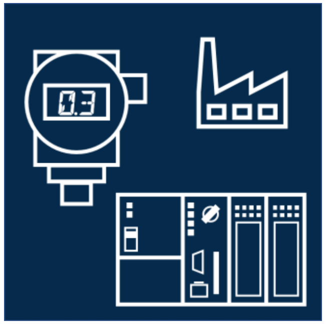
Structured Text

Industrial control

KOVÁCS Gábor
gkovacs@iit.bme.hu



Structured Text
(ST)

- High level textual language
- Clear structure of code
- Powerful execution control
- Compilation to executed code can not be directly influenced
- High level of abstraction might lead to suboptimal implementation

The whole textual code is executed during each PLC cycle or upon a call!

Expressions

- Statements work on expressions

- Elements of an expression
  - operands

    (literals, variables, even other expressions)
  - operators

    (Boolean or arithmetic operators, function calls)

Operands

- Literals

  `17, ‘my string’, T#3s`

- Variables (elementary or derived types)

  `Var1, VarArray[12]`

- Results of function calls

  `Add(2,3), sin(1.76)`

- Other functions

  `10+20 (=Add(10,20))`

Operator	Description	Example → Result	Precedence
( )	Change execution order	(3+2)*(4+1) → 25	
<fcn name>	Function call	CONCAT('PL','C') → 'PLC'	
-	Negation (arithmetic)	-10 → -10	
NOT	Complement (Boolean negation)	NOT TRUE → FALSE	
**	Power	2**7 → 128	
*	Multiplication	2*7 → 14	
/	Division	30/6 → 5	
MOD	Modulo	32 MOD 6 → 2	
+	Addition	32+6 → 38	
-	Subtraction	32-6 → 26	
<, <=, >, >=	Comparison	32<6 → FALSE	
=	Equality	T#24h = T#1d → TRUE	
<>	Inequality	2<>5 → TRUE	
&, AND	Boolean AND	TRUE AND FALSE → FALSE	
XOR	Boolean exclusive or (XOR)	TRUE XOR FALSE → TRUE	
OR	Boolean OR	TRUE OR FALSE → TRUE	

Function call

- A function call is an expression: value of the expression is the result (return value) of the function
- Formal call
  - parameters assigned to identifiers in any arbitrary order
  - if a variable is omitted, its initial value is used
  - output variables of the function can be assigned to variables
  - `LIMIT(MN:=0,MX:=10,IN:=7)`
- Informal call
  - parameters in the same order as they appear in the declaration of the function, input variables can not be omitted
  - output variables of the function can not be accessed
  - `LIMIT(0,7,10)`
  - CODESYS only supports informal call for standard functions

Statements

Keyword	Operation
:=	Assignment of value
<FB name>(parameters)	Function block call
RETURN	Return to caller POU
IF	Selection
CASE	Selection
FOR	Iteration
WHILE	Iteration
REPEAT	Iteration
EXIT	Terminate iteration

Assignment

```
VAR
   iD: INT;
   aE: ARRAY [0..9]
       OF INT;
END_VAR
```

- := operator
- Value can be assigned to
  - single-element variable
  - element of array or structure

```
iD:=4;

aE[3]:=iD**2;
```

- Data types
  - data types of the left-hand side and right-hand side need to be compatible
  - type conversion functions can be used as expressions

```
iD:=REAL_TO_INT(SIN(2))
```

Boolean assignment

- Not a Boolean function as in ladder diagram but a statement
- If the statement is not evaluated (e.g. inside an `IF` conditional block), then the value of the variable is not changed

```
IF A
 THEN Y:=1;
END_IF;
```


```
IF A
 THEN Y:=1;
 ELSE Y:=0;
END_IF;
```

Function block call

- An FB call is a statement, not allowed in an expression (there is no result of a function block)
- Formal call only
  - parameters assigned to identifiers in any arbitrary order
  - output variables can be assigned to other variables
  - parameters omitted are replaced by
    - their value assigned during the previous call or in a separate assignment
    - their initial value if not assigned before
- Interface variables can be accessed without calling the FB instance

Function block call - Example

```
PROGRAM MyProg
VAR
      MyTimer       : TON;
      xA            : BOOL;
      iMyInt        : INT;
      timMyDur      : TIME;
END_VAR
(*…*)

MyTimer(PT:=T#1s, IN:=(iMyInt=7), Q=>xA);
timMyDur:=MyTimer.ET;

(*…*)
END_PROGRAM;
```

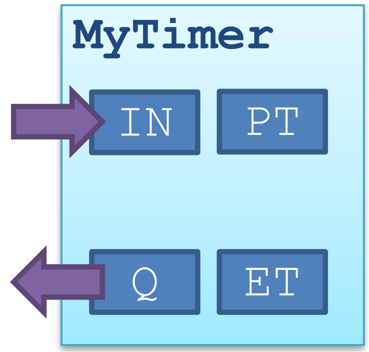
Formal function block call

Boolean expression

Accessing output variable of the instance which has not been assigned during the call

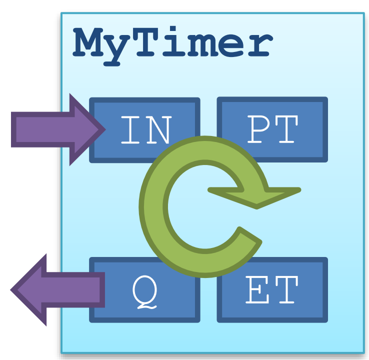
Function block call

```
MyTimer.IN:=TRUE;

xA:=MyTimer.Q;
```



- Reading or writing variables of an FB means memory access only
- Body of the FB instance is not executed
- Output variables of the FB instance are not refreshed

```
MyTimer.IN:=TRUE;
MyTimer();
xA:=MyTimer.Q;
```



- **Output variables of the FB instance are refreshed upon a call of the instance only!**

formal call (data access and FB-call in one signle line):

```
MyTimer(IN:=TRUE,Q=>xA);
```

As no value is assigned to `MyTimer.PT`, its value assigned during the previous call or assignment is used

Selection

- Selection based on a Boolean-valued expression
- Each branch contains a block of statements
- `ELSIF` and `ELSE` branches are optional
- Must be terminated by the keyword `END_IF`

```
IF <BOOL expression> THEN
     <statement block>
ELSIF <BOOL expression> THEN
     <statement block>
ELSIF <BOOL expression> THEN
     <statement block>
ELSE
     <statement block>
END_IF;
```

Selection - Example

```
IF (iA=1) THEN
     iX:=1;
     iY:=1;
ELSIF (iA=2 OR iA=3) THEN
     iX:=1;
     iY:=0;
ELSE
     iX:=0;
     iY:=0;
END_IF;
```

Case selection

- Selection based on integer (`ANY_INT`) or enumerated typed expression
- Multiple values can be defined for each case, but a value can be associated to one single case only
- If condition of a state is satisfied, further cases are not evaluated (implicit `break` at the end of each statement block)
- Default case: `ELSE` (optional)
- Terminated by keyword `END_CASE` (mandatory)

```
CASE <INT expression> OF
<value1>:             <statement block>
<value2>,<value3>:    <statement block>
<value4>..<value9>: <statement block>
ELSE  <statement block>
END_CASE;
```

Case selection - Example

```
CASE iA OF
 1:     iX:=1;
        iY:=1;
 2,3:   iX:=1;
        iY:=0;
 4..9: iX:=0;
 ELSE
        iX:=0;
        iY:=0;
END_CASE;
```

Iteration - Loops

Whole iteration is executed during one single PLC cycle (or upon one single call)

- If you use loops without caution, it can significantly deteriorate determinism or even cause a watchdog error!
- Do not use loops for waiting the occurrence of an event!
- Loops might be used for
  - iterating through elements of an array or a bit field
  - repeating an operation for a well defined number of times

While loop

- The Boolean conditional expression is evaluated before executing the body of the loop
- Body is executed if the value of the conditional expression is `TRUE`

```
WHILE <BOOL expression> DO
     <statement block>
END_WHILE;
```

While loop - example

```
VAR
      aMyArray:  ARRAY[1..10] OF UINT;
      i:         INT;
      uiMaxVal:  UINT:=0;
END_VAR
(* … *)
i:=1;
WHILE (i<=10) DO
      IF (aMyArray[i]>uiMaxVal)
            THEN uiMaxVal:=aMyArray[i];
      END_IF;
      i:=i+1;
END_WHILE;
(* … *)
```

Repeat – Until loop

- The Boolean conditional expression is evaluated after the execution of the body of the loop
- Body of the loop is executed at least once
- Iteration is terminated if the value of the conditional expression is `TRUE`

```
REPEAT
     <statement block>
UNTIL <BOOL expression>
END_REPEAT;
```

Repeat – Until loop - Example

```
VAR
      aMyArray:  Array[1..10] OF UINT;
      i:         INT;
      uiMaxVal:  UINT:=0;
END_VAR
(* … *)
i:=0;
REPEAT
      i:=i+1;
      IF (aMyArray[i]>uiMaxVal)
            THEN uiMaxVal:=MyArray[i];
      END_IF;
UNTIL (i=10)
END_REPEAT;
(* … *)
```

Checking equality: use a single equation sign (=) !

For loop

Control variable

Initial expression

End expression

Increment expression

```
FOR vC:=expI TO expE BY expInc DO
    <statement block>
END_FOR;
```

- The four variables/expressions need to be the same integer type (`SINT`, `INT`, `DINT`)
- Control variables and variables used in initial and end expressions must not be assigned a value inside the body of the loop
- Variables used in the increment expression might be assigned a value inside the body (not recommended)

For loop – Example

```
VAR
      aMyArray:  ARRAY[1..10] OF UINT;
      i:         INT;
      uiMaxVal:  UINT:=0;
END_VAR
(* … *)
FOR i:=10 TO 1 BY -1 DO
      IF (aMyArray[i]>uiMaxVal)
            THEN uiMaxVal:=aMyArray[i];
      END_IF;
END_FOR;
(* … *)
```

Skipping parts of the loop

- Further operations in the loop can be skipped by a `CONTINUE` statement
- Can be placed inside a conditional block

```
FOR j:=1 TO 10 DO
     IF aList[j]<0 THEN
          CONTINUE;
     END_IF;
     aList[j]:=SQRT(aList[j]);
END_WHILE
```

Iteration termintation

- Iteration can be terminated using the `EXIT` statement
- Terminates only the loop inside which it is executed, not ones at higher levels of hierarchy

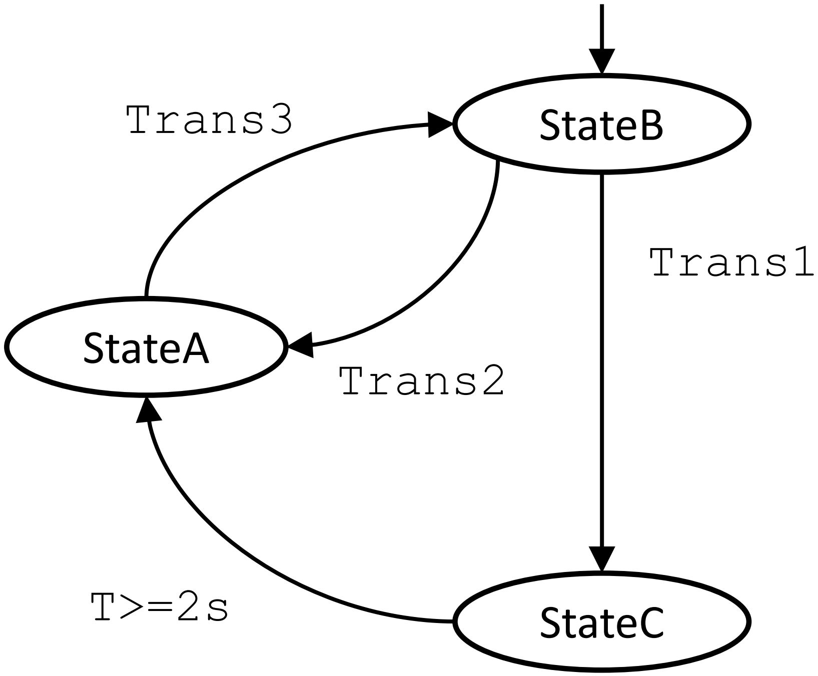
```
j:=0;
WHILE (j<10) DO
      i:=0;
      WHILE (i<10) DO
            IF (i=j) THEN EXIT;
            ELSE i:=i+1;
            END_IF;
      END_WHILE;
      j:=j+1;
END_WHILE
```

i	j
0	0
1	0
1	1
2	0
2	1
2	2
3	0
…	

Return to the caller POU

- `RETURN` keyword
- Might be used in a conditional structure (`IF` or `CASE`)
- Result (return value) of a function must be assigned before executing the `RETURN` statement
- If omitted, the POU returns to the caller after executing the last line of code

State machine-based control in ST



State machine:

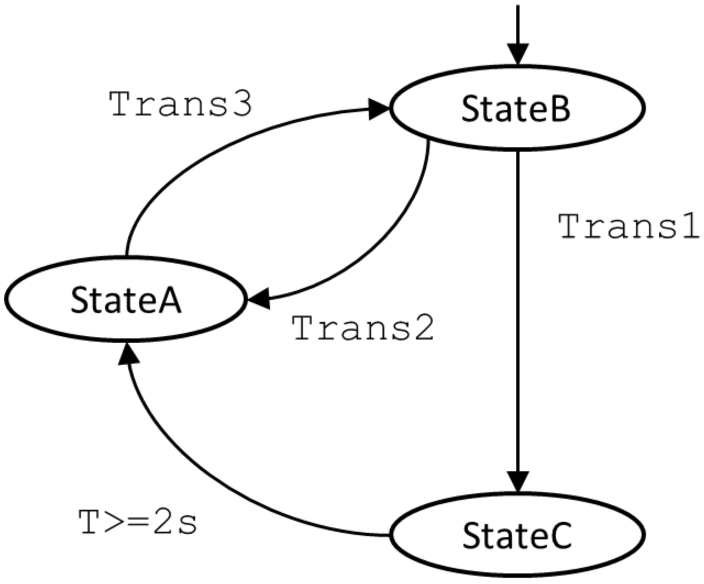
- states
- initial state
- state transitions
- output mapping

	StateA	StateB	StateC
xOut1	1	0	1
xOut2	0	1	1

Representation of states

- States shall be represented by an enumerated type variable
- States can be referenced by textual labels

- Initial state
  - first label of the enumeration by default
  - can be explicitly specified in the declaration (strongly recommended)

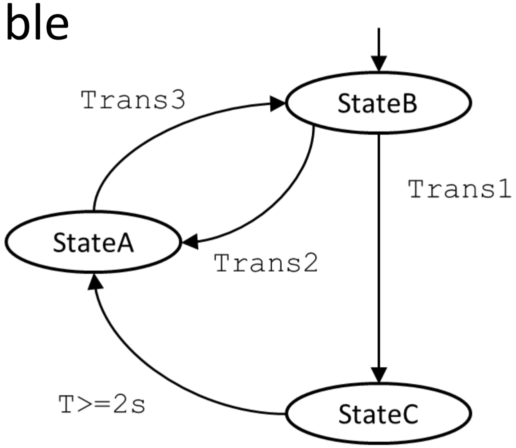



```
VAR
  eState : (StateA,StateB,
            StateC):=StateB;
END_VAR
```

Without explicite definition of the initial value, the initial state would be StateA

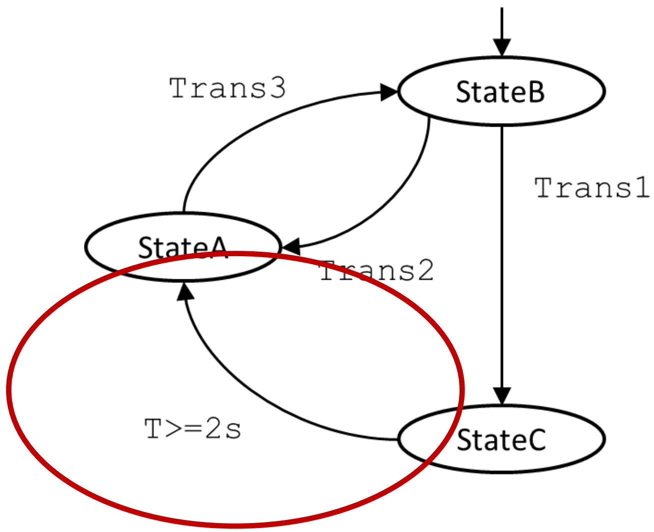
State transitions

- Transitions can be implemented in a `CASE` structure, according to their initial states
- Transition conditions are evaluated inside the cases
- If multiple transitions share the same initial state, it is strongly recommended to use mutually exclusive conditions
- A transition can be executed by assigning the label of the destination state to the state variable




```
VAR
 eState : (StateA,StateB,
           StateC):=StateB;
 TimerC : TON:=(PT:=T#2s);
END_VAR

TimerC(IN:= (eState=StateC) );
CASE eState OF
 StateA: IF Trans3 THEN
           eState:=StateB;
         END_IF;
 StateB: IF Trans1 THEN
            eState:=StateC;
         ELSIF Trans2 THEN
            eState:=StateB;
         END_IF;
 StateC: IF TimerC.Q THEN
            eState:=StateA;
         END_IF;
END_CASE;
```

Timed transitions



- Condition of timed transition: the origin state is active for a given time
- Such conditions might be realized using `TON` timers
- The input of the timer shall be active if the origin state is active (Boolean expression): `TimerC(IN:=(eState=StateC));`
- Use a dedicated timer instance for each timed transition

Timed transitions

Timer FBs of timed transitions shall be called during each cycle or call

If the timer is called only when `StateC` is active, then its input `IN` is constantly active, hence leaving and re-entering `StateC`, its output will be active so the transition is executed immediately.

```
CASE eState OF
  StateC: TimerC(IN:=(eState=StateC),PT:=T#2s);
          IF (TimerC.Q) THEN eState:=StateA; END_IF;
(* … *)
END_CASE
```

The timer is called during each call or PLC cycle, regardless the value of the state variable, so internal counter and output of the timer is reset if `StateC` is deactivated

```
TimerC(IN:=(eState=StateC),PT:=T#2s);

CASE eState OF
  StateC: IF (TimerC.Q) THEN eState:=StateA; END_IF;
(* … *)
END_CASE
```

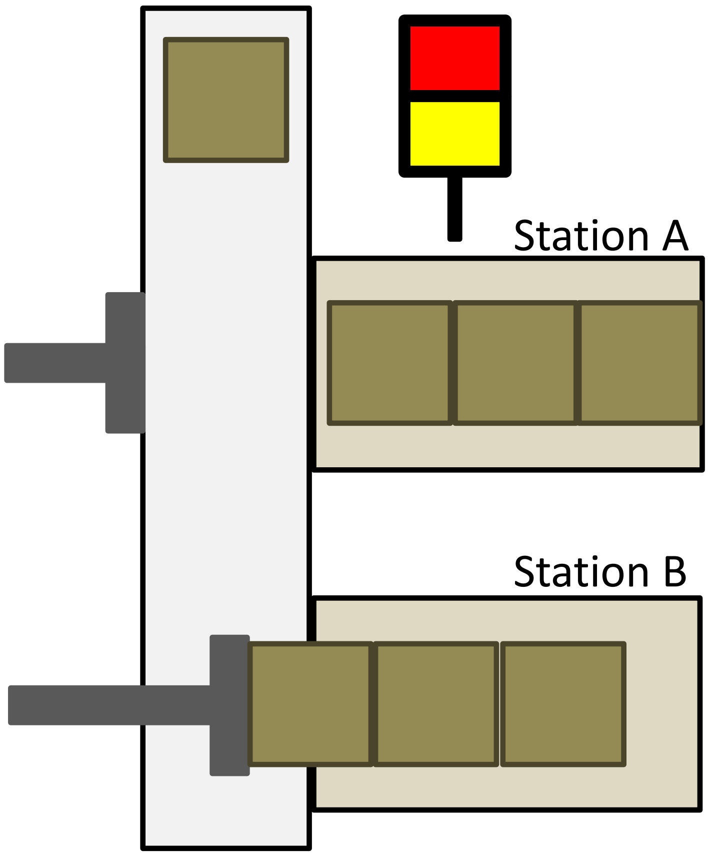
Output mapping

- Boolean functions of the output mapping can be realized by simple assignments
- Output variables shall be assigned a Boolean value during each call or PLC cycle
- Outputs can be set during state transitions, but then modification of the program requires extra care (***not recommended***)

	StateA	StateB	StateC
xOut1	1	0	1
xOut2	0	1	1

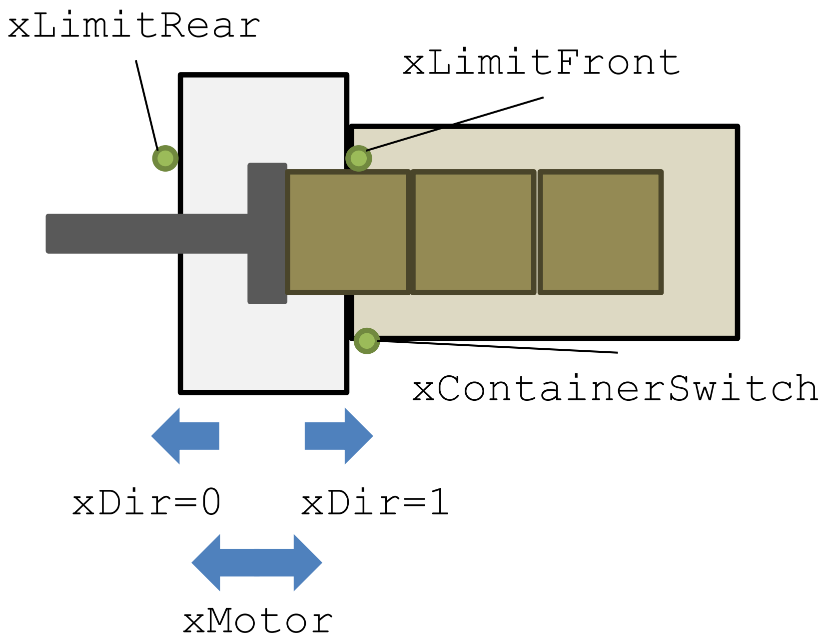
```
xOut1:=(eState=StateA) OR
       (eState=StateC);

xOut2:=(eState=StateB) OR
       (eState=StateC);
```

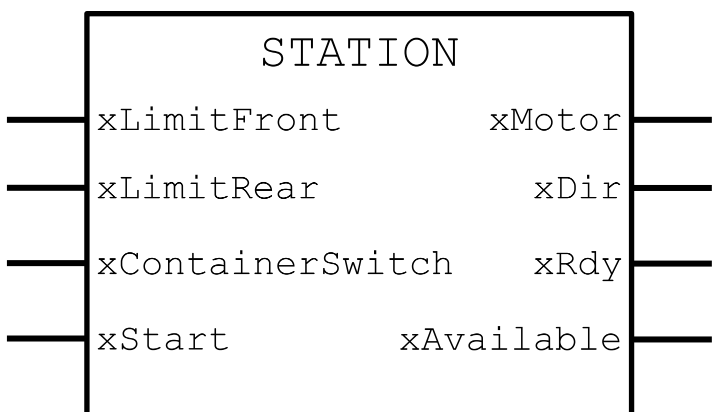
Problem – packaging line



- Packages arriving on the conveyor shall be forwarded to containers by pushers
- The conveyor can operate only if both pushers are fully retracted
- Capacity of a container is 3 packages
- Packages shall be forwarded to the container at station *A* by default
- If container at station *A* is full or missing, the packages shall be forwarded to station *B*
- If neither of the two stations can store a package, the system shall be stopped

Packaging station



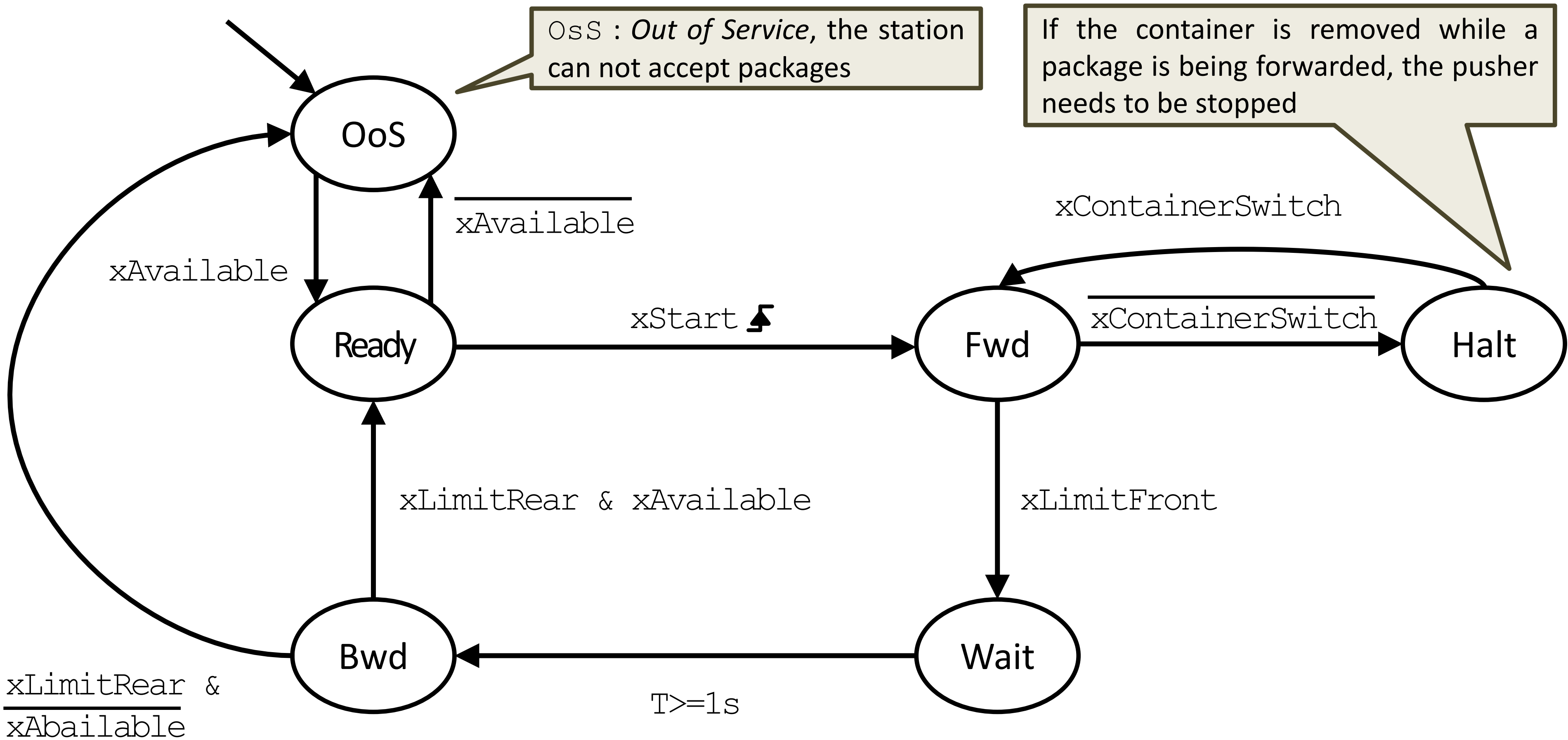


- Fully extended and fully retracted position of the pusher are reported by limit switches (`xLimitFront`, `xLimitRear`)
- The container present at the station actuates a limit switch (`xContainerSwitch`)
- The pusher is moved by the output `xMotor` while the direction of movement is set by the value of output `xDir`
- The input `xStart` of the function block is set by the main program if a package needs to be stored
- Outputs `xRdy` and `xAvailable` report if the station is idle (pusher is in the rear position and is not moving) or ready to accept a new package (a container is present and is not full), respectively

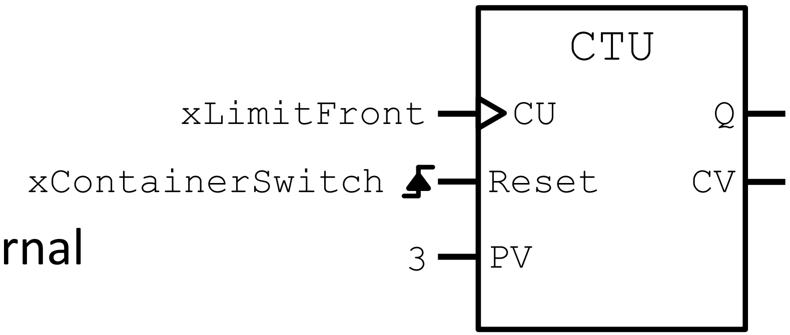
Packaging station

- Upon the rising edge of the `xStart` input, the pusher shall be fully extended and, after 1 second it, should be fully retracted
- The pusher shall be started only if a container is present and is not full
- If the container is removed while the pusher is being extended (`xContainerSwitch` changes to inactive), the pusher shall be stopped; extension of the pusher can be continued if a container is present again
- Output `xRdy` of the function block shall be set if the pusher is at its rear position and has not yet started the extension
- The station is ready to store a package (`xAvailable` output shall be active) if a container is present and is not full

Packaging station



Counting the packages stored



- Counting can be implemented by using a standard count-up counter (CTU)
- Output Q of the counter is active if the internal counter (CV) has reached the preset value connected to the input PV
- Which sensor reports that a new package is placed inside the container? If the pusher reaches its front position (rising edge of xLimitFront), a package is placed to the container (CU counter input is edge-sensing)
- Which sensor reports that the container is emptied? If the container is removed (xSwitchContainer=0) and a new container is placed (xSwitchContainer=1), we can assume that the new container is empty. The counter shall be reset by the rising edge of the limit switch of the container (Reset input of the counter is not edge-sensing)

Function block of the packaging station

```
FUNCTION_BLOCK Station

VAR_INPUT
   xLimitFront, xLimitRear : BOOL;
   xContainerSwitch        : BOOL;
   xStart                  : BOOL;
END_VAR
VAR_OUTPUT
   xDir, xMotor      : BOOL;
   xAvailable, xRdy : BOOL;
END_VAR
VAR
   eState              : (OoS,Ready,Fwd,Halt,Wait,Bwd) := OoS;
   riseContainerSwitch : R_TRIG;
   riseStart           : R_TRIG;
   TimerWait           : TON := (PT:=T#1s);
   Counter             : CTU := (PV:=3);
END_VAR
```

`ENUM` type state variable

Rising edge sensing FB for resetting the counter

FB for counting the packages (preset value is set during the instantiation)

Function blocks are called during each call of the FB `Station`

Output of the edge sensing FB is assigned the `Reset` input of the counter.

```
riseContainer(CLK := xContainerSwitch,Q => Counter.RESET);
riseStart(CLK := xStart);
Counter(CU := xLimitFront);
TimerWait(IN := (eState=Wait));
xAvailable := xContainer AND NOT(Counter.Q);
```

The `Reset` input has been assigned in a previous line while the preset value has been set during the instantiation

As `xAvailable` is used as a transition condition, its value is assigned before the `CASE` structure

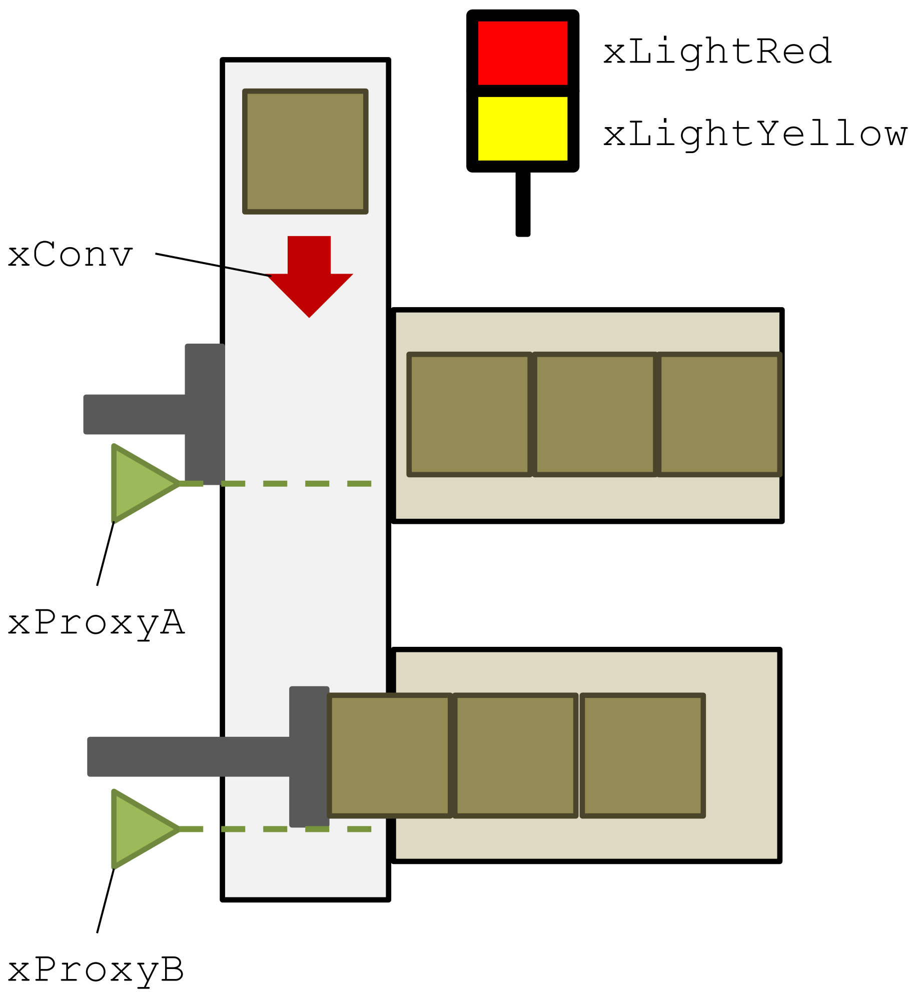
```
CASE eState OF
  OoS:   IF xAvailable THEN eState:=Ready; END_IF;
  Ready: IF NOT(xAvailable) THEN eState:=OoS;
         ELSIF xStart THEN eState:=Fwd; END_IF;
  Fwd:   IF NOT(xContainerSwitch) THEN eSate:=HALT;
         ELSIF xFront THEN eState:=Wait; END_IF;
  Halt:  IF xContainerSwitch THEN eState:=Fwd; END_IF;
  Wait:  IF TimerWait.Q THEN eState:=Bwd; END_IF;
  Bwd:   IF xRear THEN
              IF NOT(xAvailable) THEN eState:=xReady;
              ELSE eState:=xOos;
              END_IF;
            END_IF;
END_CASE;
```

State transitions

Output mapping

```
xMotor:=(eState=Bwd) OR (eState=Fwd);
xDir:=(eState=Fwd);
xRdy:=(eState=Idle) OR (eState=OoS);

END_FUNCTION_BLOCK
```

Packaging line – Main program



- The conveyor is operated by the `xConv` output
- Proximity switches (`xProxyA`, `xProxyB`) located near the conveyor report if a package arrives to one of the stations
- The yellow warning light (`xLightYellow` output) shall be activated if one of the stations is not available
- The red warning light (`xLightRed` output) shall be activated if none of the stations is available

Packaging line - Program

- We use two instances of the station function blocks in the program. Physical sensor inputs are connected to the inputs of the function block instances; their `xMotor` and `xDir` outputs are connected to physical outputs in the program.
- The input `xStart` of a station function block is activated when a package arrives to the given station (rising edge)
- The conveyor is operated if both stations are at their rear positions and at least one of the stations is available for storing a package
- The yellow warning light is activated if one of the stations is not available, while the red warning light is activated if none of the stations is available

Main program

```
PROGRAM PackagingLine

VAR_INPUT
  xProxyA            AT %IX0.0 : BOOL;
  xProxyB            AT %IX0.1 : BOOL;
  xLimitRearA        AT %IX0.2 : BOOL;
  xLimitFrontA       AT %IX0.3 : BOOL;
  xLimitRearB        AT %IX0.4 : BOOL;
  xLimitFrontB       AT %IX0.5 : BOOL;
  xContainerSwitchA  AT %IX0.6 : BOOL;
  xContainerSwitchB  AT %IX0.7 : BOOL;
END_VAR
```

Switches reporting the presence of containers

```
VAR_OUTPUT
  xConv        AT %QX0.0 : BOOL;
  xMotorA      AT %QX0.1 : BOOL;
  xDirA        AT %QX0.2 : BOOL;
  xMotorB      AT %QX0.3 : BOOL;
  xDirB        AT %QX0.4 : BOOL;
  xLightYellow AT %QX0.5 : BOOL;
  xLightRed    AT %QX0.6 : BOOL;
END_VAR

VAR
  StationA, StationB : Station;
  riseProxyA         : R_TRIG;
  riseProxyB         : R_TRIG;
END_VAR
```

FB instances used to detect the rising edges of the proximity switches

```
RiseA(IN := ProxyA, Q => StationA.Start);

RiseB(IN := ProxyB, Q => StationB.Start);

StationA(xLimitFront := xLimitFrontA, xLimitRear := xLimitRearA,
         xContainerSwitch := xContainerSwitchA,
         xMotor => xMotorA, xDir => DirA);

StationB(xLimitFront := xLimitFrontB, xLimitRear := xLimitRearB,
         xContainerSwitch := xContainerSwitchB,
         xMotor => xMotorB, xDir => DirB);

xConv := StationA.xRdy AND StationB.xRdy AND
         (StationA.xAvailable OR StationB.xAvailable);

xLightYellow := StationA.xRdy XOR StationB.xRdy;

xLightRed := NOT(StationA.xRdy OR StationB.xRdy);
```

If a package arrives to a station, its `xStart` input will be set for one single execution

Call of the station FBs during each execution of the program

Turning the conveyor belt on or off

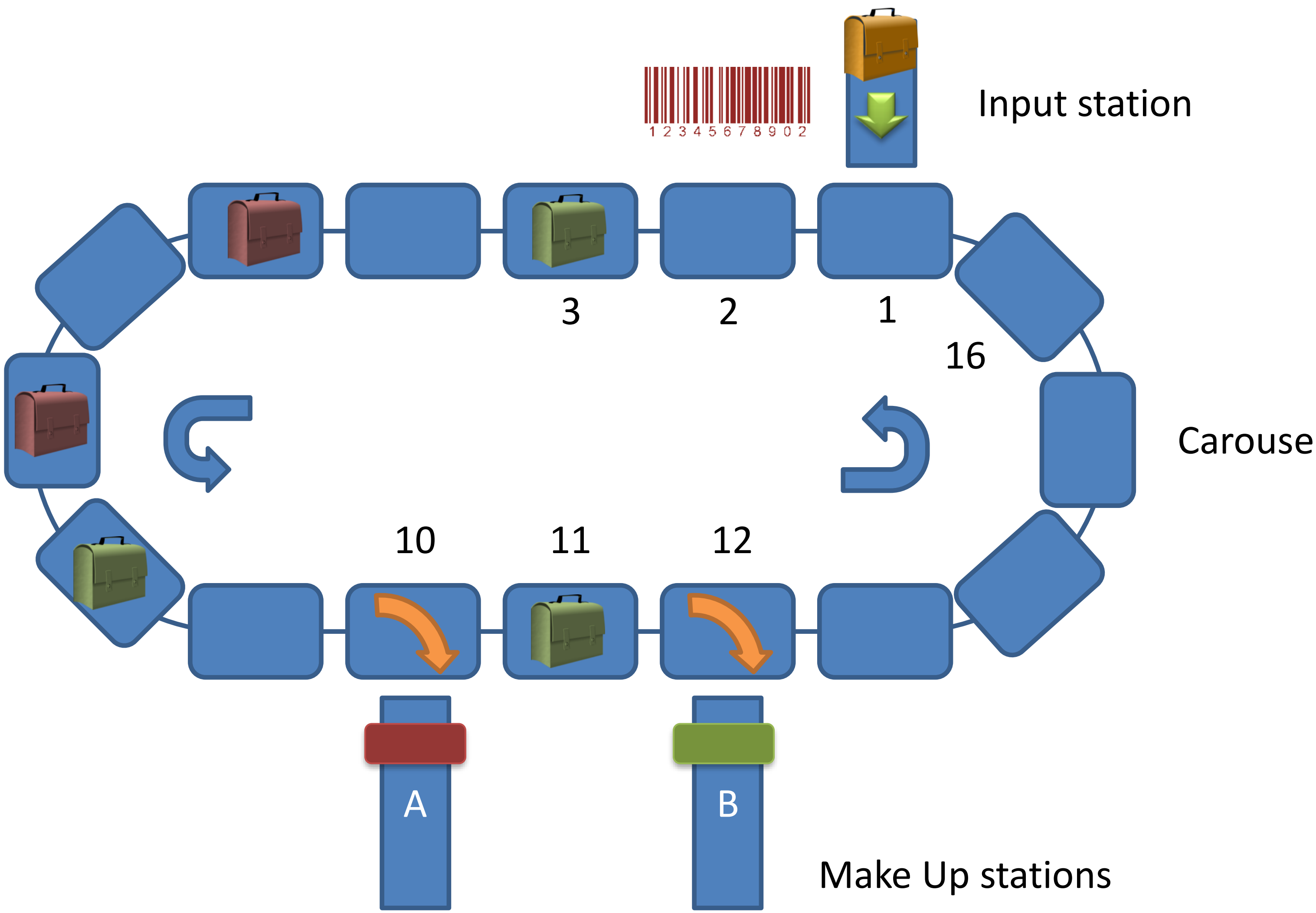
Setting the warning lights

Implementation in the template project

- The project already contains a program (`PackagingLine`) with the declarations of input and output variables, and the freewheeling task executing the program

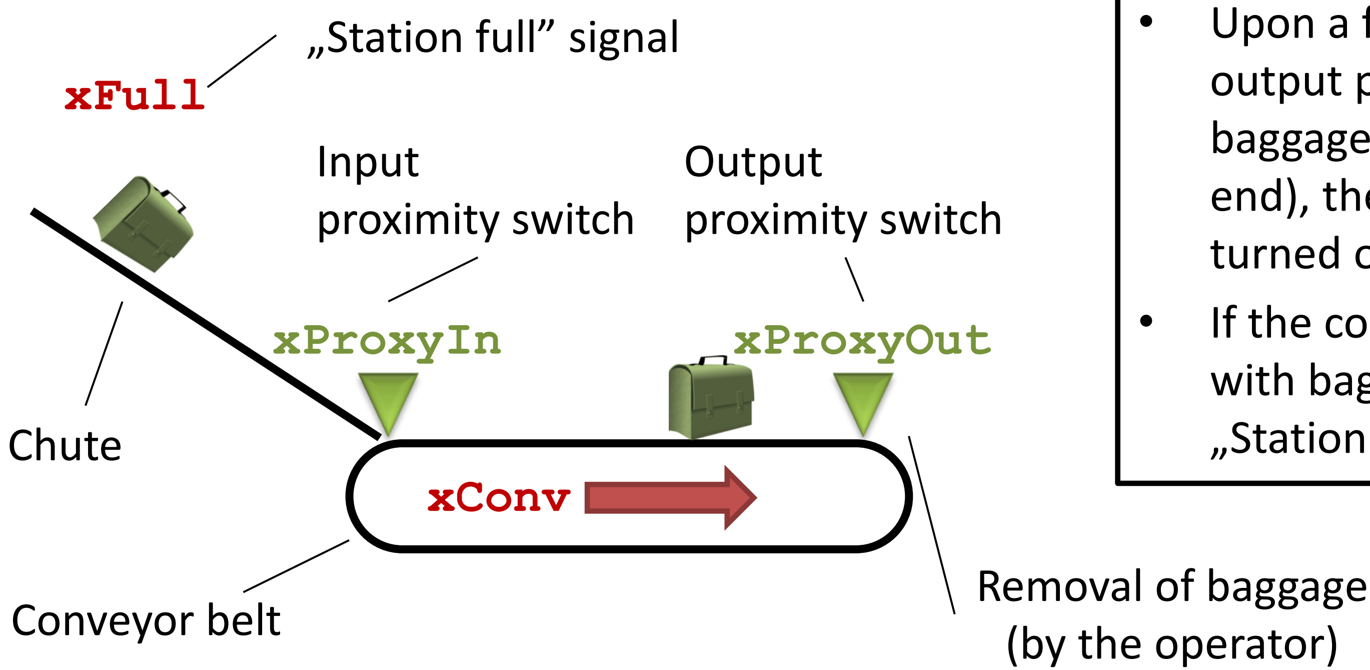
- Testing
  - the application can be tested using the VISU visualization
  - packages arrive from the top automatically
  - containers can be added or removed to or from the stations by the pushbuttons below them

Problem – Baggage handling system



Make Up station

Baggages arrive at the conveyor through a chute. Arriving baggages shall be forwarded to the end of the conveyor, where an operator removes them and place them on a pully. Proximity sensors are located at both ends of the conveyor, reporting if a baggage is present.

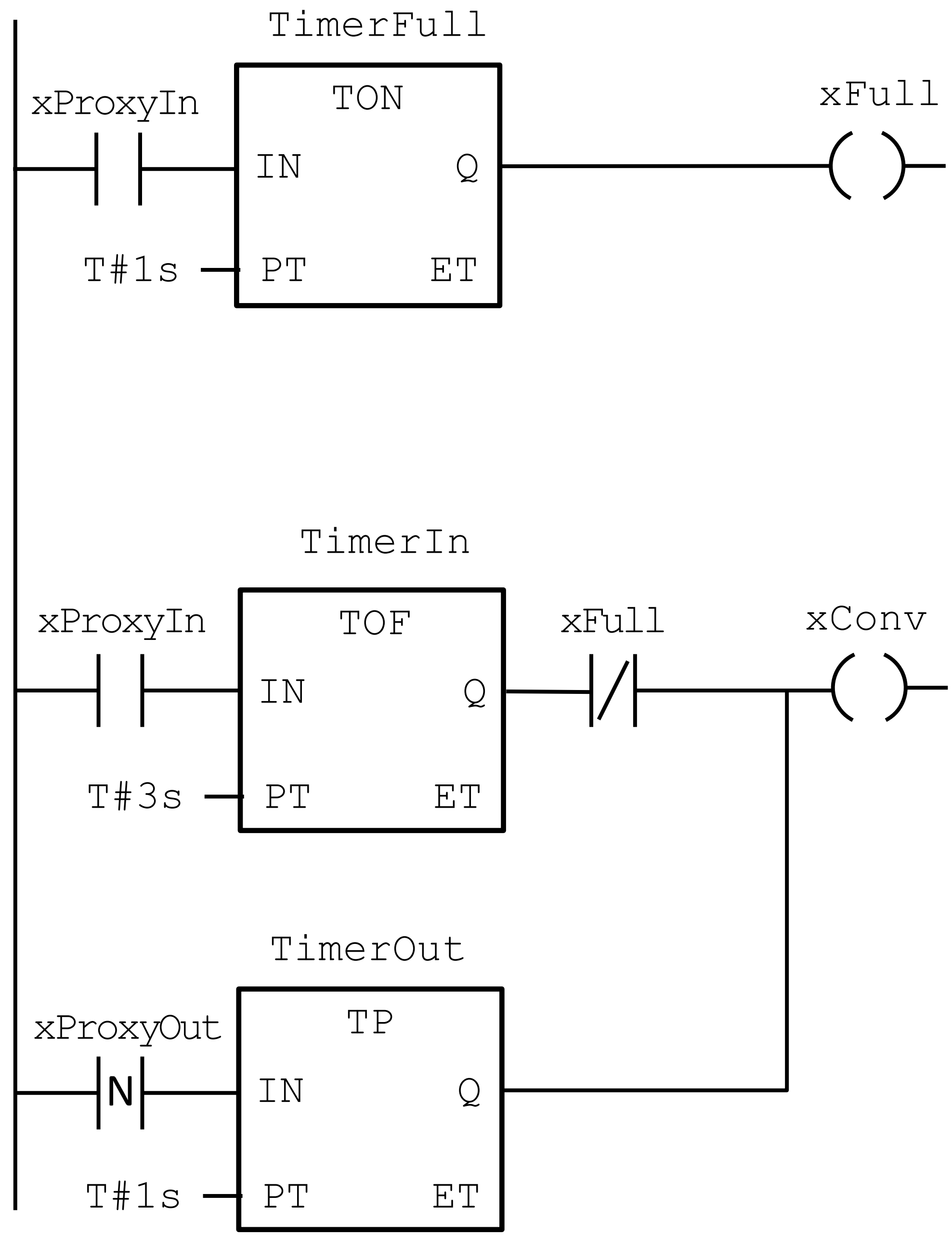


Specification

- If a baggage arrives from the chute, the conveyor shall be turned on for 3 seconds
- Upon a falling edge of the output proximity sensor (a baggage is remove from the end), the conveyor shall be turned on for 1 second
- If the conveyor is fully loaded with baggages, turn on the „Station full" light

Implementation in Ladder Diagram

```
VAR_INPUT
 xProxyIn  : BOOL;
 xProxyOut : BOOL;
END_VAR
VAR_OUTPUT
 xConv     : BOOL;
 xFull     : BOOL;
END_VAR
VAR
 TimerIn   : TOF;
 TimerOut  : TP;
 TimerFull : TON;
END_VAR
```



```
FUNCTION_BLOCK MakeUpStation

VAR_INPUT
   xProxyIn, xProxyOut: BOOL;
END_VAR
VAR_OUTPUT
   xConv, xFull: BOOL;
END_VAR
VAR
   FallOutput : F_TRIG;
   TimerIn    : TOF;
   TimerOut   : TP  := (PT:=T#1s);
   TimerFull  : TON := (PT:=T#3s);
END_VAR


FallOutput(CLK:=xProxyOut,Q=>TimerOut.IN);
TimerFull(IN:=xProxyIn, Q=>xFull);
TimerIn(IN:=xProxyIn,PT:=T#3s);
TimerOut();
xConv:=(TimerIn.Q AND NOT(xFull)) OR TimerOut.Q;

END_FUNCTION_BLOCK
```

Input variables: proximity swithces

Output variables: conveyor drive and „station full” signal

Function block instance for sensing the falling edge of `xProxyOut`

Instantiation of timers

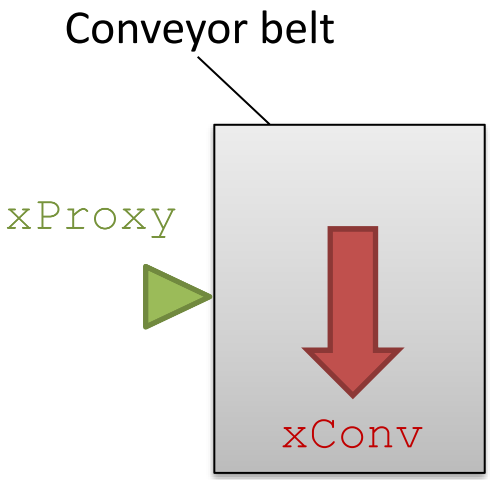
Delay of a timer can be set by assigning an initial value to its `PT` input

Checking if a falling edge has occured (FB instance shall be called) – the output of the FB is directly assigned to the `IN` input of `TimerOut`

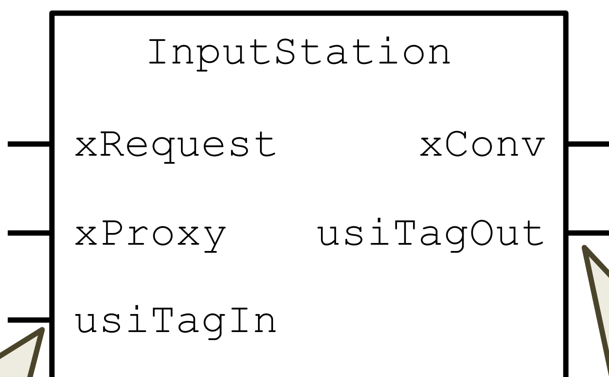
Output of the timer is assigned directly to the output `xFull`

Assigning the value of the output driving the conveyor

Input station



Barcode reader: reads the tag as USINT typed number

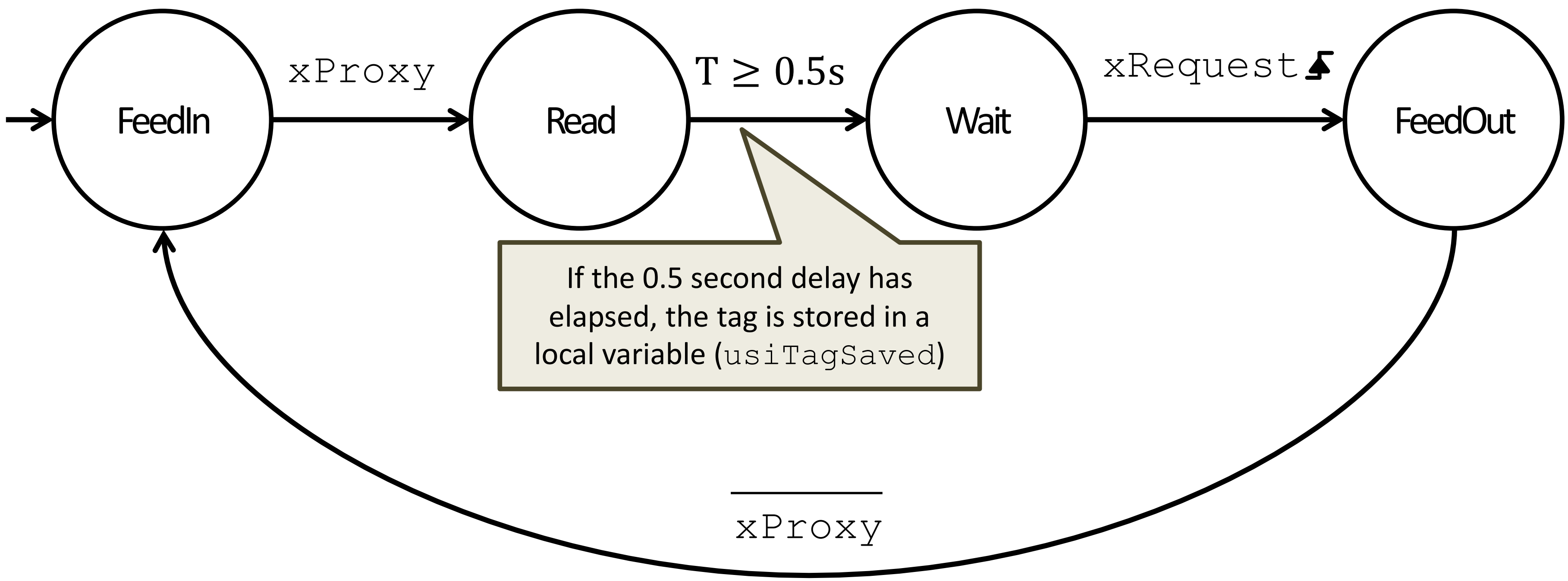


USINT type input, the main program passes the FB the tag provided by the reader

Tag of the baggage forwarded by the input station (0 if no baggage has been forwarded)

- The conveyor shall be turned on after starting the system
- If a baggage arrives at the proximity switch, the conveyor shall be stopped
- The tag reader provides a valid reading 0.5 seconds after stopping the baggage at the proximity switch (reading is invalid if the baggage is moving)
- If a rising edge of the `xRequest` input is detected, a baggage can be forwarded
  - If a baggage is waiting at the input station, it shall be forwarded and the `usiTagOut` output shall be set to its tag
  - If no package is waiting, value of `usiTagOut` shall be set to 0

InputStation



```
FUNCTION_BLOCK InputStation
VAR_INPUT
   xProxy, xRequest : BOOL;
   usiTagIn         : USINT;
END_VAR
VAR_OUTPUT
   xConv     : BOOL;
   usiTagOut : USINT;
END_VAR
VAR
   usiTagSaved : USINT;
   eState      : (FeedIn, Read, Wait, FeedOut) := FeedIn;
   RiseRequest : R_TRIG;
   TimerRead   : TON;
END_VAR
TimerRead(IN:=(eState=Read),PT:=T#500ms);
RiseTrig(CLK:=xRequest);
CASE eState OF
  FeedIn:  IF xProxy THEN eState:=Read; END_IF;
  Read:    IF TimerRead.Q THEN
              usiTagSaved:=usiTagIn;
              eState:=Wait;
           END_IF;
  Wait:    IF RiseTrig.Q THEN eState:=FeedOut; END_IF;
  FeedOut: IF NOT(Proxy) THEN State:=FeedIn; END_IF;
END_CASE;

xConv:=(eState=FeedIn) OR (eState=FeedOut);
IF (eState=FeedOut) THEN
   usiTagOut:=usiTagSaved;
ELSE
   usiTagOut:=0;
END_IF;
END_FUNCTION_BLOCK
```

Enumerated type state variable, explicit setting of the initial value is recommended

The timer is called during each call, not only if the Read state is active

State transitions

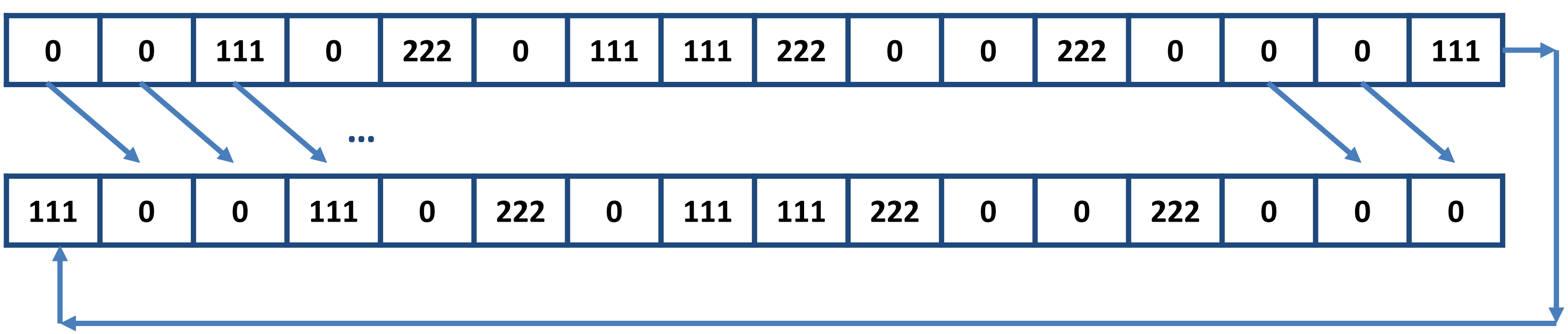
If the tag read by the barcode reader is valid, it is stored in the variable `usiTagRead`

When forwarding a baggage, its code is indicated by the output `usiTagOut`. Otherwise the output is set to 0 (no baggage is being transferred)

Keeping a record of the baggages

- Tags of baggages in trays at different positions are stored in a 16-element array
  - 111: baggage shall be forwarded to make up station „A”
  - 222: baggage shall be forwarded to make up station „B”
  - 0: empty tray

- Upon the rotation of the carousel causes a change of baggages in each position (rising edge of `xProxyCar)`, the array shall be rotated (`ROR`)



Function for rotating the array

```
FUNCTION Rotate: BOOL

VAR_IN_OUT
   aArray: ARRAY [1..16] OF USINT;
END_VAR
VAR
   i:      INT;
   usiTmp: USINT;
END_VAR

usiTmp:=aArray[16];
FOR i:=16 TO 2 BY -1 DO
   aArray[i]:=aArray[i-1];
END_FOR
aArray[1]:=usiTmp;
Rotate:=TRUE;

END_FUNCTION
```

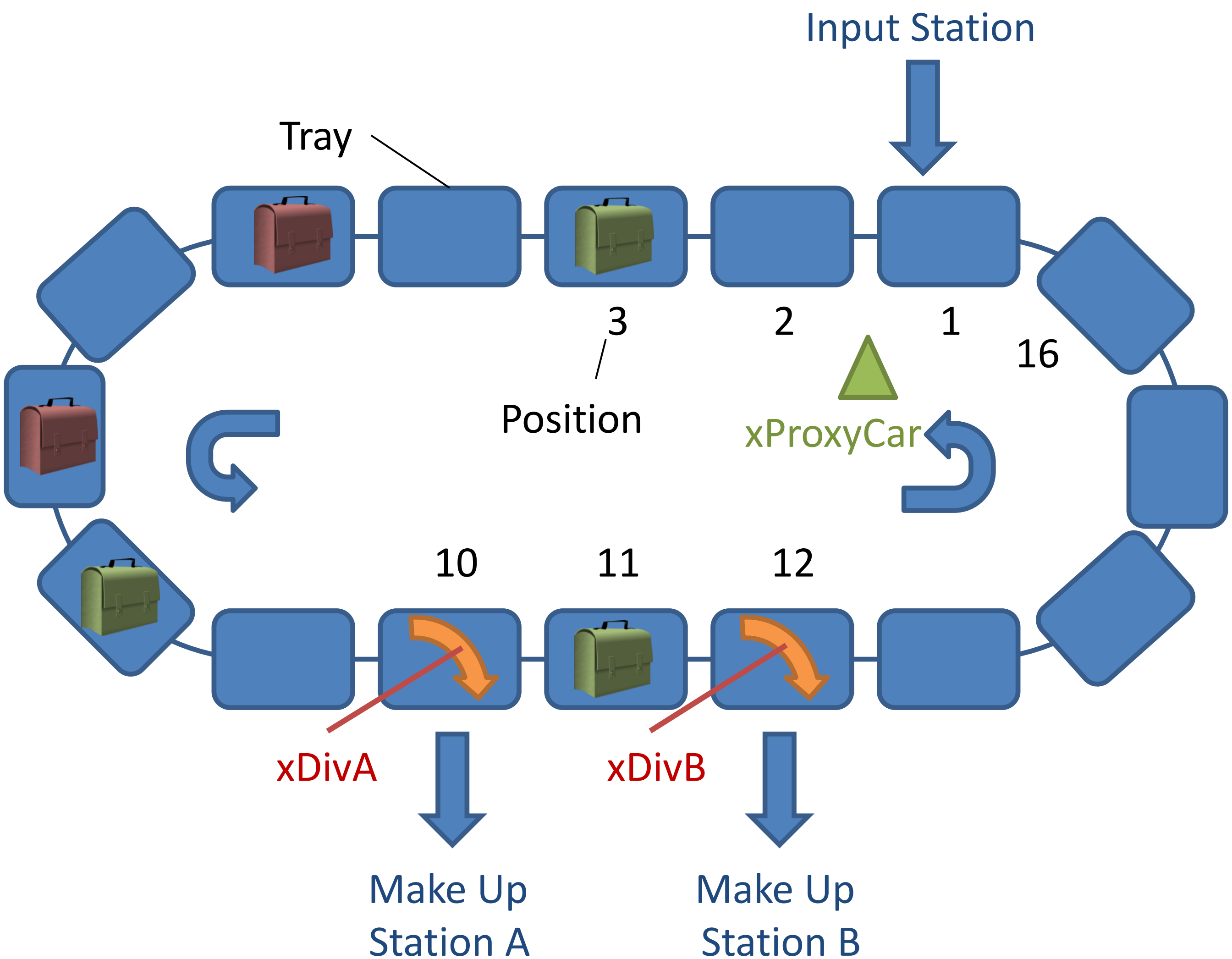
The result (return value) is Boolean (the standard allows declaration of a function without result, but CODESYS does not)

The 16-element array is passed as an input-output variable, by reference, reducing memory use of the application

Elements of the array are copied inside a `FOR` loop (rotation can be implemented several other ways)

Result (return value) of the function must be set before returning (after the last line)

Main program



- The carousel rotates continuously with the baggages in its trays
- Baggages arrive from the input station
- A baggage can be forwarded from the input station if the tray at position 1 is empty
- Rising edge of `xProxyCar` signals one step of the carousel
- Baggages are forwarded to the make up stations by the divertets (`xDivA`, `xDivB` outputs)
- No baggage can be forwarded to a make up station if that is full
- A baggage shall be forwarded to the make up station according to its tag (111: A, 222: B)

Main progrem

- Upon a rising edge of `xProxyCar`
  - the carousel array shall be rotated
  - if the tray at position 1 is empty, the FB of the input station shall be called with `xRequest=TRUE`
  - If a baggage with tag 111 is at position 10 and make up station A is not full, the tray shall be tilted by the diverter and the value 0 shall be assigned to element #10 (the baggage is forwarded to the station, the tray becomes empty)
  - If a baggage with tag 222 is at position 12 and make up station B is not full, the tray shall be tilted by the diverter and the value 0 shall be assigned to element #12
- During each execution of the program: call of station FBs (both input and make up stations)

```
PROGRAM BaggageHandling
VAR_INPUT
  xProxyInput    AT %IX0.0: BOOL;  // Proximity switch of input station
  xProxyCarousel AT %IX0.1: BOOL;  // Carousel proximity switch (PS)
  xProxyInA      AT %IX0.2: BOOL;  // Input PS of make up station A
  xProxyOutA     AT %IX0.3: BOOL;  // Output PS of make up station A
  xProxyInB      AT %IX0.4: BOOL;  // Input PS of make up station B
  xProxyOutB     AT %IX0.5: BOOL;  // Output PS of make up station B
  usiReader      AT %IB1  : USINT; // Tag reader
END_VAR
VAR_OUTPUT
  xConvInput     AT %QX0.0: BOOL; // Conveyor of input station
  xDivA          AT %QX0.1: BOOL; // Diverter at make up station A
  xDivB          AT %QX0.2: BOOL; // Diverter at make up station B
  xConvA         AT %QX0.3: BOOL; // Conveyor of make up station A
  xConvB         AT %QX0.4: BOOL; // Conveyor of make up station B
END_VAR
VAR
  aCar   :       ARRAY [1..16] OF USINT;        // Carousel array
  StationA:      MakeUpStation;  // Make up station A
  StationB:      MakeUpStation;  // Make up station B
  StationInput: InputStation;    // Input station
  RiseProxyCar: R_TRIG;          // Edge-sensing FB for carousel proxy
END_VAR
```

```
StationA(xProxyIn:=xProxyInA, xProxyOut:=xProxyOutA,
        xConv=>xConvA);
StationB(ProxyIn:=xProxyInB,xProxyOut:=xProxyOutB,
        xConv=>xConvB);
StationInput(xRequest:=FALSE, xProxy:=xProxyInput,usiTagIn:=usiReader,
       xConv=>xConvInput);

RiseProxyCar(CLK:=xProxyCarousel);
IF (RiseProxyCar.Q) THEN
   Rotate(aCar);
   IF (Carousel[1]=0) THEN
      StationInput(xRequest:=TRUE, xProxy:=xProxyInput,
                  xConv=>xConvInput, usiTagOut=>aCar[1]);
   END_IF;
   IF (aCar[10]=111 AND NOT(StationA.xFull)) THEN
       xDivA:=TRUE;
       aCar[10]:=0;
   ELSE
       xDivA:=FALSE;
   END_IF;
   IF (aCar[12]=222 AND NOT(StationB.Full)) THEN
       xDivB:=TRUE;
       aCar[12]:=0;
   ELSE
       xDivB:=FALSE;
   END_IF;
END_IF;
```

Call of station FBs during each execution

Rotating the carousel array

If the tray at position 1 is empty, the baggage at the input station is forwarded (`xRequest=TRUE`). Tag of the baggage at the input station (0 if no baggage is present) is loaded to element #1 of the carousel array.

If a baggage with the appropriate tag is present on the tray above the make up station and the station is not full, the diverter tilts the tray, and the given element of the carousel array is reset to zero. Otherwise, the diverter does not tilt the tray.

Alternative solution

```
StationA(xProxyIn:=xProxyInA, xProxyOut:=xProxyOutA,
        xConv=>xConvA);
StationB(ProxyIn:=xProxyInB,xProxyOut:=xProxyOutB,
        xConv=>xConvB);
StationInput(xRequest:=FALSE, xProxy:=xProxyInput,usiTagIn:=usiReader,
       xConv=>xConvInput);

RiseProxyCar(CLK:=x.ProxyCarousel);
IF (RiseProxyCar.Q) THEN
   Rotate(aCar);
   IF (aCar[1]=0) THEN
      StationInput(xRequest:=TRUE, xProxy:=xProxyInput,
                  xConv=>xConvInput, usiTagOut=>aCar[1]);
   END_IF;

   xDivA:=(Carousel[10]=111) AND NOT(StationA.xFull));
   aCar[10]:=BOOL_TO_USINT(NOT(xDivA))*aCar[10];
   xDivB:=(aCarousel[12]=222) AND NOT(StationB.xFull));
   aCar[12]:=BOOL_TO_UINT(NOT(xDivB))*aCar[12];
END_IF
```

Setting the diverter with a Boolean function

If the diverter is set, given element of the carousel array is multiplied by zero (reset to zero), otherwise by 1 (leave unchanged). The `BOOL_TO_USINT` function converts a Boolean value to an 8-bit unsigned integer of value 0 or 1.

Implementation in the template project

- The project already contains a program (`PLC_PRG`) with the declarations of input and output variables, and the freewheeling task executing the program

- Testing
  - the application can be tested using the VISU visualization
  - baggages can be added to the system by the pushbuttons next to the input station
  - a baggage can be removed from a make up station by the pushbutton below its conveyor