Subject: Product Note	Product: MPiec Series Controller	Doc#: PN.MWIEC.03
Title: Understanding the Execution of an IEC 61131 Program		

Product Note

Understanding the execution of an IEC 61131 program

Yaskawa Electric America
2121 Norman Drive South
Waukegan, IL 60085
1-800-927-5292

This discussion is attempts to provide casual programmers with some important insight to understand the behavior of a scanning or cyclic PLC system, and to avoid some very common programming mistakes, drastically reducing debug time.

Many novice programmers have trouble with the Structured Text language in IEC 61131. This may be partly due to prior experience with scripting languages, or other proprietary motion languages, most of which can basically be categorized as scripting languages. They typically contain a list of instructions to be interpreted by the system executing them, but with no timeframe. Only one instruction at a time has the attention of the system. By contrast, a PLC is a real time system. It cannot stop and wait for one specific activity to complete. It must scan through and execute all the code required of the system at specified intervals. The most common task in a PLC is the cyclic task. In IEC 61131, it doesn't matter which of the five languages is chosen, execution is governed by the type of task to which the code is applied. The key concept to embrace here is that a real time system, such as a PLC running a cyclic task, must run to completion in the time interval specified. If this cannot be achieved for any reason, the PLC will typically issue a watchdog fault. This leads us to rule #1.

Rule #1: Never use a WHILE instruction that may execute an infinite number of times.

An example would be programming a WHILE loop to wait until a digital input is a specific value. It may never occur. When a WHILE loop executes infinitely, the PLC cannot reach the end of the code in the task within the specified task interval. The digital input scenario is best programmed with an IF statement. The cyclic nature of the task itself will cause the IF statement condition to be re evaluated every scan interval and make the decision whether or not to execute the code in the IF condition. WHILE loops are fine for array initialization, or even a variable number of times within reasonable and finite limits. The maximum will ultimately be governed by the task interval and its priority. If you understand this concept, you're on your way to becoming an IEC 61131 expert.

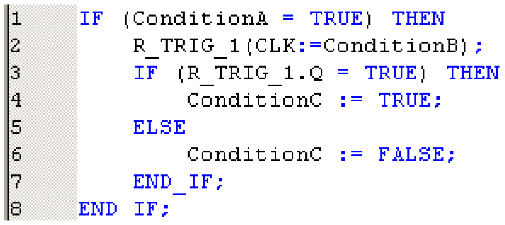
Rule #2: Never put R_TRIG or F_TRIG functions inside "IF" conditions.

To appreciate this rule, you must understand the functionality that R_TRIG and F_TRIG provide. (Hereafter referred to as a Trigger.) Trigger functions monitor the input condition provided for a change from logical zero to logical one, and vice versa for F_TRIG. Triggers have internal memory to store the previous state of the logical condition. This is how they determine if the "edge" or change has occurred. On the scan when a change is detected, the trigger function provides an output that remains high for one scan to identify the event. Now the important part: for a trigger function to continue working properly, it must be executed <u>again</u> at least until the logical condition has changed again so the process of monitoring for the next trigger can occur.

Now that we know what Triggers can do and how they do it, why shouldn't they be placed under an "IF" condition? The code inside an "IF" condition executes only when the "IF" expression is true. This could quite possibly mean intermittent execution of the Trigger functions placed inside. This means possible missing the edge or change.

The problem exposed: Imagine the following scenario.

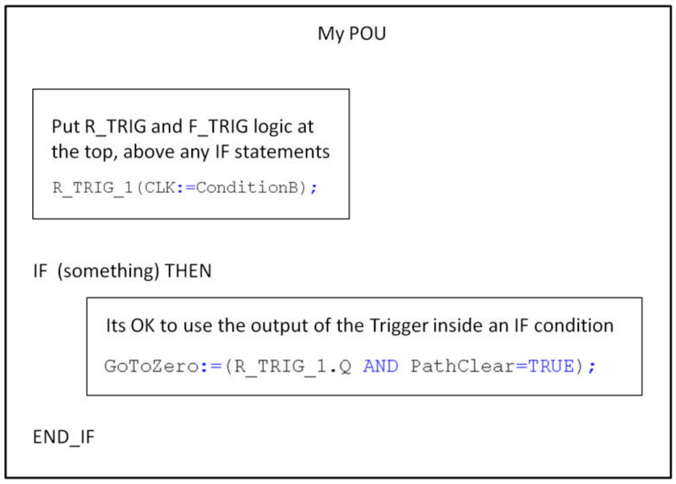
1. An IF statement with ConditionA is TRUE.
2. An R_TRIG with ConditionB is placed inside the IF condition.
3. ConditionB has changed to TRUE and the R_TRIG has changed its output to TRUE for one scan.
4. ConditionC changes to TRUE for one scan, then back to FALSE.
5. ConditionB remains TRUE.
6. Here is the problem: ConditionA becomes FALSE and the code inside the IF condition stops executing. The R_TRIG function is no longer being executed. What happens if ConditionB changes back to the FALSE state? <u>The R_TRIG function will not notice the change.</u> It is unaware that ConditionB has changed back to the 'normal' state. The ultimate failure occurs when ConditionB changes <u>back</u> to TRUE before ConditionA becomes TRUE again. When this happens, the R_TRIG executes, sees that ConditionB is still TRUE, and will not set its output TRUE for one scan. It missed the change of state.



```
1  IF (ConditionA = TRUE) THEN
2      R_TRIG_1(CLK:=ConditionB);
3      IF (R_TRIG_1.Q = TRUE) THEN
4          ConditionC := TRUE;
5      ELSE
6          ConditionC := FALSE;
7      END_IF;
8  END_IF;
```

Figure 1: Example of incorrect R_TRIG usage on line 2.

What's a good solution? Put all Trigger functions at the top of the POU. This way, the Triggers are free to evaluate the input conditions every scan without IF conditions impeding their ability to operate as expected.


My POU

Put R_TRIG and F_TRIG logic at the top, above any IF statements

```
R_TRIG_1(CLK:=ConditionB);
```

```
IF (something) THEN
```

Its OK to use the output of the Trigger inside an IF condition

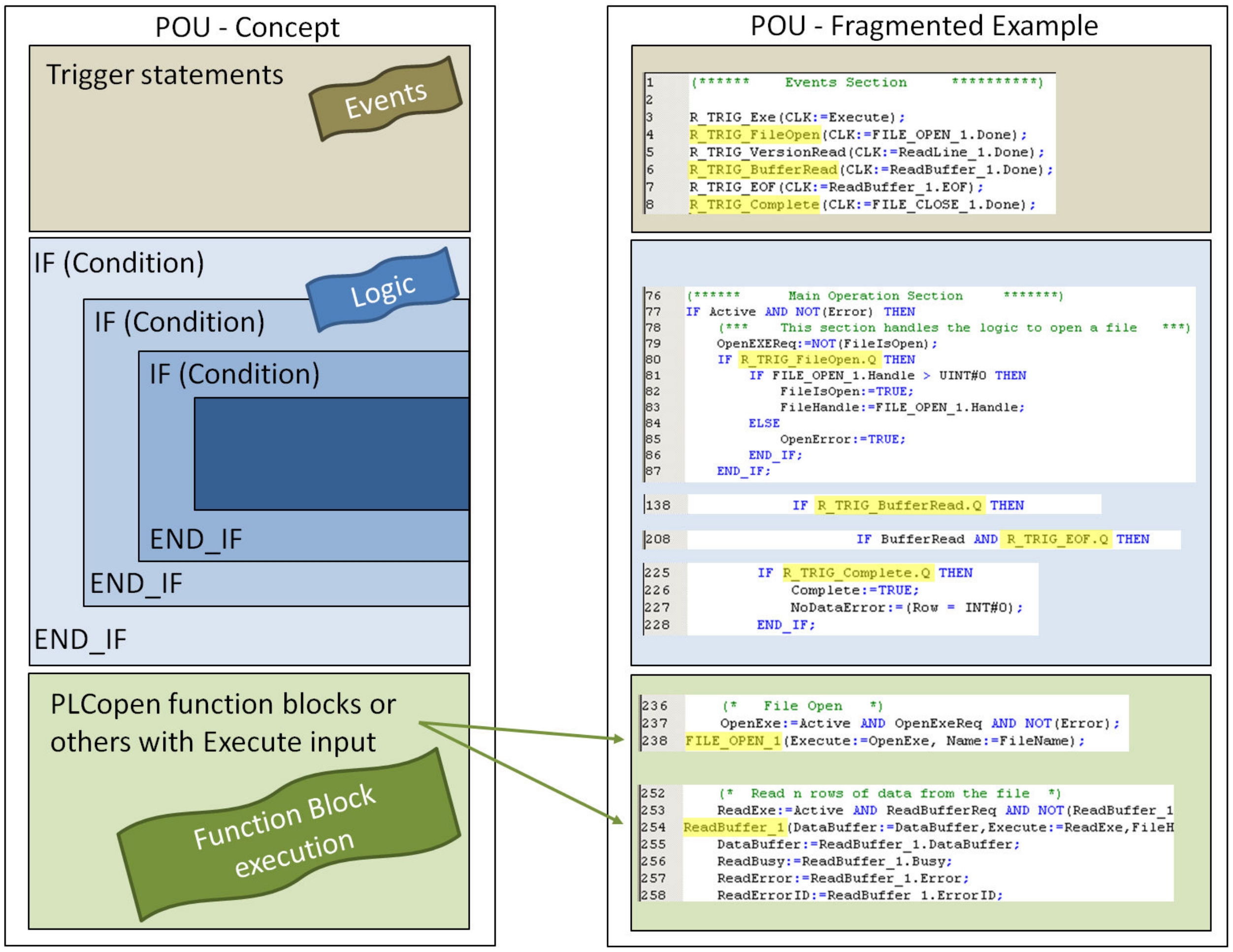
```
GoToZero:=(R_TRIG_1.Q AND PathClear=TRUE);
```

```
END_IF
```

Figure 2: Put Triggers outside of all IF conditions.

<u>Rule #3:</u> Never put Function Blocks with an 'Execute' input inside "IF" conditions.

For nearly the same reasons mentioned in Rule #2 about Triggers, function blocks such as a PLCopen motion functions or any other function that acts upon the rising edge of an input should not be placed under an IF condition. The same trouble can occur if the function is intermittently executed, and does not get a chance to see the falling edge of the input that starts its operation.



```
1    (******     Events Section      ***********)
2
3    R_TRIG_Exe(CLK:=Execute);
4    R_TRIG_FileOpen(CLK:=FILE_OPEN_1.Done);
5    R_TRIG_VersionRead(CLK:=ReadLine_1.Done);
6    R_TRIG_BufferRead(CLK:=ReadBuffer_1.Done);
7    R_TRIG_EOF(CLK:=ReadBuffer_1.EOF);
8    R_TRIG_Complete(CLK:=FILE_CLOSE_1.Done);
```

```
76   (******       Main Operation Section      *******)
77   IF Active AND NOT(Error) THEN
78       (***    This section handles the logic to open a file    ***)
79       OpenEXEReq:=NOT(FileIsOpen);
80       IF R_TRIG_FileOpen.Q THEN
81           IF FILE_OPEN_1.Handle > UINT#0 THEN
82               FileIsOpen:=TRUE;
83               FileHandle:=FILE_OPEN_1.Handle;
84           ELSE
85               OpenError:=TRUE;
86           END_IF;
87       END_IF;

138                 IF R_TRIG_BufferRead.Q THEN

208                        IF BufferRead AND R_TRIG_EOF.Q THEN

225            IF R_TRIG_Complete.Q THEN
226                Complete:=TRUE;
227                NoDataError:=(Row = INT#0);
228            END_IF;
```

```
236      (*   File Open   *)
237      OpenExe:=Active AND OpenExeReq AND NOT(Error);
238  FILE_OPEN_1(Execute:=OpenExe, Name:=FileName);

252      (*  Read n rows of data from the file  *)
253      ReadExe:=Active AND ReadBufferReq AND NOT(ReadBuffer_1
254  ReadBuffer_1(DataBuffer:=DataBuffer,Execute:=ReadExe,FileH
255      DataBuffer:=ReadBuffer_1.DataBuffer;
256      ReadBusy:=ReadBuffer_1.Busy;
257      ReadError:=ReadBuffer_1.Error;
258      ReadErrorID:=ReadBuffer_1.ErrorID;
```

Figure 3: Separating the ST code into three sections

Organizing the code as shown above allows the programmer to write code logic in the central portion of the POU that closely resembles the format and behavior typical of higher level languages such as Visual Basic or C and allows cyclic task execution to operate as intended. Note that the rule about R_TRIG functions applies to the function call only; use of the output as shown on lines 80, 138, and 208 above will not cause any problems.

Rule #4: When using the RETURN instruction, ensure nothing below it would be left in a busy state.

The RETURN instruction is useful because it can provide higher efficiency by skipping lines of code that may be unnecessary given the current mode of operation, but it must be used carefully. This problem avoidance concept is the same as described in Rules 2 & 3. Certain functions must be protected from going dormant, which could affect their ability to execute properly the next time they are required. To eliminate this potential problem point, include the BUSY output and all conditions used for any Triggers below the RETURN statement in the condition for the RETURN to take place. This will force the POU to continue executing every scan while code activity is still in progress, ensuring that no functions go dormant until they are in a quiet state.

We have presented four essential rules focusing on ST programming in IEC 61131. Remember that the rules apply because of the cyclic nature of the task executing the ST code. The two main goal are to eliminate watchdogs, and continue function block execution until all included functions are in a quiet state, allowing them to operate normally the next time they are required. It does not matter which language is chosen in IEC 61131, and the execution behavior is governed by the task type in which they are executed. These rules would also apply to Ladder Diagram (LD) and Function Block Diagram (FBD), but because WHILE and IF statements are not common in those languages, the pitfalls discussed here are less likely uncovered.