

softMC Training – Module 4

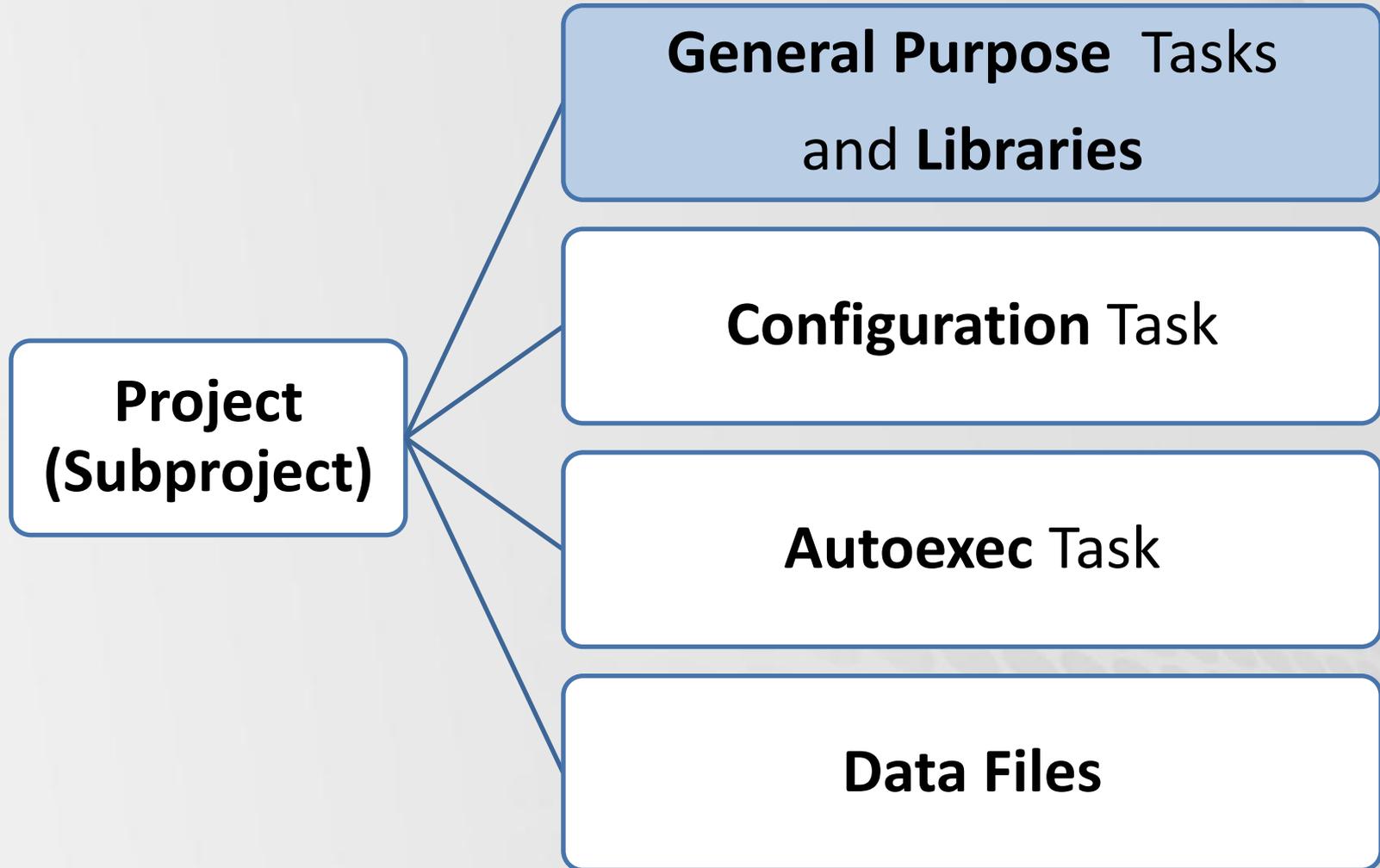
Program Structure



Contents

- General purpose tasks
- Program code blocks
- Variable declarations
- Event handling
- Error handling
- Program flow control
- Subroutines and functions
- Libraries

Project Structure



General Purpose Task Structure

- 3 main blocks

Test.prg

Task Variable Declaration

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

Main Program

```
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
```

Subroutine

```
Sub Simple  
Print I  
End Sub
```

Task Variable Declarations

- At the top of the program file, before the Program keyword
- **Common Shared** declares a variable that is visible to all tasks

```
Common Shared Sys_Var1 as Long
```

- **Dim Shared** declares a variable whose visibility is limited to the task in which it is declared

```
Dim Shared Task_Var1 as Double
```

- **Dim** is used within a program or a subroutine

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
```

Main Program

- The main program block is delimited by **Program ... End Program** keywords
- A task may have only one Program block
- Alternately: **Program Continue ... Terminate Program** block.
Program is automatically executed after loading, and automatically unloaded from memory when it ends.
- The main program block has 3 sections: Start-up, OnError, OnEvent

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
```

Main Program – Start-up Section

- Start-up section immediately follows the **Program** keyword
- Start-up is the point at which task execution begins when **StartTask** command is issued

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
```

Main Program – OnEvent Section

- Optional block of code that responds to a realtime change, such as a motor position changing or an input switch turning on.
- Event handlers reduce the effort required to make tasks respond quickly to realtime events.
- Delimited by keywords **OnEvent ... End OnEvent**
- OnEvent...End OnEvent keyword combination is required for each realtime event.

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
```

Main Program – OnError Section

- Optional block of code that responds to errors generated by the task.
- Error handlers allow program to automatically respond to error conditions, and (if possible) recover smoothly and restart the machine.
- Delimited by keywords **OnError ... End On Error**

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
```

Flow Control

Flow Control

- Instructions used to change the flow of a program based on specific conditions

`If ... Then ... Else ... End If`

`Select Case ... End Select`

`For ... Next`

`While ... End While`

`Do ... Loop`

`Goto`

Flow Control – If ... Then ... Else ... End If

- If ... Then ... Else ... End If
 - If ... Then must be followed by at least one statement
 - Else is optional; if used, it must be followed by at least one statement

- Syntax

```
If <condition> Then
```

```
    <code to execute if statement is true>
```

```
Else
```

```
    <code to execute if statement is false>
```

```
End If
```

Flow Control – If ... Then ... Else ... End If

- Example

```
If (counter < 10) Then
```

```
    Move A1 1000 Absolute = 1 VCruise = 1000
```

```
Else
```

```
    Move A1 2000 Absolute = 1 VCruise = 2000
```

```
End If
```

Flow Control – Select Case

- **Select Case ... End Select** enables one of a number of code sections to be executed, depending on the value of an expression or variable.
- Cases can be specified in one of 4 ways:
 - Exact Value
 - Logical Condition
 - Range
 - Else

Flow Control – Select Case

- Syntax

```
Select Case <SelectExpression>
  {Case <expression>
    {statement_list} }
  {Case Is <relational-operator> <expression>
    {statement_list} }
  {Case <expression> TO <expression>
    {statement_list} }
  {Case <expression> comma <expression>
    {statement_list} }
  {Case Else
    {statement_list} }
End select
```

Flow Control – Select Case

- Example

```
Select Case I
```

```
Case 0
```

```
Print "I = 0"
```

```
Case 1
```

```
Print "I = 1"
```

```
Case Is >= 10
```

```
Print "I >= 10"
```

```
Case Is < 0
```

```
'No requirement for statements after Case < 0
```

```
Case 5 To 10
```

```
Print "I is between 5 and 10"
```

```
Case 2, 3, 5
```

```
'Added in Version 4.7.1
```

```
Print "I is 2, 3 or 5"
```

```
Case Else
```

```
Print "Any other I value"
```

```
End Select
```

Flow Control – For ... Next

- For ... Next is used to define loops in programs
- Syntax

```
For <counter> = <start> To <end> {Step <size>}  
    {Loop statements}  
Next <counter>
```

- If size is not specified, it defaults to 1
- The loop is complete when the counter value exceeds *end*
For positive *size*, complete when *counter* > *end*
For negative *size*, complete when *counter* < *end*
- *counter*, *start*, *end*, and *size* may be long or double

Flow Control – For ... Next

- Example

```
For I = 2 To 5
```

```
    Print "I = " I
```

'Prints 2, 3, 4, 5

```
Next I
```

```
For I = 4 To 2 Step -0.5
```

```
    Print "I = " I  
    3.0, 2.5, 2.0
```

'Prints 4.0, 3.5,

```
Next
```

Flow Control – While ... End While

- **While ... End While** allows looping dependent on a dynamic condition (e.g., loop until input goes high; loop until velocity exceeds a certain value)
 - The condition is evaluated before any statements are executed
 - If no statements are included, While ... End While acts as a delay

- **Syntax**

```
While <condition>  
    {Loop statements}  
End While
```

- **Example**

```
While A2.VelocityFeedback < 1000  
    Print "Axis 2 Velocity feedback still under 1000"  
End While
```

Flow Control – Do ... Loop

- **Do ... Loop** allows looping dependent on a dynamic condition
 - The loop statement block is executed before the condition is evaluated
 - Loop statements are executed at least once
- Use **While** to execute while the condition is true
- Use **Until** to execute while the condition is false

- Syntax

```
Do
    {Loop statements}
Loop [While|Until] <condition>
```

- Example

```
Do
    Sleep 10
Loop Until Sys.Din.1
```

Flow Control – GoTo

- **Goto** unconditionally branches to another section of code.
 - It references a label that must appear within the same program block
 - You can only branch within a Program, Event, Function or Subroutine
 - *label* is a name followed by a colon (:)

- Syntax

```
GoTo <label>
```

```
...
```

```
<label>:
```

- **Note:** Avoid using GoTo whenever possible – it makes programs hard to understand and debug

Nesting

- Nesting means one program control command (or block of commands) is contained within another
 - There is no limit on the number of levels of nesting

- Example

```
For I = 1 to 10
  n = 5
  While n > 0
    n = n - 1
  End While
Next I
```

Subroutines and Functions

Subroutine

- Optional block of code
- A task can have any number of subroutines
- After **End Program** keywords
- Delimited by keywords **Sub ... End Sub**
- May contain local variable declarations, directly after the **Sub** keyword, using **Dim**
- Subroutines are components of a task, and can be called only from the main program within the task
- Executed when called by **Call <sub>**

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
```

Subroutine

- Syntax

```
... Sub <name> ({<par_1>([*])+ as <type_1>} ... {, <par_n>([*])+ as <type_n>})
```

End Sub

<name>

Subroutine name. Maximum 32 characters.

<par_1>, *<par_n>*

Names of array variables

<par_1>([])*, *<par_n>([*])*

Names of array variables

[*] dimension of any array without specifying limits

+ means one or more [*]

<type_1>, *<type_n>*

Type of parameters

Subroutine

- Example

```
Sub Move_Axis_One (Move_Distance as Long)
  Vcruise = 500
  Acc = 10000
  Dec = 10000
  Move A1 Move_Distance
  While A1.ismoving
    Sleep 10
  End While
End Sub
```

Function

- **Function ... End Function** delimits the function
- Functions differ from subroutines in one respect:
Functions always return a value to the task that called the function.
- Functions and subroutines use the same syntax and follow the same rules of application and behavior.
- Because functions return a value, function calls should be treated as expressions
 - Use functions within print commands, assignment statements, mathematical operations, and as conditions of flow control statements

Function

- Syntax/Examples

```
Print <function>{(<par_1>{, ...<par_n>})}  
  <variable> = <function>{(<par_1>{, ...<par_n>})}
```

```
If <function>{(<par_1>{, ...<par_n>})} > 10 Then  
  ? Log( <function>{(<par_1>{, ...<par_n>})} )
```

<variable>

Name of variable

<function>

Name of the function

<par_1>, <par_n>

Names of the parameters passed to the function

- Results are returned in line : <function >= expression
- Returned value type can be: Long, Double, String, Joint, Location, User Defined Structure, Generic Axis, Generic Group

Function

- Example

```
Program
```

```
    ?Add1 (5)
```

```
End Program
```

```
Function Add1 (ByVal a as long) as long
```

```
    Add1=a+1
```

```
End Function
```

Function

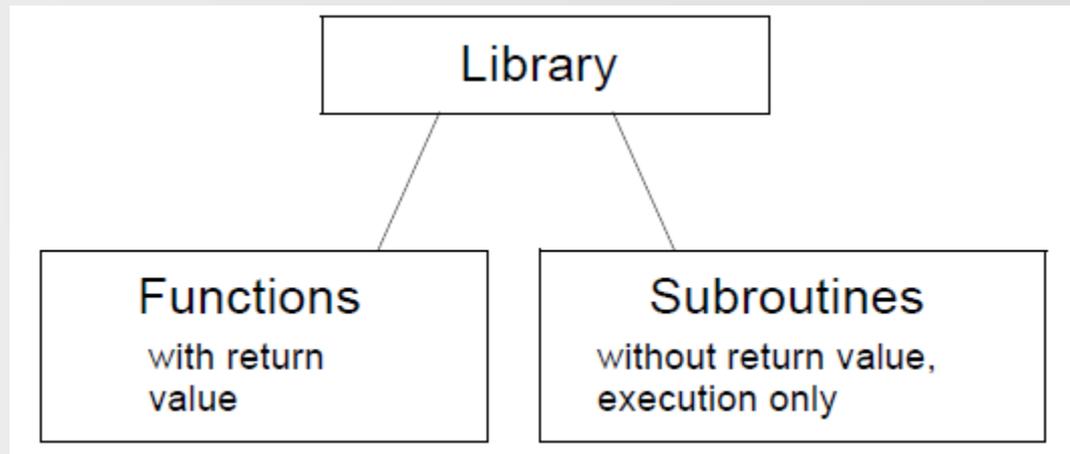
- The following example defines a recursive function to calculate the value of **N**:
- Example

```
Function Factorial (ByVal N As Long) As Double
'Declaring N to be Long truncates floating point numbers to integers
'The function returns a Double value
    If N < 3 Then
        'Statement stops the recursion
        Factorial = N '0!=0; 1!=1' 2!=2
    Else
        Factorial = N * Factorial(N-1) 'Recursive statement
    End If
End Function
```

Libraries

Library Components

- Subroutines and functions can be contained in libraries, and thus can be programmed just once and utilized by a variety of tasks.
- A library is a file that contains only the code of subroutines and functions.
- A library file does not have a main program block.
- A library file has the extension **.lib**.



Types of Libraries

- **Local Library**

- Accessible only to the program that issues the library import instruction
- Must be the first line of program

```
Import <library>.lib
```

- Can be loaded at any time

- **Global Library**

- Accessible to all programs in the system, and within terminal context
- Must be loaded during system start

```
Load < library >.lib
```

when issued from config.prg

```
LoadGlobal < library >.lib
```

when issued from the terminal

- Both types of libraries can be checked using TaskList command

Libraries

- Libraries are loaded into RAM, to be used during task execution
- Keyword **Public** makes subroutine visible outside the library file

- Syntax

```
<Declaration of static variables>  
{Public} Sub <subroutine>  
<Declaration of variables local to the subroutine>  
    <subroutine code>  
End Sub
```

- Examples

```
Import MyLibrary.lib      'Import library into task context  
  
Load MyLibrary.lib      'Load the library  
    into RAM  
  
LoadGlobal MyLibrary.lib 'Import library into system context
```

END