TOM Application Developer's Guide

November 1999

STATIONworks Version 2.1
A FASTech MES Product

Published by Brooks Automation, Inc.
15 Elizabeth Drive / Chelmsford, Massachusetts 01248 / USA
(978) 262-2400
FAX (978) 262-2500
http://www.brooks.com OR www.fastech.com

Table of Contents

Developing Simple Application

1

Topics in this chapter

This chapter presents fundaments of developing a simple TOM application to replace of TOM Explorer. The code for the application is included in the Service's Developer's Kit (SDK) in a project named *myrecipe.vbp*. You can find this project in *FASTech/TOM/Samples/apps/myrecipe*.

NOTE You must use the Professional or Enterprise Edition of Visual Basic Version 5.00 when developing TOM applications.

What Is a TOM Application?

A TOM application differs from a STATIONworks application. While STATIONworks applications run the equipment by creating an equipment manager in state machine form, a TOM application interacts with the equipment directly through the Tool Object Model, just the way TOM Explorer does. The difference between the two is the scope.

STATIONworks state machines can interact with the FACTORYworks MES and then turn around and interact with the equipment, all from the same machine.

TOM applicaitons interact with the equipment. You can write such an application to replace TOM Explorer as a graphical interface for an operator.

Writing a TOM Application in Visual Basic

To write an application, after you start up Visual Basic, carry out the tasks in each of the sections that follow, outlined below:

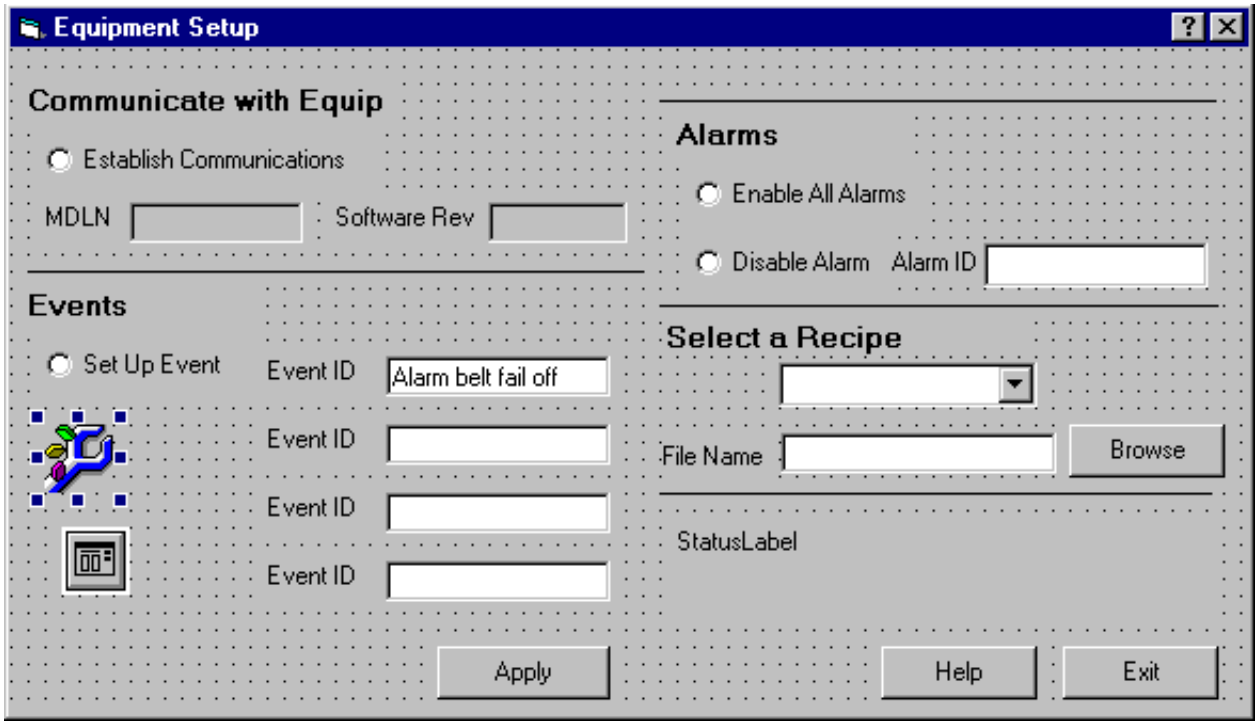
- Define the application requirements.
- Draw any Visual Basic forms required.
- Add the TOM control (tomctrl) to the Visual Basic toolbox.
- Create a reference to the Tool Object Model.
- Put the TOM control in the main Visual Basic form.
- Declare constants.
- Declare object references.
- Retrieve/set Service Attributes.
- Execute Service Methods.
- Handle Method completion notifications.
- Handle status notifications.
- Handle Event notifications.
- Compile the application.
- Test the application.

Defining the Application

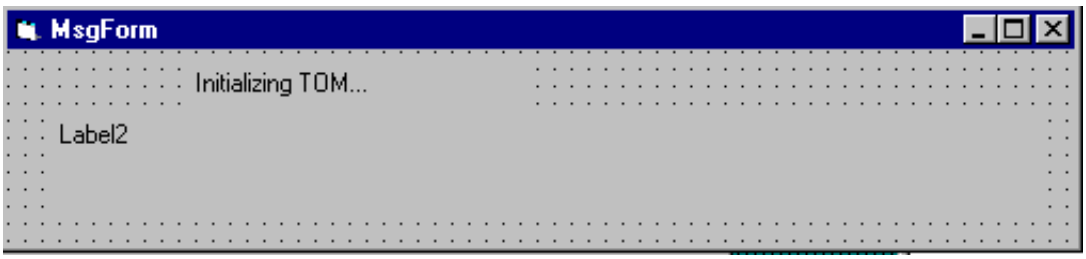
Let's suppose you want to develop a simple application to:

- Initiate communication with the equipment
- Enable all alarms on the tool
- Disable a particular alarm on the tool
- Set up particular events on the tool
- Let you select a recipe and download that recipe to the tool

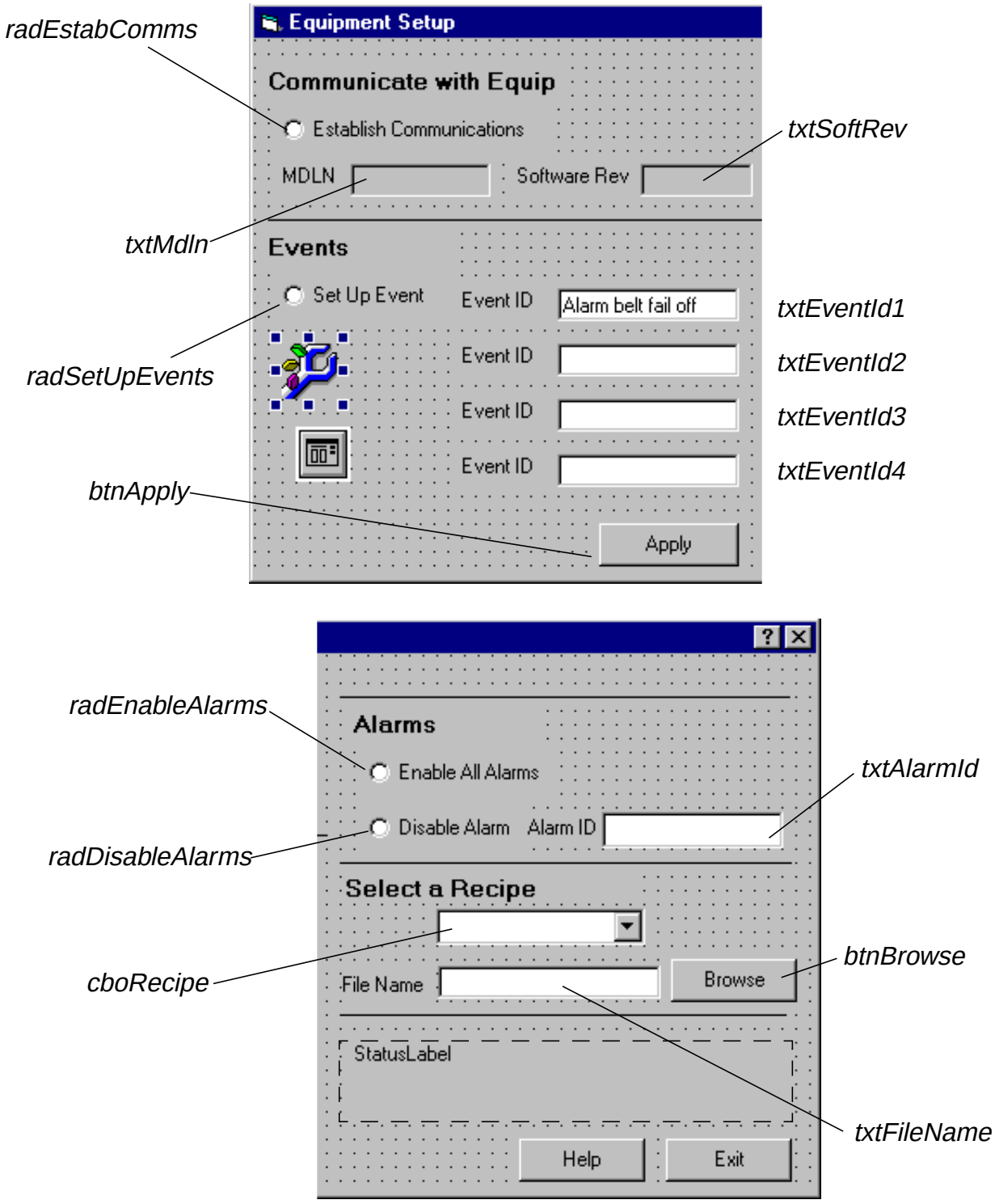
You need a form where the operator can take all of these actions. You display this form (shown below) initially when the application runs.



You also need another form to display messages while the application sets itself up. The MsgForm should be like the one that follows:



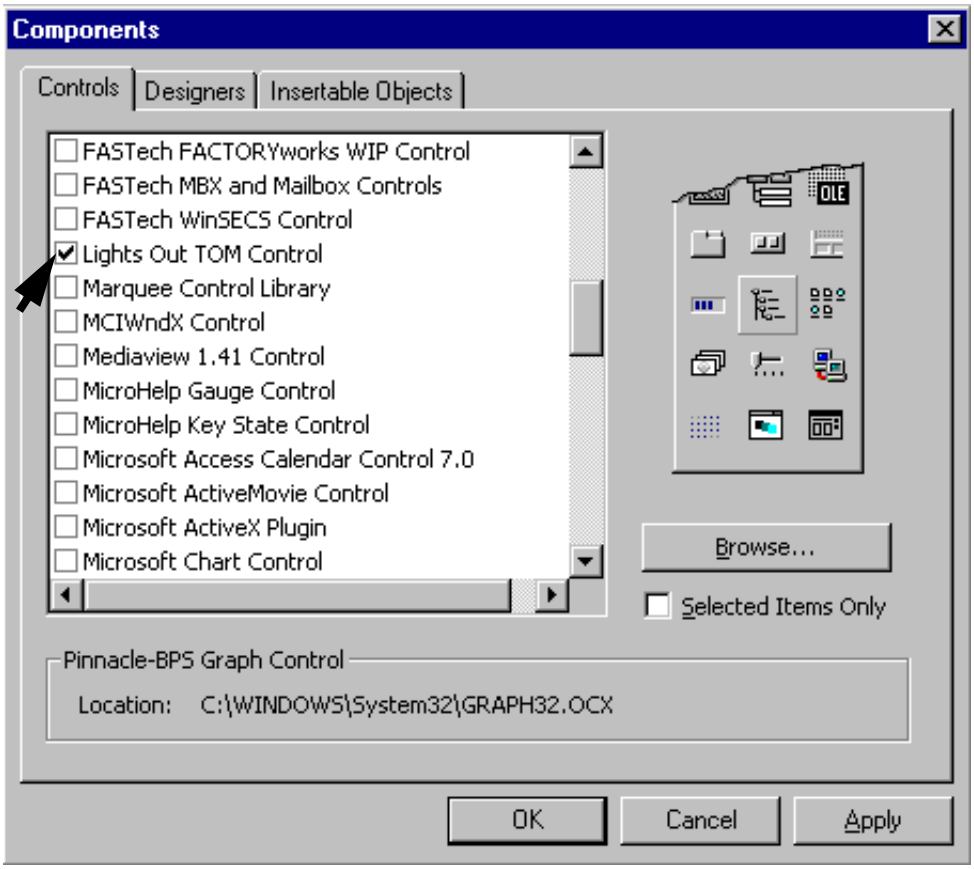
Some names for the controls on the main form are in the next illustration.


radEstabComms
Equipment Setup
Communicate with Equip
Establish Communications
txtSoftRev
MDLN
Software Rev
txtMdln
Events
Set Up Event
Event ID
Alarm belt fail off
txtEventId1
radSetUpEvents
Event ID
txtEventId2
Event ID
txtEventId3
Event ID
txtEventId4
btnApply
Apply
radEnableAlarms
Alarms
Enable All Alarms
txtAlarmId
Disable Alarm
Alarm ID
radDisableAlarms
Select a Recipe
cboRecipe
btnBrowse
File Name
Browse
StatusLabel
txtFileName
Help
Exit

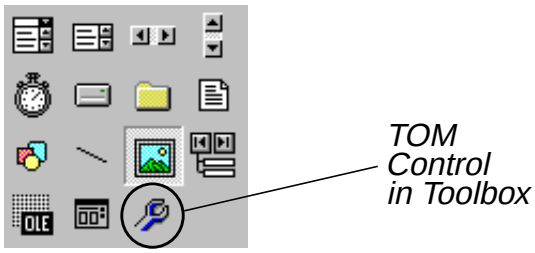
Adding TOM Control to Visual Basic Toolbox

To add the TOM control (tomctrl) to the Visual basic Controls Toolbox:

1. Select `Project` => `Components` and click the `Controls` tab.
2. Select the `Lights Out TOM Control` from the list by clicking on its check box.



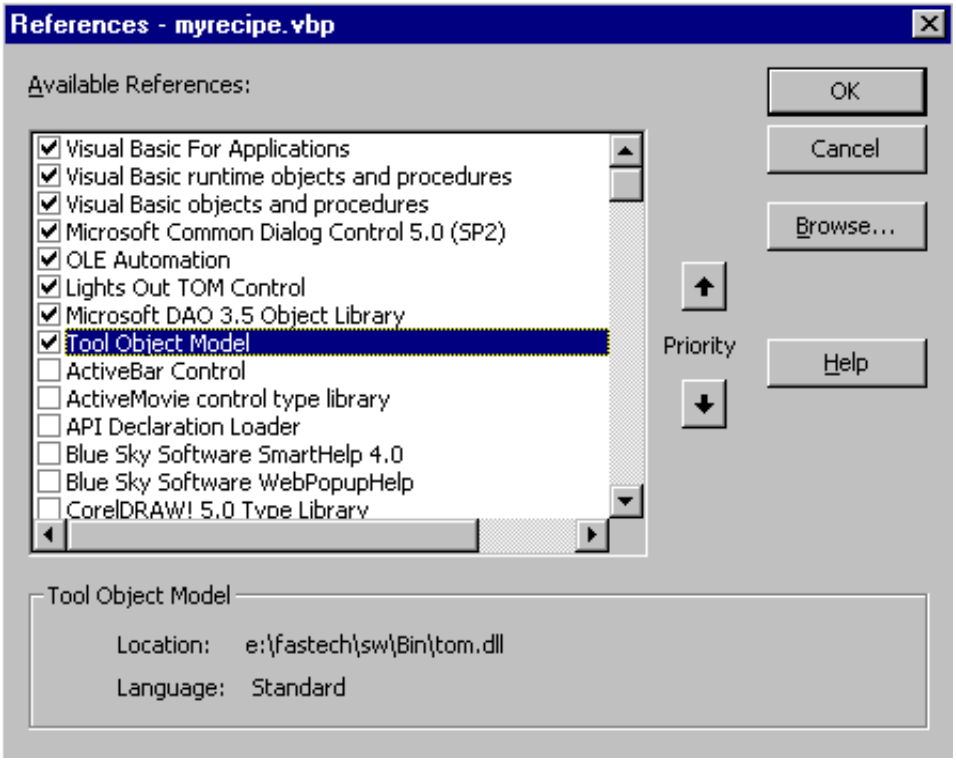
3. If you do not see the control in the list, click on `Browse` and find the control under the *system32* directory.
4. The TOM control appears in the Visual Basic Toolbox as a wrench.



Creating Reference to the Tool Object Model

To add a reference to the Tool Object Model as follows:

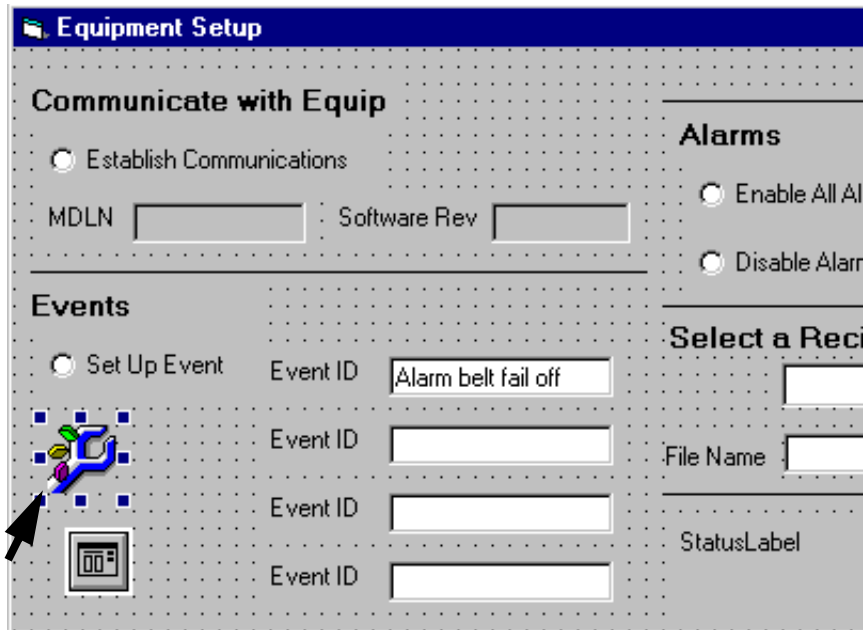
1. Select `Project` => `References` from the menu bar.
2. In the `References` dialog, select `Tool Object Model` by clicking on its check box.



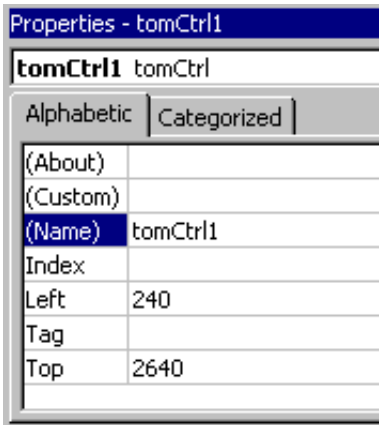
If you scroll further down in the list, you see other objects related to TOM, such as TOM Standard Services and TOM Standard Level 2 SECS Services. You do not have to select these objects; TOM already contains information about them.

Putting a TOM Control in the Application

Once you develop a Visual Basic form for the application, as long as you have added the TOM control to the custom controls and have a reference to the Tool Object Model, you can add the TOM control to your application. Even if you do not need a GUI for your application, you should have a form where you can place the TOM control (tomctrl), even if it is the only control in the form. To add the TOM control, select theTOM icon (wrench) in the Visual Basic toolbox (see the illustration to the left) and draw a rectangle for it in the form (shown below).



If you look at the properties for the TOM control, you see it is named `tomctrl1`. Later, you use this name (or whatever you change it to) in your Visual Basic code.



Selecting a Tool from the Database

In your TOM application, you start by selecting the Tool you want to work with. If the Tool exists in the TOM database, it already has particular Services associated with it so you can use those standard Services. If you need to make change in a tool (such as adding a custom Service to it), you can use TOM Builder or TOM DB Editor (see online Help files).

NOTE To run the sample application, use the Tool under */FASTech/TOM/Samples/apps/MyRecipe/Drivers*. In this location, you find the standard database directories and they contain the *.tbf* files required to build the sample tool's database. The tool is called *BTU recipe example*.

In the `General Declarations` section of the code, you should declare private constants for the following:

- The TOM Tool you are using in the application.
- The TOM database the application should use.
- The Help file and Help context number (see *Tying in a Help File*, p. 1-9).
- The Services, both standard and custom, that your application uses (see *Selecting Standard Services*, p. 1-10.)

Declare the TOM Tool Constant

For this example, you use the BTU recipe example as it exists in the alternative Tool database (see note above). To work with the Tool in your application, you declare a private constant for it:

```
' This is the TOM tool used in the TOM Application
Private Const TOOL_NAME = "BTU recipe example"
```

You can name the constant TOOL_NAME for convenience, but should set it to the exact name of the tool, as it appears in the database, in quotation marks.

In this case, the Tool is a custom one. In your code, you can select any Tool already in the database or any you have added.

Declare the Database Constant

For the application to be able to find the database that your Tool and its Services are stored in, you must declare a constant for the database:

```
Private Const DATABASE_NAME = "myrecipe.mdb"
```

The database always has a name of up to eight characters with an MDB extension. Be sure to put the name of the database in quotation marks.

Once you have a Tool and database, you can later have TOM instantiate the Tool.

Next, you tie a Help file in to your application.

Tying in a Help File

In the `General Declarations` section of the code, you need to declare private constants for the Help file and Help context number for the application.

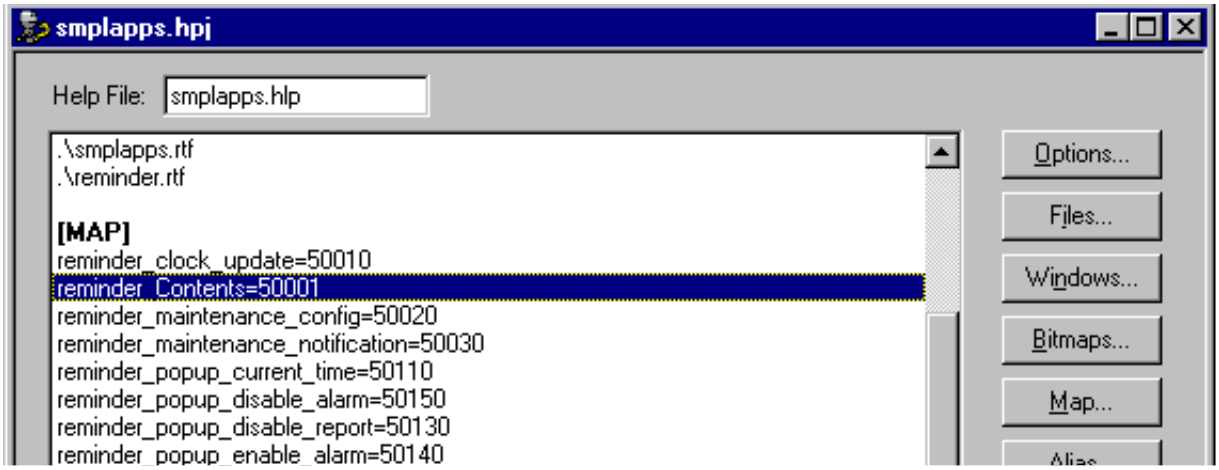
Declare Help File Constants

If you have created a Help file for your Services or your Tool, for your application to use it, you must declare a constant for it and for the help context ID (there is no help file for the sample):

```
' Where you can find help file info about this application
Private Const APP_HELPFILE = "myrecipe.hlp"
Private Const APP_HELP_CONTEXT = 50001
```

The Help file name must always end in .HLP. If you do not have a Help file yet, you can leave out these declarations and add them later.

The APP_HELP_CONTEXT value is the one associated with the contents of the Help file. You want the contents to appear when the operator presses the Help button. You associated the context ID with the Help file contents for you application the same way that it is done in the sample application. Below you can see where the 50001 context ID is associated with the contents in the HPJ file for the Help. (The illustration shows a file open in the Help Workshop, available through Microsoft.)



Next, you select the Services you want to use in your application.

Selecting Standard Services

To use standard Services in your TOM application, you start by determining which Services are available to the Tool you have selected; then choose the Services that carry out the actions you want to take. You can browse the details of the Tool's Services most easily in TOM Explorer.

Get to know the Services themselves. What kinds of Methods, Events, and Properties do they have? What kinds of Inputs does each Service need? What kinds of Outputs does it produce?

Declare Service Constants

In this application, you use a series of standard TOM Services that carry out the tasks defined earlier. You create constants for those Services in the `General Declarations` section of your project:

```
Private Const SRV_GEMPROCESS = "GemProcessPrograms"
Private Const SRV_VFEIRESXFER = "VFEIResourceTransfer"
Private Const SRV_GEMALARMMGMT = "GemAlarmManagement"
Private Const SRV_GEMREPORTS = "GemReports"
Private Const SRV_GEMESTABCOMMS = "GemEstablishCommunications"
Private Const SRV_PROTOCOLSECS = "ProtocolSECS"
```

After you declare the constants for the Services, when you declare object references (next), you declare the Service objects that correspond to these constants.

Declaring References

For all TOM applications you must declare references to TOM objects you intend to use. You create these references in the *General Declarations* section of the code. Major objects you need references to include:

- Tool Object Model object
- Tool object (to later represent the instantiation of the tool)
- Service objects
- Any controls inside forms that the application interacts with

Declare Tool Object Model Reference

While the application is operating and Tools are in use, you must always keep a reference to the Tool Object Model object. If this reference were to go out of scope, the Tools would not be able to work with TOM and the application would crash. So, you must declare this reference as a global reference and must not set it to `Nothing` until you intend to exit the application.

This single declaration is required before you can work with any TOM Tools:

```
Private m_oTOM As tom.ToolObjectModel
```

The type of a Tool Object Model is tom.ToolObjectModel. Notice that this object is different from the Tool itself, which you declare next.

Declare Tool Reference for Tool Instantiation

Once you later instantiate the Tool, you need to keep that instantiation in a Tool object, which you declare as follows:

```
Private m_oTool As tom.Tool
```

The type of this object is tom.Tool.

If this object were to go out of scope, the application would crash, so you must declare this reference as a global reference. Do not set it to `Nothing` until you intend to stop using the Tool.

An alternative is to reference the Tool through the Tool Object Model object; however, declaring a reference for the instantiation makes your code easier to read.

Declare References to Service Objects

To refer to and use Services in the application, you should declare references to each of them.

```
' References to Service objects application uses
Private m_oGemProcess As tom.Service
Private m_oVFEIResourceXfer As tom.Service
Private m_oGemAlarmMgmt As tom.Service
Private m_oGemReports As tom.Service
Private m_oGemEstablishComms As tom.Service
Private m_oProtocolSecs As tom.Service
```

The type for each Service is tom.Service.

Declare Controls in Forms

Your application must include a form, even if the only control on the form is the TOM control. However, chances are you need multiple controls in the form. To have your application work with those controls, you must declare references to them. The type you need to assign them is based on the type of control.

For the sample application (this may not be required in your application), you declare a reference to a label whose message you plan to set before displaying it:

```
Private m_oCurrStatusLabel As Label
```

Generating Code to Trigger When Form Loads

After you have declared all the constants and references, you are ready to write the Visual Basic functions. Let's start with the `Form_Load` function. This function runs immediately after you load the form:

Declare Local Variables and Set Application Properties

1. Declare the local variables for the ToolTypes collection and the ToolType of the particular Tool the application later uses; also, declare a `MsgForm` variable for the messages that need to display during the initialization process:

   ```
   Dim ToolTypes As tom.ToolTypes
   Dim ToolType As tom.ToolType
   Dim MsgForm As frmMessage
   ```

   In this routine, you use the `MsgForm` you created earlier to display the messages, so you need to refer to it using a local variable here.

2. To ensure that errors are handled, you should always have an `On Error` statement early in the code. You can have an `On Error Resume Next` statement and then on the next line either deal with the error or proceed with the current action, depending on how you want to handle errors:

   ```
   On Error Resume Next
   ```

   To deal with the error, you can look at the Err object to find out whether or not an error has occurred and then proceed to an appropriate line of code that handles the error.

3. To have the application use the Help file (if you declared a constant for one), you can now set the `HelpFile` property of the `App` object to that constant, then set the `HelpContextID` of the current object to the context ID constant:

   ```
   App.HelpFile = APP_HELPFILE
   Me.HelpContextID = APP_HELP_CONTEXT
   ```

Create the TOM Core

Every application must create the TOM Core as follows:

1. Clear the Err object, so that any error that occurs is the only Err associated with that object:

   ```
   Err.Clear
   ```

2. Create the Tool Object Model object and set the reference to that object:

   ```
   Set m_oTOM = CreateObject("tom.ToolObjectModel")
   ```

3. If an error occurs when trying to create the Tool Object Model object, call a `ReportError` function (see appendix) and pass it a message to display:

```
If Err Then
   ErrorReport “while creating TOM object ” & Err.Description
Else...
```

4. If no error occurs, let the operator enter the name of a database as a command line argument when starting the application, as shown below:

   <app_name>`.exe c:\<`*mydirectory*`>\mydbase.mdb`

   If the operator does not enter a database, use the database that you set the DATABASE_NAME constant to earlier, which becomes the default. To set the database, you set the `DefinitionFile` property of the Tool Object Model object:

```
Else...
   If Command <> "" Then
      m_oTOM.DefinitionFile = Command
   Else
      m_oTOM.DefinitionFile = DATABASE_NAME
   End If
```

Initialize the TOM Core and Find the Tool in the Collection

After the application creates the TOM Core, it should then initialize TOM:

1. Start, as you did when you created the core, by clearing the Err object:

```
Err.Clear
```

2. Call the `Initialize` Method of the Tool Object Model object:

```
m_oTOM.Initialize tomCtrl1
```

3. To handle any error that occurs, display a message box:

```
If Err Then
   ReportError “while initializing TOM Core “ & Err.Description
```

4. Otherwise, if no error occurs, to find the Tool in the ToolTypes collection, first set the ToolTypes object to the ToolTypes property of the TOM object:

```
Else
   Set ToolTypes = m_oTOM.ToolTypes
```

   Then set the ToolType object to the particular Tool in the collection using the `Item` method and passing it the constant for the Tool, TOOL_NAME:

```
Set ToolType = ToolTypes.Item(TOOL_NAME)
```

Prepare to Instantiate the Tool

Before you have the TOM Core instantiate the Tool, you should prepare to show the action that is occurring while TOM instantiates the Tool. You show the action by displaying the `StatusNotification` events that the TOM

control sends to the application. You can display a status message in the `MsgForm` to show the action/status notifications. Here are the steps you take:

1. Set the `MsgForm` to the form that displays messages:

   ```
   Set MsgForm = New frmMessage
   ```

2. Use the `Show` and `Refresh` methods to display the form:

   ```
   MsgForm.Show
   MsgForm.Refresh
   ```

3. To be sure it displays the correct message, set the reference to the Label object you created earlier to the form's `LabelMsg` property value:

   ```
   Set m_oCurrStatusLabel = MsgForm.LabelMsg
   ```

Tell TOM Core to Instantiate the Tool

To have the TOM Core instantiate the Tool:

1. Start by clearing the Err object:

   ```
   Err.Clear
   ```

2. Add a new Tool instance to the Tools collection in your TOM Core. When you instantiate the Tool, you assign it a name. You can assign it any name you want, regardless of what name you assigned to the TOOL_NAME constant. In this case, the name is *BTU Tool*:

   ```
   Set m_oTool = m_oTOM.Tools.Add(ToolType, "BTU Tool")
   ```

3. If this process succeeds, you now have a Tool instance to work with. You should, of course, check to see if an error occurred and display an appropriate message if it did; in this case, you call `ReportError`:

   ```
   If Err Then
      ReportError " while creating tool " & Err.Description
   End If
   ```

4. To display the status notifications in the `MsgForm` by setting the label to the `LabelStatus` value. Later, as the status notifications occur, TOM sends them to your application by setting the `StatusLabel`, and this statement ensures the application displays them.

   To see how to ensure the application receives status notifications from TOM, refer to *Receiving Status Notifications from TOM Core*, p. 1-21.

5. After the status message displays, since you don't need it any more, you can unload the Message form:

   ```
   Set m_oCurrStatusLabel = LabelStatus
   Unload MsgForm
   ```

Find Required Service Objects

Next, the `Form_Load` routine should find the required Services in the database.

1. When any errors occur trying to find the required Services, you should always have an `On Error` statement that sends the program flow to an error handling section in the code, because failing to find a required Service means other problems could occur:

   ```
   On Error Resume Next
   ```

2. Then, to find the Services, you set the Service object references you created earlier each to a particular Service. Each particular Service is in the collection of Services associated with the one Resource the tool has:

   ```
   ' Find the Service objects you want/extract from Services
   ' collection
   Set m_oGemProcess = m_oTool.Resources.Item(1)._
   Services.Item(SRV_GEMPROCESS)
   ReportError "while finding the GemProcessPrograms Service"
   Set m_oVFEIResourceXfer = m_oTool.Resources.Item(1)._
   Services.Item(SRV_VFEIRESXFER)
   ReportError "while finding the VFEIResourceTransfer Service"
   Set m_oGemAlarmMgmt = m_oTool.Resources.Item(1)._
   Services.Item(SRV_GEMALARMMGMT)
   ReportError "while finding the GemAlarmManagement Service"
   Set m_oGemReports = m_oTool.Resources.Item(1)._
   Services.Item(SRV_GEMREPORTS)
   ReportError "while finding the GemReports Service"
   Set m_oGemEstablishComms = m_oTool.Resources.Item(1)._
   Services.Item(SRV_GEMESTABCOMMS)
   ReportError "while finding the GemEstablishCommunications
   Service"
   Set m_oProtocolSecs = m_oTool.Resources.Item(1)._
   Services.Item(SRV_PROTOCOLSECS)
   ReportError "while finding the ProtocolSECS Service"
   ```

   The `(1)` after `Item` indicates you want to retrieve the first Resource of the Tool. In this case, since there is only one Resource in the collection, you use `(1)` to refer to it. If there were more than one, you'd have to use the correct number or the literal name in quotation marks to refer to the particular Resource.

   Each time you try to retrieve a Service from the collection, ideally you should check for an error.

Restrictions in Form_Load

Your application cannot receive Events from the TOM control or any control during `Form_Load`. So, do not have your code wait for an Event in `Form_Load`.

Complete Code of Form_Load

```
' When the main form is loaded, the app must initialize
' the TOM Core. Then it can proceed to instantiate your tool.
Private Sub Form_Load()
    Dim ToolTypes As tom.ToolTypes
    Dim ToolType As tom.ToolType
    Dim MsgForm As frmMessage
    Dim fSuccessfulStartup As Boolean

    On Error Resume Next

    ‘While startup has not successfully completed, set local var
    ‘to False. Later, when form has loaded, set it to True.
    fSuccessfulStartup = False

    ' App.HelpFile = APP_HELPFILE
    ' Me.HelpContextID = APP_HELP_CONTEXT

    ' Create the TOM Core
    Err.Clear
    Set m_oTOM = CreateObject("tom.ToolObjectModel")
    If Err Then
        ReportError “ while creating TOM object “ & Err.Description
    Else
        ' User can specify an alternative database
        If Command <> "" Then
            m_oTOM.DefinitionFile = Command
        Else
            m_oTOM.DefinitionFile = DATABASE_NAME
        End If

        ' Now, Initialize the TOM Core
        Err.Clear
        m_oTOM.Initialize tomCtrl1
        If Err Then
            ReportError “ while initializing TOM Core“_
            & Err.Description
        Else
            ' Find our tool in the ToolTypes collection
            Set ToolTypes = m_oTOM.ToolTypes
            Set ToolType = ToolTypes.Item(TOOL_NAME)

            ' Show progress status dialog while TOM instantiates
            ' the tool. Then catch StatusNotification events sent
```

```
            ' by the TOM Control and display the events in this
            ' dialog.
            Set MsgForm = New frmMessage
            MsgForm.Show
            MsgForm.Refresh
            Set m_oCurrStatusLabel = MsgForm.LabelMsg

            ' Tell TOM Core to instantiate the tool
            ' Assign a unique name to this instantiation
            Err.Clear
            Set m_oTool = m_oTOM.Tools.Add(ToolType, "BTU Tool")
            If Err Then
                MsgBox "Unable to create tool: " & Err.Description,_
                vbExclamation, App.Title
            End If

            ' Now display status notifications in our main window
            Set m_oCurrStatusLabel = LabelStatus
            Unload MsgForm

            'Set up Service objects in separate routine
            lSetupServices

            'Now that form has loaded and services are ready,
            'set local var to True.
            fSuccessfulStartup = True
        End If
    If Not fSuccessfulStartup Then
        Unload Me
    End If
End Sub
```

Complete Code of lSetupService Subroutine Form_Load Calls

The example uses a separate routine that it calls from within `Form_Load` to get a reference to each Service it uses:

```
Private Sub lSetupServices()
    On Error Resume Next

' Find the Service objects you want/extract each from collection

    Set m_oGemProcess = m_oTool.Resources.Item(1)._
    Services.Item(SRV_GEMPROCESS)
    ReportError "while finding the GemProcessPrograms Service"
    Set m_oVFEIResourceXfer = m_oTool.Resources.Item(1)._
    Services.Item(SRV_VFEIRESXFER)
    ReportError "while finding the VFEIResourceTransfer Service"
    Set m_oGemAlarmMgmt = m_oTool.Resources.Item(1)._
    Services.Item(SRV_GEMALARMMGMT)
```

```
    ReportError "while finding the GemAlarmManagement Service"
    Set m_oGemReports = m_oTool.Resources.Item(1)._
    Services.Item(SRV_GEMREPORTS)
    ReportError "while finding the GemReports Service"
    Set m_oGemEstablishComms = m_oTool.Resources.Item(1)._
    Services.Item(SRV_GEMESTABCOMMS)
    ReportError "while finding the GemEstablishCommunications
    Service"
    Set m_oProtocolSecs = m_oTool.Resources.Item(1)._
    Services.Item(SRV_PROTOCOLSECS)
    ReportError "while finding the ProtocolSECS Service"

    Exit Sub
End Sub
```

Now that the form is loaded and you have a reference to each Service you'll be using, you are ready to work with Service Attributes.

Retrieving and Setting Service Attributes

In most applications you need to retrieve Attributes settings from the database or change Attribute settings in memory. You may need to set an Attribute that a Service uses. To set such Attributes:

1. Be sure you have a reference to the Service that owns the Method.
2. To set the Attribute's value, retrieve it from the Attributes collection of that Service. For example, to set the `Baud` Attribute of the *ProtocolSECS* Service, you would access it using the following structure:

   ```
   m_oProtocolSecs.Attributes.Item("Baud").Value = "9600"
   ```

   This setting is in memory only and does not affect the database.

   For details on the Attributes the sample application uses, refer to *Carrying Out Tasks on Equipment*, p. 2-1.

Executing a Service Method

In most applications you need to be able to execute a Service Method. Once you verify that the Service is available, you can execute Methods of the Service.

Clone Method

To execute a Method of a Service, you must clone it first, then execute the clone. It is important that you clone it rather than executing the original Method, because another application or a Service could be trying to execute the original Method. By cloning the Method, your applictionhas its own copy and does not interfere with another application or Service trying to clone or execute the original. If you have all applications and Services clone Methods, you eliminate potential conflicts.

Let's look at an example. To execute the `Connect` Method of the *GemEstablishCommunications* Service, you first clone the Method. You use the reference to the Service and access its Methods collection (`m_oGemEstablishComms.Methods`). You then use the `Item` method of a Methods collection to select the `Connect` Method of the Service. You can then clone `Connect` using the `Clone` method of a Method object:

```
Dim clonedMeth As tom.Method
Set clonedMeth = m_oGemEstablishComms.Methods.Item("Connect")_
.Clone
```

Execute Method

Once you have the clone, you execute the clone of the Method using the `Execute` method of a Method object:

```
clonedMeth.Execute
ReportError "while opening the SECSProtocol Communication Port"
```

After you call the `Execute` method of a Method object, the TOM Core has a reference to the object only as long as the routine is executing. After the routine finishes executing, TOM Core no longer retains the reference—it goes away.

In addition, whenever this Method or any TOM Method executes, TOM sends control of the program to the `MethodNotification` routine for the TOM control (tomCtrl) object you embedded in the application form. In this case, that TOM control was called `tomctrl1`, so the routine would be `tomctrl1_MethodNotification` routine.

It is in this routine that you retrieve the Outputs the Method has produced.

Receiving Method Completion Notifications from TOM Core

When a Method executes, the application needs to know when the Method completes. But how can your application know when a Method is complete? You set it up to receive Method completion notifications from TOM. You do that by creating a routine called `tomCtrl1_MethodNotification`. The routine receives a TOM Method as an argument:

```
Private Sub tomCtrl1_MethodNotification(ByVal tomMethod As
Object)
```

Identify the Method Completing

In the `MethodNotification` routine, you need to determine which Method in the application is completing and carry out the final tasks for that Method.

A standard way of setting up this routine is to use a `Case` statement and enter the particular code required to complete each Method:

```
Private Sub tomCtrl1_MethodNotification(ByVal tomMethod As
Object)
    Dim counter
    Select Case tomMethod.Name
        GemAlarmManagement Enable All method
        Case "Enable all"
            ...
        'GemAlarmManagement Disable method
        Case "Disable"
            ...
        'GemEstablishCommunications
        Case "Connect"
            ...
    End Select
```

If you have more than one Service's Method with the same name to contend with, you can determine the Service that owns the Method by using:

```
strService = tomMethod.Service.Name
```

Retrieve an Output from a Method

Usually you want to use the Outputs of a Method you have executed. You retrieve the Outputs here in the `MethodNotification` routine. For instance, suppose you want to retrieve the `MDLN` and `SOFTREV` Outputs from the `Connect` Method of *GemEstablishCommunications*. You could set corresponding fields in a GUI to the value of each as follows:

```
Case "Connect"
    txtMdln.Text = tomMethod.Outputs.Item("MDLN").Value
    txtMdln.Refresh
    txtSoftRev.Text = tomMethod.Outputs.Item("SOFTREV").Value
```

```
    txtSoftRev.Refresh
    ...
```

For more on working with Outputs and setting Inputs of Methods, refer to *Selecting and Downloading a Recipe*, p. 2-14.

Receiving Status Notifications from TOM Core

In addition to sending Method notifications when a Method executes, TOM Core also sends status notifications. You may want to have your application receive those notifications and display them in your form.

To ensure your application receives status notifications as they occur, you create a routine named `tomCtrl1_StatusNotification` that takes a text string containing a notification as an argument:

```
Private Sub tomCtrl1_StatusNotification(ByVal StatusText As
String)
```

TOM Core sends the status text string to this routine.

Check for Notification

To determine whether or not there is a need to display a status notification, check the value of the `m_oCurrStatusLabel` reference that you created early in the application. If its value is `Nothing`, it does not contain a status, so you should exit the routine:

```
If m_oCurrStatusLabel Is Nothing Then Exit Sub
```

Display Notification

Otherwise, you should set the reference to the string passed to the routine and refresh the display:

```
m_oCurrStatusLabel = StatusText
m_oCurrStatusLabel.Refresh
```

In this example, the `Form_Load` routine takes care of actually displaying the label.

Complete Code for tomCtrl1_StatusNotification

```
Private Sub tomCtrl1_StatusNotification(ByVal StatusText As
String)
    If m_oCurrStatusLabel Is Nothing Then Exit Sub
    m_oCurrStatusLabel = StatusText
    m_oCurrStatusLabel.Refresh
End Sub
```

Receiving Event Notifications from TOM Core

You may want to have your application receive Event notifications when a TOM Event occurs in a Service it is using.

To have your application "catch" Events, you must have it use the `EventNotification` event of the TOMctrl object.

NOTE **Terminology—Collection Events vs. Events**

Collection events occur on the equipment. Another type of Event is a TOM object event. `EventNotifications` are TOM object Events, usually referred to as simply *Events*.

Prepare for Event Notifications

The `tomCtrl1_EventNotification` routine accepts a tomEvent as an argument:

```
Private Sub tomCtrl1_EventNotification(ByVal tomEvent_
As Object)
...
End Sub
```

Identify the Event

When the Event in `tomEvent` matches an Event you are interested in, your code can take specific action. Be sure to include a case for every Event you are interested in responding to:

```
Select Case tomEvent.Name
   Case "Alarm set"
      ...
   Case "Alarm clear"
      ...
End Select
```

Take Specific Action for Each Event

For example, if the *GemAlarmManagement* Service receives either an `Alarm set` or `an Alarm clear` Event, your Service can display Output DataItems from the Event in a GUI by retrieving the value of one of the Event's Output DataItems, then putting it in the reference to the Event notification display area and refresh the display:

```
m_oCurrStatusLabel = tomEvent.Outputs.Item("ALTX").Value &_
" Alarm set"

m_oCurrStatusLabel.Refresh
```

You could take similar action for any other Events that occur.

Full Code of tomCtrl1_EventNotification Routine

```
Private Sub tomCtrl1_EventNotification(ByVal tomEvent As Object)
   Select Case tomEvent.Name
      Case "Alarm set"
         If m_oCurrStatusLabel Is Nothing Then Exit Sub
         m_oCurrStatusLabel = tomEvent.Outputs.
         Item("ALTX").Value & " Alarm set"
         m_oCurrStatusLabel.Refresh
      Case "Alarm clear"
         If m_oCurrStatusLabel Is Nothing Then Exit Sub
         m_oCurrStatusLabel = tomEvent.Outputs.Item_
         ("ALTX").Value & " Alarm cleared"
         m_oCurrStatusLabel.Refresh
   End Select
End Sub
```

Creating Code for Help Button

View the `ButtonHelp_Click` function and fill in the following code to associate the Help file with the Help button:

```
Private Sub ButtonHelp_Click()
  HelpByContext Me.hwnd, APP_HELPFILE, APP_HELP_CONTEXT
End Sub
```

Because you used constants in this situation, if the Help file or Help context numbers change, you can set the constants for them under `General Declarations` and leave this piece of the code intact.

Unloading the Application Form

View the `Form_Unload` routine and add text like the following to it to ensure that the application's objects go away before you exit the application:

```
Private Sub Form_Unload(Cancel As Integer)
   Set m_oTool = Nothing
   Set m_oTOM = Nothing
   Set m_oCurrStatusLabel = Nothing
End Sub
```

Be sure to set the Tool object reference, TOM object reference, and `Label` reference to `Nothing`.

Create Code for Exit Button

To be sure that the exit button also brings the application down, you should have the following routine to respond to that button being pressed:

```
Private Sub ButtonExit_Click()
  Unload Me
End Sub
```

Compiling the Application in Visual Basic Project

Your application can be a standard .EXE server, an in-process OLE server (DLL), or an out-of-process OLE server (EXE).

NOTE

TOM Tip—Synchronous Blocking Operations and Modal Dialog Boxes

Do not use a synchronous blocking operation inside your Visual Basic code—such an operation stops all action in TOM.

The most common type of file you compile your application into is a standard .EXE file. To create a standard .EXE file, go to the menu bar and select `File` => `Make EXE File`. Visual Basic creates the .EXE file in the project directory.

Try running the .EXE file to see it in action.

For an explanation of how the sample application carries out the equipment tasks, proceed to the next chapter, which covers the remaining details of the sample code.

Carrying Out Tasks on Equipment

2

Topics in This Chapter

This chapter presents how to carry out some common tasks within an application using standard Services available in TOM. The techniques it illustrates include passing data from one Method to another within a TOM application. This chapter assumes you have read Chapter 1.

The code for the application is included in the Service's Developer's Kit (SDK) in a project named *myrecipe.vbp*. You can find this project under */FASTech/TOM/apps/myrecipe*.

NOTE You must work with the Professional or Enterprise Edition of Visual Basic Version 4.00 when developing TOM Services or applications.

The sample uses a graphical user interface (GUI) for convenience. Your application may talk directly to an MES system or to other applications rather than working with a GUI.

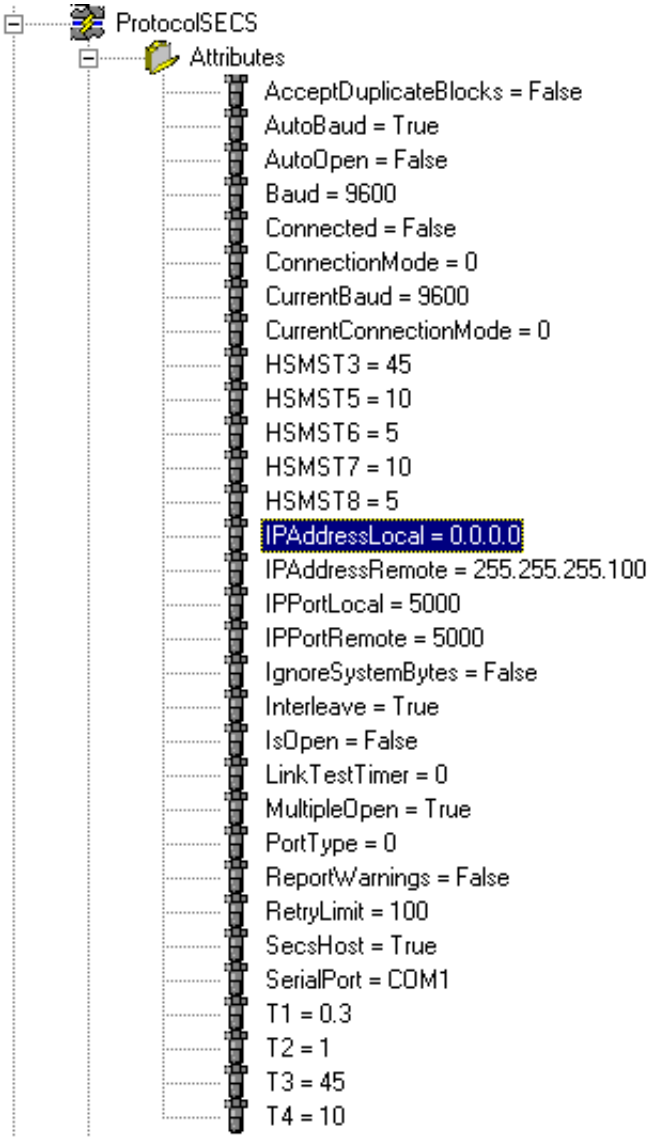
Establishing Communication with Equipment

To communicate with the tool, you always need to work with the *ProtocolSECS* Service. *ProtocolSECS* is a level 1 Service that communicates directly with the equipment.

Understand Required Services

Before you can establish communication with the equipment, you need to set Attributes of the *ProtocolSECS* Service so that it can communicate with the equipment.

To see what the Attributes of this Service are, you can expand the Attributes collection under *ProtocolSECS* in TOM Explorer.



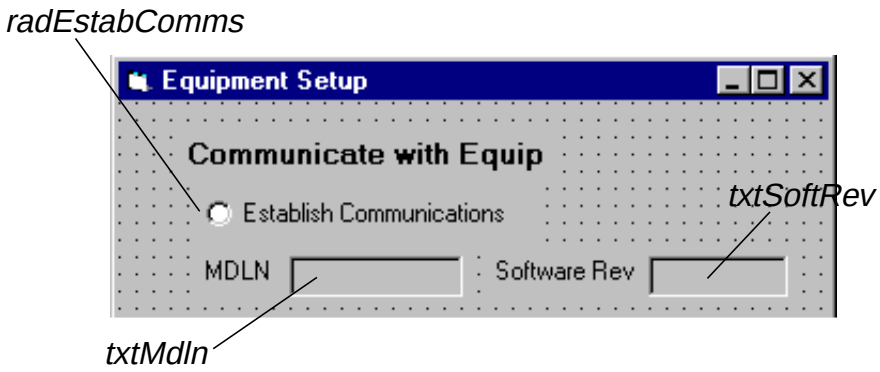
Retrieve Level 1 Service from Database

Depending on the the physical setup of the Tool, you may need to change the default settings of these Attributes. First, you must have declared the Service's constant, generated a reference to it, and retrieved the Service from the database, all in the `Form_Load` routine:

```
Private Const SRV_PROTOCOLSECS = "ProtocolSECS"
Private m_oProtocolSecs As tom.Service
Set m_oProtocolSecs = m_oTool.Resources.Item(1)._
Services.Item(SRV_PROTOCOLSECS)
```

Create Routine for Communicating with Equipment

To have the application establish comunication with the Tool when the operator clicks the `Establish Communications` radio button (see below)



you need to create a routine associated with that button:

```
Private Sub radEstabComms_Click()
```

In your facility you most likely do not need GUI, but you could have a similar routine that the MES triggers.

Set Attributes of Level 1 Service

Now, in the `radEstabComms_Click` routine you can set the Attribute values by referring to the Attributes collection of the Service and using the `Item` Method of an Attribute object.

Some Attributes you are required to set:

1. Set the `Baud` Attribute as follows:

   ```
   m_oProtocolSecs.Attributes.Item("Baud").Value = "9600"
   ```

2. You must always set the `PortType` to the appropriate number, depending on your Tool's physical setup:

   - `0`—SECS1 connection using RS-232
   - `1`—HSMS connection
   - `2`—SECS1 connection to a terminal server using TCP/IP protocol
   - `3`—SECS1 connection to a terminal server using TELNET protocol

Let's set the Attribute to the default setting of 0 to run the example:

```
m_oProtocolSecs.Attributes.Item("PortType").Value = "0"
```

3. If you are using an RS-232 connection to the Tool, you must set the `SerialPort` Attribute to the correct comm port (here you see the default setting):

```
m_oProtocolSecs.Attributes.Item("SerialPort").Value = "COM1"
```

4. If the tool is connected over a LAN (using HSMS) rather than over an RS-232 connection, you must set the `IPAddressLocal`, `IPAddressRemote`, `IPPortLocal`, and `IPPortRemote`(here you see the default settings):

```
m_oProtocolSecs.Attributes.Item("IPAddressLocal").Value =_
"0.0.0.0"

m_oProtocolSecs.Attributes.Item("IPAddressRemote").Value =_
"255.255.255.100"

m_oProtocolSecs.Attributes.Item("IPPortLocal").Value =_
"5000"

m_oProtocolSecs.Attributes.Item("IPPortRemote").Value =_
"5000"
```

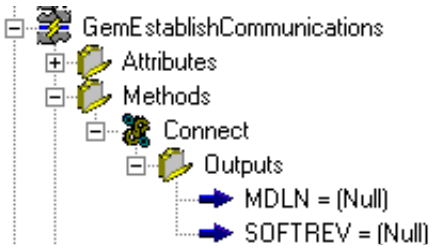
5. For a terminal server, you need to set only the `IPAddressRemote` and `IPPortRemote`.

You may want to set other Attributes, depending on your Tool setup, but these are the mandatory settings.

Retrieve the Communication Service

Now you are ready to establish communication by using a higher level Service that calls the *ProtocolSECS* Service—*GemEstablishCommunications*.

To be able to actually establish the communication, you execute the Service's `Connect` Method. When you look at this Service in TOM Explorer, notice that it returns two Outputs after it executes, `MDLN` and `SOFTREV`:



You can retrieve the values from these Outputs and put them into the application's GUI, in the text boxes `txtMdln` and `txtSoftRev` shown in the next illustration:

You carry out the steps of creating the private constant, creating a reference to the Service, and retrieving the Service from the database all in the `Form_Load` procedure:

```
Private Const SRV_GEMESTABCOMMS = "GemEstablishCommunications"
Private m_oGemEstablishComms As tom.Service
Set m_oGemEstablishComms = m_oTool.Resources.Item(1)._
Services.Item(SRV_GEMESTABCOMMS)
```

Clone and Execute the Connect Method

Now, as part of the `radEstabComms_Click()` routine, you can clone the `Connect` Method. First, you use the reference to the Service and access its Methods collection (`m_oGemEstablishComms.Methods`). You then use the `Item` method of a Method object (`m_oGemEstablishComms.Methods.Item`) to select the `Connect` Method of the Service. Once you have accessed the Method this way, you clone `Connect` using the `Clone` method of a Method object (see *Cloning Method Objects* in the TOM Help file):

```
Dim clonedMeth As tom.Method
Set clonedMeth = m_oGemEstablishComms.Methods.Item_
("Connect").Clone
```

Once you have the clone, you execute the clone of the Method using the `Execute` method of a Method object:

```
clonedMeth.Execute
ReportError "while opening the SECSProtocol Communication Port"
```

When this Method executes, TOM sends control of the program to the `MethodNotification` routine for the TOM control (tomctrl) object you embedded in the application form.

Let's proceed to see how you use the `MethodNotification` routine to retrieve the Outputs the Method has produced.

Finish Establish Communications in the MethodNotification Routine

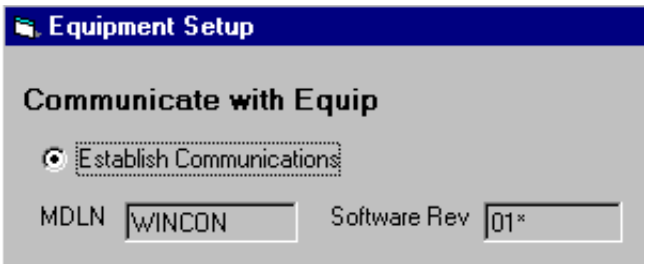
The `tomctrl1_MethodNotification()` routine must contain any action that remains for any routine in the application after that routine executes a Method. In the case of the `radEstabComms_Click()` routine, after the `Connect` Method executes, you need to be able to set the `txtMdln` and `txtSoftRev` values in the GUI.

Inside the routine, you need a `Case` statement to check for each case of the Method name, then take the appropriate action. For instance, if the Method just executed is `Connect` and its Service is *GemEstablishCommunications*, then you must set the values for the two Outputs displayed in the GUI:

```
Select Case tomMethod.Name
    'GemEstablishCommunications
    Case "Connect"
    txtMdln.Text = tomMethod.Outputs.Item("MDLN").Value
    txtMdln.Refresh
    txtSoftRev.Text = tomMethod.Outputs.Item("SOFTREV").Value
    txtSoftRev.Refresh
    ...
End Select
```

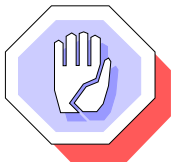
See the Results

Now, when the operator establishes communication with the equipment, the MDLN and revision of the software on the equipment display:



In addition, when the application receives the status notification TOM sends, it displays the status message shown below:

Complete Code of radEstabComms_Click



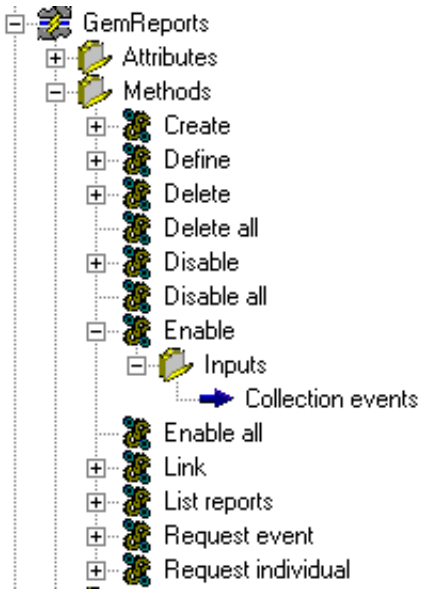
CAUTION

This listing reflects the latest info on attribute settings for ProtocolSECS. The data here supersedes that shown in the sample code included with the product.

```
Private Sub radEstabComms_Click()
   Dim clonedMeth As tom.Method
'Set attributes of the ProtocolSECS service.
'The GemEstablishCommunications service then uses runs the
'ProtocolSECS service when it communicates with the tool.
'For your tool, you may need to set additional attributes
'of this service.
   m_oProtocolSecs.Attributes.Item("Baud").Value = "9600"
'For an HSMS connetion, set the following attributes:
   m_oProtocolSecs.Attributes.Item("IPAddressLocal")._
   Value = "0.0.0.0"

   m_oProtocolSecs.Attributes.Item("IPAddressRemote")._
   Value = "255.255.255.100"

   m_oProtocolSecs.Attributes.Item("IPPortLocal").Value_
   = "5000"
   m_oProtocolSecs.Attributes.Item("IPPortRemote").Value_
   = "5000"
'You would set only IPRemoteAddress and IPPortRemote attributes
'for a terminal server
'You set the PortType to HSMS by setting it to 1.
'Since you need to be able to test this sample without a tool,
'the PortType is being set to the default of 0 for an RS-232
'connection. For RS-232, you also needs to set the SerialPort.

   m_oProtocolSecs.Attributes.Item("PortType").Value = "0"
   m_oProtocolSecs.Attributes.Item(“SerialPort”).Value = “COM1”
'Establish communication with the tool.
'Use the reference to the GemEstablishCommunications service.
   Set clonedMeth = m_oGemEstablishComms.Methods.Item_
   ("Connect").Clone
   clonedMeth.Execute
   ReportError "while opening the SECSProtocol Serial Port"
End Sub
```

Setting Up Collection Events

To set up events on the equipment, let's use the *GemReports* Service. If you look in TOM Explorer, you can see that this Service has an `Enable` Method that requires one or more collection event IDs.



First, you must have declared the Service's constant, generated a reference to it, and retrieved the Service from the database, all in the `Form_Load` routine:

```
Private Const SRV_GEMPROCESS = "GemProcessPrograms"
Private m_oGemReports As tom.Service
Set m_oGemReports = m_oTool.Resources.Item(1)._
Services.Item(SRV_GEMREPORTS)
```

Respond to Click of Apply Button

You can have the code retrieve the collection event IDs when the operator presses the `Apply` button by putting the appropriate code into the `btnApply_Click()` routine.

In this routine, the local variables for the collection events and the clone of the Method are:

```
Dim objDataItemCollectionEvents As DataItem
Dim clonedMeth As tom.Method
```

Test to see if the operator has toggled the `Set Up Events` radio button to on, and as long as it is not on, exit the routine:

```
If radSetUpEvents.Value = 0 Then Exit Sub
```

Clone the Method

Once the `Set Up Events` radio button has been toggled on, to work with the Service's collection event ID DataItems you need to first clone the `Enable` Method of *GemReports*:

```
Set clonedMeth = m_oGemReports.Methods.Item("Enable").Clone
```

Retrieve Collection Events Data from Method

Now, you need to get the `Collection event` data from the `Enable` Method's Inputs collection:

```
Set objDataItemCollectionEvents = _
clonedMeth.Inputs.Item("Collection events")
```

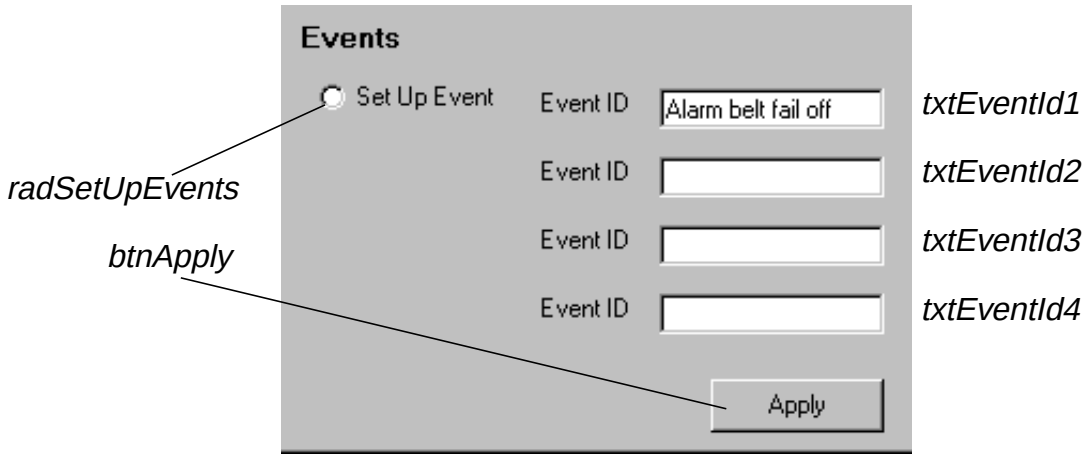
Then, to clear out any collection event IDs already in the collection, use the `Clear` Method of a DataItem object:

```
objDataItemCollectionEvents.Clear
```

Now you are ready to set the event IDs to the input from the GUI.

Set the Event IDs Using GUI Entries

Once the operator enters event IDs into the `Event ID` text boxes of the GUI, your code can retrieve the event IDs (see below).



Test each `Event ID` text box and if the text box does not contain an empty string, add the collection event to the DataItem collection using the `Add` method of a DataItem object. The `Add` method takes the name of a DataDef as an argument, in this case the name from the GUI:

```
If txtEventId1.Text <> "" Then
   objDataItemCollectionEvents.Add (txtEventId1.Text)
End If
If txtEventId2.Text <> "" Then
   objDataItemCollectionEvents.Add (txtEventId2.Text)
```

```
End If
```

NOTE If you run the sample program, you must enter CEIDs from the collection. You must spell the name exactly, including blank spaces. Refer to the dictionary of the sample Tool (*BTU recipe example*) in TOM Explorer; look under `Collection Events` for a complete list. Some of those used to test the sample include those below (all can be `off` or `on`, as shown for the first one):

```
Alarm belt fail off
Alarm belt fail on
Alarm conveyor speed off
Alarm package dropped off
Alarm rail position 1 off
```

```
If txtEventId3.Text <> "" Then
   objDataItemCollectionEvents.Add (txtEventId3.Text)
End If
If txtEventId4.Text <> "" Then
   objDataItemCollectionEvents.Add (txtEventId4.Text)
End If
```

Execute Clone of the Method

Once you have set all the collection event IDs, you can execute the cloned `Enable` Method of *GemReports*:

```
clonedMeth.Execute
ReportError "while enabling a list of events"
```

When TOM sends program control to the `tomctrl1_MethodNotification` routine, no other actions are required to complete the event setup.

Later, when the message executes, you see the following message:

Complete Code of btnApply_Click()

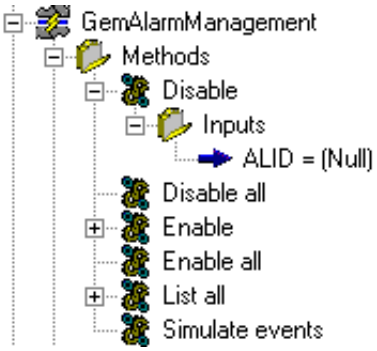
```
Private Sub btnApply_Click()

   On Error Resume Next

   Dim objDataItemCollectionEvents As DataItem
   Dim clonedMeth As tom.Method

   If radSetUpEvents.Value = 0 Then Exit Sub

   Set clonedMeth = m_oGemReports.Methods.Item_
   ("Enable").Clone

   'Get the Collection event data item from Enable Method

   Set objDataItemCollectionEvents = _
   clonedMeth.Inputs.Item("Collection events")

   'Clear out the possible previous list of event names.

   objDataItemCollectionEvents.Clear

   'Set Event Ids Names in the "Collection events" collection.

   If txtEventId1.Text <> "" Then
      objDataItemCollectionEvents.Add (txtEventId1.Text)
   End If

   If txtEventId2.Text <> "" Then
      objDataItemCollectionEvents.Add (txtEventId2.Text)
   End If

   If txtEventId3.Text <> "" Then
      objDataItemCollectionEvents.Add (txtEventId3.Text)
   End If

   If txtEventId4.Text <> "" Then
      objDataItemCollectionEvents.Add (txtEventId4.Text)
   End If

   'Execute the Enable Method from the GemReports service.

   clonedMeth.Execute

   ReportError "while enabling a list of events"

End Sub
```

Enabling and Disabling Alarms

To enable and disable alarms, you can use the *GemAlarmManagement* Service. To use the Service, you should declare its constant, generate a reference to it, and retrieve the Service, all under `Form_Load`:

```
Private Const SRV_GEMALARMMGMT = "GemAlarmManagement"
Private m_oGemAlarmMgmt As tom.Service
Set m_oGemAlarmMgmt = m_oTool.Resources.Item(1)._
Services.Item(SRV_GEMALARMMGMT)
ReportError "while finding the GemAlarmManagement service"
```

If you look in TOM Explorer, you can see that the Service has several Methods. The one that would disable an alarm, logically named `Disable`, has a single Input of an alarm ID. The Method that you use to enable all alarms is `Enable All` and has no Inputs. Let's start with enabling alarms.



Enable Alarms

To enable all alarms, you can have a routine that responds when the operator selects the `Enable Alarms` radio button (see illustration). Have the routine

first clone the `Enable All` Method, then execute it:

Complete Code of radEnableAlarms_Click

```
Private Sub radEnableAlarms_Click()
    Dim clonedMeth As tom.Method
    Set clonedMeth = m_oGemAlarmMgmt.Methods.Item_
    ("Enable All").Clone
    clonedMeth.Execute
    ReportError "while enabling GEM alarms"
```

```
End Sub
```

After `Enable All` executes, you need not take any other action, so the section of `tomctrl1_MethodNotification()` that tests for this Method name and its Service name need not contain any additional code.

Disable Alarms

To disable alarms, you can have a routine that responds when the operator selects the `Disable Alarm` radio button and fills in the `Alarm ID`. You can associate the routine with the `Disable Alarm` radio button and have it check that the `Alarm ID` text box is not empty:

```
Private Sub radDisableAlarms_Click()
   Dim clonedMeth As tom.Method

   If txtAlarmId.Text = "" Then Exit Sub
...
End Sub
```

You can then clone the `Disable` Method using the `Clone` method of a Method object:

```
Set clonedMeth = m_oGemAlarmMgmt.Methods.Item("Disable").Clone
```

Once you have the clone of the Method, you also have clones of all of its Inputs/Outputs. So, you can set the value of its alarm ID Input using the `txtAlarmId` from the GUI:

```
clonedMeth.Inputs.Item("ALID").Value = txtAlarmId.Text
```

Once you have set the required Input, you can then execute the Method:

```
clonedMeth.Execute
```

When TOM sends control to the `tomctrl1_MethodNotification` routine, no special action needs to take place there.

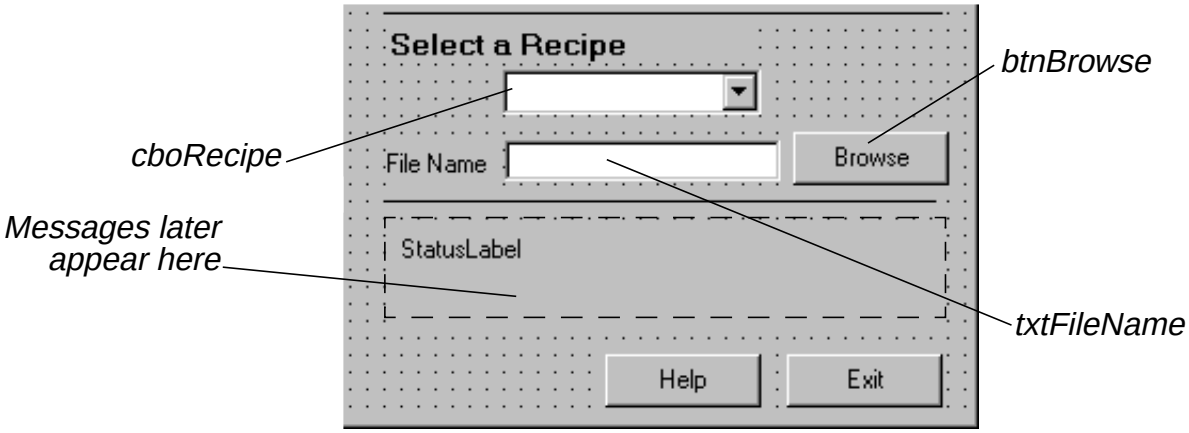
Complete Code of radDisableAlarms_Click

```
Private Sub radDisableAlarms_Click()
   Dim clonedMeth As tom.Method
   If txtAlarmId.Text = "" Then Exit Sub
   Set clonedMeth = m_oGemAlarmMgmt.Methods.Item_
   ("Disable").Clone
   clonedMeth.Inputs.Item("ALID").Value = txtAlarmId.Text
   clonedMeth.Execute
   ReportError "while disabling alarm" & Str(txtAlarmId)
End Sub
```

Selecting and Downloading a Recipe

Next, you want to have code for:

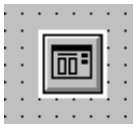
- Browse button with a text box that receives a recipe file
- Combo box where the operator can pull down a list of recipes



Use Browse Button to Retrieve Recipe File

To retrieve the file name through a browse button, the sample application includes the Common Dialog control in its form (see illustration at beginning that follows).

Common Dialog control

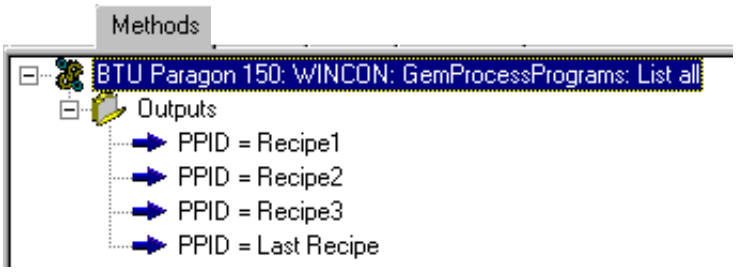


The sample application uses the `FileName` method of the Common Dialog control (`commdlg`) to retrieve the file the operator selects and place it in the GUI text box named `txtFileName`; the routine's code appears below (for more detail, refer to the Visual Basic documentation):

```
Private Sub btnBrowse_Click()
    commdlg.DialogTitle = "Find Recipe file"
    commdlg.ShowOpen
    txtFileName = commdlg.FileName
End Sub
```

List Recipes in Combo Box

To see what kind of recipes the program should list in the combo box, you can execute the `List all` Method of *GemProcessPrograms* in TOM Explorer and then check under the Methods tab to see the resulting recipes:



You can see that there are four recipes. When the `List all` Method executes from inside your code, it also produces this list of Outputs. You must, of course, clone the Method, then execute it, in response to an operator clicking on the combo box:

Complete Code of cboRecipe_Click

```
Private Sub cboRecipe_Click()
   Dim cloneMethLocal As tom.Method
   If txtFileName.Text = "" Then Exit Sub
   Set cloneMethLocal = m_oGemProcess.Methods.Item_
   ("List all").Clone
   cloneMethLocal.Execute
   ReportError "while listing all recipes"
End Sub
```

To put the recipes resulting from the `List all` Method into the pulldown menu, you retrieve the Outputs from the Method *after* the Method executes, which means you must retrieve those Outputs after TOM sends a Method completed notification to the application. In the `tomCtrl1_MethodNotification` routine, you develop a case for when the Method is `List all` and the Service is *GemProcessPrograms*:

Code of MethodNotification for List all Method

```
Private Sub tomCtrl1_MethodNotification(ByVal tomMethod As
Object)
   Dim counter
   ...
   Select Case tomMethod.Name
      'GemProcessPrograms
      Case "List all"
         If tomMethod.Service.Name = SRV_GEMPROCESS Then
            For counter = 1 To 4
               cboRecipe.AddItem tomMethod.Outputs._
               Item(counter).Value
            Next counter
            cboRecipe.Refresh
```

```
                Call Upload_Recipe(tomMethod, cboRecipe.Text)
            End If
        End Select
End Sub
```

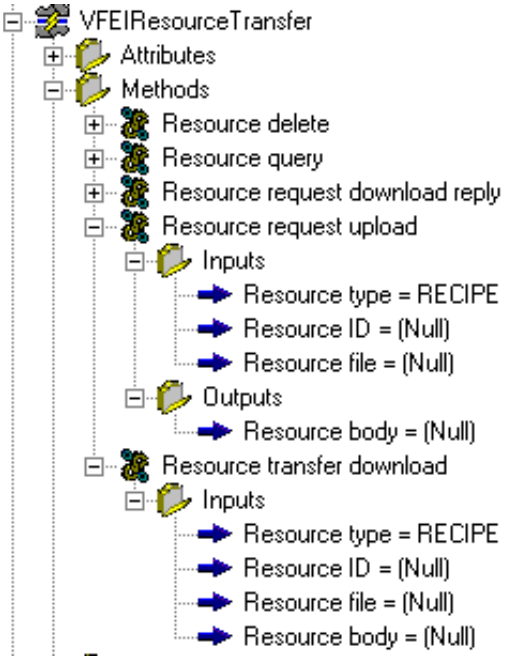
Using a `For/Next` loop, you can retrieve the recipes from the `List all` Method's Outputs. Using the `Item` method of a DataItem object (remember, Inputs and Outputs are DataItems), you get the `Value` property and associate it with its counter number to position it in the pulldown list. Using the `AddItem` method in Visual Basic, you add the recipes to the combo box list.

Upload/Download Recipe Operator Selects

Then you can call another routine. You pass it the name of the Method last executed and the name of the recipe the operator selected in the GUI:

```
Call Upload_Recipe(tomMethod, cboRecipe.Text)
```

In the `Upload_Recipe` routine, you work with yet another Service, called *VFEIResourceTransfer*. You use its `Resource request upload` Method to send some fundamental Inputs to the Tool so that you can retrieve the Output from this Method and use it as an Input to the next Method, `Resource transfer download`. You can see the relationship between the two Methods in the illustration from TOM Explorer shown below:



Pass Data from One Method to Another

Passing data from one Method to another inside a TOM application is useful when you are daisy-chaining Methods together in a sequence. Although in this example, you pass data between Methods of the same Service, you can also use this same technique to pass data from one Service's Method to another Service's Method.

Let's see how you can use the `Resource body` Output from `Resource request upload` as an Input to `Resource transfer download`.

You also pass some Inputs from the upload to the download Method, to ensure the Methods use the same Inputs.

Let's see how that works. Begin by defining the the routine so it receives the two arguments:

```
Private Sub Upload_Recipe(RecipeMethod As tom.Method, Recipe As
String)
```

Next, you set clone `Resource request upload`:

```
Dim clonedMeth As tom.Method
Set clonedMeth = m_oVFEIResourceXfer.Methods._
Item("Resource request upload").Clone
```

Then you set its Inputs using the `Item` method of a DataItem object and setting the `Value` property for each:

```
clonedMeth.Inputs.Item("Resource type").Value = "Recipe"
clonedMeth.Inputs.Item("Resource ID").Value = Recipe
clonedMeth.Inputs.Item("Resource file").Value = txtFileName
```

Finally, you execute the Method using the `Execute` method of a Method object:

```
clonedMeth.Execute
ReportError "while requesting recipe upload"
```

Once the Method executes, TOM sends program control to the `tomCtrl1_MethodNotification` routine, where you create a case for when the TOM Method is `Resource request upload` and the Service is *VFEIResourceTransfer*. In that case, you deal with the Output from the Method. You set the `Tag` property of the TOM Method to the value of the Output by using the `Item` method of a DataItem object and retrieving that object's `Value` property:

Code of MethodNotification for Resource request upload Method

```
Private Sub tomCtrl1_MethodNotification(ByVal tomMethod As
Object)
...
    'VFEIResourceTransfer Resource Request Upload method
    Case "Resource request upload"
    If tomMethod.Service = SRV_VFEIRESXFER
        tomMethod.Tag = tomMethod.Outputs.Item("Resource_
```

```
        body").Value
        Call Download_Recipe(tomMethod, cboRecipe.Text)
    End Select
End Sub
```

Once you have set the tag, you call the `Download_Recipe` routine and pass it the TOM Method and the recipe name from the `cboRecipe` text box:

```
Call Download_Recipe(tomMethod, cboRecipe.Text)
```

Remember the Method being passed in to the routine is the `Resource request upload` Method, so the next routine can retrieve the values from both its Inputs and its Outputs. Now, let's see how you would have `Download_Recipe` use those Inputs and Outputs.

NOTE

TOM Tip—Copying Data from One Service to Another

To pass data from one Service to another, you should map the Outputs of the Service Method that has the information to the Inputs of the Service Method you want to pass the data into.

Use Inputs and Outputs from Another Method

First, you generate the routine and clone the `Resource transfer download` Method within the routine:

```
Private Sub Download_Recipe(RecipeMethod As tom.Method, Recipe
As String)
    Dim clonedMeth As tom.Method
    Set clonedMeth = m_oVFEIResourceXfer.Methods.Item_
    ("Resource transfer download").Clone
End Sub
```

Now, you want to copy both the Inputs and the Outputs from previously executed Method to this one. Because you passed in the name of the previously executed recipe in `RecipeMethod` and you have the soon-to-be-executed Method stored in the `clonedMeth` variable, you can work with the two Services and set the cloned Method's Input items using the Inputs from the previously executed Method:

```
clonedMeth.Inputs.Item("Resource type").Value = RecipeMethod._
Inputs.Item("Resource type").Value
clonedMeth.Inputs.Item("Resource ID").Value = RecipeMethod._
Inputs.Item("Resource ID").Value
clonedMeth.Inputs.Item("Resource file").Value = RecipeMethod._
Inputs.Item("Resource file").Value
```

You use the `Item` method of each Input on both sides of the assignment operator.

To set the last Input to the Output from the previously executed Method, you set it to the `Tag` from that Method, which you can still access because you have passed that other Method's name to this Method:

```
clonedMeth.Inputs.Item("Resource body").Value =_
RecipeMethod.Tag
```

Finally, you can execute the cloned Method:

```
clonedMeth.Execute
ReportError "while downloading recipe"
```

When you execute this Method, you receive no Outputs from it and need not take any action in the `tomCtrl1_MethodNotification` routine.

Complete Code of Upload_Recipe

```
Private Sub Upload_Recipe(RecipeMethod As tom.Method, Recipe As
String)

   Dim clonedMeth As tom.Method

   Set clonedMeth = m_oVFEIResourceXfer.Methods.Item_
   ("Resource request upload").Clone

   clonedMeth.Inputs.Item("Resource type").Value = "Recipe"

   clonedMeth.Inputs.Item("Resource ID").Value = Recipe

   clonedMeth.Inputs.Item("Resource file").Value = _
   txtFileName

   clonedMeth.Execute

   ReportError "while requesting recipe upload"

End Sub
```

Complete Code of Download_Recipe

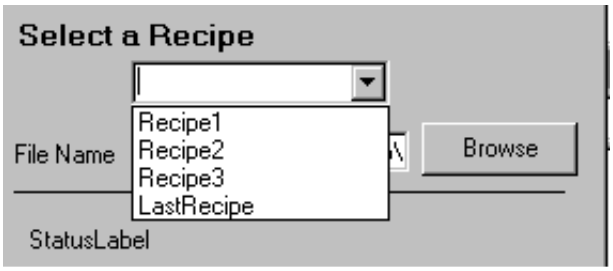
```
Private Sub Download_Recipe(RecipeMethod As tom.Method, Recipe
As String)

   Dim clonedMeth As tom.Method

   Set clonedMeth = m_oVFEIResourceXfer.Methods.Item_
   ("Resource transfer download").Clone

   'Copy inputs from previously exeuted method to this one.
   'Also copy outputs from previously executed method to this
   'one. RecipeMethod contains the previously executed method.

   clonedMeth.Inputs.Item("Resource type").Value = _
   RecipeMethod.Inputs.Item("Resource type").Value

   clonedMeth.Inputs.Item("Resource ID").Value = _
   RecipeMethod.Inputs.Item("Resource ID").Value

   clonedMeth.Inputs.Item("Resource file").Value = _
   RecipeMethod.Inputs.Item("Resource file").Value
```

```
    clonedMeth.Inputs.Item("Resource body").Value = _
    RecipeMethod.Tag

    clonedMeth.Execute

    ReportError "while downloading recipe"
End Sub
```

See the Results

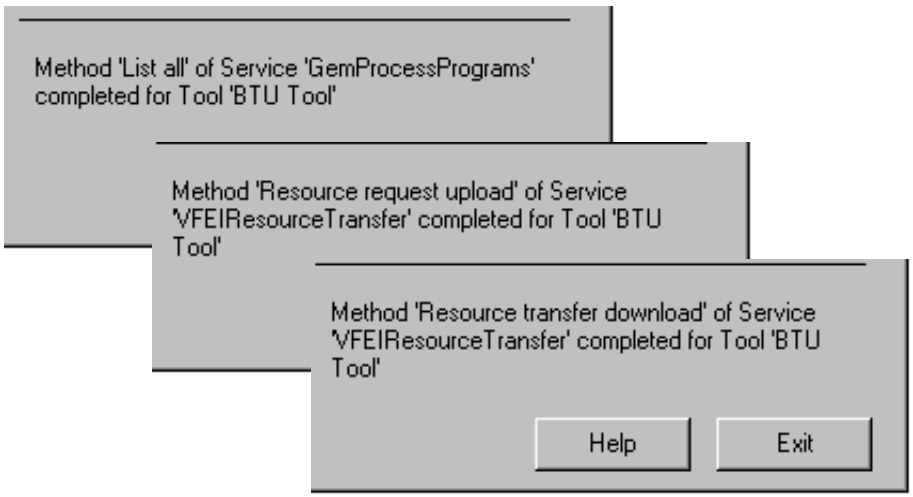
When an operator runs the application, you can browse the network for the file containing the recipe:

Now the pulldown list of recipes becomes available in the combo box:



After you select the file containing that recipe, when you select the recipe name, the application executes the Methods. When TOM sends the status notifications to the application, you see each related status message in the GUI. Due to the 10 second delays added to the sample code, you have time to read each message before the next Method executes.

The three messages you see, in sequence, are shown below:



To generate a delay between messages, the sample application uses the `Sleep` statement from a Windows Visual Basic library. Refer to the Appendix, *Application Code*, p. A-1, for more information and for a complete list of the code.

Tips and Tricks

3

Topics in this chapter

This chapter presents a few tips and tricks to follow when writing TOM applications.

Using Non-Modal Dialog Boxes

To display a dialog box from the application, you need to be sure it is a non-modal dialog box. For example, to display a form named NotifyForm, you would use Show 0:

```
NotifyForm.Show 0
```

CAUTION

You should avoid using modal dialog boxes in TOM applications. Do not display a form that waits for a reply using `Show 1`, because that makes the form a modal dialog box. When you use a modal dialog box in a TOM application, TOM Core becomes suspended—all action stops.

To display the form in a non-modal dialog box, you should always use the `Show 0` method rather than `Show 1`.

Using Daisy-Chained Services/Methods

The best approach to using multiple Services is daisy-chaining Services by having one Service call another. This technique is essentially the technique that *GemEstablishCommunications* uses when it calls *ProtocolSECS*. You cannot see this action taking place—it happens in the background. Your own custom Services can take similar action. For more information on writing Services, refer to the *Tool Object Model (TOM) Service Developer's Guide*.

NOTE

TOM Tip—Using Daisy-Chained Services/Methods

You can and should daisy chain Services by having one Service call the Method of the next Service, and the Method of the next Service call the Method of another Service, and so on.

Using Variables to Maintain Context

A recommended way you can use local variables is to maintain context across multiple routines or Services. Remember, you can use the `Tag` property of a Service to pass information from that Service to another Service.

NOTE **TOM Tip—Maintaining Context across Routines/Services**

To maintain context across multiple routines/Methods, you should declare a local variable and pass it from routine to routine (or Service to Service). Each routine (or Service) then knows what the previous routine has done or can use information from the Method that just executed.

Waiting for Events

NOTE **TOM Tip—Waiting for Events**

Your application cannot receive Events from the TOM control or any control during `Form_Load`. So, do not have your code wait for an Event in `Form_Load`.

Stopping an Application

NOTE **TOM Tip—Stopping an Application While Method Is Active**

You should always stop an application between Method invocations, rather than while a Method is running.

Application Code
Appendix

A

Topics in This Appendix

This appendix presents the full code for the sample application.

The code for the application is included in the Service's Developer's Kit (SDK) in a project named *myrecipe.vbp*. You can find this project under *FASTech\Sw\Dev\Samples\apps\MyRecipe*.

Complete Code of Recipe Application

NOTE The `sleep` statement in this code is available only if you declare the appropriate library by having the following `Declare` statement in a *.bas* file (the sample one is in *recipe.bas*):

```
Declare Sub Sleep Lib “kernel32” (ByVal dwMilli-
seconds As Long)
```

```
' ---- FASTech Integration.  Copyright 1996-1997
' Sample code is provided to customers for unsupported
' use only. Technical Support will accept notification
' of problems in sample services and applications, but
' FASTech will make no guarantee to fix the problems in
' current or future releases.
```

General Declarations

```
Option Explicit
' This is the TOM tool used in the TOM Application
Private Const TOOL_NAME = "BTU recipe example"

' This is the database file where the tool is defined
Private Const DATABASE_NAME = "myrecipe.mdb"

' Where you can find help file info about this application
' For a detailed explanation of this application, please refer
' to the TOM Application Developer’s Guide
‘ on the CD in Acrobat PDF format.
' Private Const APP_HELPFILE = "myrecipe.hlp"
' Private Const APP_HELP_CONTEXT = 50001

' Specify needed services
' using their generic service names.
Private Const SRV_GEMPROCESS = "GemProcessPrograms"
Private Const SRV_VFEIRESXFER = "VFEIResourceTransfer"
Private Const SRV_GEMALARMMGMT = "GemAlarmManagement"
Private Const SRV_GEMREPORTS = "GemReports"
Private Const SRV_GEMESTABCOMMS = "GemEstablishCommunications"
Private Const SRV_PROTOCOLSECS = "ProtocolSECS"

' A TOM application must keep a reference to the top-level
' TOM object over the lifetime of TOM tools.
Private m_oTOM As tom.ToolObjectModel

' After our tool is instantiated, this reference keeps it
```

```
Private m_oTool As tom.Tool

' References to service objects used in application
Private m_oGemProcess As tom.Service
Private m_oVFEIResourceXfer As tom.Service
Private m_oGemAlarmMgmt As tom.Service
Private m_oGemReports As tom.Service
Private m_oGemEstablishComms As tom.Service
Private m_oProtocolSecs As tom.Service

' References to other objects
Private m_oCurrStatusLabel As Label
```

ReportError Function

```
'ReportError function called when an error occurs
Public Sub ReportError(strMessage As String)
   If (Err.Number <> 0) Then
   MsgBox "Error " & Str(Err.Number) & "(" & Err.Description_
   & ")" & strMessage
   End If
End Sub
```

btnApply_Click Routine

```
Private Sub btnApply_Click()
   Dim objDataItemCollectionEvents As DataItem
   Dim clonedMeth As tom.Method

   On Error Resume Next
   ' Set up events on the tool.
   ' Use the reference to the GemReports service.
   If radSetUpEvents.Value = 0 Then Exit Sub
      Set clonedMeth = m_oGemReports.Methods.Item_
      ("Enable").Clone
'Get "Collection event" data item from "Enable" Method
   Set objDataItemCollectionEvents = _
   clonedMeth.Inputs.Item("Collection events")

'Clear out the possible previous list of event IDs.
   objDataItemCollectionEvents.Clear

'Set Event Ids Names in the "Collection events" collection.
   If txtEventId1.Text <> "" Then
      objDataItemCollectionEvents.Add (txtEventId1.Text)
   End If
   If txtEventId2.Text <> "" Then
```

```
        objDataItemCollectionEvents.Add (txtEventId2.Text)
    End If
    If txtEventId3.Text <> "" Then
        objDataItemCollectionEvents.Add (txtEventId3.Text)
    End If
    If txtEventId4.Text <> "" Then
        objDataItemCollectionEvents.Add (txtEventId4.Text)
    End If
'Execute Enable Method from GemAlarmManagement service.
    clonedMeth.Execute
    ReportError "while enabling a list of events"
End Sub
```

btnBrowse_Click Routine

```
Private Sub btnBrowse_Click()
    commdlg.DialogTitle = "Find Recipe file"
    commdlg.ShowOpen
    txtFileName = commdlg.filename
End Sub
```

ButtonHelp_Click Routine

```
Private Sub ButtonHelp_Click()
'  HelpByContext Me.hwnd, APP_HELPFILE, APP_HELP_CONTEXT
End Sub
```

cboRecipe_Click Routine

```
Private Sub cboRecipe_Click()
    Dim cloneMethLocal As tom.Method
'The following statement clones the "List all" method of the
'GemProcessPrograms service, which it extracts and stores in
'm_oGemProcess when the form loads and the Form_Load()
'routine executes. To clone the Method, the statement below
'uses the Clone method of the Method object. Since the clone
'has local scope, when the routine is over, the clone is gone.
    If txtFileName.Text = “” Then Exit Sub
    Set cloneMethLocal = m_oGemProcess.Methods._
    ("List all").Clone
    cloneMethLocal.Execute
    ReportError "while listing all recipes"
End Sub
```

Form_Load Routine

```
' When the main form is loaded, the app must initialize
' the TOM Core. Then it can proceed to instantiate your tool.
Private Sub Form_Load()
    Dim ToolTypes As tom.ToolTypes
    Dim ToolType As tom.ToolType
    Dim MsgForm As frmMessage
    Dim fSuccessfulStartup As Boolean

    On Error Resume Next

    'While startup has not successfully completed, set local var
    'to False. Later, when form has loaded, set it to True.
    fSuccessfulStartup = False

    ' App.HelpFile = APP_HELPFILE
    ' Me.HelpContextID = APP_HELP_CONTEXT

    ' Create the TOM Core
    Err.Clear
    Set m_oTOM = CreateObject("tom.ToolObjectModel")
    If Err Then
        ReportError " while creating TOM object " & Err.Description
    Else
        ' User can specify an alternative database
        If Command <> "" Then
            m_oTOM.DefinitionFile = Command
        Else
            m_oTOM.DefinitionFile = DATABASE_NAME
        End If

        ' Now, Initialize the TOM Core
        Err.Clear
        m_oTOM.Initialize tomCtrl1
        If Err Then
            ReportError " while initializing TOM Core"_
            & Err.Description
        Else
            ' Find our tool in the ToolTypes collection
            Set ToolTypes = m_oTOM.ToolTypes
            Set ToolType = ToolTypes.Item(TOOL_NAME)

            ' Show progress status dialog while TOM instantiates
            ' the tool. Then catch StatusNotification events sent
```

```
            ' by the TOM Control and display the events in this
            ' dialog.
            Set MsgForm = New frmMessage
            MsgForm.Show
            MsgForm.Refresh
            Set m_oCurrStatusLabel = MsgForm.LabelMsg

            ' Tell TOM Core to instantiate the tool
            ' Assign a unique name to this instantiation
            Err.Clear
            Set m_oTool = m_oTOM.Tools.Add(ToolType, "BTU Tool")
            If Err Then
                MsgBox "Unable to create tool: " & Err.Description,_
                vbExclamation, App.Title
            End If

            ' Now display status notifications in our main window
            Set m_oCurrStatusLabel = LabelStatus
            Unload MsgForm

            'Set up Service objects in separate routine
            lSetupServices

            'Now that form has loaded and services are ready,
            'set local var to True.
            fSuccessfulStartup = True
        End If
    If Not fSuccessfulStartup Then
        Unload Me
    End If
End Sub
```

lSetupService Routine That Form_Load Calls

The example uses a separate routine that it calls from within `Form_Load` to get a reference to each Service it uses:

```
Private Sub lSetupServices()
    On Error Resume Next

' Find the Service objects you want/extract each from collection

    Set m_oGemProcess = m_oTool.Resources.Item(1)._
    Services.Item(SRV_GEMPROCESS)
    ReportError "while finding the GemProcessPrograms Service"
    Set m_oVFEIResourceXfer = m_oTool.Resources.Item(1)._
    Services.Item(SRV_VFEIRESXFER)
    ReportError "while finding the VFEIResourceTransfer Service"
    Set m_oGemAlarmMgmt = m_oTool.Resources.Item(1)._
    Services.Item(SRV_GEMALARMMGMT)
```

```
    ReportError "while finding the GemAlarmManagement Service"
    Set m_oGemReports = m_oTool.Resources.Item(1)._
    Services.Item(SRV_GEMREPORTS)
    ReportError "while finding the GemReports Service"
    Set m_oGemEstablishComms = m_oTool.Resources.Item(1)._
    Services.Item(SRV_GEMESTABCOMMS)
    ReportError "while finding the GemEstablishCommunications
    Service"
    Set m_oProtocolSecs = m_oTool.Resources.Item(1)._
    Services.Item(SRV_PROTOCOLSECS)
    ReportError "while finding the ProtocolSECS Service"

    Exit Sub
End Sub
```

Form_Unload Routine

```
Private Sub Form_Unload(Cancel As Integer)
' Remember to set m_oTOM to nothing, so it can go away.
    Set m_oTool = Nothing
    Set m_oTOM = Nothing
    Set m_oCurrStatusLabel = Nothing
End Sub
```

radDisableAlarms_Click Routine

```
Private Sub radDisableAlarms_Click()
    Dim clonedMeth As tom.Method
    If txtAlarmId.Text = "" Then Exit Sub
'Disable a single GEM alarm.
'Use the reference to the GemAlarmManagement service

'First clone Disable method of GemAlarmManagement service.
'Then set the ALID variable in the input collection of the
'Disable Method.

    Set clonedMeth = m_oGemAlarmMgmt.Methods.Item_
    ("Disable").Clone
    clonedMeth.Inputs.Item("ALID").Value = txtAlarmId.Text
'Execute Disable method from GEMAlarmManagement service.
    clonedMeth.Execute
'After you call Execute method of a Method object,
'the TOM Core has a reference to the object as long as the
'routine is executing. After the routine finishes executing 'TOM
Core no longer retains the reference.
'After this routine ends, the reference goes away. You can
'also save the reference after the method is complete in order
'to maintain access to outputs from the method. If you want to
'save a method after it has executed, you should create a
```

```
'global reference for it rather than a reference local
'to the routine.

   ReportError "while disabling alarm" & Str(txtAlarmId)
End Sub
```

radEnableAlarms_Click Routine

```
Private Sub radEnableAlarms_Click()
   Dim clonedMeth As tom.Method
   ' Enable all GEM alarms.
   ' Use the reference to the GemAlarmManagement service
   Set clonedMeth = m_oGemAlarmMgmt.Methods.Item_
   ("Enable All").Clone
   ' Run 'Enable All' Method from GEMAlarmManagement service.
   clonedMeth.Execute
   ReportError "while enabling GEM alarms"
End Sub
```

radEstabComms_Click Routine

```
Private Sub radEstabComms_Click()
   Dim clonedMeth As tom.Method
'Set attributes of the ProtocolSECS service.
'The GemEstablishCommunications service then uses runs the
'ProtocolSECS service when it communicates with the tool.
'For your tool, you may need to set additional attributes
'of this service.
   m_oProtocolSecs.Attributes.Item("Baud").Value = "9600"

   m_oProtocolSecs.Attributes.Item("IPAddressLocal")._
   Value = "0.0.0.0"

   m_oProtocolSecs.Attributes.Item("IPAddressRemote")._
   Value = "255.255.255.100"

'You would set the IPPortLocal and IPPortRemote attributes for
'a terminal server
   m_oProtocolSecs.Attributes.Item("IPPortLocal").Value_
   = "5000"
   m_oProtocolSecs.Attributes.Item("IPPortRemote").Value_
   = "5000"
'You set the PortType to HSMS by setting it to 1.
'Since you need to be able to test this sample without a tool,
'the PortType is being set to the default of 0 for an RS-232
'connection. For RS-232, you also needs to set the SerialPort.

   m_oProtocolSecs.Attributes.Item("PortType").Value = "0"
   m_oProtocolSecs.Attributes.Item(“SerialPort”).Value = “COM1”

'Establish communication with the tool.
```

```
'Use the reference to the GemEstablishCommunications service.
   Set clonedMeth = m_oGemEstablishComms.Methods.Item_
   ("Connect").Clone
   clonedMeth.Execute
   ReportError "while opening the SECSProtocol Serial Port"
End Sub
```

tomCtrl1_EventNotification Routine

```
Private Sub tomCtrl1_EventNotification(ByVal tomEvent As Object)
   Select Case tomEvent.Name
      Case "Alarm set"
         If m_oCurrStatusLabel Is Nothing Then Exit Sub
         m_oCurrStatusLabel = tomEvent.Outputs.
         Item("ALTX").Value & " Alarm set"
         m_oCurrStatusLabel.Refresh
      Case "Alarm clear"
         If m_oCurrStatusLabel Is Nothing Then Exit Sub
         m_oCurrStatusLabel = tomEvent.Outputs.Item_
         ("ALTX").Value & " Alarm cleared"
         m_oCurrStatusLabel.Refresh
   End Select
End Sub
```

tomCtrl1_MethodNotification Routine

```
Private Sub tomCtrl1_MethodNotification(ByVal tomMethod As
Object)
   Dim counter
   Select Case tomMethod.Name
      'GemAlarmManagement Enable All method
      Case "Enable all"

      'GemAlarmManagement Disable method
      Case "Disable"

      'GemEstablishCommunications
      Case "Connect"
      'Retrieve the model number and software revision from the
      'outputs of the method.
      'The statements shown below set MDLN and SOFTREV fields
      'in the GUI using the Value property of the outputs from
      'the Connect method:
      If tomMethod.Service.Name =SRV_GEMPROCESS Then
         txtMdln.Text = tomMethod.Outputs.Item("MDLN").Value
         txtMdln.Refresh
         txtSoftRev.Text = tomMethod.Outputs.Item("SOFTREV").Value
```

```
            txtSoftRev.Refresh
        End If

    'GemReports
    Case "Enable"

    'GemProcessPrograms List All method
    Case "List all"
    'Retrieves the Recipes from the Outputs that the application
    'set when it executed the "List all" method of the
    'GemProcessPrograms service(in cboRecipe_Click() routine)
    If tomMethod.Service.Name = SRV_GEMPROCESS Then
        For counter = 1 To 4
            cboRecipe.AddItem tomMethod.Outputs.Item(counter)._
            Value
        Next counter
        cboRecipe.Refresh
        ‘ Delay -- Strictly for demo purposes. Delays next Method
        ‘ so you may read the status message that is displaying.
        Sleep 5000
        ' Calling Upload Recipe
        Call Upload_Recipe(tomMethod, cboRecipe.Text)
    End If

    'VFEIResourceTransfer Resource Request Upload method
    Case "Resource request upload"
    If tomMethod.Service.Name = VFEIRESXFER Then
        tomMethod.Tag = tomMethod.Outputs.Item_
        ("Resource body").Value
        ‘ Delay -- Strictly for demo purposes. Delays next Method
        ‘ so you may read the status message that is displaying.
        Sleep 5000
        Call Download_Recipe(tomMethod, cboRecipe.Text)
    End If

    'VFEIResourceTransfer Resource Transfer Download method
    Case "Resource transfer download"

    End Select
End Sub
```

Upload_Recipe Routine

```
Private Sub Upload_Recipe(RecipeMethod As tom.Method, Recipe As
String)
    Dim clonedMeth As tom.Method

    Set clonedMeth = m_oVFEIResourceXfer.Methods.Item_
    ("Resource request upload").Clone
    clonedMeth.Inputs.Item("Resource type").Value = "Recipe"
```

```
    clonedMeth.Inputs.Item("Resource ID").Value = Recipe
    clonedMeth.Inputs.Item("Resource file").Value = _
    txtFileName
    clonedMeth.Execute
    ReportError "while requesting recipe upload"
End Sub
```

Download_Recipe Routine

```
Private Sub Download_Recipe(RecipeMethod As tom.Method, Recipe
As String)
    Dim clonedMeth As tom.Method
    Set clonedMeth = m_oVFEIResourceXfer.Methods.Item_
    ("Resource transfer download").Clone
    'Copy inputs from previously exeuted method to this one.
    'Also copy outputs from previously executed method to this
    'one. RecipeMethod contains the previously executed method.
    clonedMeth.Inputs.Item("Resource type").Value = _
    RecipeMethod.Inputs.Item("Resource type").Value
    clonedMeth.Inputs.Item("Resource ID").Value = _
    RecipeMethod.Inputs.Item("Resource ID").Value
    clonedMeth.Inputs.Item("Resource file").Value = _
    RecipeMethod.Inputs.Item("Resource file").Value
    clonedMeth.Inputs.Item("Resource body").Value = _
    RecipeMethod.Tag
    clonedMeth.Execute
    ReportError "while downloading recipe"
End Sub
```

tomCtrl1_StatusNotification Routine

```
Private Sub tomCtrl1_StatusNotification(ByVal StatusText As
String)
    If m_oCurrStatusLabel Is Nothing Then Exit Sub
    m_oCurrStatusLabel = StatusText
    m_oCurrStatusLabel.Refresh
End Sub
```

ButtonExit_Click Routine

```
Private Sub ButtonExit_Click()
    Unload Me
End Sub
```

txtEventId_Change Routine

```
Private Sub txtEventId_Change()
    If radEnableAlarms.Value = 0 Then
        Exit Sub
    Else
        Call btnApply_Click
    End If
End Sub
```

txtAlarmId_Click Routine

```
Private Sub txtAlarmID_Click()
    If radDisableAlarms.Value = False Then
        Exit Sub
    Else
        Call radDisableAlarms_Click
    End If
End Sub
```

Index