Conveyor belts automation with the IEC 61131-3 standard

Course: Programmable Controllers for Industrial Automation

David Savev, ID: 14057

February 10, 2021

1 Requirements

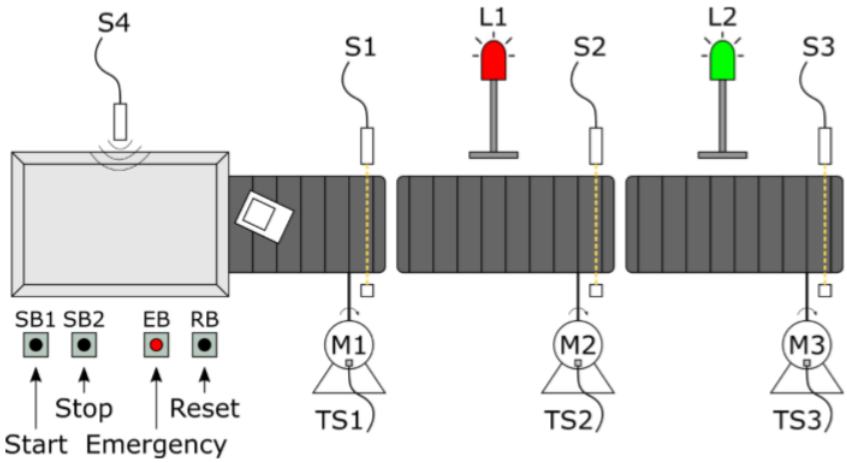

Figure 1: Illustration of the conveyor belt exercise

The task is to design and implement, using the CodeSys IDE, the control system of the three sections conveyor belt illustrated in Figure 1. The conveyor belts will be moved by three electrical motors M1, M2 and M3, each of them embedding a thermostat (temperature-controlled switch) that can be used to detect overheating (TS1, TS2, TS3). The presence of objects is detected by photocells S1, S2 and S3, and inductive sensor S4. The human-machine interface consists of 3 pushbuttons, start, SB1, stop, SB2, and reset, RB, one emergency button, EB, and two lamps, L1 and L2, for signalling operations.

All the motors M1, M2 and M3 are started when a certain sensor detects an

object. For M1 this sensor is S4, for M2 and M3 they are S1 and S2. Each belt must stop if there are no packets on it (condition which can be deduced using S1, S2 and S3, respectively). The stop button lets the machine to complete the delivery operation of packets already on the belt. When the machine is in operation, the L2 light must be constantly turned on. In case of problem with any of the motor, all belts are stopped and the alarm light L1 blinks. Also, when the emergency button is pressed, all the motors are stopped and L1 is turned on constantly. The reset button is supposed to reset the alarms and the machine internal state.

The candidate must prepare a design document including the PLC solution (manufacturer and model of the PLC, number and type of inputs and outputs, external IO modules if needed) and be ready to discuss it.

Note that during the oral exam, the candidate must perform a live presentation of how the code works using a virtual panel visualization, proving that the code works correctly.

2 Introduction

In order to obtain a better understanding about the system needs and, in order to perform a first formalization of the problem, we started by analyzing the components needed in order to develop such a system; more specifically, we can recognize the following input and output components:

Digital inputs:

- *StartButton*: N.O. pushbutton;
- *StopButton*: N.O. pushbutton;
- *ResetButton*: N.O. pushbutton;
- *EmergencySwitch*: N.C. switch;
- *InductiveSensor*: N.O. proximity sensor (typology: inductive).
- *Photocell1, Photocell2, Photocell3*: Block composed by a laser transmitter and a photo-transistor as receiver.
  The main principle behind the component is to let the photo-transistor conduce (similarly to a closed switch) in case of beam detection, while in case of beam interruption to act as an open switch.
- *TemperatureSwitch1, TemperatureSwitch2, TemperatureSwitch3*: N.O. thermostats that signal an overheating on the motor; we have 3 of this components since each one is related to one of the motors of the conveyor belt.

Digital outputs:

- *Motor1, Motor2, Motor3*: mono-phase motors with annex a mechanical reducer to decrease their *rpm* parameter;
- *Light_OK, Light_Emergency*: respectively, a green and red 24V signal light for updating the operator about the status of the system (operative or in emergency).

Hence, in total we have the need to manage and control **11** digital inputs and **5** digital outputs.

3 States and logic

3.1 State machines

We can basically extract the following different states out of the system working requirements: **Idle, Wait, Operative, Emergency**.

1. **Idle:** the system is powered but is in a dormant status waiting to be started by the operator.

2. **Wait:** the system is active and is in busy waiting for incoming objects on the first belt.
3. **Operative:** the system is working and at least one of the conveyor belt motors is switched on
4. **Emergency:** exceptional status thrown either by one of the motor thermostats or from the emergency switch.

The following table is expressing the system states and their relationship with the different actuators:

State	**Output**				
	Motor1	***Motor2***	***Motor3***	***Light Emerg.***	***Light OK***
Idle	0	0	0	0	0
Wait	0	0	0	0	1
Operative	$0 \vee 1$	$0 \vee 1$	$0 \vee 1$	0	1
Emergency	0	0	0	$1 \vee$ Blink	0

Table 1: States-output relationship

We can recognize that the table is not providing ***uniqueness***, since we may have two different outcomes for the same status: in case of the motors, the motor may or not be active depending whether it has objects on the belt, while the emergency light can either blink or be active in the emergency status, depending whether the thermostats of the motors or the emergency switch has changed status.

However, in order to switch from one state to the other, we need an abstract model able to provide a relation between states and inputs; for performing this, a top level state machine was developed:

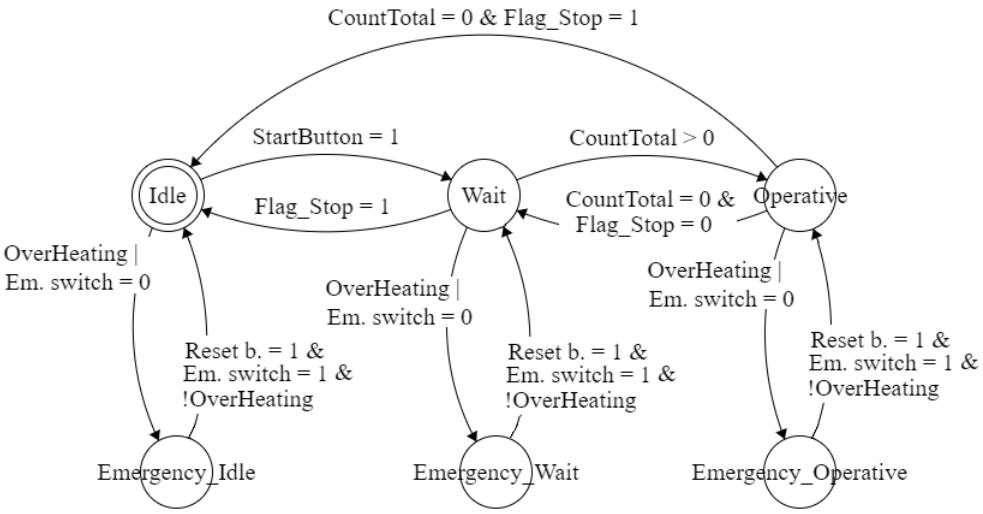

Figure 2: Top level State Machine

We can recognize that the Emergency status was further reduced to **Emergency_Idle, Emergency_Wait and Emergency_Operative** to maintain the reference to the previous status.
Basically:

$$Emergency := Emergency_{Idle} \cup Emergency_{Wait} \cup Emergency_{Operative}$$

OverHeating is simply a function which takes into consideration a logical combination of the motor thermostats, **CountTotal** is a global integer variable obtained by summing up the objects on the conveyor belts and is used to enter or leave the operative status, and last but not least, **Flag_Stop**, which is simply a boolean variable that is set on the pressure of the StopButton, is reset when we leave the current state and it basically retains the button status.

The state machine was in the reality split into two sub-categories, one for the management of the operative states, while the other for the emergency states: **ActiveStateMachine and ErrorStateMachine** and both were implemented in Ladder Diagram.

3.2 Variables

The source code implements two different files containing a declaration of variables:

- **IOVars**;
- **GlobalVars**.

IOVars is concerned with performing a 1:1 mapping to all the physical input and output devices, while GlobalVars specifies the states of the machine (Idle, Wait, Operative, Emergency) and, moreover, contains variables used for internal logic purposes.

```
//Digital inputs controlled by user
StartButton : BOOL := FALSE;
StopButton : BOOL := FALSE;
ResetButton : BOOL := FALSE;
EmergencySwitch : BOOL := TRUE;

//Sensing units (digital inputs)
InductiveSensor : BOOL := FALSE;
Photocell1 : BOOL := FALSE;
Photocell2 : BOOL := FALSE;
Photocell3 : BOOL := FALSE;
TemperatureSwitch1 : BOOL := FALSE;
TemperatureSwitch2 : BOOL := FALSE;
TemperatureSwitch3 : BOOL := FALSE;

//Digital outputs
Light_Emergency : BOOL := FALSE;
Light_OK : BOOL := FALSE;
Motor1 : BOOL := FALSE;
Motor2 : BOOL := FALSE;
Motor3 : BOOL := FALSE;
```

Figure 3: IOVars

```
OverHeating : BOOL := FALSE; //OverHeating condition regarding the motors.
countLine1 : INT := 0; //Objects present on the line nr. 1
countLine2 : INT := 0; //Objects present on the line nr. 2
countLine3 : INT := 0; //Objects present on the line nr. 3
countTotal : INT := 0; //Total number of objects present.

//States of the system; IDLE is the starting state after powering on
IdleState : BOOL := TRUE;
WaitState : BOOL;
OperativeState : BOOL;
Emergency_Idle_State : BOOL;
Emergency_Wait_State : BOOL;
Emergency_Operative_State : BOOL;

//Retained flags
Flag_Emergency : BOOL := FALSE; //Flag used to recognize whether the
//emergency button or OverHeating is active.
Flag_Stop : BOOL := FALSE; //Flag used to know whether the stop
//button was pressed in the current cycle
```

Figure 4: GlobalVars

3.3 Functions and Function Blocks

In order to develop such state machine, we firstly created the following different functions and function blocks to increase the **maintainability, readability and re-usability** of the code:

1. **OverHeating** (Function in Structured Text), which basically takes into account the following logical condition:

   $$TemperatureSwitch_1 \vee TemperatureSwitch_2 \vee TemperatureSwitch_3$$

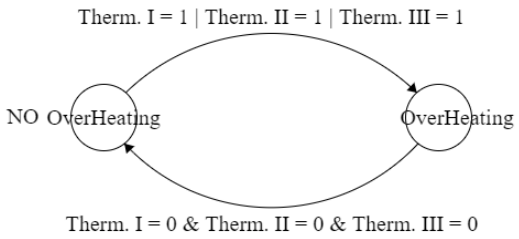

Figure 5: OverHeating function

2. **ObjectCounter** (Function Block in Structured Text): permits the control of the number of objects on the conveyor belts by using 3 up-down counters, 3 global variables related to the different belts (countLine1, countLine2, countLine3) and 1 other variable as result of the sum of the previous three (countTotal).

```
Fall_1(CLK := IOVars.Photocell1);
Fall_2(CLK := IOVars.Photocell2);
Fall_3(CLK := IOVars.Photocell3);

CTLine1(CU := (IOVars.InductiveSensor AND NOT(GlobalVars.Flag_Stop)), CD := Fall_1.Q);

IF(GlobalVars.countLine1 > 0) THEN;
    CTLine2(CU := Fall_1.Q, CD := Fall_2.Q);
ELSE;
    CTLine2(CU := FALSE, CD := Fall_2.Q);
END_IF;

IF(GlobalVars.countLine2 > 0) THEN;
    CTLine3(CU := Fall_2.Q, CD := Fall_3.Q);
ELSE;
    CTLine3(CU := FALSE, CD := Fall_3.Q);
END_IF;

GlobalVars.countLine1 := CTLine1.CV;
GlobalVars.countLine2 := CTLine2.CV;
GlobalVars.countLine3 := CTLine3.CV;

GlobalVars.countTotal := GlobalVars.countLine1 + GlobalVars.countLine2 + GlobalVars.countLine3;
```

Figure 6: Object counter function block

3. **Delay** (Function in Structured Text), which basically delays the off switching of the motors by a θ constant through the usage of a TOFF timer;
4. **Multivibrator** (Function Block in Structured Text), which basically provides a PWM signal with a configurable duty cycle in order to manage the blinking of the Light_OK;

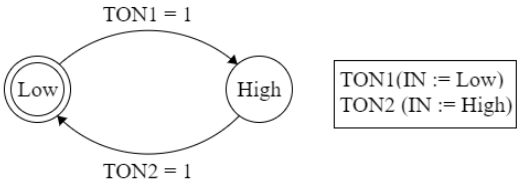

Figure 7: Multivibrator function block

3.4 Output

In order to manage the output actuators, we massively used the functions and function blocks and we divided our code into: ***OutputActive and OutputEmergency***, both writter in Ladder Diagram; the main difference between them is that the first acts in case of normal behaviour of the system (motor control and management of the Light_OK), while the other only for the control of the emergency state (hence managing only the Light_Emergency output).

We can use abstraction and represent the two pieces of code in the following way:

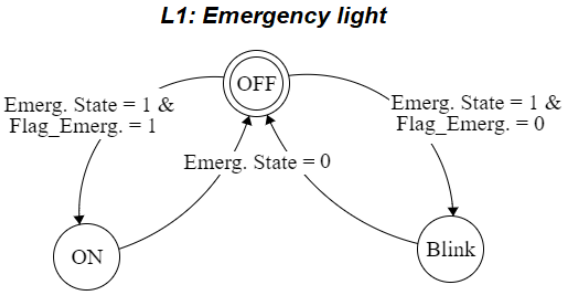

Figure 8: Output emergency

We can recognize the boolean variable **Flag_State**, which is actually retained and directly set by the Emergency Switch; its purpose is to recognize whether the Emergency state is thrown by the motor thermostats (0) or by the Emergency switch (1) in order to blink or be always on.

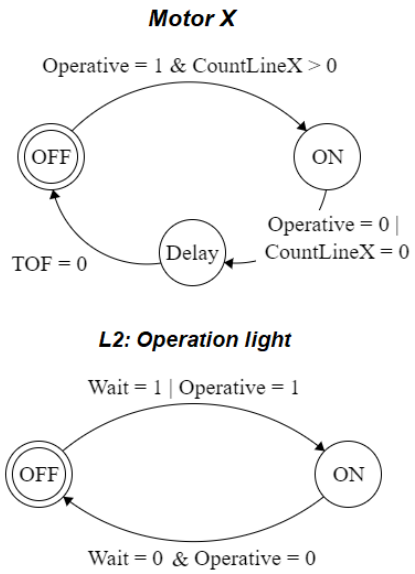

Figure 9: Output operative

For what concerns the Operative output, each single motor is turned on in case the Operative state is active and in case on its belt has at least one object. The motor is switched off after a specific delay θ when no objects is on the related belt or when the system is not anymore in the Operative state.
For what concerns the L2 (Light_OK), it is active in case of Wait or Operative state.

4 Hardware

4.1 Initial considerations

In order to standardize the working logic of the photocell devices (it is more natural to have logical high level in case of beam interruption -hence in case of object detection-, while a logical low is more suitable in case of object absence), I decided to use at the receiver side a **pull-up configuration**, in such a way:

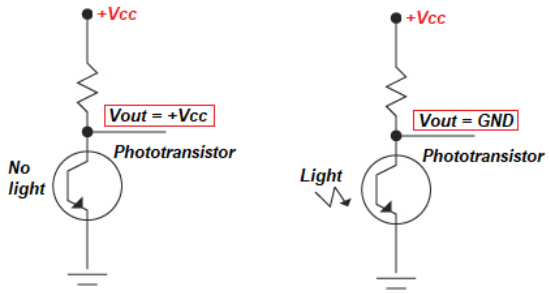

Figure 10: Pull-up configuration

4.2 PLC selection

The Logic controller that was selected for managing sensors and actuators is the **Zelio Logic Smart Relay SR3B261BD**, selled by Schneider electric.
The device can be supplied by 24V DC, has **16** input ports (10 digital and 6 analog/digital) and **10** relay-based output ports supporting a quite large current (8A or 5A), permitting so a direct control of some actuators (lights and laser transmitters). The device was selected after a precise comparison with big competitors (Siemens, Beckhoff, Omron, Mitsubishi) since it was the **cheapest on the market (only 232 euros)** and was having enough input/output ports to control the system **without the need of expansion modules**.

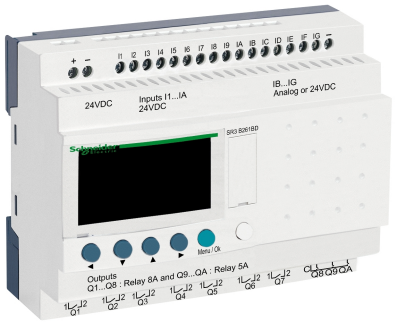

Figure 11: Zelio Logic Smart Relay

4.3 Electric circuit

The electric circuit and connections of the system result in the following way:

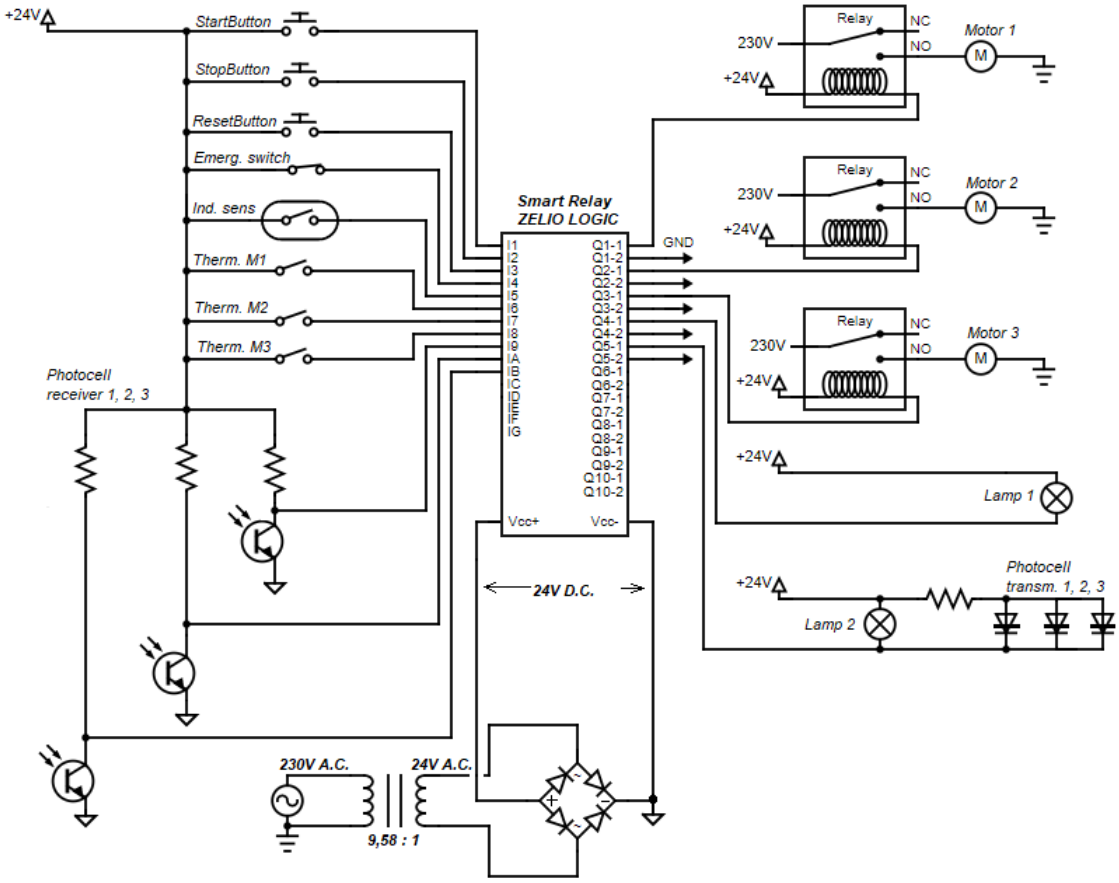

Figure 12: Electric circuit

We can see that the PLC is supplied by 24V D.C. together with the two lights and the laser transmitters of the photocells (which are actually in parallel with the Light_OK since need to work only in the Wait and Operative state) and that the motors are controlled by a power stage composed by 3 relays at 230V A.C. due to their inductive nature and to the high power needed.

5 Visualization

The following graphical user interface was developed in order to control and manage virtually the system through the CodeSys IDE:

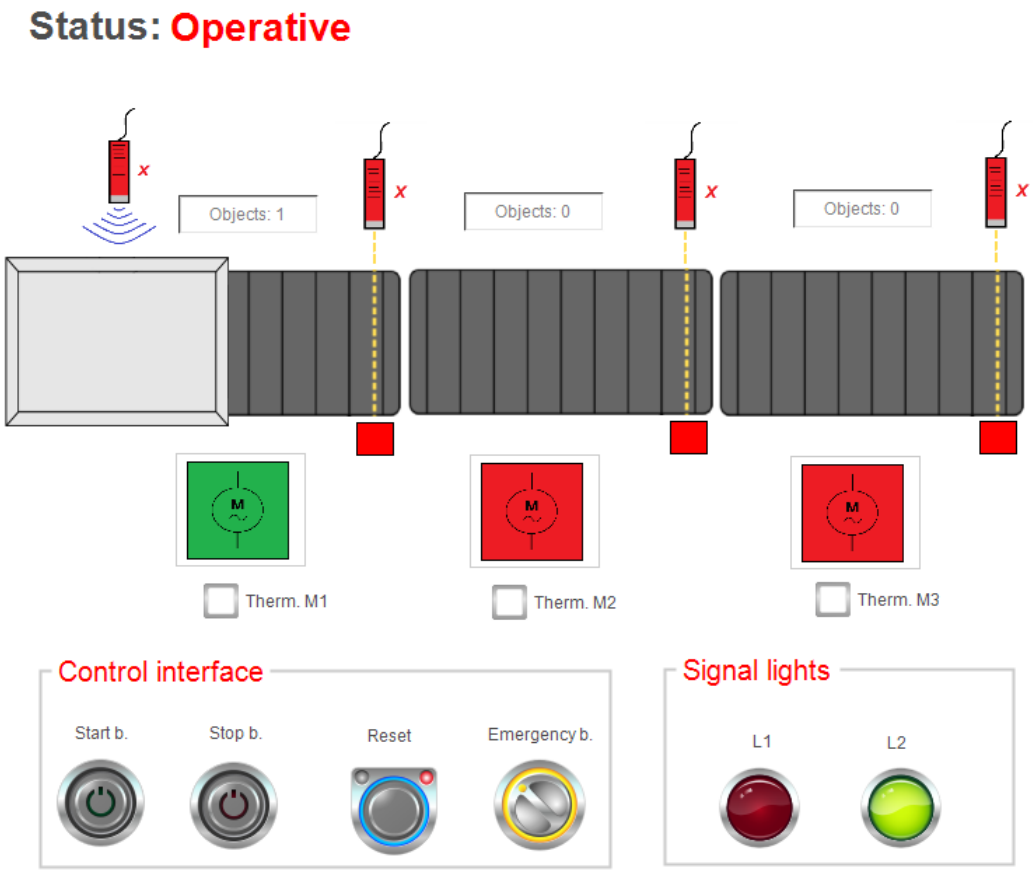

Figure 13: GUI

6 Final considerations

The system can be further optimized by using **one single digital input** for the motor thermostats by basically connecting them in a series configuration; in this way, we will reduce the number of needed digital inputs and the complexity of the source code.

Another important part, is that we are relying to much on the photo-transistors

of the photocells; what if one of this components is in fault? **The system will clearly not work as expected.**
A possible solution for this is to use 3 redundant photo-transistors instead of one and to realize a **Triple-Modular Redundancy circuit**.
The purpose of the circuit is to recognize the sensor status also if one of the components is faulty by taking into consideration the status of the other two.
This technique will however increase the complexity of the circuit and also of the source code.

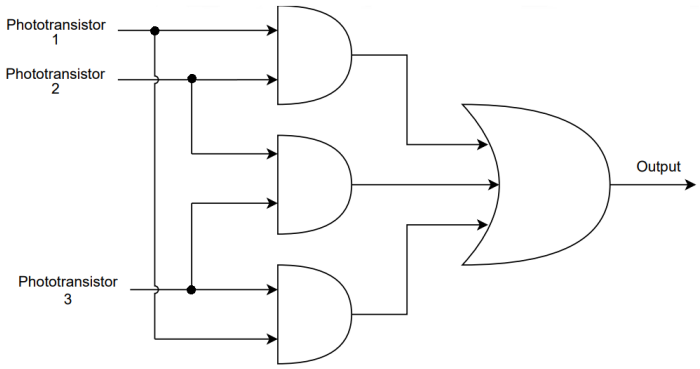

Figure 14: Triple-Modular Redundancy circuit