

softMC Training – Module 9

Events & Error Handling



Topics

- Events
- Errors

Events

Event Handling

- Events can also be thought of as interrupts
- Events are subroutines that are started by a real-time condition
- Events run in their own context (like a separate task) but have access to task local variables
- Use priority carefully to control execution (default priority is 3)
- Events consist of an event condition and an event action

OnEvent Command Syntax

- Syntax

```
OnEvent <event> {<condition>} {Priority=<priority>} {ScanTime=<time>}  
    <code block that defines the event action>  
End OnEvent
```

event

Any valid name that is otherwise not used in the task

condition

Any logical expression, such as `System.Dout.1 = 1`.

The event is triggered when condition switches from false to true

Priority

An integer from 1 (highest) to 16 (lowest). Default is 1. The priority should always be higher than the priority of the task or the event will never run.

ScanTime

An integer indicating the number of motion bus cycles between each scan. Default is 1 (every cycle).

OnEvent Command Example

- Example
 - An event action is triggered whenever input 1 goes high.
 - The event causes axis **X-axis** to move **10000** counts.

```
OnEvent MOVE_ON_TRIGGER System.Din.1=ON  
    Move X-axis 10000  
End OnEvent
```

OnEvent Program Example

Test.prg

```
Dim Shared I as Long
Dim Shared X[10] as Double

Program
OnEvent PRINTER I = 5
Print "Event: I = 5"
End OnEvent

OnError
  Catch 8001      'Div by zero
  Print "Divided by zero"
  ContinueTask Test.prg
End OnError

EventOn PRINTER 'Turn event on
for I = 1 to 10
  Call Simple
Next I
I = 1/0          'Show OnError
Print "Task continued after error"
End Program

Sub Simple
Print I
End Sub
```

EventOn – EventOff

- **EventOn** enables event scanning
 - OnEvent (name and condition) must be defined before EventOn is issued
 - Syntax

```
EventOn <event>
```

- **EventOff** disables event scanning
 - Syntax

```
EventOff <event>
```

- View list of defined events (from Terminal)

```
?EventList
```

Program Flow and OnEvent

- **GoTo** commands can be used within an **OnEvent** block of code
- **GoTo** commands cannot be used to branch out of or into the event handler (because **OnEvent** interrupts the main program)
- **OnEvent...End OnEvent** cannot be used within program flow commands
- Local variables cannot be declared or used within an **OnEvent** block of code

Error Handling

Error Handling

- An error handler instructs the softMC how to respond to an error in the system
- softMC has a default error handler
- To change how the softMC responds to certain errors or even all errors, you must write an error handler

Severities of Errors

- Errors
 - Errors that can be trapped and handled by the user
 - If not handled by the user, the system has a default error handler
- Fatal Faults
 - Errors that cannot be trapped by the user
 - Severe internal firmware errors
- Notes
 - Messages informing user that something did not occur as expected
 - Does not halt execution

Types of Errors

- Multi-level error handling
 - Program line context (Try ... End Try)
 - Task-wide context (OnError ... End OnError)
 - System-wide context (OnSystemError ... End OnSystemError)
- **Synchronous** errors
 - Associated with a specific program line
 - Example: division by zero or out of range parameters on a command
- **Asynchronous** errors
 - Not associated with a specific program line
 - Example: a following error and overspeed error

Error Handling – In a Program Line

- Try ... End Try is used to handle an error on a specific line of code
- Syntax

Try

<code being monitored>

{Catch *<error number X>*

<code to execute when error X occurs>}

{Catch *<error number Y>*

<code to execute when error Y occurs>}

{Catch else

<code to execute for all other errors>}

{Finally

<code to execute if error occurred and was trapped>}

End Try

Error Handling – In a Program Line

- Example

Try

```
MyVariable = 1/A1.PFB
```

```
Catch 27 'Divide by zero
```

```
Print "Divided By Zero"
```

```
Finally
```

```
Print "An error occurred and it was handled"
```

End Try

Error Handling – In a Task

- **OnError ... End OnError** is used to handle an error that occurs in a specific task
- Syntax

OnError

```
{Catch <error number X>
```

```
  <code to execute when error X occurs>}
```

```
{Catch <error number Y>
```

```
  <code to execute when error Y occurs>}
```

```
{Catch else
```

```
  <code to execute for all other errors>}
```

End OnError

Error Handling – In a Task

- Examples

```
OnError
  Catch 27
    Print "Divided By Zero"
End OnError
```

```
OnError                                'Start of OnError block
  Catch 27                              'Division by zero
    Print "Divided by zero"
  ContinueTask MyTask.Prg
End OnError                              'End of OnError
block
```

Error Handling – In Any Task (System)

- **OnSystemError ... End OnSystemError** is used to handle an error that occurs within the system that is not caught by OnError/Try.
- For example:
 - position error on an axis not attached to a task
 - Velocity over-speed
- Syntax

```
OnSystemError
  {Catch <error number X>
    <code to execute when error X occurs>}
  {Catch <error number Y>
    <code to execute when error Y occurs>}
  {Catch else
    <code to execute for all other errors>}
End OnSystemError
```

Error Handling – In Any Task (System)

- Examples

```
OnSystemError                                'Start of
OnSystemError block
    Catch 27                                    'Divide by zero
        Print "Divided By Zero"
End OnSystemError                            'End of OnSystemError block
```

```
OnSystemError
    Catch 3017                                'Position Error Overflow
        Print "Position error exceeded"
ContinueTask MyTask.Prg
End OnSystemError
```

END