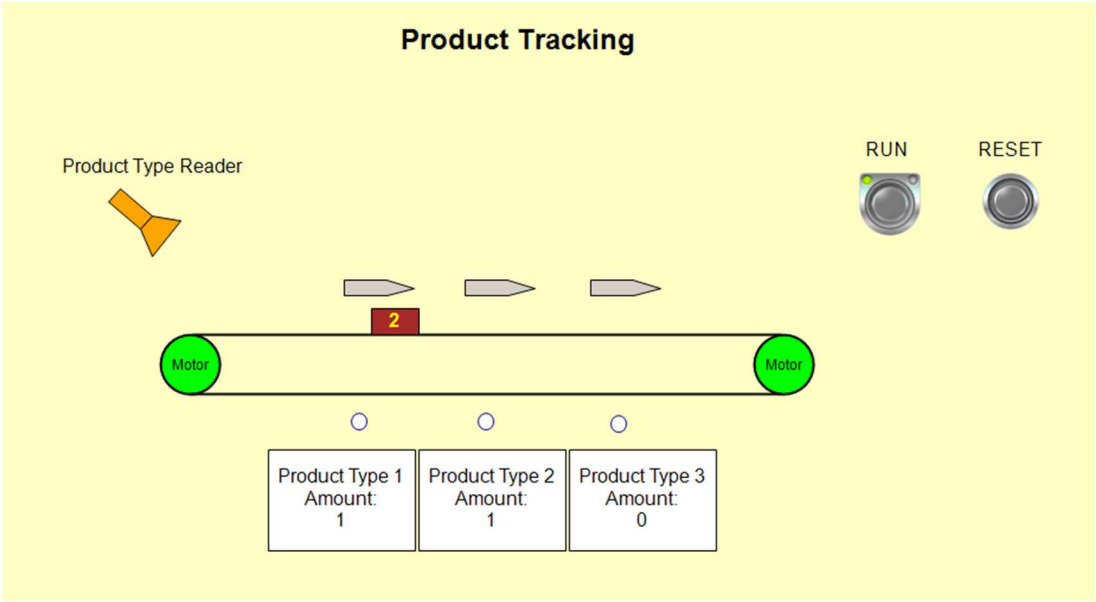
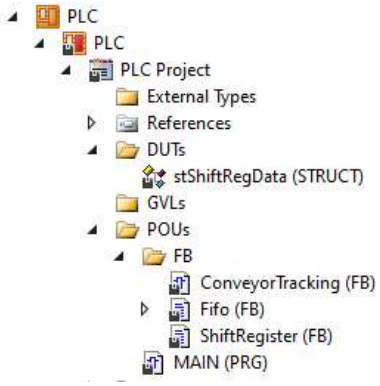
Product Tracking and Sorting

Product and its type are tracked on the conveyor using encoder and shift register.
Three diverters placed on different locations sort out respective product by swinging its arm.

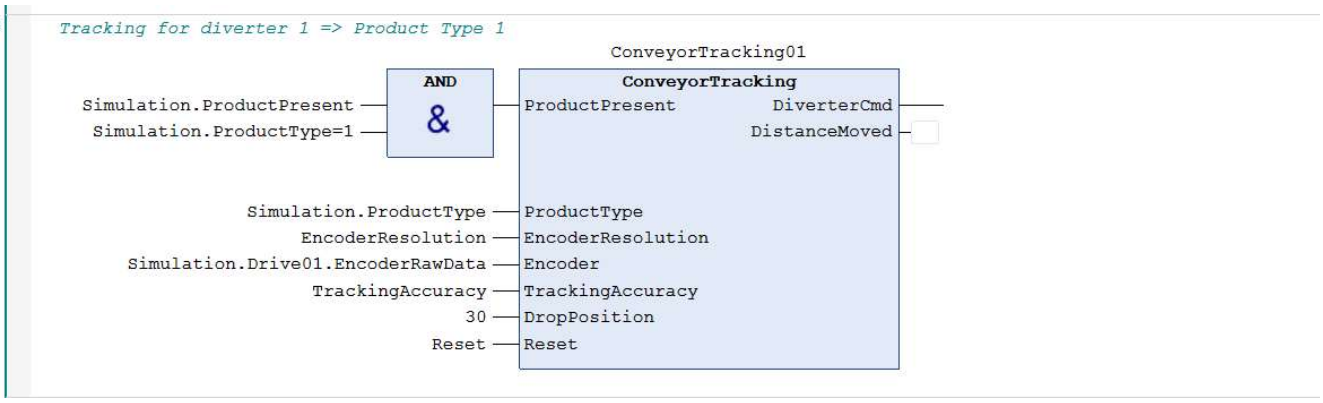


PLC Project Structure:

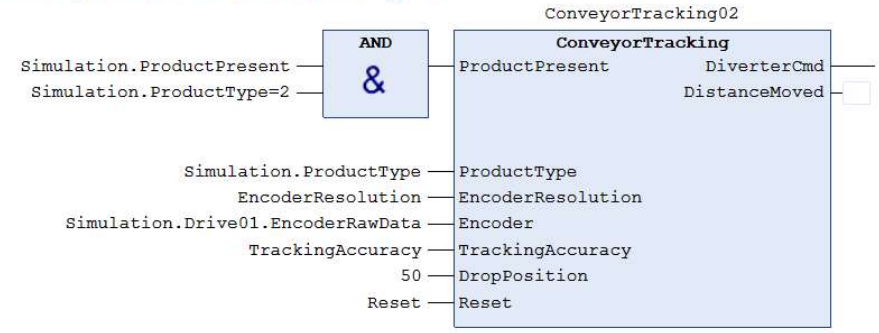


```
(*Shift Register Data Structure*)
TYPE stShiftRegData :
STRUCT
      ProductSts   :   BOOL;          // True=product present
      ProductType  :   INT:=-1;       // 1=big product; 2=middle product; 3=small
product
END_STRUCT
END_TYPE
```

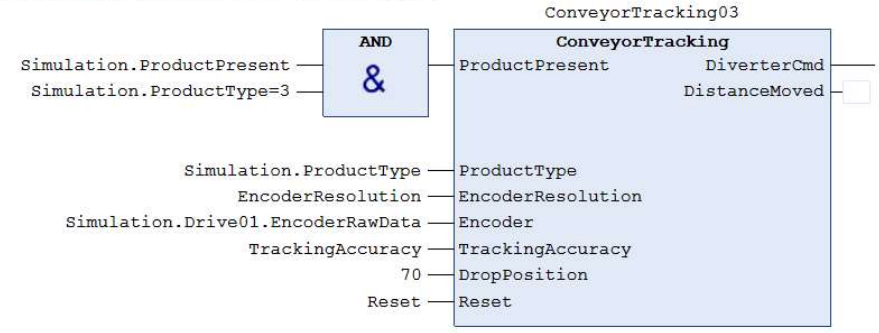
```
PROGRAM MAIN
VAR
      ConveyorTracking01            :     ConveyorTracking;
      ConveyorTracking02            :     ConveyorTracking;
      ConveyorTracking03            :     ConveyorTracking;
      EncoderResolution             :     REAL:=0.2;
      TrackingAccuracy              :     REAL:=1.0;
      Reset                         :     BOOL;
      tpDrvStop                     :     TP;
      DroppedProducts1              :     CTU;
      DroppedProducts2              :     CTU;
      DroppedProducts3              :     CTU;
END_VAR
```



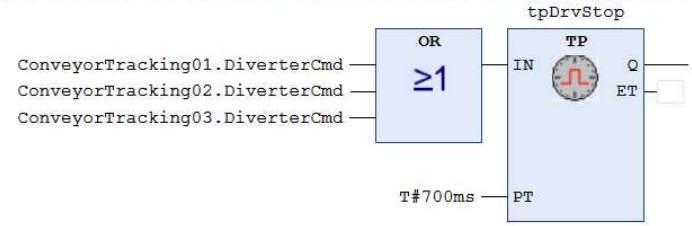
3 *Tracking for diverter 2 => Product Type 2*



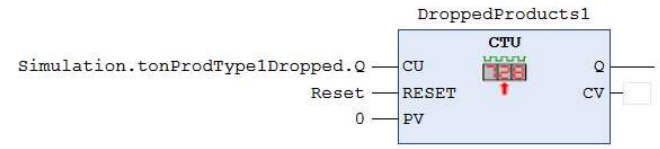
4 *Tracking for diverter 3 => Product Type 3*



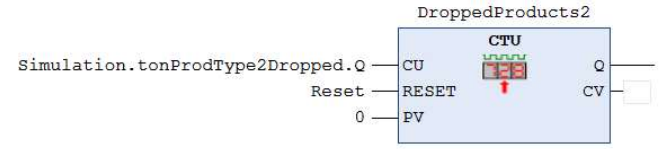
5 *Stop the conveyor drive for a while when any diverter activated*



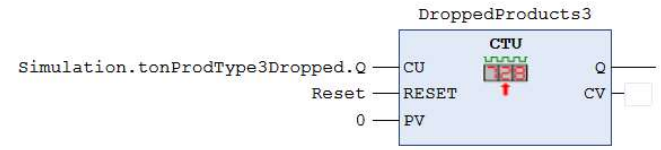
6



7



8

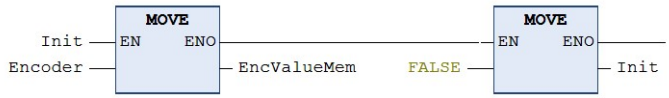


```
FUNCTION_BLOCK ConveyorTracking
VAR_INPUT
      ProductPresent                  :     BOOL;       // sensor detecting the product at the start of the conveyor
      ProductType                     :     INT;        // product type
      EncoderResolution               :     REAL;       // how many mm distance per one encoder pulse [mm/pulses]
      Encoder                         :     LINT;       // raw encoder value in pulses
      TrackingAccuracy                :     REAL;       // needed product tracking accuracy [mm]
      DropPosition                    :     UINT;       // diverter position in [mm]

      Reset                           :     BOOL;       // reset all tracking logic
END_VAR
VAR_OUTPUT
      DiverterCmd                     :     BOOL;       // command to activate the diverter
      DistanceMoved                   :     REAL;       // distance a product has moved since product presence sensor
END_VAR
VAR
      Init                            :     BOOL:=TRUE;
      EncValueMem                     :     LINT;
      EncDiff                         :     LINT;

      rtTrack                         :     R_TRIG;
      TrackTrigger                    :     BOOL;

      Register                        :     ARRAY[0..500] OF stShiftRegData;
      ShiftRegister01                 :     ShiftRegister;
      ctuProductLen                   :     CTU;
      ftrigProduct                    :     F_TRIG;
      ProductLen                      :     WORD;
      FifoLoad                        :     Fifo;
      MiddlePointPos                  :     REAL;

      rtrigProductPresent             :     R_TRIG;
      ProductStarted                  :     BOOL;
      ftrigProductGone                :     F_TRIG;
END_VAR
```

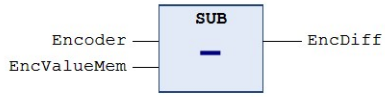
1

```
(*
This is product/package tracking FB.
'TrackTrigger' writes current 'ProductPresent' and 'ProductType' to the Shift Register.
'TrackTrigger' frequency depends on EncoderResolution and TrackingAccuracy.
'DropPosition' defines position in [mm] where the tracked item should be dropped.
*)
```

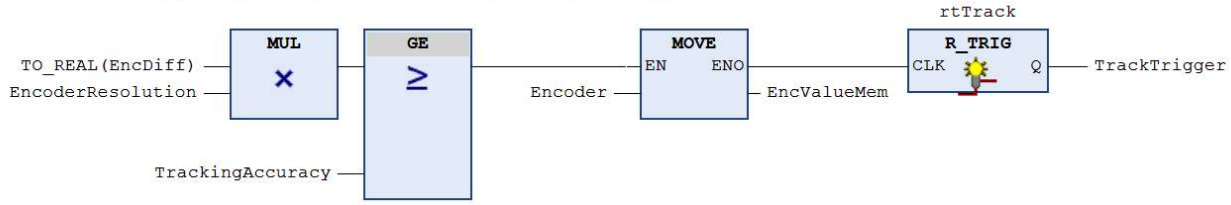
2 *Store current Encoder value to EncValueMem during first CPU scan*



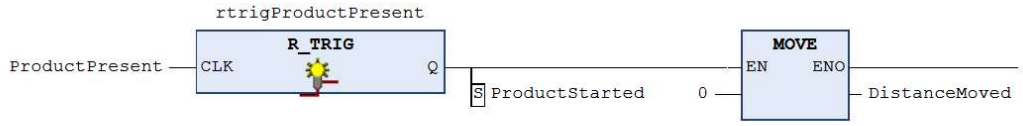
3 *Evaluate raw encoder pulses difference value since last comparison*



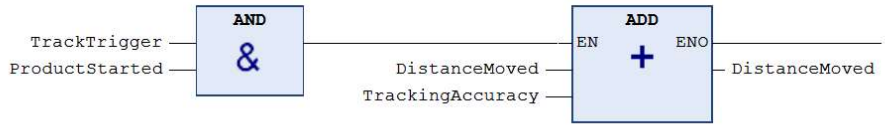
4 *Create one scan pulse when required distance (TrackingAccuracy in mm) moved*



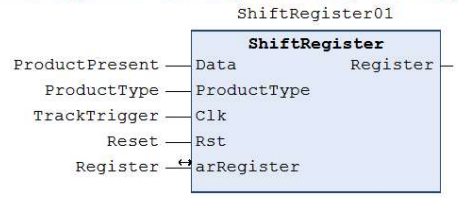
5



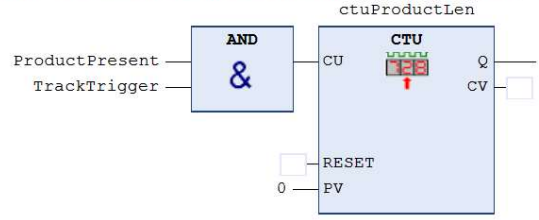
6 *Measure distance passed from product begin*



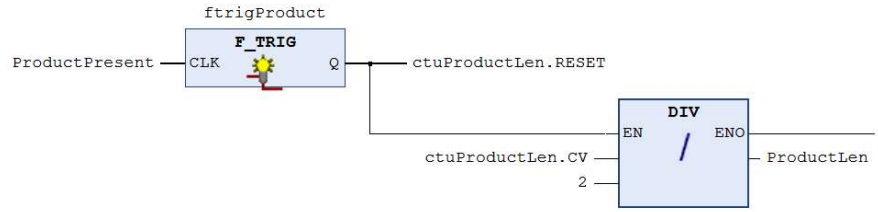
7 *Shift right all tracking data every TrackTrigger pulse*



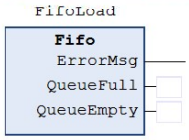
8 *Count the amount of encoder pulses (track triggers) needed for the product to completely pass the 'Product Present' Sensor.*



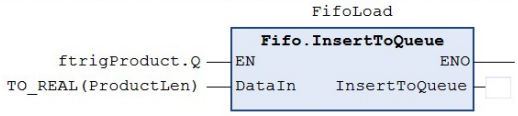
9 *When the product sensor has cleared, calculate the middle point of the product by dividing the length in half.*
Next reset the counter



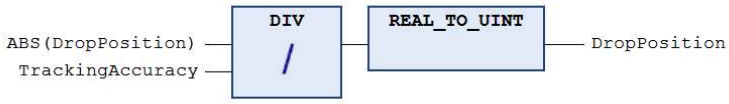
10 *FIFO needed for next network*



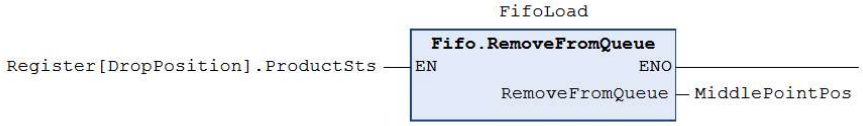
11 *When the product sensor has cleared, store the middle point value into FIFO queue (call the method of FifoLoad FB)*



12 *Convert Diverter Position in mm to proper index in Register array*

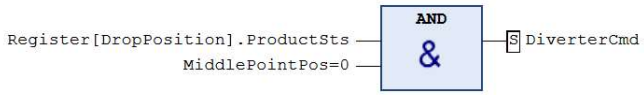


13

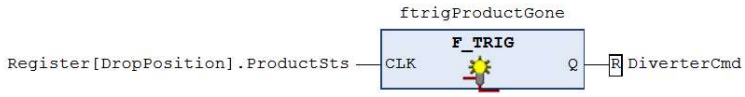


14

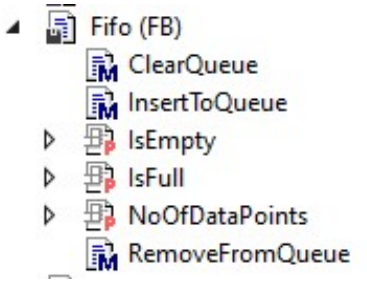
15



16



```
FUNCTION_BLOCK ShiftRegister
VAR_INPUT
	Data			:	BOOL;
	ProductType		:	INT;
	Clk			:	BOOL;
	Rst			:	BOOL;
END_VAR

VAR
	rtrigClk		:	R_TRIG;
	Idx			:	UINT;
END_VAR

VAR_IN_OUT
	arRegister		:	ARRAY[0..500] OF stShiftRegData;
END_VAR

(*
Shift register for complex(STRUCT) data type

Each CLK signal stores 'Data' and 'ProductType'
into first position of the register as STRUCT data type.
All other entries from the register will be then shifted by one position.
*)

// detect rising edge on clk signal
rtrigClk(clk := Clk);

IF rtrigClk.Q THEN
	FOR Idx := 500 TO 1 BY -1 DO
		arRegister[Idx] := arRegister[Idx-1];
	END_FOR
	arRegister[0].ProductSts := Data;
	arRegister[0].ProductType := ProductType;

END_IF

IF Rst THEN
	MEMSET(ADR(arRegister), 0, SIZEOF(arRegister) );
END_IF
```




```
FUNCTION_BLOCK Fifo

VAR_OUTPUT
	ErrorMsg		:	STRING(50);	// error message
	QueueFull		:	BOOL;
	QueueEmpty		:	BOOL;
END_VAR

VAR CONSTANT
	QueueMin		:	INT:=0;
	QueueMax		:	INT:=29;
END_VAR

VAR
	ptrHead			:	INT ;		// pointer to first element (head)
	ptrTail			:	INT ;		// pointer to last element (tail)
	Idx			:	INT;
	InitBit			:	BOOL:=TRUE;
	QueueSize		:	INT;		// size of circular queue
	QueueSizeOk		:	BOOL;
	Queue			:	ARRAY[QueueMin..QueueMax] OF REAL;
	// test:
	QueueLength		:	INT:=8;
END_VAR
```

```
(*
This FB acts as FIFO i.e. circular queue.
Size of the queue must be customized with constants QueueMin and QueueMax.
*)
IF InitBit THEN

	InitBit := FALSE;

	IF QueueMax > QueueMin THEN
		ptrHead := -1;
		ptrTail := -1;
		QueueSize := TO_INT(SIZEOF(Queue) / SIZEOF(REAL));
		QueueSizeOk := TRUE;
	END_IF
END_IF
```

```
PROPERTY IsEmpty : BOOL
```

Fifo.IsEmpty.Get:

```
IsEmpty := ptrHead = -1 AND ptrTail = -1;
```

```
PROPERTY IsFull : BOOL
```

Fifo.IsFull.Get:

```
IsFull := ptrHead = (ptrTail + 1) MOD QueueLength;
```

```
PROPERTY NoOfDataPoints : INT
```

Fifo.NoOfDataPoints.Get:

```
// number of data points in queue
IF ptrTail>ptrHead THEN
      NoOfDataPoints:=ABS(ptrTail - ptrHead)+1;
ELSIF ptrTail<ptrHead THEN
      NoOfDataPoints:=(ptrTail+1) + (QueueLength-ptrHead);
ELSE
      NoOfDataPoints:=0;
END_IF
```

```
METHOD InsertToQueue : BOOL
VAR_INPUT
      DataIn            :     REAL;             // data to insert into circular queue
END_VAR

// Insert new data into queue

IF QueueSizeOk THEN
      IF IsFull THEN
            // queue is full. can't insert new data
            ErrorMsg := 'Queue is full. Can not insert new data';
            QueueFull := TRUE;
      ELSIF IsEmpty THEN
            ErrorMsg := '';
            QueueFull := FALSE;
            ptrHead := 0;
            ptrTail := 0;
            Queue[ptrTail] := DataIn;
      ELSE
            ErrorMsg := '';
            QueueFull := FALSE;
            IF (ptrTail+1) MOD QueueLength <= QueueLength THEN
                  ptrTail := (ptrTail + 1) MOD QueueLength;
                  Queue[ptrTail] := DataIn;
            END_IF

      END_IF
END_IF
```

```
METHOD RemoveFromQueue : REAL;
VAR_INPUT
END_VAR

(* remove one data from the queue *)

IF QueueSizeOk THEN
      IF IsEmpty THEN                                    // zero element in queue
            ErrorMsg := 'Queue is empty';
            QueueEmpty := TRUE;
      ELSIF ptrHead=ptrTail THEN                         // only one element in queue
            ErrorMsg := '';
            QueueEmpty := FALSE;
            RemoveFromQueue := Queue[ptrHead];
            Queue[ptrHead]:=0;
            ptrHead := -1;
            ptrTail := -1;
      ELSE                                              // more then one element in queue
            ErrorMsg := '';
            QueueEmpty := FALSE;
            IF (ptrHead + 1) MOD QueueLength <= QueueLength THEN
                  RemoveFromQueue := Queue[ptrHead];
                  Queue[ptrHead]:=0;
                  ptrHead := (ptrHead + 1) MOD QueueLength;
            END_IF
      END_IF
END_IF
```

```
METHOD ClearQueue : BOOL
VAR_INPUT
END_VAR

(* reinit the queue *)

IF QueueSizeOk THEN
      ptrHead := -1;
      ptrTail := -1;

      FOR Idx := QueueMin TO QueueMax DO
            Queue[Idx] := 0;
      END_FOR

      ErrorMsg := '';
      QueueEmpty := FALSE;
      QueueFull := FALSE;
END_IF
```