Contents

Overview

The Wizard

Advanced issues

Welcome

Have you ever wondered what is the most common operation you perform when writing VB code?

The answer is simple. Creating procedures.

The event-driven programming paradigm requires that the logic of your applications is spread among many event procedures. And if you embraced the object-oriented functionality of Visual Basic 4.0, you suddenly found yourself writing a huge number of properties and methods.

Yes, you can count on the Insert-Routine command of the VB environment, but it surely can't satisfy the needs of more demanding programmers. It is especially unsatisfactory when writing Property procedures, since you cannot automatically create properties that return a non-Variant value, and have to manually delete code if you wish to generate read-only or write-only properties, just to name the two most frustrating shortcomings.

After you have written hundreds and thousands of Sub, Function and Property procedures, you realize that - while you surely cannot automate the creation of "intelligent" code - there are a number of actions that are similar even in the most different procedures. For instances, many procedures have a remark at the beginning that states the purpose of the routine, the meaning of the arguments passed to the routine and what the routine returns (in case of Functions). Moreover, many procedure must include an error handler, and routinely declare one or more local variables, such as "Dim i As Integer" or "Dim temp As String", and so on.

Writing properties is especially time-consuming, because you are to create two procedures at the same time (Get and Let/Set), duplicate the parameter list, ensure that the type of value assigned in the Let/Set Property procedure is the same as the value returned by the Get Property procedure, and so on. When you dive more deeply into class modules, you will see that many Property procedures are simply "wrappers" around Private member variables, that validate the values being assigned or do other programming chores.

While this activity cannot be actually labeled as "intelligent", it surely takes some time that you could spend in a better way, for instance drinking coffee or going fishing. Besides, this activity is error-prone, so you can't even do it while thinking of how good it would be getting tanned in the Bahamas.

If you too are bored of all this, relax. *The Routine Master* has arrived.

Features

Welcome to *The Routine Master*, the most advanced wizard for automated creation of procedure templates.

This program is meant to work as a stand-alone program or as an addin for Visual Basic 4.0 and is a replacement for the Insert-Routine command, with many more options. Summarizing its many features, The Routine Master lets you:

- create Sub, Function and Property procedures, including read-only and write-only properties
- easily define the type of value returned by a Function or a Property, correctly using Set keywords for routines that return objects
- create procedures that cannot be called recursively
- build a list of parameters, using sensible hotkeys for the most common VB keywords, e.g. Optional, ByVal and all the usual As clauses
- automatically generate all the IsMissing commands for dealing with optional arguments
- optionally add error handling
- quickly create a list of local variables
- add a comment banner, with intelligent word-wrapping and fill-the-blank descriptions of parameters
- automatically add name of author, creation date and time *(registered version only!)*
- create a property that works as a "wrapper" around a private member variable, array or collection - and correctly declare the variable itself
- create "enumerated" properties, i.e. properties that return a String or a Variant value but that are based on a private variable of Integer type (an example: the months of the year) - you can even generate integer values in the form of symbolic constants
- optionally add tracing commands, so that you can have a log of which procedures have been called, and when, together with the value of parameters, when and where errors occurred - you can output to the Debug window or call a user-defined trace procedure, and can even enclose such trace commands in #IF blocks for conditional compilation
- insert generated code to Clipboard or to the current code window
- save your preferences, such as comment banner, rules for generating the name of variables and labels generated by the wizard, and so on *(registered version only!)*
- create multiple procedures during the same session *(registered version only!)*

The shareware version of The Routine Master includes all the features of the registered version, with a few exceptions:

- each time you start the program you have to type some characters into a Nag Screen window: such characters must perfectly match those requested by the program, otherwise you won't be allowed to continue (note that case *IS* significant)
- you can't generate a comment with author name and creation date/time
- you can't create multiple procedures in the same session
- you can't save your preferences to disk
- you cannot alter the templates used by The Routine Master to generate code items

Installation

The Routine Master is designed to work as a VB4 addin, but also works in stand-alone mode, though a few features won't be available in the latter case.

You will find two versions of the program: TRM.EXE is for 32-bit Windows platforms (win95 and NT), TRM16.EXE is for Windows 3.1.

To install The Routine Master, simply copy the TRM.EXE (or TRM16.EXE) and TRM.HLP files in a directory of your choice. If you did not install VB4 bit you should also copy TABCTL32.OCX (or TABCTL16.OCX) into your SYSTEM directory and use the REGOCX32.EXE (or REGOCX16.EXE) utility to register it, issuing the following command from the Dos prompt

```
REGOCX32 TABCLT32
```

or

```
REGOCX16 TABCLT16
```

At this point you may run the TRM.EXE executable file once. This will install the program as a OLE server and will notify Visual Basic that it is ready to work as an addin. You may now close the program, start VB4 and enable it in the Add-In menu.

If you are working under Win95 or NT, the 32-bit version of TRM will work fluently with both versions of VB4.

To uninstall The Routine Master you may run it with the /UNREGSVR flag on the command line. The next time VB4 will be executed it will issue a warning that the addin is missing, and will remove it from the list of available addins.

That's it! Enjoy the program.

Registering

The Routine Master is distributed as shareware. This means that you can install and use the program, and liberally copy it and give to friends and other VB programmers. However, if you find it useful you should register it.

Registration fee is only $10, plus $2 for each additional licensed user. It is not requested that multiple users are from the same company, the only important thing is that all of them are registered at the same time.

When you register you **must** provide the name(s) of registered user(s). In return for your registration fee you will receive one or more passwords, one for each registered user. You will then use this password in the opening nag screen of The Routine Master to complete the registration. Such password will be immediately saved to disk in the TRM.INI file, and you won't see this nag screen anymore.

IT IS STRONGLY SUGGESTED THAT YOU MAKE A BACKUP COPY OF THE TRM.INI FILE IMMEDIATELY AFTER REGISTRATION IS COMPLETE

The registered version of The Routine Master exposes a number of features that are missing or inhibited in the shareware version, namely:

- the creation of a comment holding author name and creation date/time
- the capability to create more procedures in the same session
- the capability to save your preferences to disk
- the chance to modify the templates used internally by the wizard to generate name of variables, labels, etc. and to control the behavior of the wizard itself

You may send the registration fee as US dollars bills, or the equivalent sum in any major currency. If you prefer to mail an international check, please add $4.00 for covering bank charges.

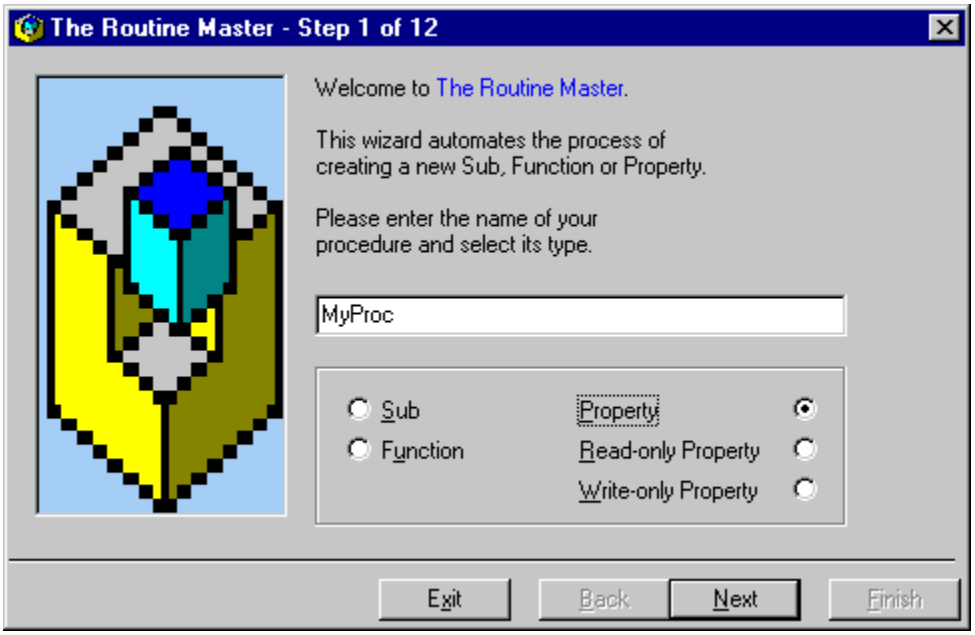
Passwords will be provided by email (preferred), by fax or ordinary mail. If you wish, I can also send you a diskette with a version of the program already registered to you: in this case please add $5 for s/h.

Send suggestions, remarks, bug reports, bills, checks, postcards, complains or whatever to

Francesco Balena
Via Papa Pio XII, 50
70124 BARI - ITALY

email: fbalena@infomedia.it
fax: +39-80-5045490

Step 1 - Name and Type

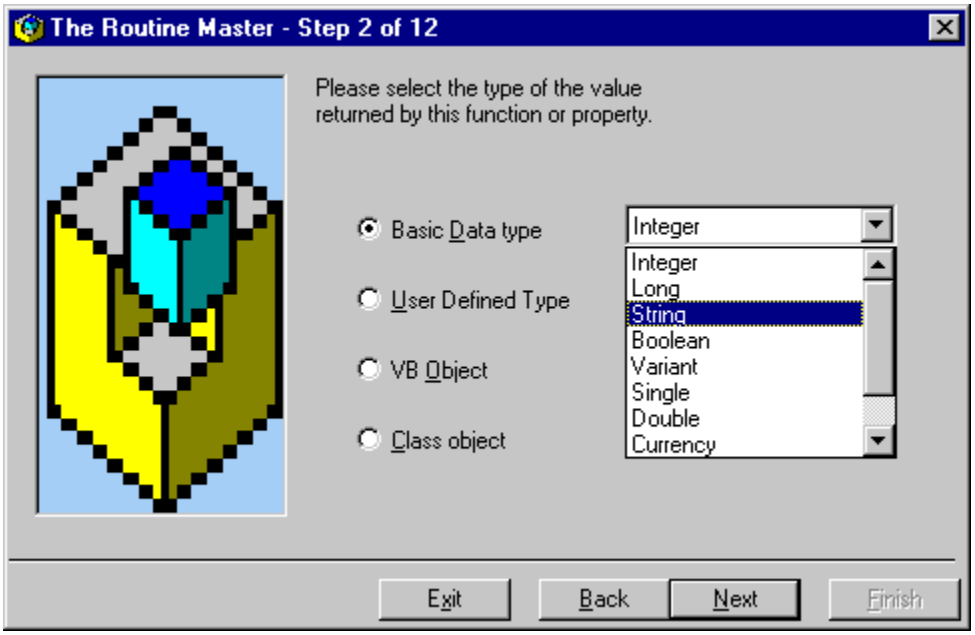


In the first step you have to type the name of the procedure you wish to create and select its type.

If you select the **Property** option you will create both Get Property and Let Property (or Set Property) procedures at the same time. Differently from the Insert-Routine of the VB4 environment, you can opt for read-only or write-only Property procedures.

The Next button stays disabled until you enter a valid name into the first field.

Step 2 - Return value



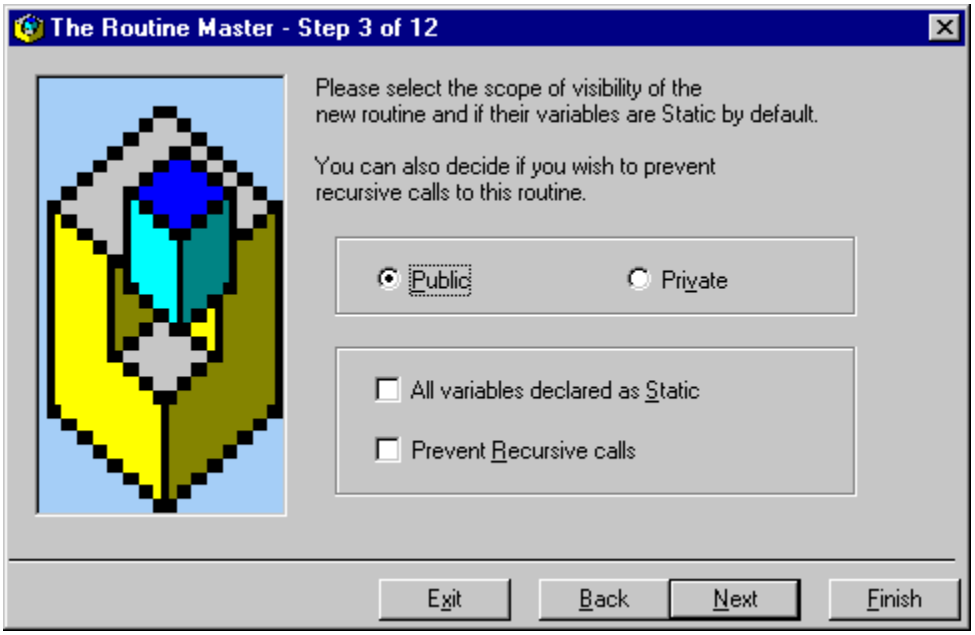
In the second step you have to select the type of the value returned by the Function or the Property procedure. This step is skipped if you selected a Sub procedure type in Step 1.

You can select one of the following data type:

Basic Data Types: choose among Integer, Long, String. Boolean, Variant, Single, Double, Currency, Byte and Date.
User Defined Type : type the name of a UDT declared in your program.
VB Object: select one of the most common type of objects, including Collection, Object, Form, Control - this combo box includes the name of most controls that are provided with the Enterprise Edition of Visual Basic.
Class Object: type the name of a class defined in your program - if The Routine Master has been installed as a VB4 addin, you can select from the list the name of a class module included in your current project.

If you selected a User Defined Type or a Class Object, the Next button stays disabled until you enter a valid name in the control to the right.

Step 3 - Scope and Recursion



In the third step you have to decide the scope of visibility of your routine, if the variables it uses are of Static type and if you do not wish to allow recursive calls to it.

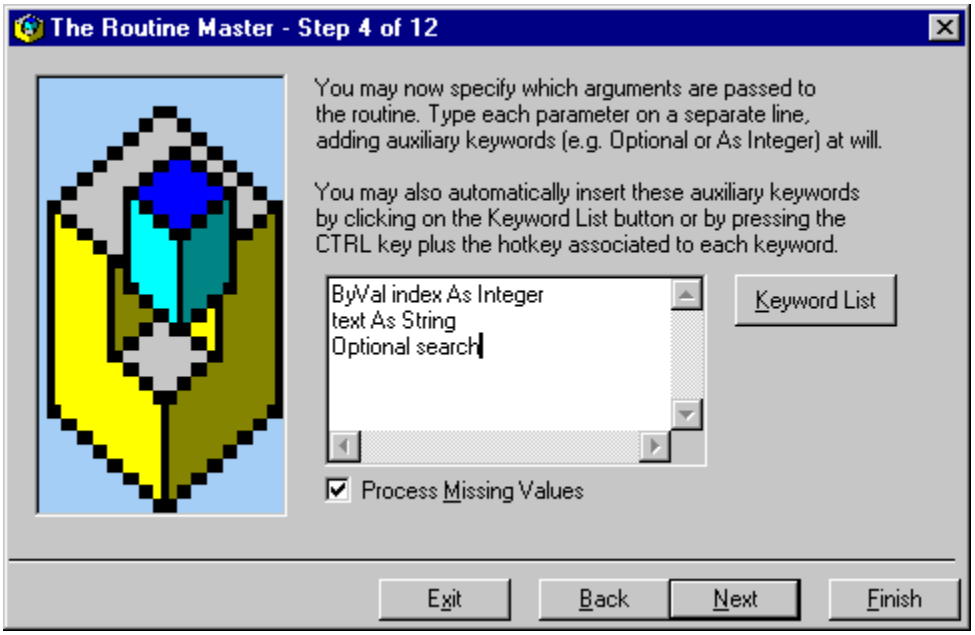
Public, Private: the scope of visibility of the routine. Public routines can be called from other modules, whereas Private routines can only be invoked from within the module where they are defined.

All variables declared as Static: if you mark this checkbox a Static keyword will be added in the first line of the procedure, thus automatically making all its local variable of type Static. Static variables retain their values between consecutive calls to the routine, whereas non-static local values are re-initialized at each call.

Prevent Recursive Calls: if you mark this checkbox, The Routine Master will automatically insert a few statements that prevent recursive calls to the procedure. This is achieved adding a Static Boolean variable that is set to True on entering the routine, and is reset to False on exiting:

```
Sub MyProc ()
Static callInProgress As Boolean
If callInProgress Then Exit Sub
callInProgress = True
'
' here goes the body of the procedure
'
callInProgress = False
End Sub
```

Step 4 - Parameters



In the fourth step you may enter a list of one or more parameters.

To enter a parameter you simply type its name in the textbox, one on each line and using Ctrl+Enter to terminate the current line and start a new one. You can also type all the keywords that VB allows you to specify as a prefix to a parameter (ByVal, ByRef, Optional and ParamArray), and an As clause for specifying the type of the parameter.

To help you in this job, The Routine Master provides a list of the most commonly used keywords, in the form of a menu that you recall with a click on the **Keyword List** button. As soon as you have practiced with this menu, you will find that you can reach the same result by pressing the CTRL key together with the key corresponding to the character underlined in the menu. For instance, you can automatically enter the string "As Variant" by pressing the CTRL key and the "V" key at the same time.

Here is the complete list of supported keywords

Modifiers:
    ByVal
    Optional, ParamArray (these are disabled when defining Property procedures)
As clauses (simple types):
    Integer, Long, String, Boolean, Variant, Single, Double, Currency, Byte, Date
As clauses (objects):
    Object, Collection, Form, Control

Please note that no check is performed on the data entered in the textbox, so it is up to you entering valid parameter names, and add keywords in the correct order.

Specifically, the wizard accepts optional parameters of a type different from Variant, even though everything is automatically corrected when the source code is finally generated (see below).

Process Missing Values: if you mark this checkbox The Routine Master will automatically generate one statement for each Optional parameter that assign a default value to it if it is missing, as in:

```
If IsMissing(search) Then search = ""
If IsMissing(index) Then index = 0
```

By default, optional parameters are treated as if they were strings, so they are assigned to a null string. However, if you specified a numeric type of that optional parameter it will be assigned zero. Moreover, since VB4 only accepts Optional parameters of Variant type, during source code generation the type of the parameter is automatically reset to Variant.

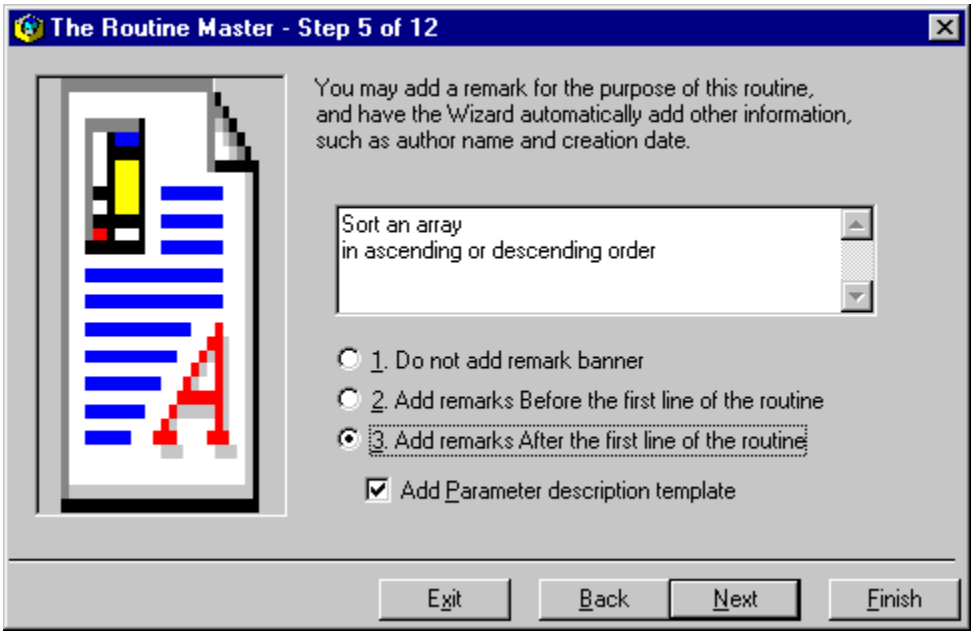
If the parameter list includes a ParamArray item, the wizard automatically generates a loop that iterates on all elements in the array, as follows:

```
Dim i As Integer
For i = LBound(args) To UBound(args)
    ' process parameter array
Next
```

It is your responsibility to complete the loop with proper code.

Please note that this checkbox is not enabled until the textbox contains the words "Optional" or "ParamArray".

Step 5 - Remarks



In the fifth step you may enter a remark for the routine and decide to add a remark banner and its position.

You may enter any comment in the first multilined textbox, using Ctrl+Enter to add new lines. All sentences longer than the allowed length of remarks (60 character by default, but can be modified acting on the TRM.INI file - *registered version only*)

Do not add remark banner: if you select this option the text you have entered in the above textbox will be inserted immediately after the first line of the procedure. No other information will be added. If you are running the shareware version of The Routine Master, this is the only option available.

Add remarks Before/After the first line of the routine: if you select one of these options, The Routine Master will insert a remark banner with several information, such as author name, creation date and time, etc. You can also decide if the banner will go before or after the first line of the routine *(registered version only).*

Parameter Description: if you mark this checkbox the wizard will generated a line for each parameter in the list, and for the return value in case of Functions, that you will then fill with a description for documentation purposes. This checkbox is only active if a remark banner has been selected *(registered version only)*

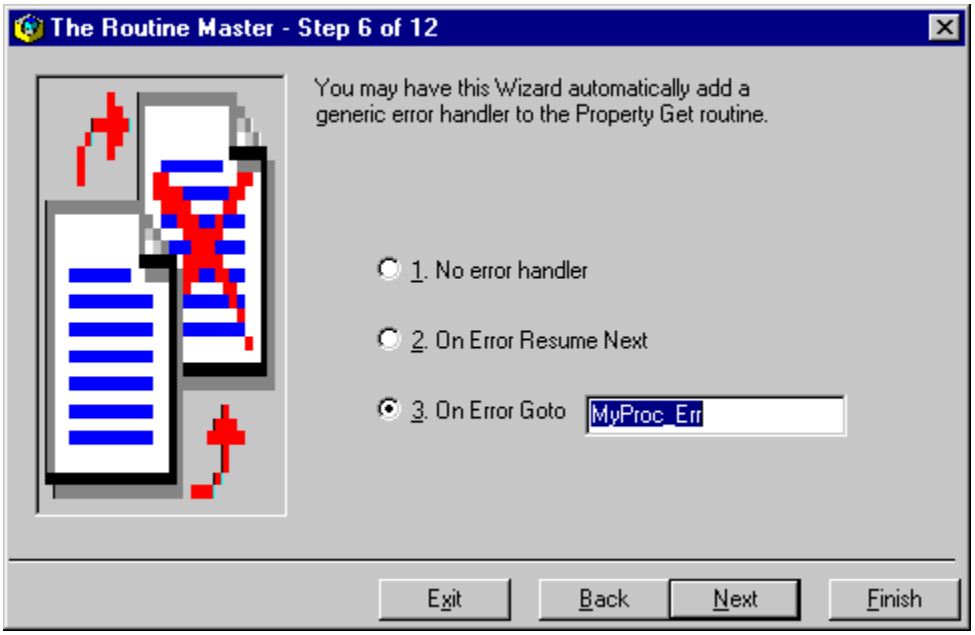
Here is an example of a remark generated by the full version:

```
'************************************************************
' Sort an array
' in ascending or descending order
'
' Author:      Francesco Balena
' Created on: 09-27-1996   (time 15:54)
'
' Parameters
'  text :
```

```
'  index :
' Return Value:
'*************************************************************
```

Please note that the remark banner can be customized by modifying the TRM.INI file *(registered version only)*.

Step 6 - Error Handler



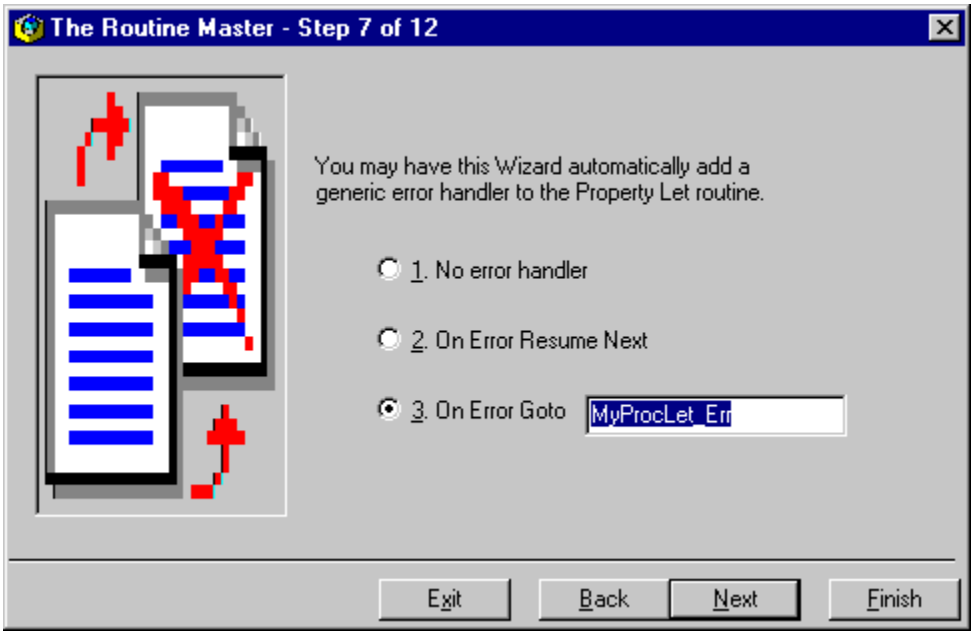
In the sixth step you may add an error handler to your procedure.

You may decide to add an On Error Resume Next statement, or an On Error Goto statement that jumps to the label of your choice, on no error handler at all (the default choice).

If you are building a read-write Property procedure, this step refers to the error handler in the Get Property routine. Click the Next button to set the error option for the Let/Set Property procedure.

By default, The Routine Master builds the error handler label name by appending "_Err" to the name of your procedure. You may change this behavior by modifying the TRM.INI file *(registered version only).*

Step 7 - Error Handler (Let/Set Property procedures)

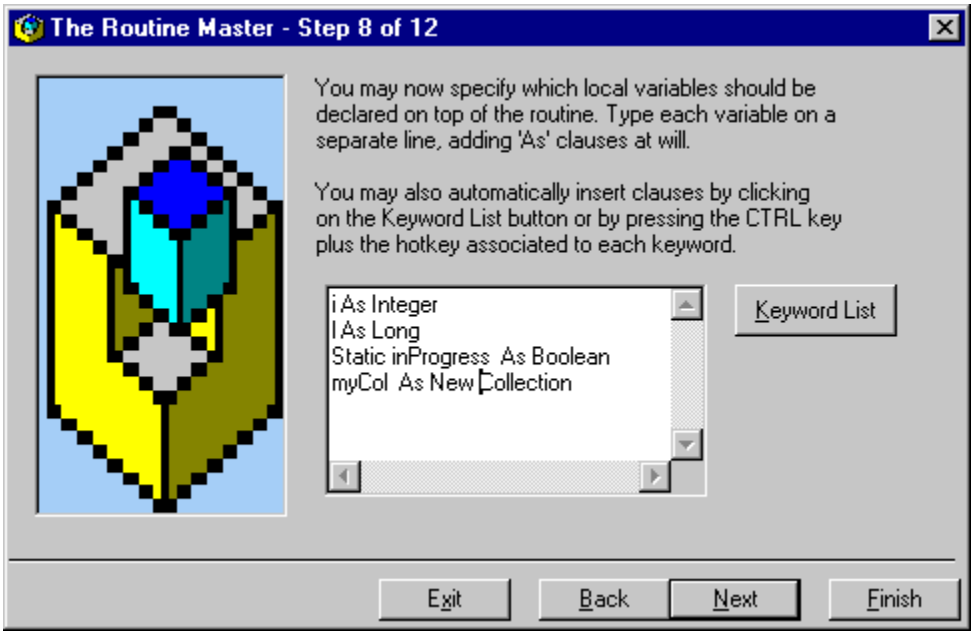


In the sixth step you may add an error handler to your Let/Set Property procedure. This step is skipped if you are defining a Sub, a Function or a read-only Property procedure.

You may decide to add an On Error Resume Next statement, or an On Error Goto statement that jumps to the label of your choice, on no error handler at all (the default choice).

By default, The Routine Master builds the error handler label name by appending "_Err" to the name of your procedure. You may change this behavior by modifying the TRM.INI file *(registered version only).*

Step 8 - Local variables



In the eighth step you may insert one or more local variables.

To enter one or more local variables you simply type their name in the textbox, one on each line and using Ctrl+Enter to terminate the current line and start a new one. You can also type all the keywords that VB allows you to specify as a prefix to a parameter (Dim and Static), and As clauses for specifying the type of the variable.

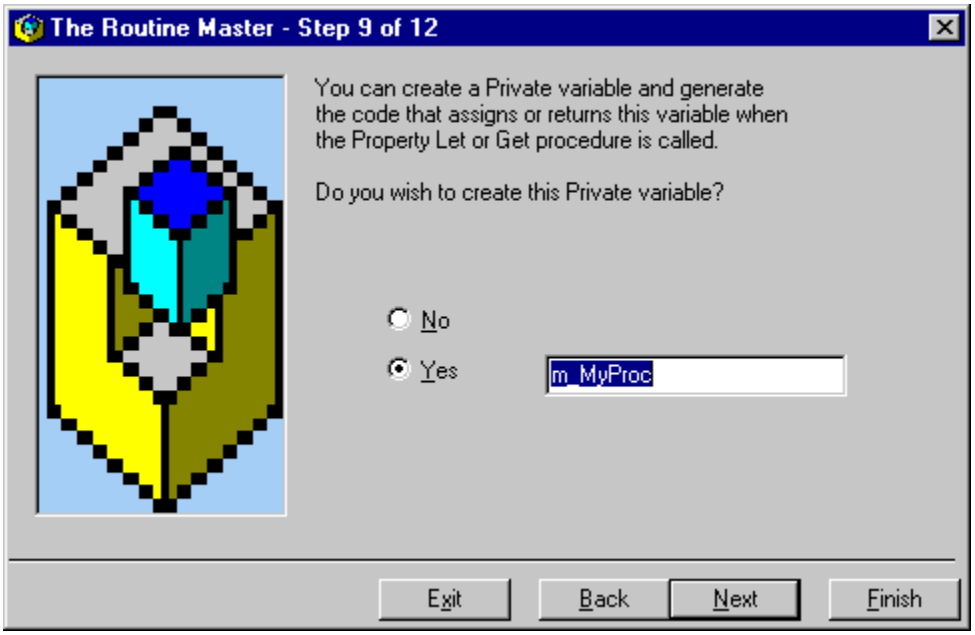
To help you in this job, The Routine Master provides a list of the most commonly used keywords, in the form of a menu that you recall with a click on the **Keyword List** button. As soon as you have practiced with this menu, you will find that you can reach the same results by pressing the CTRL key together with the key corresponding to the character underlined in the menu. For instance, you can automatically enter the string "As Variant" by pressing the CTRL key and the "V" key at the same time.

Here is the complete list of supported keywords

Modifiers:
    Static
As clauses (simple types):
    Integer, Long, String, Boolean, Variant, Single, Double, Currency, Byte, Date
As clauses (objects):
    Object, Collection, Form, Control

Please note that no check is performed on the data entered in the textbox, so it is up to you entering valid variable names, and add other keywords in the correct order.

Step 9 - Wrapper Property Procedures



In the ninth step you may decide if your Property is a "wrapper" around a Private member variable.

Since there is no official term for this kind of properties, by *wrapper property* this documentation refers to a property that encapsulates the value of a Private variable, which is usually named *member variable*. There are a number of reasons for encapsulating the value of a variable into a couple of Property procedures: (1) you can validate the values assigned to the property, (2) you can write read-only or write-only properties, (3) you can perform a given action any time the property is accessed, and (4) you can assign a HelpContextID value to the property. All of the above advantages cannot be achieved if you create the property by simply exposing a Public variable.

If you select the **Yes** option, The Routine Master will automatically create a Private variable of the same type of the Property, and will generate the correct statements that assign the value to such variable (in Let/Set Property procedures) or return its current value (in Get Property procedures). If you want to validate this value or perform any other action when it is read or modified, you just have to add a few statements to the code generated by the wizard.

The type of the Private variable generated by The Routine Master depends on the number and type of parameters associated to the property. If the property accepts no arguments, a simple variable is created. If the property accepts one or more Integer or Long arguments, an array or a matrix is generated. Lastly, if the Property is read-only and accepts one String or Variant argument, a collection is created.

If neither one of the above conditions is met, this step is simply skipped over because The Routine Master assumes that such a property can't be a wrapper property. Of course, this step is also skipped if you are generating a Sub or a Function procedure.

Here is an example of the code generated for a read-write property that accepts one Integer argument and for which you requested to create a memory variable:

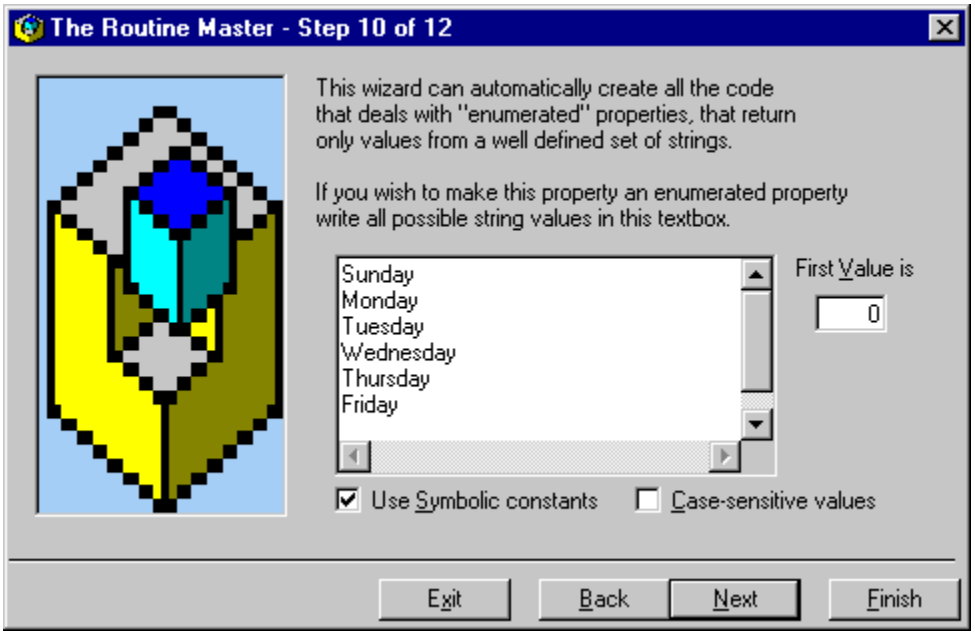
```
' Form-level variable used by PanelWidth Property (modify as needed)
Private m_PanelWidth(0) As Single

Public Property Get PanelWidth(index As Integer) As Single
```

```
      PanelWidth = m_PanelWidth(index)
End Property

Public Property Let PanelWidth(index As Integer, newValue As Single)
      m_PanelWidth(index) = newValue
End Property
```

When an array is generated, as in the above example, it will initially contain only one item. It is up to you to modify the declaration to create the proper number of elements.

Please note that The Routine Master creates the name of the Private variable by prefixing the name of the property with "m_". However, you can change this default behavior by modifying the TRM.INI file *(registered version only)*.

Step 10 - Enumerated Properties



In the tenth step you may decide if this should be an "enumerated" property.

Since there is no official term for this kind of properties, by *enumerated property* this documentation refers to a String or Variant wrapper property that can only return a value from a well defined set of possible values. These properties are special because you can store their current value in the form of a Private variable of type Integer.

For instance, you might create a DayOfWeek property that can be assigned to and that returns the name of the weekday in string format, but stores internally this information in a Private variable of Integer type, using the convention that 0 stands for Sunday, 1 stands for Monday, and so on. Similarly, you might create a property that returns the strings "True" or "False" according to the value of a Private variable that can assume only the values -1 and 0.

There are a number of reasons for creating enumerated properties: (1) you can deal with an Integer variable more easily than a with set of string values, and (2) you can easily test for invalid values, i.e. those values that are not in the set of expected values.

If you wish to create an enumerated property, just type all possible values in the multilined textbox. If you do not want to make the current property an enumerated property, skip this step simply clicking on the Next button without typing anything in this textbox. All the values you enter are assumed to be string values, and you do not have to enclose them within double quotes (if you type a double quote, the wizard assumes that it belongs to the string).

First Value: this wizard automatically assign increasing integer values to all the strings you have typed in the textbox. You can enter a numeric type into this field to set the value assigned to the first string of the set.

TIP: you may use a first value of -1 to create enumerated properties that are based on boolean values. For instance, you could create a m_Done private variable, and type two possible values "Done" (-1) and "Pending" (0).

Use Symbolic Constants: if you mark this checkbox, The Routine Master will generate one symbolic

constant for each string value entered in the textbox. This option produces more code, but it also makes it more readable and gives you the opportunity to later modify the values that correspond to each string.

Case-sensitive Values: the wizard normally generates code in the Let Property procedure that recognizes all possible string values in a case unsensitive fashion, making no distinction between uppercase and lowercase characters. If case is significant to your application, you should mark this checkbox to generate statements that only recognize strings in the correct case.

This is an example of code that implements a DayOfWeek enumerated property, using the settings visible in the figure above. Please note how symbolic constants have been created and how the Let property deals with case unsensitive values.

Also note that both Get and Let Property procedures include a Case Else clause that will traps invalid values. Filling this clause with proper code is your responsibility (e.g. you might add a MsgBox statement or issue a Err.Raise command):

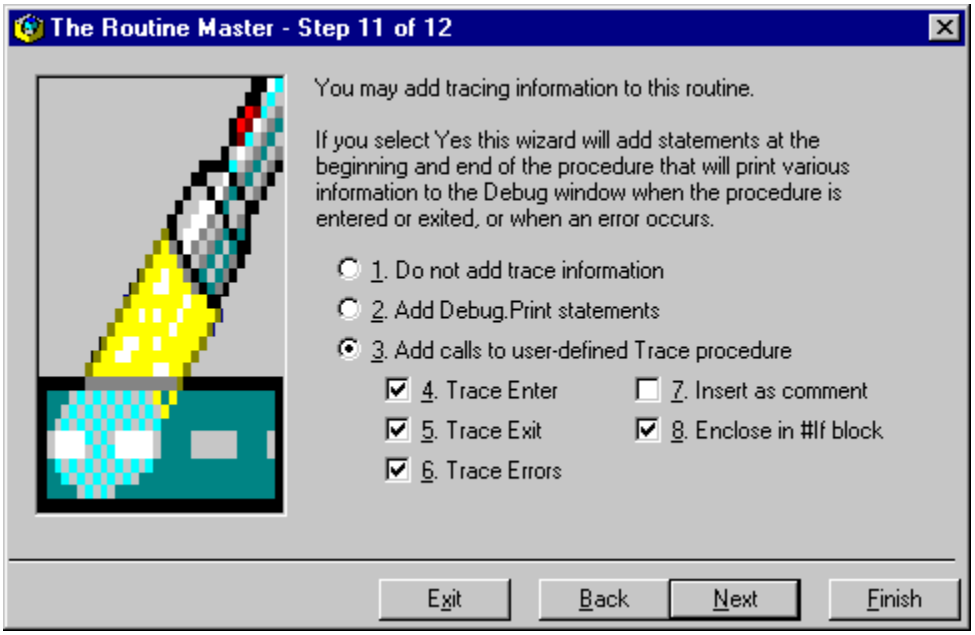
```
' Form-level variable used by DayOfWeek Property
Private m_DayOfWeek As Integer

Const DayOfWeek_Sunday = 0
Const DayOfWeek_Monday = 1
Const DayOfWeek_Tuesday = 2
Const DayOfWeek_Wednesday = 3
Const DayOfWeek_Thursday = 4
Const DayOfWeek_Friday = 5
Const DayOfWeek_Saturday = 6

Public Property Get DayOfWeek() As String
      Select Case m_DayOfWeek
Case DayOfWeek_Sunday
            DayOfWeek = "Sunday"
Case DayOfWeek_Monday
            DayOfWeek = "Monday"
Case DayOfWeek_Tuesday
            DayOfWeek = "Tuesday"
Case DayOfWeek_Wednesday
            DayOfWeek = "Wednesday"
Case DayOfWeek_Thursday
            DayOfWeek = "Thursday"
Case DayOfWeek_Friday
            DayOfWeek = "Friday"
Case DayOfWeek_Saturday
            DayOfWeek = "Saturday"
Case Else
            ' Invalid value
End Select
End Property

Public Property Let DayOfWeek(newValue As String)
      Select Case LCase$(newValue)
Case "sunday"
            m_DayOfWeek = DayOfWeek_Sunday
Case "monday"
            m_DayOfWeek = DayOfWeek_Monday
Case "tuesday"
            m_DayOfWeek = DayOfWeek_Tuesday
```

```
Case "wednesday"
            m_DayOfWeek = DayOfWeek_Wednesday
Case "thursday"
            m_DayOfWeek = DayOfWeek_Thursday
Case "friday"
            m_DayOfWeek = DayOfWeek_Friday
Case "saturday"
            m_DayOfWeek = DayOfWeek_Saturday
Case Else
            ' Invalid value
End Select
End Property
```

If you are not satisfied with the names of the symbolic constants created by The Routine Master, you can easily modify them by using the Find & Replace capabilities of the Visual Basic environment. Similarly, you can change the integer value corresponding to each possible string value by simply changing one Const directive.

The wizard generates the name of symbolic constants using a very simple algorithm. First, it discards all symbols off the string value, then it converts blanks into underscore characters, and finally appends the result to the name of the property. If you do not like this, you may modify the template in the TRM.INI file *(registered version only)*.

Step 11 - Trace commands



In the eleventh step you decide if you wish to add trace commands to your procedure. Tracing commands may report a given event by displaying data in the Debug window or by invoking a user-defined trace routine.

Do not add trace information: this is the default option; if you do not want to add trace statements simply click the Next button.

Add Debug.Print statements: select this option to include Print commands that show the value of parameters and error codes to the Debug window

Add calls to user-defined Trace procedure: select this option to generate Call commands that invoke a user-defined routine called "Trace", that receives all the arguments passed to the procedure.

Trace Enter / Exit / Errors: The Routine Master currently recognize three events: when the procedure is entered, when it is exited and when an error occurs. In the former two cases, the value of parameters are also reported, while in the last case the trace commands report on which error occurred, and where.

Please note that the Trace Errors checkbox is enabled only if a On Error Goto statement has been requested.

Insert as comment: you may request that the trace statement is generated but remarked out, making it actually inactive, but ready to become an executable as soon as you remove the leading apostrophe.

Enclose in #If block: if you mark this checkbox, the trace statement will be enclosed in a #If block, so that you can activate trace by setting a compilation symbolic constant to True and deactivate it - and remove commands from compiled programs - by setting it to False.

This is an example of tracing code, generated using the settings visible in the figure above.

```
Public Sub Testproc(text As String, Optional search As Variant, _
      Optional index As Variant)
#If Trace Then
```

```
      Trace "Entering Testproc", text, search, index
#End If
On Error Goto Testproc_Err

Testproc_Exit:
#If Trace Then
      Trace "Exiting Testproc", text, search, index
#End If
Exit Sub

Testproc_Err:
#If Trace Then
      Trace "Error in Testproc", Err
#End If
Resume Next

End Sub
```

If you opted for displaying trace information on the Debug window, the format of the trace statement is different, because it is necessary to deal with optional arguments

```
Debug.Print "Entering TestProc " & text & ", " _
      & IIf(IsMissing(search), "", search) & ", " _
      & IIf(IsMissing(index), "", index)
```

If there is a ParamArray parameter, the format is still different, and only the first three items of the array are traced to the Debug window.

```
Debug.Print "Entering TestProc " & _
      IIf(IsMissing(args(0)), "", args(0)) & ", " & _
      IIf(IsMissing(args(1)), "", args(1)) & ", " & _
      IIf(IsMissing(args(2)), "", args(2))
```

For more information see "User defined Trace routines".

Step 12 - Generating code

In the twelfth and last step you can select where the generated source code should be copied to, and can choose if you wish to append a new procedure and/or save current settings to TRM.INI file.

Clipboard: the default choice is copying the source code to the Clipboard. This is the only possible choice when The Routine Master has not been installed as a VB4 addin, and also when there is no code window currently open.

Current Code Window: you can also decide to insert the source code into the code window that is currently opened. Please note that the code is added to the current module as if an Insert File command where issued, therefore you will find the freshly created routine at the beginning of the module itself, and any form level statement - e.g. variable and symbolic constants declarations - at the end of the declare section of the file. This option is disabled if The Routine Master is not working as a VB4 addin or if there is no code window that is currently open.

Add New: you can restart the wizard and add other procedures, and then insert the entire generated code in your program in one single operation. *This button is active only in the registered version of the program.*

Save Settings: you can save current settings to TRM.INI file, so that they will be automatically enforced in future executions of the wizard. *This button is active only in the registered version of the program.*

For more information see "The TRM.INI file".

Finish: click on the Finish button to finally generate the code and close the wizard.

There are other possible actions that are not mentioned in the visible user interface:

- if you click with the mouse on the Finish button while you press the SHIFT key the generated code does not include the first and the last line of the procedure (i.e. the Sub/Function/Property declaration and the End statement). For more information see "Create code for event procedures".

- if you click with the mouse on the Finish button while you >press the CTRL key the generated code is

copied to the clipboard but the wizard is not exited. This allows you to paste the code in the VB environment and see if it satisfies you, and to return to the wizard to modify one or more settings if this is not the case *(registered version only)*

- if you click with the mouse on the Finish button while you press the ALT key the generated code is copied to the clipboard but the wizard is not exited. Instead, it is restarted on page one, so that it is ready to create a new routine. Use this option if you plan to generate other procedures in the immediate future *(registered version only)*

- if you right-click on the Finish button a popup menu will appear, that includes all the above mentioned options *(registered version only)*.

Create code for event procedures

If you have created only one procedure - with no form level variables or symbolic constants - and you are copying generated code to the clipboard, you can click with the mouse on the **Finish** button while keeping the SHIFT key pressed at the same time.

The effect of such operation is that the first and the last line of the generated code (i.e. the Sub, Function or Property declaration and the End statement) are not included in the code.

This hidden feature has been added to The Routine Master to ease the writing of code inside event procedures, without also requiring you to manually create the header of the procedure. The Sub/End Sub template of any event procedure can be easily created using the Visual Basic environment, and it would not make sense to duplicate such functionality, since it cannot be sensibly improved.

Therefore, you can use the VB editor to create the outline of an event procedure, and then use this wizard to fill it with code.

Say you have just created a Text1_Change event procedure, but would like to use The Routine Master to fill the empty outline with some significant and useful code. You just need to activate the wizard and fill in the routine name (you can also use cut & paste the procedure name); since Sub is the default procedure type you can advance in the wizard by simply pressing the Enter key.

There are a number of steps that are particularly useful with event procedures:

Step 3: non-recursive events
Step 4: definition of parameters
Step 5: remarks
Step 6: error handler
Step 8: local variables
Step 11: trace statements

Please note that steps 2, 7, 9 and 10 are automatically skipped by the wizard when you are creating a Sub, so you can complete your job even more quickly.

Besides, you can omit to specify the parameter list, unless you also wish to add trace commands.

This is an example of the code you can generate and insert in the middle of a Text1_KeyPress event procedure, after asking to generate remarks and trace commands, a local variable called *char* and a local error handler. Note that we had to manually specify the KeyPress parameter, so that if can be properly traced

```
'********************************************************
'
' Author:     Francesco Balena
' Created on: 09-27-1996  (time 15:54)
'
'********************************************************

Static callInProgress As Boolean
Dim char As String

If callInProgress Then Exit Sub
callInProgress = True
```

```
Trace "Entering Text1_KeyPress", KeyAscii
On Error Goto Text1_KeyPress_Err

Text1_KeyPress_Exit:
      Trace "Exiting Text1_KeyPress", KeyAscii
      callInProgress = False
      Exit Sub

Text1_KeyPress_Err:
Resume Next
```

User-defined Trace Procedures

If you choose to trace capabilities to all the procedures you create with The Routine Master you have two options:

- you can automatically create Debug.Print statements that log procedure calls and errors to the Debug window for a quick-and-dirty list of program events

or

- you can have The Routine Master generate calls to a user-defined Trace routine. The default name for this procedure is "Trace", and it will be passed the values of all the parameters received by the traced procedure

In the latter case you have to provide the Trace procedure yourself. Since that procedure will be called from within many different routines, with possibly different type and number of arguments, your only choice is to let it accept any number of optional arguments. You can do using a ParamArray parameter.

The implementation of this Trace procedure is completely left to you. It could log every information it receives into a file, or open a window and use it for trace purposes. It could log enter and exit time, so that you can even get profiling information. The limit is only your imagination.

This is an example of a user defined Trace procedure, that logs all information to a file, together with timing data:

```
Public Sub Trace (event$, ParamArray args())
Static fileHandle As Integer
Dim i As Integer

If fileHandle = 0 Then
  fileHandle = FreeFile()
  Open "program.log" For Output As #fileHandle
End If

Print #fileHandle, event$;
For i = LBound(args) To UBound(args)
  Print #fileHandle, ", " & args(i);
Next
Print #fileHandle, ""

End Sub
```

Note that you do not really need to explicitly close the log file, since it will be closed by VB when the application terminates.

Another use of the Trace procedure is parameter validation. Say you want to ensure that the value of the first argument passed to a procedure named "MyProc" is always greater than the value of the second argument. Here is a solution

```
Public Sub Trace (event$, ParamArray args())
If event$ = "Entering MyProc" Then
  If arg(0) <= arg(1) Then
    Debug.Print "Wrong arguments in MyProc"
  End If
End If
```

```
End Sub
```

Of course, you can obtain the same information using the watch capabilities of VB, but a user-defined Trace procedure surely adds extra functionality. And all trace settings are stored to disk along your source code, so you do not have to set them up at the beginning of each session.

The following example shows how you can show how much time is spent within the "MyProc" procedure:

```
Public Sub Trace (event$, ParamArray args())
Static startTime As Single
If event$ = "Entering MyProc" Then
  startTime = Timer
ElseIf event$ = "Exiting MyProc" Then
  Debug.Print "MyProc took " & (timer - startTime) & " sec."
End If
End Sub
```

You can build more complex trace procedures if you are willing to take the burden to manage an array of pending calls. For instance, you might build your trace procedure to print the value of the arguments of "MyProc" only when it is called from within "Command1_Click". In this case, however, you must ensure that both "MyProc" and "Command1_Click" include trace commands

```
Public Sub Trace (event$, ParamArray args())
Static calls(99) As String, callsNdx As Integer
Dim i As Integer, found As Boolean

' record all procedures pending on the call stack
If Left$(event$, 9) = "Entering " Then
  callsNdx = callNdx + 1
  calls(callsNdx) = Mid$(events$, 10)
ElseIf Left$(event$, 8) = "Exiting " Then
  callsNdx = callNdx - 1
End If

' if we are entering MyProc
If event$ = "Entering MyProc" Then
  For i = 1 To callNdx
    If calls(i) = "Command1_Click" Then
      found = True: Exit For
    End If
  Next
  If found Then
    ' if this is the case, show parameter values
    For i = 0 To UBound(args)
      Debug.Print args(i),
    Next
  End If
End If

End Sub
```

The TRM.INI file

What follows only applies to the registered version of the program.

When you click the **Save Settings** button in the last step of the wizard, you create or update a TRM.INI file with all the most significant current options that you selected in the previous steps. This file can be found in the same directory as the executable file TRM.EXE, and will be automatically reloaded at the beginning of each session.

Besides saving your preferences for future sessions, the TRM.INI gives you an opportunity to deeply influence the behavior of the wizard and the style of the source code that it generates.

Here is a condensed description of all the values that you can found (and modify) in the TRM.INI file. As you see, all items are commented, so it's very easy to spot the ones you are looking for:

```
[Pages]
; show/hide Local variable page
ShowLocalVariables=1
; show/hide Trace option page
ShowTrace=1
```

The above settings influence the behavior of the wizard itself. If you are sure that you will never want to insert local variables using this wizard (step 8) assign zero to the **ShowLocalVariables** entry. Similarly, if you wish that the wizard always skips step 11 (trace information) set the **ShowTrace** entry to zero.

```
[ErrorHandler]
; template for error handler label (%P=procedure name)
ErrorLabelTemplate=%P_Err
; this stmt will be added to error handlers (not exposed by UI)
ErrorHandlerStatement=Resume Next
```

ErrorLabelTemplate is the "template" string used to generate the label of the error handler. The default label is built by appending the "_Err" suffix to the name of the procedure, but you can change this if you with. Just keep in mind that the "%P" sequence in the template will be substituted with the name of the procedure. If you save settings from within the wizard, The Routine Master tries to deduct the new template and stores it in the INI file.

Whatever follows the **ErrorHandlerStatement** item is appended to the error handler routine. The default statement is "Resume Next" but you can change it at will. To suppress automatic generation of this statement you have to break the line immediately after the "=" symbol. Do not delete the whole line, since this would restore the default setting. Note that you can modify this value only by directory manipulating the INI file, because this setting is not exposed by the user interface of The Routine Master.

```
[Trace]
; the name of the routine that will be called if trace is enabled
TraceRoutine=Trace
; other trace templates
TraceEnterTemplate=Entering %P
TraceExitTemplate=Exiting %P
TraceErrorTemplate=Error in %P
; default state of options on trace page
TraceEnter=1
TraceExit=1
TraceErrors=1
```

```
TraceRem=0
TraceIfBlock=0
```

TraceRoutine is the name of the user defined routine that will be called if you enable tracing. You can enter whatever you like in this field, but keep in mind that the wizard does not validate this item, so if you type an incorrect value the generated code won't work as expected.

TraceEnterTemplate, **TraceExitTemplate** and **TraceErrorTemplate** are the templates of the strings printed to the Debug window or passed to the user-defined trace routine when the routine is entered or exited, or when an error occurs, respectively. You can change these template at will, knowing that the "%P" sequence will be substituted with the name of the procedure you are creating.

TraceEnter, **TraceExit** and **TraceErrors** reflect the default state of the checkboxes with the same name. **TraceRem** is the default state of the "Insert as comment" checkbox, while **TraceIfBlock** is the default state of the "Enclose in #If block" checkbox. If you save settings after modifying such fields, your choices will be stored in the INI file.

```
[Code]
; number of tabs used to indent code (not exposed by UI)
IndentTabs=1
; number of spaces for each tab
TabSpaces=4
; template for exit label (%P=procedure name)
ExitLabelTemplate=%P_Exit
; template for the name of static variable that prevents recursion
RecursionVariableTemplate=callInProgress
```

IndentTabs is the number of tab characters used to indent all the code inside the routine, except the initial declaration statement, the ending End statement and all labels. The default value is one, but you can manually reset it to zero if your coding style requires that.

TabSpaces is the number of spaces for each tab character: this value is only used when the generated code is to be inserted in the current module (when code is pasted from the clipboard it is not necessary to convert tabs into spaces).

ExitLabelTemplate is the template string used to build the name of the label that marks the exit point of the procedure; as usual the wizard will substitute the "%P" sequence with the name of the procedure being generated.

RecursionVariableTemplate is the template string used to build the name of the Static variable used to prevent recursive calls to the procedure. Even if the wizard correctly recognizes the "%P" sequence, you can use a constant name, because multiple procedures can declare local variables with the same name.

```
[Properties]
; template for the name of Private property (%P=procedure name)
PrivatePropTemplate=m_%P
; default state of option for creating enumerated constants
AddEnumConst=1
; default first value for enumerated properties
EnumFirstValue=1
; template for symbolic constants for enumerated properties
; (%P=procedure name, %V=string value, %N=auto-increment counter)
EnumConstTemplate=%P_%V
```

PrivatePropTemplate is the template used to build the default name of the Private member variable used by wrapper procedures. As usual, the "%P" sequence will be substituted with the name of the procedure

being generated. You can modify this template by editing the INI file, or you can type a new value in this field and then click on the Save Settings button to let the wizard evaluate the template itself.

AddEnumConst is the default state for the "Use Symbolic Constants" checkbox that you find in step 10 (enumerated properties). You can modify this value by editing the INI file, or you can mark/unmark the checkbox as desired and then click the Save Settings button.

EnumFirstValue is the default starting value for enumerated constants. You can alter this default by typing a new value in the "First Value is" textbox (step 11), or by directly editing of the INI file.

EnumConstTemplate is the template used to build the names of all symbolic constant related to enumerated properties. You can use three special sequences in this template: "%P" will be replaced by the name of the procedure, "%N" is an auto-incrementing counter, and "%V" is a contracted form of the value itself (all symbols are removed, and blanks are replaced by underscores). You can modify this value only by editing the INI file.

Suppose we have an enumerated "Color" property that accepts the three values "Black", "White" and "Bright White" values. Using a first value of 1 and EnumConstTemplate equal to "%P_%V" the wizard will generate the following symbolic constants

```
Const Color_Black = 1
Const Color_White = 2
Const Color_Bright_White = 3
```

If we assign EnumConstTemplate the string "Proc%P_%N" and a starting value of zero, the wizard will generate these values instead

```
Const PropColor_0 = 0
Const PropColor_1 = 1
Const PropColor_2 = 2
```

The last block in the INI file is devoted to all the settings that influence remarks

```
[Comments]
; encrypted user name - DO NOT MODIFY NEXT LINE !!!
EncryptedLicensedUser=KNece::>CJK|n~xpq.àlkfdrdq`k%Dfdldj
; add remark banner (0=none, 1=add before, 2=add after)
RemarkBanner=2
; max length for remarks - used for wrap-around
RemarkMaxLength=60
; default state for comment options
AddParamDescription=1
; remark banner (%A=author name, %D=date, %T=time,
; %R=user-defined remark, %L=argument list, %N=newline)
Rem1=************************************************************
Rem2= %R
Rem3=
Rem4= Author:     %A
Rem5= Created on: %D  (time %T)
Rem6=
Rem7= Parameters%N %L %N Return Value:
Rem8=************************************************************
```

EncryptedLicensedUser is the crippled name of the user name that was provided at registration time. IT IS VERY IMPORTANT THAT YOU DO NOT MODIFY THIS ITEM. If you do it, you will ruin your registered copy of The Routine Master, and will be left with the shareware version.

For this reason, IT IS STRONGLY SUGGESTED THAT YOU MAKE A BACKUP COPY OF THIS INI FILE IMMEDIATELY AFTER REGISTRATION IS COMPLETE.

RemarkBanner is the default option for remark banners. You can use 0 for no banner, 1 to insert a remark banner before the first line of the procedure, and 2 to insert a remark banner immediately after the first line of the procedure, but before the declaration of local variables and the other executable lines.

RemarkMaxLength is the maximum length of remarks; this value is used to wrap around long comments that you typed in the multilined textbox in step 5.

AddParamDescription is the default state of the "Parameter Description" checkbox. You can modify this setting on the user interface and then clicking the Save Setting button, or directly editing the INI file.

Rem lines states which commented lines will be generated when a remark banner has been selected. You can include up to 20 distinct lines, from Rem1 to Rem20. In each line you can use one of the following special codes to let The Routine Master automatically insert variable information:

- **%P** is replaced by the name of the procedure
- **%R** is replaced by the text you entered as a remark to the procedure (correctly word-wrapped)
- **%A** is replaced by the name of the author
- **%D** is replaced by creation date
- **%T** is replaced by creation time (in hh:mm format)
- **%L** is replaced by the list of paramters
- **%N** is replaced by a newline caracter

You can modify the default remark banner, editing or removing existing lines, or adding new ones (up to 20 lines in total). You only have to check that lines are numbered sequentially, otherwise the wizard will create empty lines in between.

The only special code that required an additional explanation is %L. This line will be printed only if the routine accepts arguments, and if the user marked the "Parameter Description" checkbox. Otherwise it is ignored. Moreover, what follows the %L code is included in the banner only if the routine is a Function, and is ignored in the case of Properties and Subs.

The Rem lines showed above will produce a remark banner like the following

```
'************************************************************
' The text you entered in the textbox, word-wrapped because
' it is longer than the value RemarkMaxLength
'
' Author:     Francesco Balena
' Created on: 09/25/1996  15:03
'
' Parameters
'    text :
'    index :
' Return Value:
'************************************************************
```