connector. Some other models of the Micro 800 family are also equipped with a further Ethernet interface.

As a special feature, selected digital inputs are capable of high-speed counting and selected digital outputs support pulse-width modulation (PWM), so these models can be used in standalone servo and stepper motor control applications. The 2080-LC30-24QBB model can control two axis, motion control functions are supported by functions and function-blocks provided by the manufacturer.

Development for Micro 800 PLCs is supported by the Rockwell Connected Components Workbench (CCW) integrated development environment, which also provides configuration and programming tools for selected HMI (Human-Machine Interface) and drive devices. CCW supports ladder diagram (LD), structured text (ST) and function block diagram (FBD) programming languages for Micro 800 PLCs.

2 Ladder Diagram






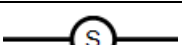
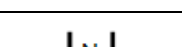

For simple automation problems, ladder diagram is one of the most commonly used among the programming languages defined by the IEC 61131-3 standard. In the sequel, the fundamentals of ladder diagram programming are summarized using the notations of the Rockwell Connected Components Workbench development environment.

Like relay logic circuits, basic elements of the ladder diagram programming language are the **rungs** (circuits connecting the power and the neutral rail). In each cycle, the PLC reads its physical inputs, then processes the rungs from the top to the bottom, and finally sets the values of the physical outputs. The control algorithm can be defined by placing elements in such rungs.

2.1 Boolean logic

Basic Boolean elements used in rungs of ladder diagrams are given in Table 1¹.

Table 1 - Boolean logic elements (*terminology used by CCW*)

Contacts		Coils	
	Normally open (NO) contact (<i>Direct Contact</i>)		Normally open (NO) coil (<i>Direct Coil</i>)
	Normally closed (NC) contact (<i>Reverse Contact</i>)		Normally closed (NC) coil (<i>Reverse Coil</i>)
	Rising edge sensing contact (<i>Pulse Rising Edge Contact</i>)		Set coil (<i>Set Coil</i>)
	Falling edge sensing contact (<i>Pulse Falling Edge Contact</i>)		Reset coil (<i>Reset Coil</i>)

Rungs of the diagram mostly realize Boolean functions: inputs of the functions are represented by the contacts while its outputs are represented by the coils. According to the analogy with relay logic circuits, current flows from the power rail (left-hand side of the rung) through contacts and coils towards the neutral rail (right-hand side of the rung). Operation of a normally open (NO) contact can be imagined as a simple pushbutton: it conducts (i.e. current can flow through it) if the Boolean variable associated to the contact (indicated above the element in the diagram) is **true** (1). A normally closed (NC) contacts acts inversely: it conducts if the variable associated to the contact is **false** (0). Series connection of NO contacts (NC contacts) realize an AND (NOR) function on their associated variables, while parallel connection of NO contacts (NC contacts) realize an OR (NAND) function on their associated variables.

If current flows through a normally open (NO) coil, it assigns the value **true** (1) to the variable associated to it, and assigns the value **false** (0) if no current flows. A normally closed (NC) coil acts **inversely**: it assigns the value **true** (1) if there is no current flowing through it, otherwise it assigns the value **false** (0).

¹ Development environments of Rockwell traditionally denote coils by a full circle instead of the standard $-()-$ notation.

Figure 2 shows the implementation of the Boolean function $xY = \text{NOT}(xA \text{ OR } (xB \text{ AND NOT } xC))$ in ladder diagram. Out of the two horizontal branches in the left, the topmost conducts if the value of variable xA is `true`, while the bottom branch conducts if the value of xB is `true` and value of xC is `false` simultaneously ($xB \text{ AND NOT } xC$). Current flows through the coil if either of the branches conducts, i.e. $xA \text{ OR } (xB \text{ AND NOT } xC)$. As the coil at the right is a normally closed (NC) one, the variable xY is assigned the value `false` (0) if one of the branches at the left conduct and is assigned the value `true` (1) if neither of the branches conduct, hence the rung realizes the Boolean function $\text{NOT}(xA \text{ OR } (xB \text{ AND NOT } xC))$.



Figure 2 - Ladder diagram implementation of a simple Boolean function

It is important to underline that the rung shown by Figure 2 realizes a Boolean function, which assigns a value to its output every time it is evaluated. Beginners often consider that operation of such a rung is the same as the operation of the code segment

```
IF (xA OR (xB AND NOT xC)) THEN xY:=0
```

However, there is a significant difference between the two: if the condition is not fulfilled, the textual code segment does not assign any value to xY (i.e. xY keep its previous value), while the ladder rung assigns the value `true` (1) to xY if the expression $xA \text{ OR } (xB \text{ AND NOT } xC)$ evaluates to false. The operation of the ladder rung hence can be defined by the code segment

```
IF (xA OR (xB AND NOT xC)) THEN xY:=0 ELSE xY:=1.
```

Normally open and normally closed coils assign their associated variable a value according to the state of the elements on their lefts in each PLC cycle. If a variable needs to be associated a memory-like behavior, i.e. its value needs to be changed only if a given condition is met, retentive coils, so-called Set and Reset coils can be used for assignment. A Set coil assigns the value `true` (1) to a Boolean variable if current flows through it, and it does not change the value of the variable otherwise. Similarly, a Reset coil assign the value `false` (0) to the associated Boolean variable if current flows through it and does not change its value otherwise.

It is important to keep in mind that the user program does not change values of the outputs directly: operations in the program modify bits of the output map stored in the memory only. Bits of the output map are copied to the physical outputs only after finishing the evaluation of the last rung. Therefore, if an output is assigned a value in multiple rungs, the last assignment overwrites results of previous assignments. It is hence strongly advised to assign a value to an output in one single rung only.

2.2 Most important functions and function blocks

Beside simple bit logic operations, ladder diagrams can realize more complex operations in forms of function and function block calls. This chapter summarizes the use of standard timer and counter function blocks, comparison functions and non-Boolean assignment operations.

Timers

Connected Components Workbench supports the use of all three standard function blocks (TON, TOF, TP). Each type of timer has two input parameters: **IN** (BOOL), corresponding to the Boolean value to be delayed and **PT** (TIME), which defines the duration of delay. Outputs include **Q** (BOOL), which is the delayed value of the Boolean input, and **ET** (TIME), which reports the duration elapsed since the start of the timer. Implementation of a TON timer in the Connected Components Workbench environment is illustrated by Figure 3, with delay specified by the 3 seconds valued duration literal (T#3s) connected to the **PT** input of the function block.

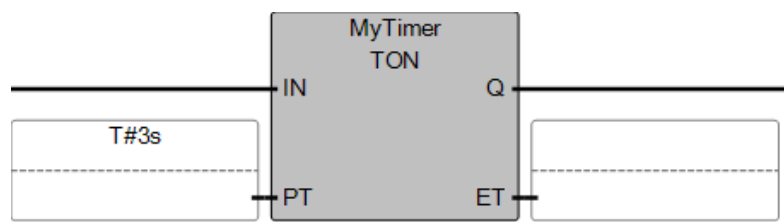


Figure 3 - Timer function block

The on-delay timer (TON) delays the rising edge of its input signal (**IN**) by the duration specified by the value connected to its **PT** input. The output of the timer (**Q**) is set to **true** (1) if the value of the input has been constantly **true** for the required duration and is immediately reset to **false** (0) when the input changes to **false** (see Figure 4).

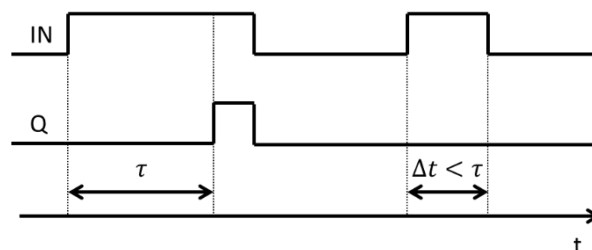


Figure 4 - Time diagram of a TON timer

The output of an off-delay timer (TOF) is set to **true** (1) immediately if a rising edge is sensed on its input and is set to **false** (0) after the specified duration following the falling edge of the input. Note that after it has been set to **true** (1), the output of the timer is reset to **false** (0) only if the input has been constantly **false** (0) during the period defined by the input **PT** (see Figure 5).

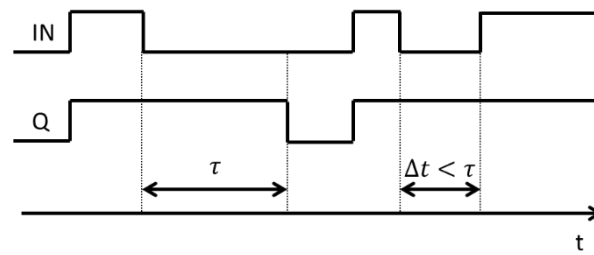


Figure 5 - Time diagram of a TOF timer

Output of a pulse timer (TP) is set to `true` (1) if a rising edge is sensed at the input and is kept active for the duration specified by the value connected to the input `PT`, then is reset to `false` (0). The output is set to `true` (1) again upon the next rising edge of the input. Changes of the input while the output is active do not influence the state of the timer (see Figure 6).

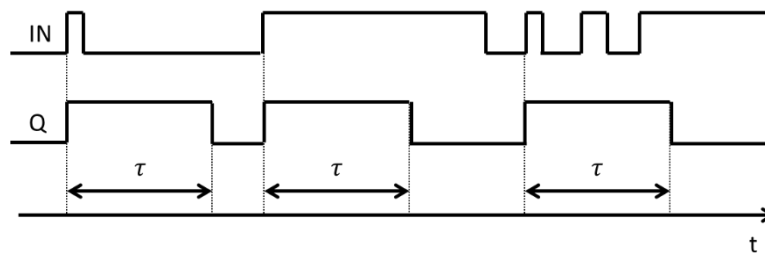


Figure 6 - Time diagram of a TP counter

Beside the standard timer function blocks, Connected Components Workbench also provides a timer implementing both on-delay and off-delay timing (TONOFF) and a retentive on-delay timer (RTO).

Counters

Connected Components Workbench supports all standard counter types: up-counters (CTU), down-counters (CTUD) and bidirectional up-down-counters (CTUD). As the former two can be considered as special subtypes of the most complex CTUD block, only the operation of the latter one is detailed here.

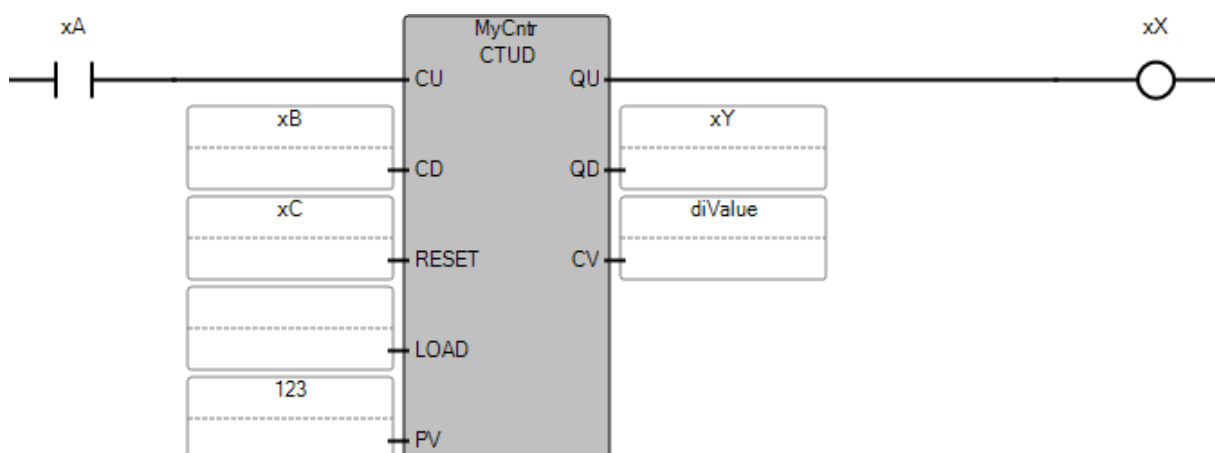


Figure 7 - Bidirectional up-down counter (CTUD)

Boolean counting inputs of the block are rising edge-sensing, hence the internal counter register is incremented every time a *rising edge* of the signal connected to the CU (Count Up) input is detected and decremented every time a *rising edge* of the signal connected to the CD (Count Down) input is detected. As consequence, if the value of the signal connected to, for example, the input CU is kept constantly active (`true`, 1), the value of the counter register does not change.

Active (`true`, 1) level of the signal connected to the Boolean RESET (R) input of the counter resets the value of the internal counter register to 0, while active level of the signal connected to the Boolean LOAD (LD) input sets the value of the internal counter register to the numerical value connected to the DINT-typed PV (Preset Value) input.

Value of the DINT-typed internal counter register is available at the CV (Current Value) output of the function block. The value of the Boolean output QU is true if $CV \geq PV$, while value of the Boolean output QD is true if $CV \leq 0$. Unlike the behavior defined in the standard, CCW limits the value of the internal counter register between zero and PV: $0 \leq CV \leq PV$. Therefore, if a counter is used for up-counting, a value greater or equal to the expected maximal counter value shall be connected to the PV input of the function block.

In case of up-counting CTU function blocks, inputs CD and LOAD and output QD are not available. Similarly, inputs CU and RESET as well as the output QU are not available for CTD function blocks.

Comparison operators

Comparison of non-Boolean (e.g. numeric) variables and literals is supported by comparator functions, which evaluate the selected relational operator on two values of the same data type, connected to the inputs i1 and i2. Relational operators available are <, <=, =, >=, >, <> (not equal). The first operand is always the value of i1, while the second is i2, i.e. result of a < operator is false if 42 is connected to i1 and 31 is connected to i2 (42<31 evaluates to false).

Comparison functions can be inserted to a rung by their Boolean EN (enable) input and o1 output, latter reporting the result of comparison. If the value of the EN input is false, the comparison is not carried out and the value of the output is set to false. If the block is enabled (i.e. EN is true), o1 equals the result of the relational operator on the two non-Boolean inputs, i1 and i2.

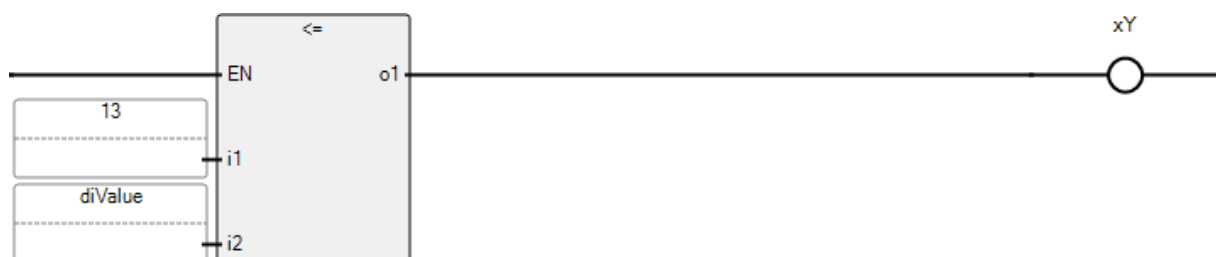


Figure 8 - Comparison function (example)

Assignment

Non-Boolean (e.g. numeric) assignments can be carried out by the MOV function, which can be inserted into a rung by its Boolean EN input and ENO output. If the value of the EN input is true, the value of the variable or the literal connected to the input i1 is assigned to the variable connected to the output o1. Types of the variables connected to input i1 and output o1 need to be the same (e.g. both INT). Figure 9 illustrates the implementation of the assignment `diValue := 17` in ladder diagram.

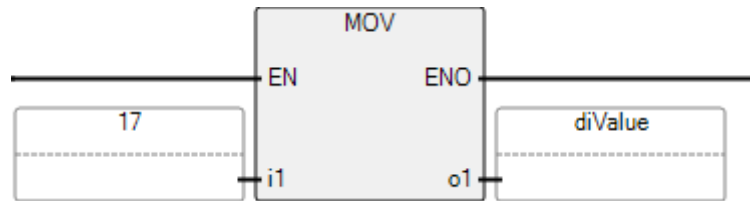


Figure 9 - Non-Boolean assignment using the MOV function

2.3 State machine-based control

In many automation tasks, the desired operation can be described in a form of a state machine, which can be then implemented as a ladder diagram. Such processes are mostly described as Moore-machines, i.e. value of each output is a function of the active state only.

Consider the control of a hand dryer machine, like the one you can find in many public restrooms. The machine is equipped with a blower, which provides high-speed air flow, and a heating element to keep the temperature of the air warm. These elements can be controlled (i.e. turned on and off) by the outputs `xBlower` and `xHeater`, respectively. There is also a proximity sensor installed at the bottom of the equipment, which sets the value of the corresponding input (`xProxy`) to `true`, if the hand of the user is under the machine (otherwise the value of the input is `false`).

In the idle state of the hand dryer, both the heater and the blower are turned off. When the user puts his or her hand under the machine, both the blower and the heater need to be turned on and hot air should be blown (*Hot* state `H`). Upon removing the hand, the heater is turned off immediately, but the blower keeps running for 5 further seconds (*Warm* state `W`). If the value of the proximity sensor returns to `true` before 5 seconds elapse (i.e. the hand reappears), the heater is turned on again. Otherwise, 5 seconds after the falling edge of the proximity sensor, blower is turned off. The state transition diagram of the hand dryer is depicted by Figure 10.

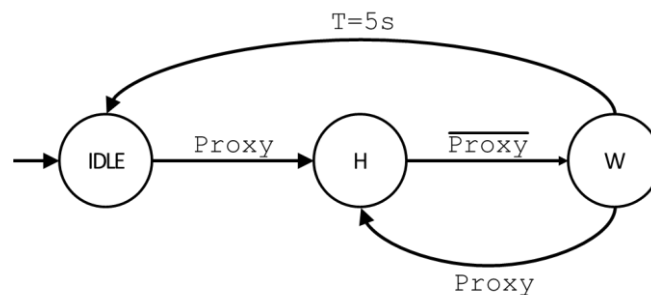


Figure 10 - State transition diagram of the hand dryer

To implement the state machine-based control algorithm in ladder diagram, the four steps below should be followed:

1. Define Boolean state variables
2. Implement state transitions
3. Implement output mapping
4. Ensure correct initialization of the state machine

At first, a Boolean variable (memory bit) needs to be associated to each state. This variable shall be set to `true` (1) if the given state is the active one and `false` (0) otherwise. Note that since only one state of a state machine can be active at a time, if a state variable is `true`, all others need to be `false` simultaneously. Considering the hand dryer, the three Boolean state variables will be denoted by `xStateIdle`, `xStateHot` and `xStateWarm` in the sequel.

Transitions of the state machine define a mapping from state-condition pairs to states, i.e. it gives whether if a state is active and a given condition is fulfilled, which state shall be activated. Therefore, to check whether a transition might take place, it shall be both verified whether the origin state of the transition is active, and the condition of the transition is met. If yes, the origin state of the transition needs to be deactivated (the corresponding Boolean state variable shall be set to `false`) and the destination state needs to be activated (the corresponding state variable shall be set to `true`). Ladder diagram implementation of a generic state transition is shown by Figure 11.

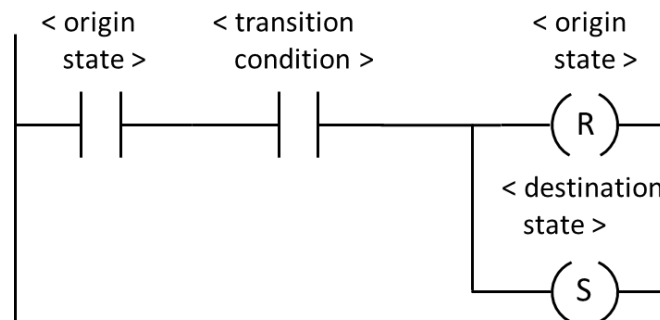


Figure 11 - Ladder diagram implementation of a state transition

In the figure `<origin state>` and `<destination state>` denote the Boolean state variables associated to the origin and destination states of the transition, while the contact labeled `<transition condition>` represents the condition of the transition. Note that, although represented by a single contact in the figure, condition of the transition might be any Boolean-valued expression, even an output of a function or a function block.

Figure 12 shows the ladder diagram implementation of state transitions for the hand dryer. Note that the condition of the transition from state `W` to state `IDLE` is the duration for which the state `W` has been active (recall that the blower needs to be switched off 5 seconds after the user removes his or her hand from below the machine).

Such a timed transition can be implemented with ease by using an on-delay (TON) timer function block with the state register of the origin state connected to its Boolean input. When the origin state is activated, the timer is started, and it sets its output `true` after the duration specified by the value of its PT input has elapsed. When the output of the timer becomes `true`, the state variable of the origin state is reset and the state variable of the destination state is set. Note that, according to the operating principle of the on-delay timer (see Figure 4), if the origin state is active for a duration shorter than the duration setting of the timer, the timer is deactivated and its output remains `false`, hence the transition is not executed.

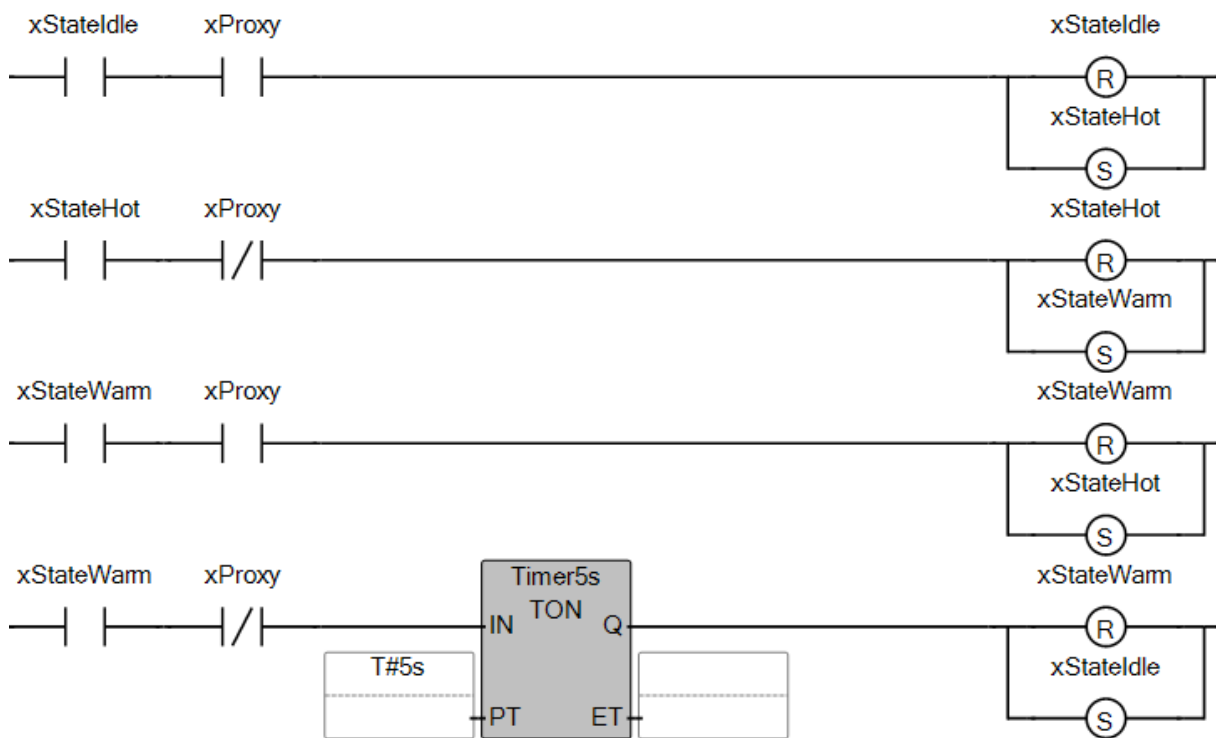


Figure 12 - Implementation of transitions for hand dryer control

Implementation of output mapping shall assign values to outputs according to the actual state. The mapping can be described by a table, like the one shown by Table 2 for the hand dryer.

Table 2 - Output mapping for hand dryer control

State	IDLE	HOT	WARM
xBlower	0	1	1
xHeater	0	1	0

It is strongly advised to implement the output mapping according to the rule that each output shall be assigned in one single rung only. The best practice is to define a rung for each output, containing a single coil associated to the output variable (see Figure 13). In such rungs, the state variables of states in which the given output is active, shall be associated to contacts in parallel connection. Hence if any of these state variables is active, the coil assigns the value `true` to the output variable. Since only one state variable can be active at one time, if a state not associated to any contact in the rung is active, none of the contacts conduct, hence the output is assigned a `false` value. Alternatively, an NC coil can also be used to assign value to the output, but in that case the state variables of states in which the output is inactive shall be associated to the contacts.

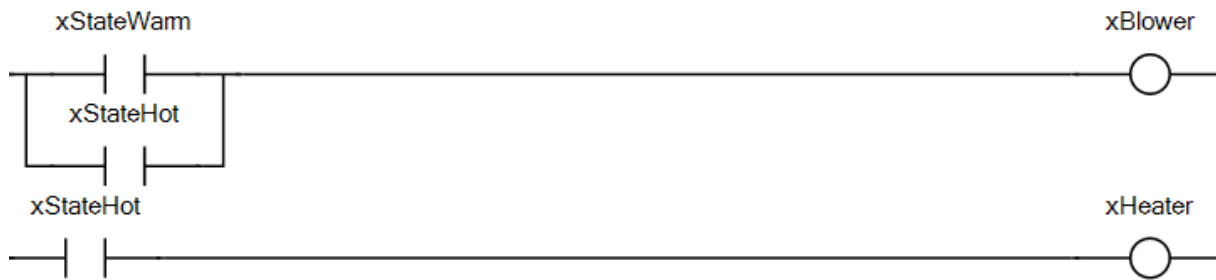


Figure 13 - Implementation of output mapping for the hand dryer

When the PLC is started, Boolean variables are initialized to `false` by default, hence no state is marked as active and the state machine is unable to operate. To ensure the desired behavior of the program, the state register associated to the initial state shall be initialized to `true` when the PLC starts.

Like most development environments, Connected Components Workbench allows setting instance-specific initial values to variables, so correct initialization of the state machine can be ensured by simply setting the initial value of the initial state register (`xStateIdle` for the hand dryer) to `true` during declaration. Note that if no instance-specific initial value is specified, the variable is initialized to the default value of its data type (`false` for Boolean variables).

3 Connected Components Workbench

Connected Components Workbench (CCW) is an integrated development environment (IDE) providing a uniform platform for configuring and programming selected Rockwell PLCs, HMI devices and drives.

Full-feature Standard edition of the IDE can be downloaded free of charge from the following URL (from version 12, the standard edition of CCW contains also a simulator with limited capabilities):

<https://www.rockwellautomation.com/global/capabilities/industrial-automation-control/overview.page?pagetitle=Design-and-Configuration-Software&docid=6b8425507e679dd6b2587d385b7b26e1>

3.1 Creating a new project

Like other state-of-the-art environments, CCW is also project-based. The first step of development is therefore to create a project, which stores the application along with its settings and hardware configuration.

A new project can be created by clicking on the *New Project...* link of the welcome window, by the *File > New...* menu command or by using the first icon of the toolbar. Name and location of the project can be specified in the New Project dialog shown by Figure 14. If the *Add Device on Create* option is checked, the wizard for adding hardware components is launched automatically after the project has been created.

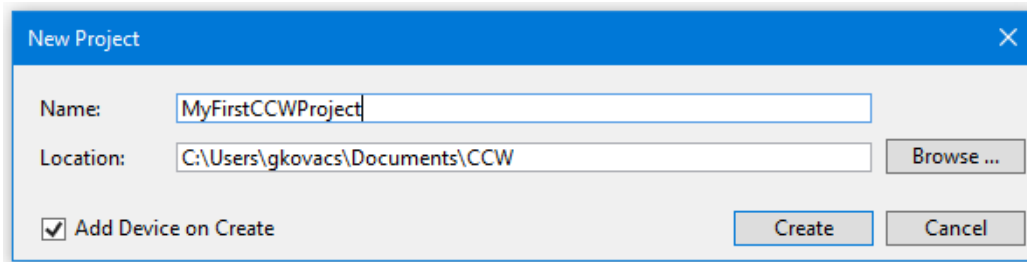


Figure 14 - New Project dialog

PLCs and other devices can be added to the project using the *Add Device* dialog. Figure 15 shows how the model used during the laboratory practice can be added to the project by selecting the *Controllers > Micro 830 > 2080-LC30-24QBB* item from the list and finally clicking the *Add To Project* button.

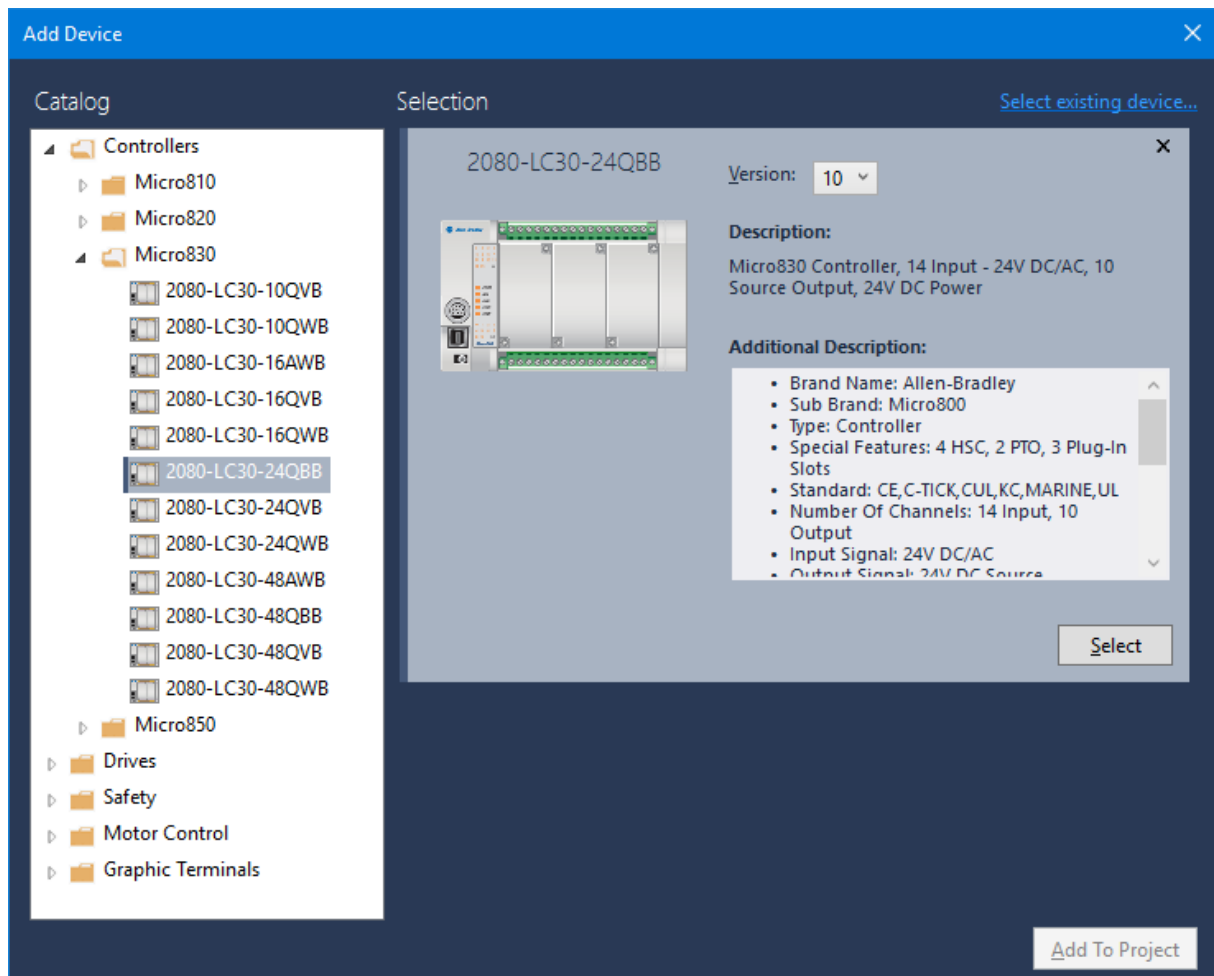


Figure 15 - Adding the PLC to the project

After creating the project and adding the PLC, the main window shown by Figure 16 is displayed. The left panel shows the project tree (*Project Organizer*) which lists devices included in the project (here the only element is the Micro 830 PLC). Project elements associated to the given device (program organization units, global variables and type definitions in case of a PLC) are shown as nodes of the project tree (see Figure 17).

The main panel of the window shows the configuration screen of the PLC by default. If an element (e.g. a program organization unit or a variable block) is double-clicked in the project tree, the corresponding screen (e.g. program editor or variable declaration table) is displayed in a tab of the main panel.

The *Toolbox* panel, located at the right-hand side of the window, shows the tools corresponding to the active tab of the central panel (e.g. contacts, coils and other elements can be added from the *Toolbox* panel if a ladder diagram editor tab is selected). Layout of the screen can be reconfigured, panels can be closed and reopened using the buttons at the header of panels or the commands of the *View* menu. It is recommended to pin the *Toolbox* panel, so elements of the ladder diagram can be simply dragged to the editor pane.

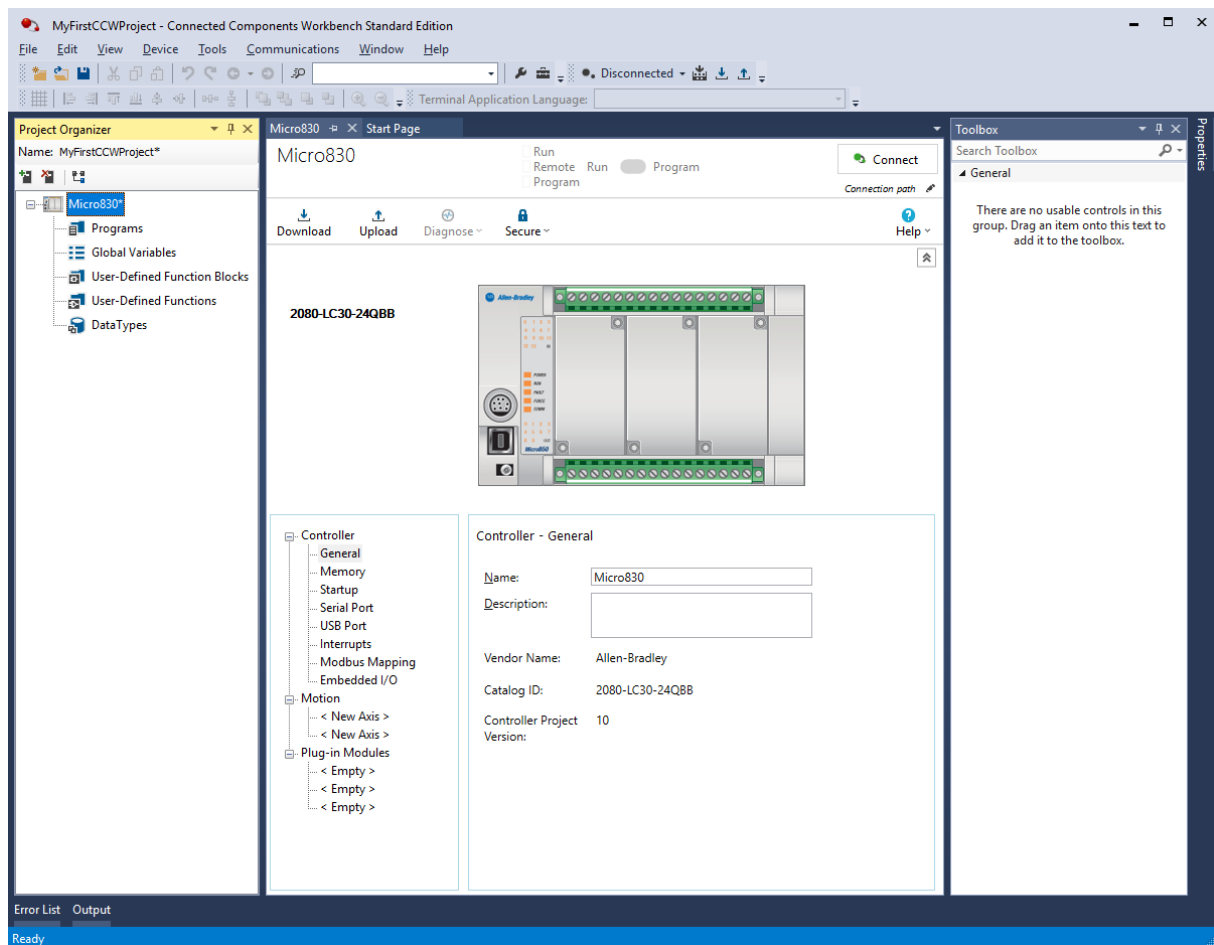


Figure 16 - Default layout of CCW

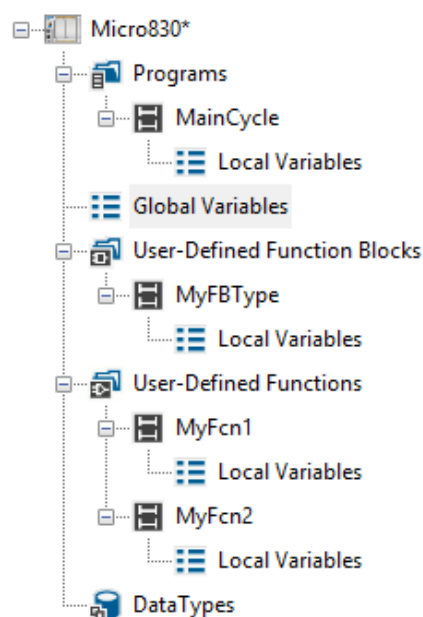


Figure 17 - Structure of the project tree

3.2 Program organization units

Connected Components Workbench supports all program organization unit (POU) types defined by IEC 61131-3: programs, function blocks (FBs) and functions.

The highest-level POU is the program, which is executed cyclically by default (tasks are not supported by CCW, however, programs can be configured to run periodically or upon an IO interrupt). Function block instances and functions can be called from user programs.

Beside standard (timers, counters, bistable elements, edge-sensing) and Rockwell-specific function blocks, the user can define further FB types (see the *User-Defined Function Blocks* branch of the project tree). Note that beside the definition of their types, the instances need to be declared and called in a program or function block to use such FBs.

User-defined functions can be defined in a similar way (*User-Defined Functions* branch) and can be called in any other POU.

Program organization units can be added to the project by right-clicking on the appropriate node of the project tree (*Programs*, *User-Defined Function Blocks*, *User-Defined Functions*) with the `Add > New LD: Ladder Diagram` option (ST and FBD languages are also supported). Newly created POUs are assigned an automatically generated name by default (e.g. `Prog1`), which can be changed by right-clicking the corresponding node in the project tree and selecting the `Rename` option.

3.3 Variables

Terminology of Connected Components Workbench uses the terms *global* and *local* variables. Global variables are associated to the PLC itself and can be used in every POU running on it without further declarations. Local variables are assigned to POUs (programs, function blocks or functions) and depending on their types, might be local or interface variables according to standard terminology. Local variables in CCW can be input variables (*VarInput*), output variables (*VarOut*) or internal local variables (*Var*). Input and output variables can be declared for functions or function blocks but not for programs. Internal local variables are however available for all three types of program organization units.

Internal variables (local variables according to the standard nomenclature) can be accessed only inside the POU in which they have been declared. Input variables can be read only inside the POU they are declared, but they can be written from the caller POU. Output variables can be read by the caller but they can be written only inside the POU they are declared in. The caller POU can reference interface variables of a function block by the name of the instance and the identifier of the variable divided by a dot, e.g. `MyTimer.Q`.

Variables declared in a POU are available in the *Local Variables* node under the branch of the given POU in the project tree. Figure 18 shows the declaration table which is displayed upon double-clicking the *Local Variables* node under a POU.

	Name	Alias	Data Type	Direction	Dimension	Initial Value
	xIn1		BOOL	VarInput		
	wIn2		WORD	VarInput		
	xOut		BOOL	VarOutput		
	iValue	IntegerValue	INT	Var		17
-	Timer1		TON	Var		...
	Timer1.IN	IN	BOOL	VarInput		
	Timer1.PT	PT	TIME	VarInput		
	Timer1.Q	Q	BOOL	VarOutput		
	Timer1.ET	ET	TIME	VarOutput		
	Timer1.Pdate	Pdat	TIME	Var		
	Timer1.Redg	Redg	BOOL	Var		

Figure 18 - Variable declaration table

New variables or function block instances can be added by clicking and typing in the last row of the table. Columns of the table are the following (optional fields are in *italics*):

- **Name:** unique identifier of the variable
- **Alias:** alternative name for the variable (the variable can also be referenced by its alias, e.g. the physical input with identifier `_IO_EM_DI_01` can be referenced by the more descriptive alias `xLevelSwitch`)
- **Data Type:** data type or FB type. The type can be selected from a drop-down list.
- **Direction:** type of variable (*VarInput*: input, *VarOutput*: output, *Var*: internal local)
- **Dimension:** can be used to define the indexing range (e.g. `[1..3]`) for arrays
- **Initial Value:** instance-specific initial value of the variable. If omitted, the variable is initialized to the initial value according to its data type (`false`, `0` etc.).
- **Comment:** textual comment assigned to the variable
- **String Size:** length of string variable (available only if String data type is selected)

When opening the declaration table of a program, the *Direction* column is missing since programs support only *Var* type local variables.

Physical IOs and status information of PLCs are available in forms of global variables. Identifiers of digital inputs and outputs are `_IO_EM_DI_xx` and `_IO_EM_DO_xx`, respectively, where `xx` denotes the 0-based two-digit number of the channel. In order to improve legibility of the code, it is strongly recommended to assign descriptive aliases (e.g. `xBtn2`) to the IO variables.

Status information, including scan time (`__SYSVA_TCYCURRENT`) and the first scan bit (`__SYSVA_FIRST_SCAN`), which is active for a single cycle after starting the PLC, are also available as global variables. These special, read-only variables are distinguished by the `__SYSVA_` prefix.

The user can also define global variables by adding a new row to the *Global Variables* table. Such global variables can be accessed in each program organization unit. However, their use is strongly discouraged – it is recommended to pass information between the POU's by interface variables.

3.4 Ladder Diagram editor

The program code editor can be opened by double-clicking a POU node in the project tree. Figure 19 shows the editor for an LD-language program organization unit. The diagram is displayed in the main panel, elements (contacts, coils etc.) can be dragged from the *Toolbox* panel at the right. Elements of the toolbox can be deleted or rearranged, the original state can be restored by right-clicking inside the toolbox panel and selecting the *Reset Toolbox* command.

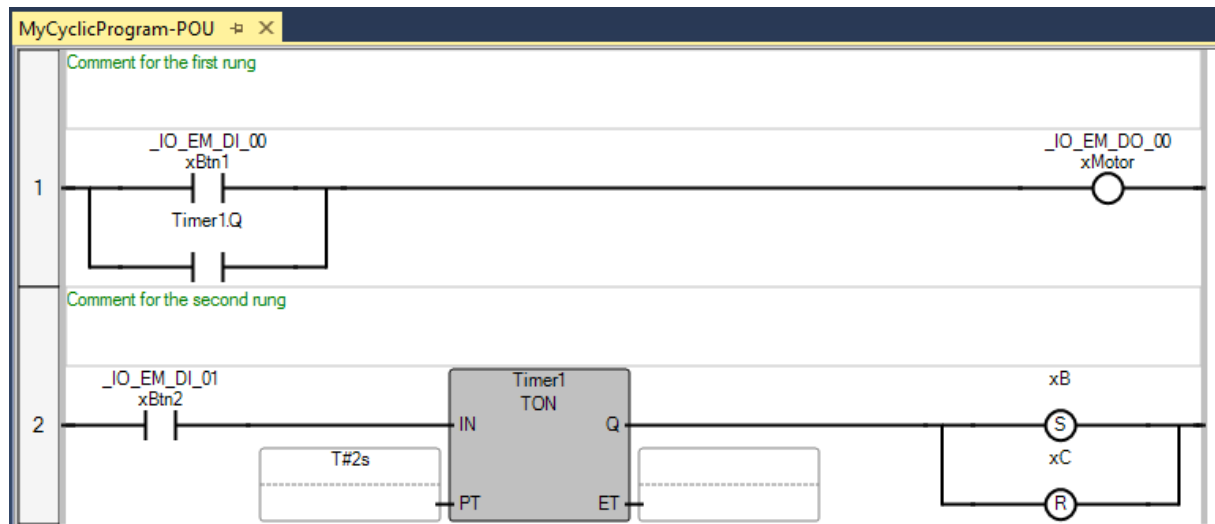


Figure 19 - Ladder diagram editor

A newly created LD-language program organization unit contains an empty rung. Further rungs can be added by dragging the *Rung* element of the toolbox to a place inside the diagram marked by an encircled plus sign. Rungs can be selected by clicking on the numbered field on their left. After selecting a rung, it can be dragged to another location (below or under other rungs) or deleted by pressing the *Del* key. A comment field is located above each rung – it is recommended to describe the operation realized by the rung in a few words. Such comments are also included in the automatically generated documentation.

Elements of the ladder diagram (contacts, coils etc.) can be added in a similar way, by dragging the selected symbol from the toolbox to a place in the diagram marked by an encircled plus sign. Note that coils can be added only as last elements of the rung and they can be connected with other coils in parallel only.

When adding a contact or coil, the *Variable Selector* dialog is opened automatically (see Figure 20), which provides a user-friendly way to select the variable associated to the element. Different types of variables are organized to tabs:

- *Local Variables* – local variables declared inside the POU being edited
- *User Global Variables* - global variables defined by the user
- *System Variables* – *Micro 830* - system and status variables (e.g. cycle time)
- *I/O Variables* – *Micro 830* – variables associated to physical input and output channels

After selecting the tab, the variable to be associated to the element can be selected from the list or by typing in the *Name* field.

Variables can be assigned to elements also by clicking the input field above the contact or coil and typing the name (or alias) of the variable, which provides a faster way than using the *Variable Selector* dialog. Automatic opening of the *Variable Selector* and the *Instruction Block Selector* (see later) dialogs can be turned off by opening the IEC Languages > Ladder Diagram (LD) menu and setting the Editor Settings > Automatically Invoke Variable / Block Selector option to False. Even if the automatic display of selector windows is turned off, such dialogs can be opened by double-clicking an element (contact, coil, instruction block) or by right-clicking and invoking the Show Variable Selector or Show Block Selector command.

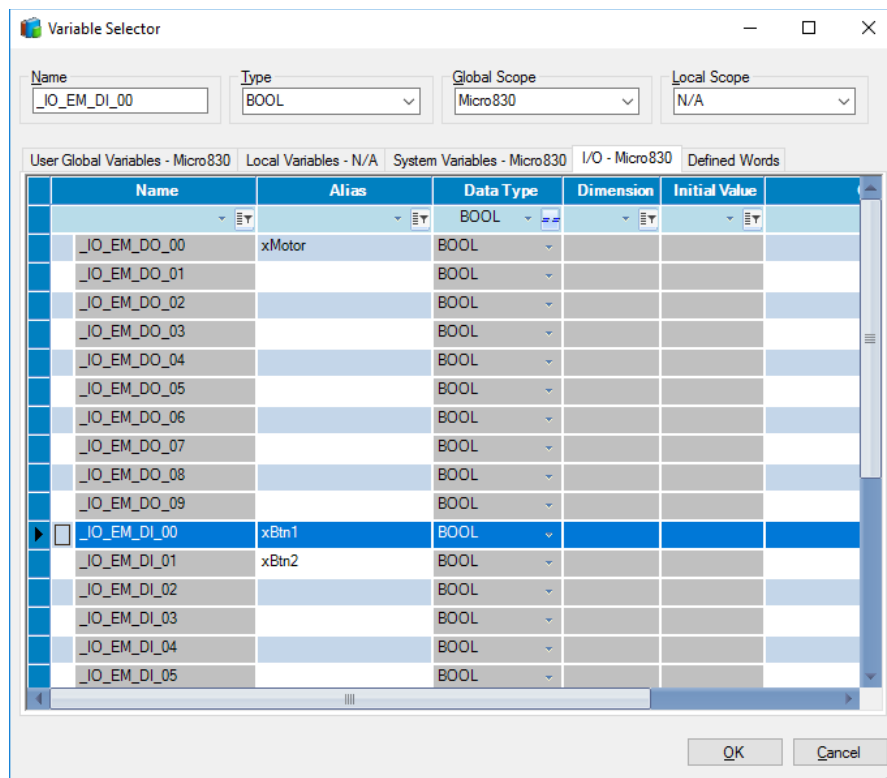


Figure 20 - The Variable Selector dialog

The Branch tool adds a short parallel branch below the selected section of a rung, to which further elements can be added. An alternative way to add a parallel branch is to right-click an element and select the Insert Ladder Elements > Surround with Branch command from the context menu.

Calls of functions and function block instances can be added to a rung by inserting an *Instruction Block* element. The function or function block instance to be called can be selected in the *Instruction Block Selector* dialog (see Figure 21), which is opened automatically by default when an *Instruction Block* element is dragged onto a rung. The dialog lists all standard and user-defined functions and function block types, the list can be filtered by typing in the *Search* field.

If the Show Parameters option is activated, the window shows interface and internal variables of the selected POU. If the mouse pointer is placed above a variable at the right-hand side panel, a short description of its role is also displayed.

If a function block type is selected, the instance to be called can be selected in the *Instance* field located at the bottom of the window. If the instance to be called is already declared, it can be selected from the drop-down list, while a new instance can be declared simply by typing the desired instance name to the *Instance* field (it contains an automatically generated type-specific identifier by default, e.g. TON_1).

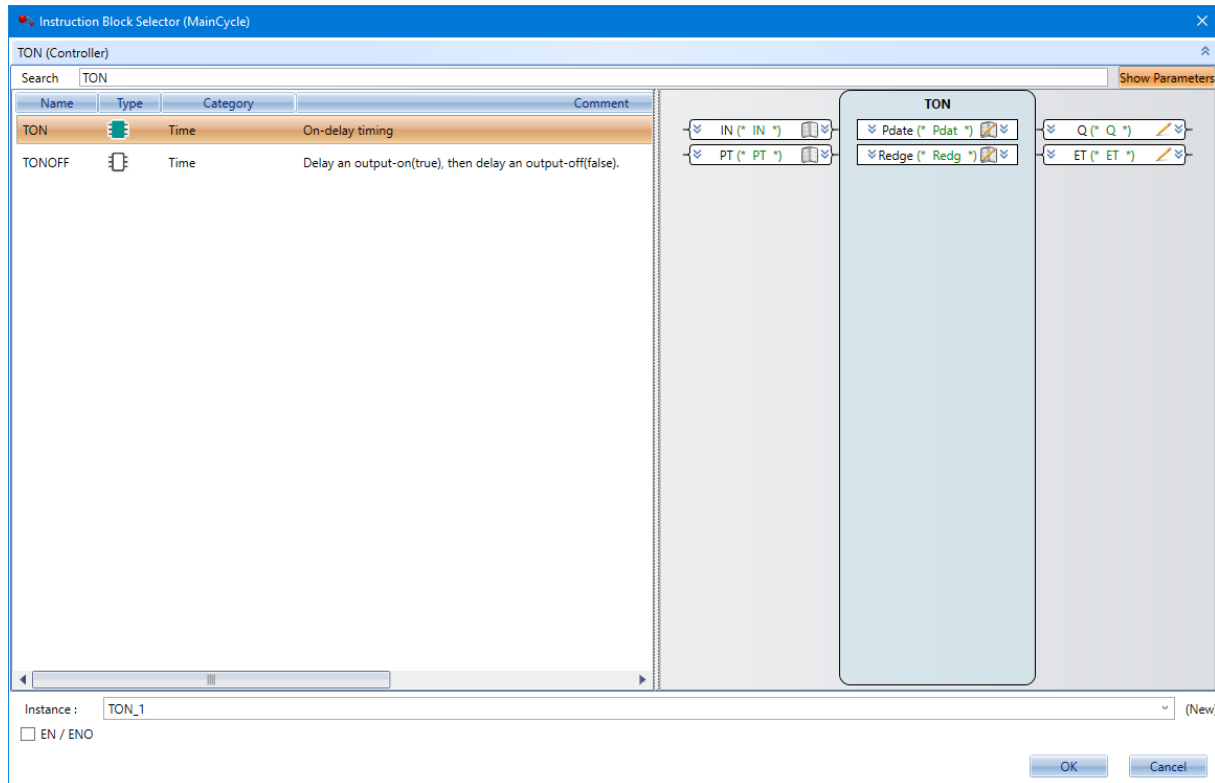


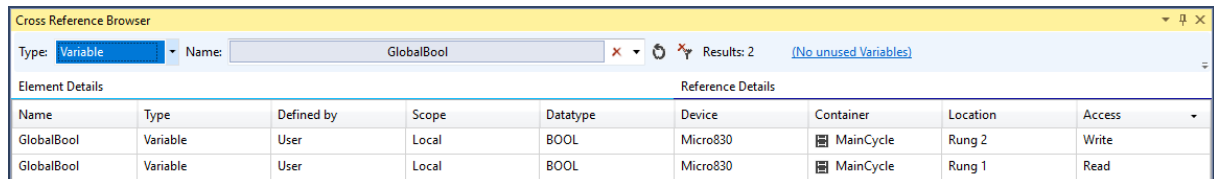
Figure 21 - The Instruction Block Selector dialog

If the checkbox at the bottom-left corner is checked, the *EN/ENO* enable input and output parameters are displayed for functions and function blocks with Boolean inputs and outputs also (for blocks without Boolean inputs and outputs, *EN* and *ENO* are displayed automatically to allow inserting the call to the rung).

Connected Components Workbench allows only the topmost Boolean input and output of an Instruction Block to be inserted to the rung. For other inputs and outputs, only variables (or literals for inputs, e.g. T#3s) can be connected by typing the identifier or literal to the top input field of the box connected to the input or by double-clicking it and using the *Variable Selector* window.

Debugging is assisted by the *Cross Reference Browser* tool, which lists the locations (program organization unit and number of rung within) where the selected variable or function block instance is accessed. The tool can be opened using the *View > Cross Reference Browser* menu command or by right-clicking an element inside the ladder diagram and selecting the *Cross Reference Browser* command. Latter option only displays information about the variable which is associated to the selected element.

The *Cross Reference Browser* panel shows the table depicted by Figure 22, where each row shows a location of access to the selected variable(s). First columns recalls declaration of the variable or the function block instance, most important information are shown by the last three fields. Location of access is identified by the program organization unit (*Container* field) and the number of rung (*Location* field), while the *Access* field shows whether the given variable or function block instance is read, written or called (*invoked*) at the given location. Double-clicking a row of the table opens the editor window of the referenced POU and highlights the rung of the access.



The screenshot shows the 'Cross Reference Browser' window. At the top, there's a search bar with 'Type' set to 'Variable' and 'Name' set to 'GlobalBool'. It shows 'Results: 2' and a link to '(No unused Variables)'. Below this is a table with two main sections: 'Element Details' and 'Reference Details'.

Element Details				Reference Details				
Name	Type	Defined by	Scope	Datatype	Device	Container	Location	Access
GlobalBool	Variable	User	Local	BOOL	Micro830	MainCycle	Rung 2	Write
GlobalBool	Variable	User	Local	BOOL	Micro830	MainCycle	Rung 1	Read

Figure 22 - Window of the Cross Reference Browser

3.5 Downloading the project to the PLC

In order to run the project on the PLC, it needs to be compiled and downloaded to the device (the project is compiled automatically when a download operation is initiated).

Micro 800 series PLCs are equipped with a three-position mode selector switch on their front panel. Switch-selectable operating modes are the following:

- *Run* – The downloaded project is running on the PLC, modification or download of the program is not possible.
- *Program* – The PLC is stopped; a new project can be downloaded to the device.
- *Remote* – The operating mode can be selected from the development environment if the PC is connected to the PLC. The project can be downloaded if the *Remote Program* mode is selected in the IDE and the PLC is started if the operating mode is changed to *Remote Run* (in *Remote Run* mode the program keeps running after disconnecting the PLC from the PC).

Tasks related to the connected PLC can be managed from the configuration screen of the device (see Figure 23), which can be opened by double-clicking the *Micro 830* node in the project tree. Connection to the PLC can be initiated by clicking the *Connect* button in the top right corner (when the PLC is connected, the same button can be used for disconnecting). If the physical mode switch is set to *Remote*, the *Remote Program* or *Remote Run* operating mode can be selected by the switch located at the top of the window.

Download of the project can be initiated by clicking the *Download* button. If the PLC has not been connected before, the connection is established automatically.

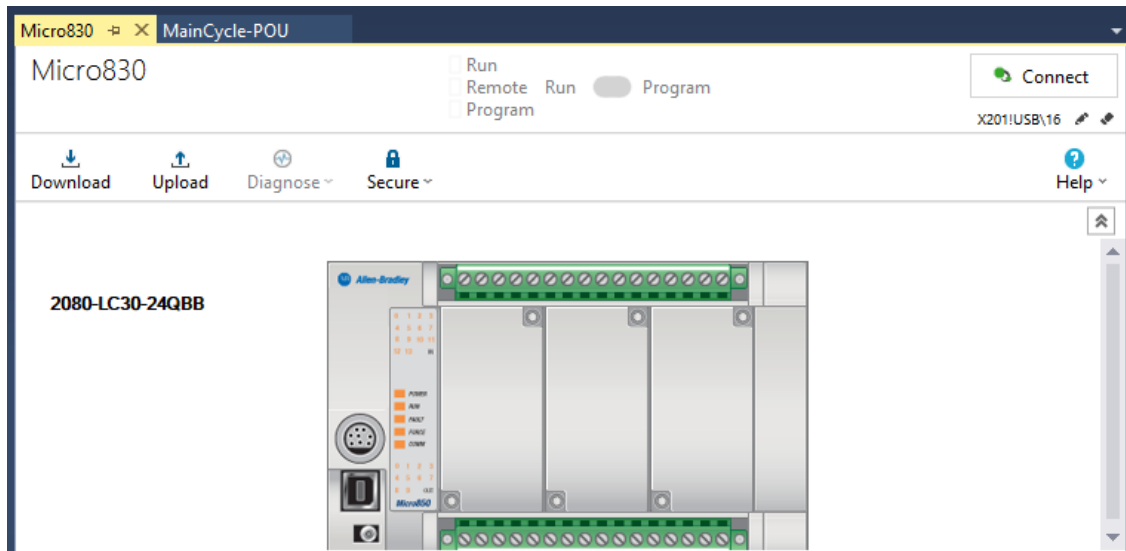


Figure 23 - Connection, download and mode selector controls

Before the first connection to the PLC, the *Connection Browser* dialog shown by Figure 24 is displayed, where the communication channel between the PC and the device needs to be configured. As the programming interface of Micro 830 PLCs is USB, the USB > Micro830 shall be selected.

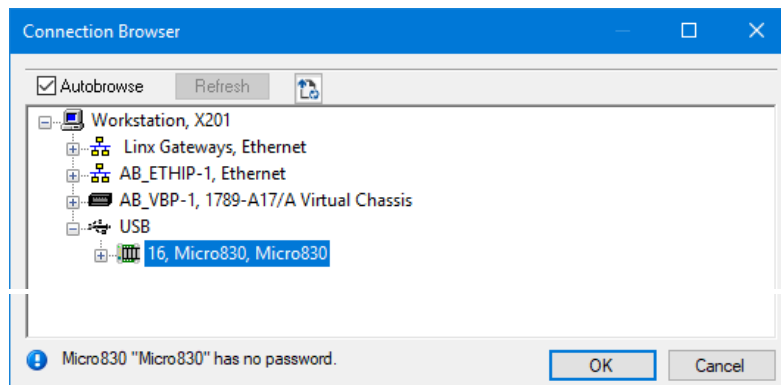


Figure 24 - Selecting the communication channel for connection

Before downloading the project, it is compiled and built automatically (building the project without downloading it can be initiated by the Device > Build menu command). Results of the build process are displayed in the *Output* pane located at the bottom of the screen. If the project is not correct, it can not be downloaded to the PLC and error messages can be displayed by clicking the *Error List* link next to the *Output* pane.

Upon downloading the project, a dialog is displayed to confirm that the current project on the PLC is to be overwritten. If the download is successful, another dialog is displayed which allows switching the PLC to *Remote Run* mode, hence starting the program.

If the PLC is connected and is in *Remote Run* mode, both program editor and variable declaration windows show the online state, i.e. values of variables and execution of program organization units inside the PLC (refresh time is about 500ms, which is much slower than the cycle time). Figure 25 shows the program editor in *Remote Run* mode: parts of the rung in which “current flows” are displayed in red while inactive parts are shown in blue.

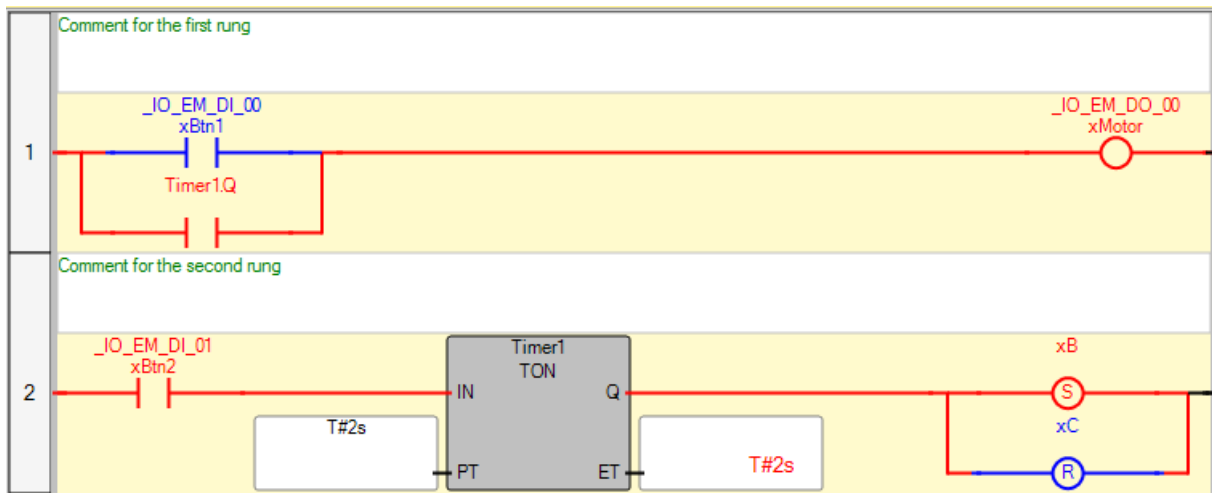


Figure 25 - Tracing the evaluation of the program in Remote Run mode

Figure 26 depicts the variable declaration window if the PLC is connected and is in *Remote Run* mode. Current values of the variables are displayed in the *Logical Value* column (physical state of IO channels are reported in the *Physical Value* column). Variables can be locked by checking the *Lock* checkbox. Value of a locked variable can be changed manually from the declaration window only, assignments in program organization units do not change its value. This method (called *forcing*) can be used for debugging and testing purposes, but extra care needs to be taken especially if physical outputs are affected.

	Name	Alias	Logical Value	Physical Value	Initial Value	Lock	Data Type
	xA		<input type="checkbox"/>	N/A		<input type="checkbox"/>	BOOL
	xB		<input checked="" type="checkbox"/>	N/A		<input type="checkbox"/>	BOOL
	xC		<input type="checkbox"/>	N/A		<input type="checkbox"/>	BOOL
▶	Timer1		<input checked="" type="checkbox"/>	TON
	Timer1.IN	IN	<input checked="" type="checkbox"/>			<input type="checkbox"/>	BOOL
	Timer1.PT	PT	T#2s			<input type="checkbox"/>	TIME
	Timer1.Q	Q	<input checked="" type="checkbox"/>	N/A		<input type="checkbox"/>	BOOL
	Timer1.ET	ET	T#2s	N/A		<input type="checkbox"/>	TIME
	Timer1.Pdate	Pdat	T#12m44s105m	N/A		<input type="checkbox"/>	TIME
	Timer1.Redg	Redg	<input checked="" type="checkbox"/>	N/A		<input type="checkbox"/>	BOOL

Figure 26 - Variable declaration window in Remote Run mode

The project can not be modified in *Remote Run* mode. To change the program, the PLC needs to be switched to *Remote Program* mode (the PLC is stopped immediately) or be disconnected from the PC. In the latter case, the program keeps running and can be updated upon the next download.

3.6 Automated generation of project documentation

Like other state-of-the-art development environments, Connected Components Workbench supports automatic generation of project documentation. Such a document contains configuration settings, variable declarations and code of program organization units along with comments. If the comments attached to rungs and variables are descriptive, the automatically generated documentation is self-contained, operation of the application can be understood and reconstructed, hence further modifications can be carried out by other developers.

The documentation can be generated by right-clicking the root of the project tree (*Micro 830*) and selecting the `Document Generator (print)` command. Connected Components Workbench saves the documentation to a Microsoft Word (*.docx*) document.

Prerequisite of generating a comprehensive documentation is the use of short but descriptive comments both for the variables declared and program organization units implemented.

4 Measurement tasks

The task of the laboratory session is to implement the control of a manufacturing line model depicted by Figure 27. The line consists of an input conveyor, an output conveyor, two machining stations with conveyors and two pushers which connect the perpendicular conveyor sections.

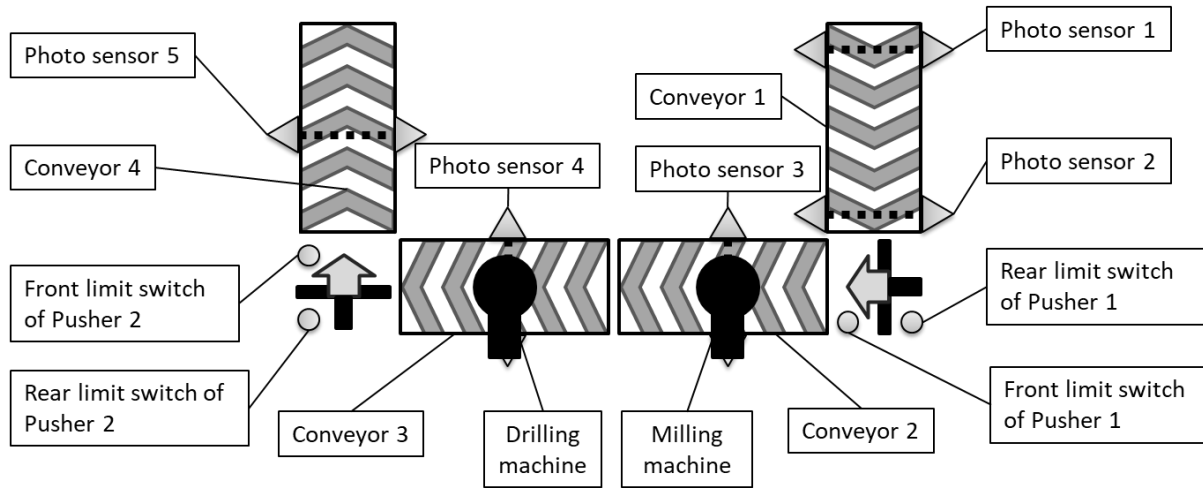


Figure 27 - Layout of the manufacturing cell

The line is equipped with five photoelectric proximity switches (photo sensors). If a piece interrupts the light beam of the sensor, the value of the corresponding PLC input changes to `false` (inverse logic). Limit switches of the pushers are active (value `true`) if they are pressed down by the pusher.

Motors of conveyors and machines are running if the value of the corresponding PLC output is set to `true`. Pushers are driven by two outputs: one turns their motor on, the other specifies the direction of the motion (`true`: forward, `false`: backward).

The desired operation of the manufacturing line is as follows.

When a piece is placed onto the input conveyor, sensed by the 1st photo sensor, the conveyor shall be started and forward the piece to the pusher. The conveyor shall run for 2 seconds after the piece has reached the 2nd photo sensor. The pusher shall be then fully extended (until the front limit switch is reached) and then retracted to its rear position.

The conveyor of the milling station shall be started when the pusher starts moving forward and shall run until the piece arrives to the machine (sensed by the 3rd photo sensor). The machine shall be operated for 3 seconds, then the conveyor shall be turned on again for 3 seconds.

The conveyor of the drilling station shall be started when the tool of the milling station is stopped. The operation of the drilling station is the same as of the milling station, except that the drilling tool shall be operated for 5 seconds.

The second pusher operates the same way as the first one, and it shall be started when the conveyor of the drilling station is stopped.

The output conveyor shall be started when the second pusher starts moving forward and shall be stopped 2 seconds after the piece has been sensed by the 5th photo sensor.

As the manufacturing line is composed of various, functionally independent elements (stations), it is advised to implement the control application in a modular way, exploiting the features of program organization units. According to the structure of the manufacturing line, function block types shall be defined for the control of the following stations:

- Input conveyor
- Pusher
- Machining station (including tool and conveyor)
- Output conveyor

As you might notice, you can reuse your code by defining one single function block type for the pusher and for the machining station and instantiating them twice. If the function blocks are implemented according to the principles of state machine-based control, the resulting application will be easy to understand and easily extensible.

Since stations (except for the input conveyor) are not equipped with proximity sensors on their inputs, a Boolean output shall be defined for the function blocks in order to start their operation. Likewise, function blocks (except for the one controlling the output conveyor) shall be extended by a Boolean output, which is used to start the next element (see Figure 28). For example, output of the input conveyor is connected to the input of the first pusher, so the pusher can be started when the input conveyor has forwarded the workpiece to it).

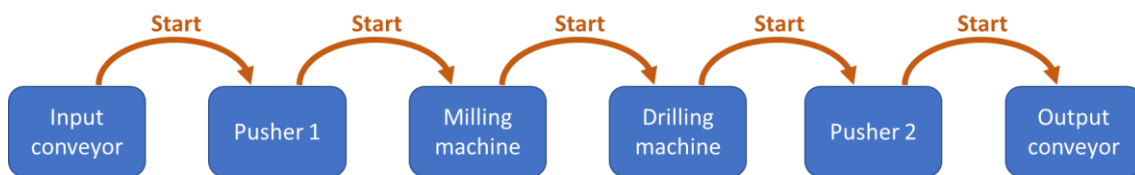


Figure 28 - Daisy-chaining start inputs and outputs

Keep in mind that physical IO variables shall not be used in function block types. Physical inputs and outputs shall be declared only in a program, which connects them to input and output variables of the appropriate function block instance.

Detailed tasks

1. Add an LD function block type to the application and implement the control of the input conveyor. The function block shall start the conveyor if a piece is placed onto the input conveyor (signal of the first photo sensor changes to `FALSE`), and stop the conveyor 2 seconds after the piece passes the second photo sensor. This duration is enough for the piece to arrive at the pusher. Note that the piece is in the range of the second photo sensor for a short time only.

Declare input variables of the function block for the two photo sensors and an output variable for the conveyor (on/off). Define an additional output (for starting the pusher), which is active when the two second delay has elapsed (you can connect the Boolean output of the timer to this variable).

Instantiate the function block in an LD program, assign its inputs and outputs to the appropriate physical input and output variables and test the application.

2. Add a function block type for controlling a pusher. When started by the appropriate input, the pusher shall start moving forward until reaching the front limit switch, and then retract (move backward) until the rear limit switch is reached.

Beside the signals of the limit switches, define a Boolean input variable which can be used to start the pusher. As there is no photo sensor, which can report when a piece arrives at the pusher, it needs to be started by another station (the first pusher is started by the input conveyor, using the additional input which is activated when the 2 second delay has elapsed). Define Boolean output variables for the physical outputs (motor on/off, direction) and for starting the next station. This latter signal shall be active while the pusher is moving forward.

Instantiate the function block in the program, connect the IOs of the first pusher to it and test the application.

3. Add a function block type which implements the control of a machining station (conveyor and machine). When the station is started, the conveyor shall be turned on and operated until the piece arrives at the photo sensor. Then the conveyor shall be turned off and the tool (drill or mill) shall be operated for a given time. After that the conveyor shall be started again and operated for 3 seconds (the piece leaves the station in less than 3 seconds).

Beside the input connected to the photo sensor, declare an additional input variable for starting the station and also a duration (`TIME`) type input, which defines how long the tool shall be operated. Declare Boolean outputs for controlling the conveyor and the tool. Beside that, declare two additional Boolean inputs for starting the next station. One of these shall be active while the conveyor is running after the machining has been finished, the other shall be active when the conveyor is turned off after the 3 second delay (you can connect the outputs of the corresponding timers to these signals).

In the program, declare two instances of the function block for the drilling and milling station. The milling station shall be started by the starting signal generated by the first pusher. The drilling station shall be started by the milling station when the tool is turned off (use its first start signal). The milling tool shall operate for 5 seconds while the drilling tool shall operate for 8 seconds.

Declare an other instance of the pusher function block to control the second pusher. The second pusher shall be started by the milling station when the conveyor is stopped after the 3 second delay. Test the application.

4. Add a function block type to the application, which controls the output conveyor. When started by the second pusher, the conveyor shall be turned on and stopped 2 seconds after the piece has passed the photo sensor. Define a Boolean input for the photo sensor and also for the starting signal, as well as a Boolean output for turning the conveyor on or off.

Instantiate the function block, connect its start input to the start signal generated by the second pusher and test your application.

Tips

- Define state machines for the stations! Draw the state transition diagrams at first and then implement them!
- You shall implement and test your application incrementally, i.e. implement the control of the input conveyor and verify its operation at first. Then implement the control of the first pusher and test it along with the input conveyor, etc.
- If you experience any malfunction, stop the PLC and ask for assistance!

Lab report

In the lab report, you shall briefly describe the measurement task and the principle of the solution, then present the function block types you have developed. For each of these function blocks, give the state transition diagram and the output mapping of the underlying state machine. You shall not include any program code in the lab report but you shall attach the documentation generated by CCW.