



Service Developer's Guide

November 1999

STATIONworks Version 2.1
A FASTech MES Product



This document contains information that is the property of Brooks Automation, Inc., Chelmsford, MA 01842, and is furnished for the sole purpose of the operation and the maintenance of FASTech products of Brooks Automation, Inc. No part of this publication is to be used for any other purpose, and is not to be reproduced, copied, disclosed, transmitted, stored in a retrieval system, or translated into any human or computer language, in any form, by any means, in whole or in part, without the prior express written consent of Brooks Automation, Inc.

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

Copyright © 1999 by Brooks Automation, Inc. All rights reserved.

Though at Brooks Automation, Inc., we make every effort to ensure the accuracy of our documentation, Brooks assumes no responsibility for any errors that may appear in this document. The information in this document is subject to change without notice.

Sample code that appears in documentation is included for illustration only and is, therefore, unsupported. This software is provided free of charge and is not warranted by Brooks in any way. FASTech Products Technical Support will accept notification of problems in sample applications, but Brooks will make no guarantee to fix the problem in current or future releases.

FASTech's CELLman, CELLtalk, CELLguide, Grapheq, WINclient, TOM, STATIONSworks, and FASTspc are trademarks of Brooks Automation, Inc. FASTech, FASTech's CELLworks and FACTORYworks are registered trademarks of Brooks Automation, Inc.

Acrobat Reader is a trademark of Adobe Systems Incorporated.

CodeCenter, ObjectCenter, and TestCenter are trademarks of CenterLine.

DIGITAL UNIX is a trademark of Digital Equipment Corporation.

Glance is a trademark of Hewlett-Packard

HP-UX and Glance are trademarks of Hewlett-Packard Company.

Ingres is a trademark of Ingres Corporation.

ORACLE, ORACLE 7, SQL*Net, and SQL*Plus are registered trademarks of Oracle Corporation.

OSF/Motif is a trademark of Open Software Foundation, Inc.

POLYCENTER is a trademark of Computer Associates International, Inc.

PostScript is a registered trademark of Adobe Systems, Inc.

Purify, Quantify, PureCover are trademarks of Pure Software

Seagate Crystal Reports and Seagate Crystal Info are trademarks or registered trademarks of Seagate Technology, Inc. or one of its subsidiaries

SEMI is a trademark of Semiconductor Equipment and Materials International.

Solaris is a trademark of Sun Microsystems, Inc.

SPARCCompiler, UltraSPARC, and all other SPARC trademarks are registered trademarks of SPARC International, Inc.

Sun is a trademark of Sun Microsystems, Inc.

Sybase is a trademark of Sybase, Inc.

System V and SVID (System V Interface Definition) are trademarks of American Telephone and Telegraph Co.

TIB is a trademark of Teknekron Software Systems, Inc.

Tools.h++ and DB.h++ are trademarks of RogueWave Software, Inc.

UNIX is a registered trademark of X/Open Company, Ltd.

SmartShapes and Visio are registered trademarks of Visio Corporation.

Windows NT, Active X, and Visual Basic are trademarks of Microsoft Corporation.

Workstream is a trademark of Consilium, Inc.

X Window system is a trademark of the Massachusetts Institute of Technology.

XRrunner is a trademark of Mercury Interactive.

All other product names referenced are believed to be the registered trademarks of their respective companies.

Table of Contents

About This Manual

Chapter 1 Getting Started with Samples

Combining Tools and Services	1-2
Establishing Database Components.....	1-3
Working with Sample Services/Applications	1-4
Building a Database of Sample Tools	1-5
Compiling Sample Services.....	1-7
Removing Samples	1-8
Sample Services for Testing/Debugging.....	1-9
Understanding Objects in TOM.....	1-10

Chapter 2 Developing Service Infrastructure

Defining the Service's Role in the Application	2-2
Writing a Custom Service—Steps to Take	2-2
Creating the Visual Basic Project	2-4
Adding Custom Controls to Your Project	2-4
Adding Required Files to Your Project.....	2-5
Creating References for Your Project	2-5
Assigning the Project Name and Title	2-6
Creating a Class Module and Declaring Service Name.....	2-10
Understanding References, Variables, & Constants Required.....	2-12
Creating References, Variables, and Constants	2-13
Fitting Services Together in Visual Basic Project	2-14

Chapter 3 Writing Your Handler Methods

Writing Required Handler Methods That TOM Triggers.....	3-2
Understanding the OnCreate Handler Method	3-5
Writing the OnCreate Handler Method	3-6
Defining Method Objects for Your Service in OnCreate.....	3-9
Defining Event Objects for Your Service in OnCreate	3-11
Writing the LetAttribute Handler Method	3-14
Writing the GetAttribute Handler Method	3-15
Writing the OnInitialize Handler Method.....	3-16
Writing the OnExecute Handler Method	3-20

Executing Existing Methods in OnExecute	3-22
Writing the OnMethodCompleted Handler Method	3-25
Writing the OnSubscribedEvent Handler Method	3-30
Triggering Your Service Event	3-31
Writing the OnVerify Handler Method	3-33
Verifying a Service—The Nuts and Bolts.....	3-36
Writing the Version Handler Method.....	3-40
Writing the OnTerminate Handler Method.....	3-41
Writing a Terminate Class Method	3-41
Chapter 4 Creating a Tool for Your Service	
Working with TOM Builder.....	4-3
Creating a New Tool	4-4
Creating a New Resource	4-5
Adding Resources to the Tool.....	4-6
Adding Your Custom Service to Database	4-8
Setting Properties of Your Service.....	4-9
Assigning Services to Tool Resources	4-11
Creating a New Service Dictionary	4-13
Assigning the Dictionary to a Service	4-16
Creating a New Resource Dictionary	4-17
Assigning the Dictionary to Resources.....	4-20
Creating DataDefs.....	4-21
Cloning DataDefs	4-27
Creating Attributes.....	4-28
Finalizing Tool by Releasing It.....	4-31
Building TOM Database (Containing New Tool).....	4-32
Chapter 5 Debugging/Testing Your Service	
Preparing to Use Your Service in TOM Explorer.....	5-2
Running Your Service in Debug Mode.....	5-4
Executing Methods through TOM Explorer	5-8
Verifying the Service from TOM Explorer.....	5-11
Exiting TOM Explorer	5-14
Compiling Your Service—Final Compile.....	5-15
Testing Your Service	5-15
Using Your Service in an Application	5-15
Chapter 6 Reusing Existing Services in Yours: Containment	
Choosing a Related Standard Service	6-2
Writing the Container Service	6-3
Writing Handler Methods for Low Level Services.....	6-6

Chapter 7 Dealing with Errors

Deciding to Raise, Extend, or Trigger an Error 7-2
 Extending an Error 7-3
 Raising an Error 7-6
 Triggering an Error 7-8

Chapter 8 Creating Service to Initialize Tool

Planning the Approach..... 8-2
 Create Constants and References in Declarations 8-3
 Creating Method Object in OnCreate 8-3
 Checking Required Services in OnInitialize 8-3
 Subscribing to Events in OnInitialize 8-4
 Setting Up TOM Notifications 8-4
 Starting the StartTool Method in OnExecute 8-4
 Continuing to Chain Methods in OnMethodCompleted 8-5
 Executing Last Method in OnSubscribedEvent 8-6
 Creating the Service DLL 8-7
 Creating Service, Tool, Dictionaries in Database 8-7
 Running Service in Visual Basic Debugger..... 8-8

Appendix A Template/Sample Service Code

Complete Code for the Service A-2

Appendix B Container Service Code

Complete Code for Container Service B-2

Appendix C Developing Equipment Services: Using Sample Services

Finding Sample Equipment Services/Tools C-2
 Building Replacement Services Tool Database C-2
 Using Sample (Replacement) Equipment Services C-2
 Examining Sample Level 1 Service C-3
 Examining Sample Level 2 Service C-3
 Examining Sample Level 3 Service C-4
 Removing Samples C-4

Appendix D Developing Help Files for Services: Documentation Kit

Writing Help Files for Your Custom Services D-2

Appendix E Using Testing Services

FTIAttributeForms E-2
 FTISizeInfo E-8

Chapter F Code for Initialize Tool Service

Complete Code for the Init.sample2 Service F-2

Index

About This Manual

Introduction

Topics in This Chapter

Purpose of This Manual, p. viii

Prerequisites and Related Manuals, p. viii

Conventions, p. ix

Information Included in This Manual, p. ix

Service Developer's Guide provides basic information required to start developing Services for STATIONworks.

The manual takes you step-by-step through the process of developing a Service using a sample “template” Service provided with the product.

Purpose of This Manual

This manual is designed for Service developers with background in:

- Coding in Visual Basic
- Knowledge of SECS and/or GEM messaging standards
- Knowledge of VFEI drivers

To develop a driver, which combines a Tool with one or more Services, you should have a copy of the equipment manufacturer's manual.

Prerequisites and Related Manuals

Before you read this manual, you should be familiar with the following manuals for TOM:

- *STATIONworks Tool Deployment Guide*
(or *Using TOM Explorer* in TOM Help file)
- *TOM Object Reference*
(or *Tool Object Model Reference* in TOM Help file)

Companion Manuals/Help Files

If you plan to use Methods from any standard TOM Services in your Service, you can learn more about those Services in:

- *TOM Standard Services Reference*
(or *Standard Services Reference* in TOM Help file)

For details on the handler support routines you use in developing a Service and a quick reference on the handler methods, refer to:

- *Service Developer's Reference*
(or *Authoring a TOM Service* in TOM Help file)

When you are ready to create a Tool in the database or add your Service to the database, you use TOM Builder and refer to:

- *TOM Builder User's Guide*, a distinct Help file in the STATIONworks menu.
- For information on working with Services that interact with FACTORYworks and the MBX, refer to the *STATIONworks Host Service Developer's Guide*.

Conventions

This manual uses the Visual Basic syntax conventions, including *italics* for text that should be replaced, the word **Optional** before appropriate arguments, and **bold** for required literal text.

This manual alters one Visual Basic convention—for convenience it uses the underscore character to continue lines of sample code even in middle of strings or paths, breaking lines of code wherever necessary to fit them in the text column.

`Courier` font distinguishes names of the following in the text: handler methods, handler support routines, Methods, Events, Properties, and Attributes.

Information Included in This Manual

The information in this manual is divided into the chapters described below.

The first five chapters cover developing a Service and Tool from scratch and take it from the first lines of code through the debugging process. Those chapters illustrate the process using a template Service provided with the product. After that, each chapter explores a particular topic, first writing a container Service, then handling errors in a Service. All sample code is included in the appendixes.

Chapter	Description
About This Manual	Explains the purpose of the manual and how it is organized. Also presents a list of related manuals.
1 Getting Started with Samples	Explains where to find the sample Services and applications shipped with STATIONworks and covers how to build a database of the sample Tools those Services/applications use. You may use the same techniques described here to create an actual Tool database.
2 Developing Service Infrastructure	Takes you step-by-step through the process of laying down the foundation for your Service, from conceiving the idea to selecting the controls to coding the general declarations.
3 Writing Your Handler Methods	Takes you through the process of writing the handler methods TOM expects to find in your Service. Covers several commonly used handler support routines.

About This Manual

Chapter	Description
4 Creating a Tool for Your Service	Covers how to create a conceptual Tool for a high level Service (like the one in this manual) and add it to the database. For information on developing drivers, contact Brooks Automations' FASTech Products deployment or LightsOut Software, Inc.
5 Finalizing Your Service	Covers adding your Service to the database, compiling it, and debugging it. Takes you step-by-step through debugging the sample Service using TOM Explorer and the Visual Basic debugger.
6 Reusing Existing Services in Yours: Containment	Explains how you can reuse an existing Service within your Service code by containing the Service, so your Service becomes a <i>Container Service</i> . This technique saves you work when the Service you plan to write closely resembles an existing standard Service.
7 Dealing with Errors	Covers how to use the handler support routines designed to handle errors in your Service.
A Template/Sample Service Code	Provides a complete listing of the template sample Service code.
B Container Service Code	Provides a complete listing of the sample container Service code.
C Developing Equipment Services	Presents information about the sample replacement Services included with STATIONworks. These Services replace the standard TOM equipment (SECS, GEM, VFEI) Services for equipment that is not completely standard. You may need to develop Services like these and can use the samples provided as a starting point.
D Developing Help Files for Services	Introduces the documentation kit provided with STATIONworks, which instructs you on how to develop Help files for your custom Services.
E Using Testing Services	Presents some information on how to use sample testing services included with the product. These samples are intended as guidelines to developing your own testing Services.
Index	Contains a complete index.

Getting Started with Samples

1

Introduction

Topics in This Chapter

Combining Tools and Services, p. 1-2
Establishing Database Components, p. 1-3
Working with Sample Services/Applications, p. 1-4
Building a Database of Sample Tools, p. 1-5
Compiling Sample Services, p. 1-7
Removing Samples, p. 1-8
Understanding Objects in TOM, p. 1-10

This chapter presents vital information for using the sample Services and applications that ship with STATIONworks. It also explores the relationship between Tools and Services.

Combining Tools and Services

Usually, you write a Service to control a Tool. If the Tool is SECS/GEM compliant, you may not even need to write a Service, because you can use standard Services.

However, you may need to write a new Service for a non-standard Tool or write a higher level service, a Service that does not necessarily directly control a piece of equipment.

Say you want to write a higher level service. So, you don't need a Tool for that Service, right? Wrong. You always need a Tool, even though it may be only a conceptual Tool.

The combination of a Tool and a Service is called a TOM driver. A higher level driver (that uses higher level Services) may use a Tool that has Resources that map to virtual devices. For instance, a Resource called *ProcessControlDevice* might be a virtual device for a series of statistical process control Services.

You define your Tool using TOM Builder. You must coordinate certain information that links the Tool to the Service:

- Define a Tool in the database
- Define Resources for the Tool
- Create the Service in the database
- Assign the Service to the Tool in the database
- If your Service requires Attributes, you must add them to the database

Your Service may also require access to existing Dictionaries or a unique Dictionary of its own. To handle this situation, you need to:

- Create your own Dictionaries if you need them—Service Dictionaries and Resource Dictionaries
- Add DataDefs to the Dictionaries
- Assign Dictionaries to Services
- Assign Dictionaries to Resources

This manual presents some basics on developing a sample database (next section) and later ties in how to create DataDefs and Attributes for the Tool; however, for the complete picture on how to develop a Tool refer to the *TOM Builder User's Guide* Help file.

Establishing Database Components

Before you develop a new driver, you must establish the driver database components from the existing database using TOM Builder:

1. When you first load TOM Builder, the database full of existing drivers is located under `\FASTech\Sw\Drivers` and is in the following subdirectories full of several components for each driver:
 - ◆ Dictionaries
 - ◆ Manufacturers
 - ◆ Resources
 - ◆ Services
 - ◆ Tools
2. Under each of these directories, you see `.tbf` (TOM builder file) files for each component. The `.tbf` files have easy to understand names, like `BTU Thermal Process.tbf` or `tomss.Verification.tbf`. You can check each of these components in to a revision control system to keep track of versions of that component. To use the original database, you do not have to build it; it has already been built using these components. But to alter the database, create a new one, or work with one you already have, you must do one of the following:

NOTE Before you proceed to edit the database, you should either check all `.tbf` files into a revision control system or copy them to a backup directory.

- ◆ Alter the components, then build the database
- ◆ Create a new series of components, then build them into a new database

From here, let's take a look at how to create an entirely new database that you can have contain the Tools and Resources required for the sample Services in this manual. You would follow the same steps to create a database for your custom Services.

Working with Sample Services/Applications

After you install STATIONworks, if you installed the Developer version of the software, you can find the sample Services and applications for this manual and the *TOM Application Developer's Guide* in the following directories:

NOTE Sample code that appears in documentation or is included with the product is included for illustration only and is, therefore, unsupported. This software is provided free of charge and is not warranted by Brooks in any way. Brooks Technical Support will accept notification of problems in sample applications, but will make no guarantee to fix the problem in current or future releases.

Samples Documented in Manual Chapters

Sample Code, Manual, & Directory Location	Associated Tool & Location
Template Service (<i>Service Developer's Guide</i> , Chapters 2-5 and Appendix A) <code>\FASTech\Sw\Dev\Samples\Services\demo.vbp</code>	Stepper (not a real tool) <code>\FASTech\Sw\Dev\Samples\Services\Drivers\</code>
Container Service (<i>Service Developer's Guide</i> , Chapter 6 and Appendix B) <code>\FASTech\Sw\Dev\Samples\Container\nv10\nv10.vbp</code>	NV10 <code>\FASTech\Sw\Dev\Samples\Container\Drivers\</code>
Init Service with StartTool Method (<i>Service Developer's Guide</i> , Chapter 7 and Appendix F) <code>\Fastech\Sw\Dev\Samples\StartTool\init.vbp</code>	GenTool (not a real Tool, a generic Tool) <code>\Fastech\Sw\Dev\Samples\StartTool\Drivers</code>
Sample MyRecipe Application (<i>TOM Application Developer's Guide</i>) <code>\FASTech\Sw\Dev\Samples\apps\MyRecipe\myrecipe.vbp</code>	BTU recipe example <code>\FASTech\Sw\Dev\Samples\apps\MyRecipe\Drivers\</code>

Each documented sample comes with a series of *.tbf* files for its required custom Tool. They are in the directories under *..\Drivers* (see preceding table for details). You must build a database containing both these *.tbf* files and those from the Standard SECS Dictionary.

The code for each sample has been shipped as a Visual Basic project file that you must compile. You can compile and run each using the Visual Basic

compiler. To see the effects of the *Template Service* (see table above) you should:

1. Build the database containing its Tool (location indicated in preceding table) using TOM Builder.
2. Compile the Service in the Visual Basic debugger.
3. Then open the stepper Tool in TOM Explorer following the instructions provided in Chapter 5.
4. You use TOM Explorer and the Visual Basic debugger to step through the code and see it print in the Debug window indications of what portion of the code is executing.

Building a Database of Sample Tools

To build the database, you use TOM Builder. Full instructions for how to use TOM Builder are included in the *TOM Builder User's Guide* Help file, available from the Start menu by selecting:

```
Start => Programs => FASTech STATIONworks Beta =>
TOM Builder User's Guide
```

To build a database that contains the sample Tools for the sample Services and/or applications or to build your own database:

1. Start TOM Builder by selecting Start => Programs => FASTech STATIONworks Beta => TOM Builder.
2. Proceed to the `\FASTech\Sw\Dev\DriversCore` directory. In this directory, you find the `Dictionaries` and `Services` subdirectories. The `Dictionaries` subdirectory contains the SECS Standard Dictionary and the `Services` directory contains all standard TOM Services. The `Dictionaries` and `Services` are in `.tbf` files. When you create new Tools or Services, you should use the `Dictionaries` and `Services` in these directories as a foundation for your database. Later, when you build the database using these `.tbf` files (along with the additional ones provided for the samples), the resultant database contains both standard and custom Tools and Services.
3. To include the standard `Dictionaries` and `Services` in your sample database, copy `DriversCore` and the `Dictionaries` and `Services` subdirectories from beneath it to the location where you'd like your new database. **Never work on the original database.** Leave the original intact to be sure you can return to it in the event of a serious error.

You also find the `Manufacturers` directory under `DriversCore` and may copy that directory to your samples database as well. If you copy the `manufacturers`, when you edit the database with TOM Builder, these

manufacturers later appear in the list you can choose from when you add a Tool to the database.

4. Create two additional subdirectories under `Drivers`—`Resources` and `Tools`. Make them parallel to the other directories under your copy of `DriversCore`.
5. To work with the sample Services, copy the sample `.tbf` files from under their `..\Drivers\Dictionaries`, `..\Drivers\Resources`, and `..\Drivers\Tools` directories to the corresponding directories in your new database location. You may want to include all sample Tools in a single database for convenience.
6. To work with a new database that does **not** contain the sample Services, copy any existing Tools or Resources you'd like to use in your new database from `\FASTech\Sw\Drivers\Tools` and `\FASTech\Sw\Drivers\Resources` to the new directories you have created. You can usually identify the Tool and Resource by its name, like `Disco Saw 600.tbf`, the same name for Tool and Resource. In unusual cases, the Tool and Resource have different names, such as the `MBX Tool.tbf` and the `Messaging Resource.tbf`, both required for *ProtocolMBX*.
7. Indicate where you would like TOM Builder to find the new database components (`.tbf` files) by selecting `File => Directory Locations` from the menu bar and filling in the `Component File Directory Location` field. (You can use `Browse` to find the correct path to your new directory.) The path to your new directory should go down to the `DriversCore` level, so if you put your copy of `DriversCore` under ***D:\Databases***, the path to the directory should be `D:\Databases\DriversCore`.
8. Once you have indicated the location of the directories, the sample Tools appear with the others under the `Tools` tab in the right-hand pane of TOM Builder (also called the `Component View`).
9. To build the database, return to the menu bar and select `File => Build Database...`
10. When the dialog box displays, enter the name of the database to build. The database name must have a `.mdb` extension.

Give the database a few minutes to build. When TOM Builder has finished, it displays a message on the status bar along the bottom of the window indicating the database build is complete.

Compiling Sample Services

NOTE When you build the samples, you do not build the sample shown in this manual. Instead, you compile and debug the code from this manual in Chapter 5, where you step through the debugger to see it perform.

You do not have to compile the samples to use them. However, you can “build” a large number of the samples provided by executing the following steps:

1. Add the *vbpath* environment variable to your system properties. Set this environment variable to where the build script should find Visual Basic installed on your machine. For instance, if Visual Basic is installed in `c:\VisualBasic`, that is what you should set this variable to.

You set environment variables by selecting `Start => Settings => Control Panel`, then double clicking on the `System` icon to open the `System Properties`, and selecting the `Environment` tab to find the system variables.
2. Edit the *database.bat* file (under *FASTech\Sw\Dev\Samples*) to be sure the following variables point to the correct directories on your hard drive:

```
set TOM_BUILDER_DIR=..\..\Components\TomBuilder1
set TOM_DRIVER_DIR=..\..\Components\TomBuilder\TmpDriver
set CORE_DRIVER_DIR=..\DriversCore
```
3. Run the *build.bat* script (under *FASTech\Sw\Dev\Samples*). It compiles the following samples and places the executable, DLL, database (*.mdb*), and Help files in the *FASTech\Sw\Dev\Samples\bin* directory:

Sample Application

MyRecipe.exe
MyRecipe.mdb

Sample Container Service

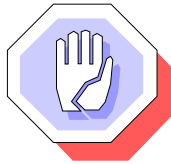
nv10.dll
nv10.mdb

If you compile the *.dll* on your machine, it is automatically registered. Otherwise, be sure to register the *nv10.dll* with *regsvr32.exe* to use this Tool.

Replacement Equipment Services

replace.mdb *replss2.dll*
replss1.dll *replss3.dll*

The replacement equipment Services are examples of Services that would each replace a particular standard TOM Service for a piece of equipment that is not quite standard. You may need to write this type of Service. For more information on these writing such Services and for more information on the samples provided, refer to Appendix C.



CAUTION

Writing replacement Services is intended for advanced developers. Brooks Automation recommends you step through this manual chapter by chapter in sequence before attempting to write a replacement Service.

Help File for Replacement Equipment Services

replace.hlp
replace.cnt

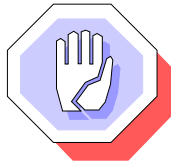
4. Run the *register.bat* script (under *FASTech\Sw\Dev\Samples\bin*). It registers the DLLs for the sample Tools.

Removing Samples

When you have finished using these sample Services, save any you would like to keep in a new location. If you compiled the samples, to remove both the source and the compiled files from your machine, execute the *cleanup_all.bat* script located in *FASTech\Sw\Dev\Samples*.

Sample Services for Testing/Debugging

Another set of sample Services that you may find useful after you have become a more advanced user of this product are available in the *FASTech\Sw\Dev\Samples\Misc\Level5* directory. Here, you find the source code for some level 5 Services designed to be useful during testing and debugging. You can also open the source code for these Services and customize them.



CAUTION

Sample code that appears in documentation is included for illustration only and is, therefore, unsupported. This software is provided free of charge and is not warranted by Brooks in any way. Brooks Technical Support will accept notification of problems in sample applications, but Brooks will make no guarantee to fix the problem in current or future releases.

These Services are contained within a single *.vbp* project named *FTIdev5.vbp*. Inside the project, you find the following *.cls* files:

- *5AttrFrm.cls*—A Service that displays a form where you can set/alter Service Attribute settings quickly during testing.
- *5SizeInf.cls*—A Service that indicates the amount of RAM the code you are running is using.

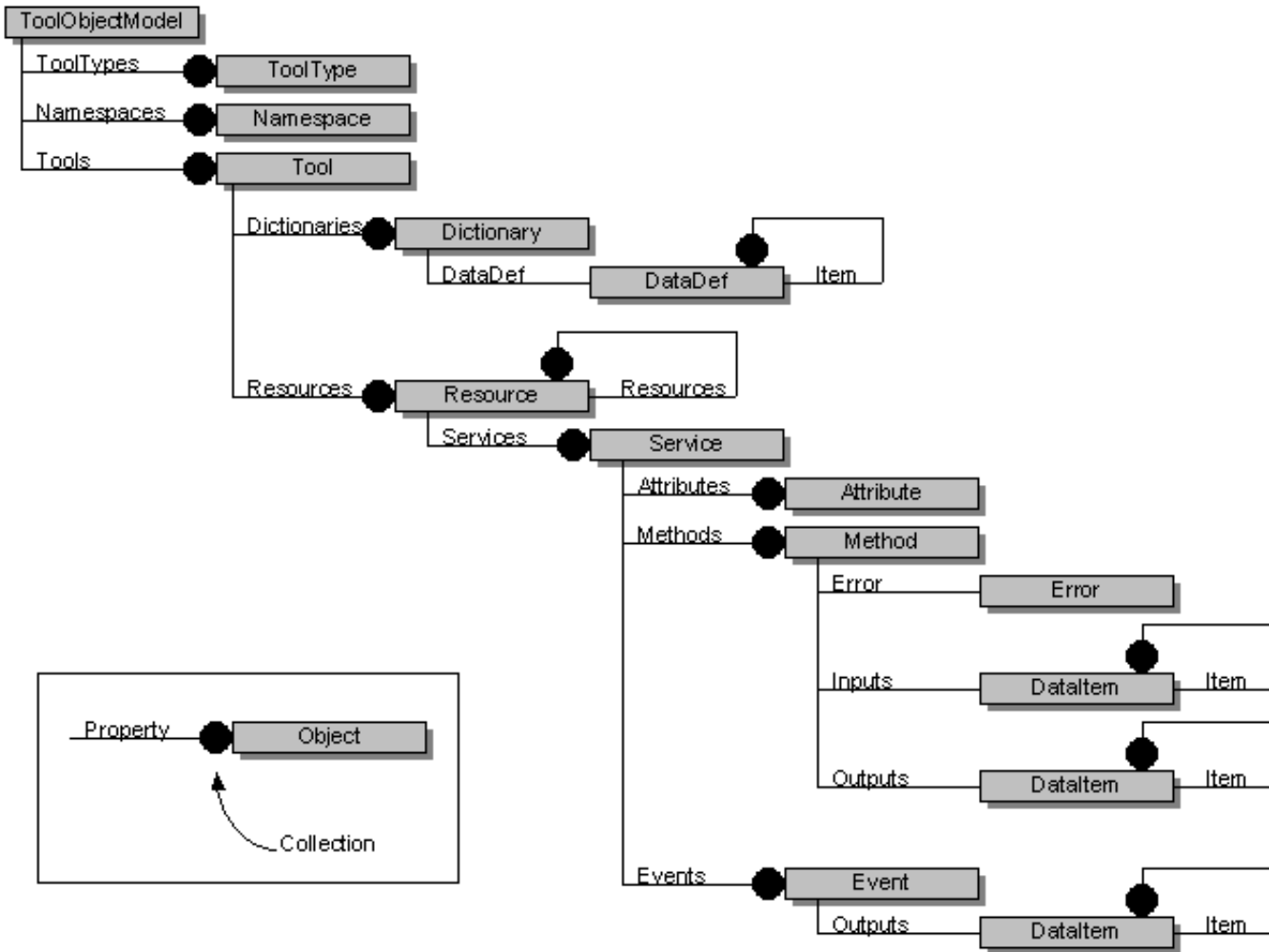
These files contain the source code for the following *.tbf* files, which you can find in the *FASTech\Sw\Dev\Samples\Misc\TBFs* directory:

- *FTIdev5.FTIAttributeForms.tbf*
- *FTIdev5.FTISizeInfo.tbf*

To use these Services, you must add these files to your *Services* directory under *Drivers* and rebuild the samples database. For details on using these Services, refer to Appendix E.

Understanding Objects in TOM

Before you proceed, you should understand the Tool Object Model (TOM). Acquaint yourself with it using TOM Explorer and following the Help file or referring to the *STATIONworks Tool Deployment Guide*.



You may also want to refer to the *TOM Object Reference*.

Introduction

Topics in This Chapter

- Defining the Service's Role in the Application, p. 2-2
- Writing a Custom Service—Steps to Take, p. 2-2
- Creating the Visual Basic Project, p. 2-4
- Adding Custom Controls to Your Project, p. 2-4
- Adding Required Files to Your Project, p. 2-5
- Creating References for Your Project, p. 2-5
- Assigning the Project Name and Title, p. 2-6
- Creating a Class Module and Declaring Service Name, p. 2-10
- Understanding References, Variables, & Constants Required, p. 2-12
- Creating References, Variables, and Constants, p. 2-13
- Fitting Services Together in Visual Basic Project, p. 2-14

This chapter covers the setup to get you started developing a TOM Service. It refers to the template Service (a dummy Service) provided under `\FASTech\TOM\Dev\Samples\Services\demo` and the Tool that accompanies it under `\FASTech\TOM\Dev\Samples\Services\Drivers`.

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

The complete code for the Service's class is listed in Appendix A.

Defining the Service's Role in the Application

Before you write your Service, you should decide exactly what you want the Service to do. Then you are in a position to see whether or not some of the tasks the Service should perform are available in the standard Services of TOM.

For instance, if your Service needs to communicate with equipment to carry out its tasks, it might be able to use methods and events in *SecsLoopBackDiagnostic* and *ProtocolSECS* to carry out those tasks.

While you develop the flowchart for your Service, prepare a list of other Services and their methods/events that you would like your Service to work with. It's a good idea to have this information handy before you start your Service.

Writing a Custom Service—Steps to Take

Any Service you write for TOM can be an in-process or out-of-process OLE server. You embed the OLE server in a TOM application to use it.

NOTE If you are not familiar with how to write an OLE server, you should look up how to in the *Programmer's Guide* and the *Professional Features* manual for *Microsoft Visual Basic*.

To write a Service, when you start up Visual Basic, create a Visual Basic project that is an ActiveX DLL (in-process) or ActiveX EXE (out-of-process). Then carry out the tasks in each of the sections that follow, outlined below:

Set Up the Service Project

- Add any custom controls your Service requires to the Visual Basic Toolbox in your project.
- Create references to the TOM control and (optionally) the FASTech WinSECS control.
- Add the required files so that you can use the various handler support routines (like APIs) in TOM (such as *handler.bas*).
- Add the *Smain.bas* file provided with TOM to your Visual Basic project. This file defines the required Sub Main routine.

Write the Service Code

- (recommended) Define constants for other Services yours might use and the methods, events, and parameters of those Services.
- Create a class module and declare the Service name in it. You need a separate class module for each Service, but should put all Services you want to distribute in a single .DLL in the same Visual Basic project.

- Define variables required by your Service.
- Define references to TOM objects and handlers.
- Write handler methods that trigger on actions and write them in the order presented below (not all of those listed are required; for more information, refer to *Writing Required Handler Methods That TOM Triggers*, p. 3-2):
 - ◆ OnCreate
 - ◆ GetAttribute
 - ◆ LetAttribute
 - ◆ OnInitialize
 - ◆ OnExecute
 - ◆ OnMethodCompleted
 - ◆ OnSubscribedEvent
 - ◆ OnVerify
 - ◆ Version
 - ◆ OnTerminate

TOM uses these handler methods to run your Service. In addition, you can write two other handler methods that are not illustrated in this manual:

- ◆ OnStartup
- ◆ OnTimerEvent
- Write additional private functions your Service requires—ones the handler methods call that extend the action that occurs in each handler method. For instance, `OnVerify` might call a private function to carry out some deeper detail of the verification process or `OnCreate` might call a private function to create events within the Service.

Make the DLL

- Make an the DLL or EXE (using the Visual Basic menu selection).

Modify the Database

- Add your Service to the TOM database.
- In the database, associate the Service with Resources of the Tool.

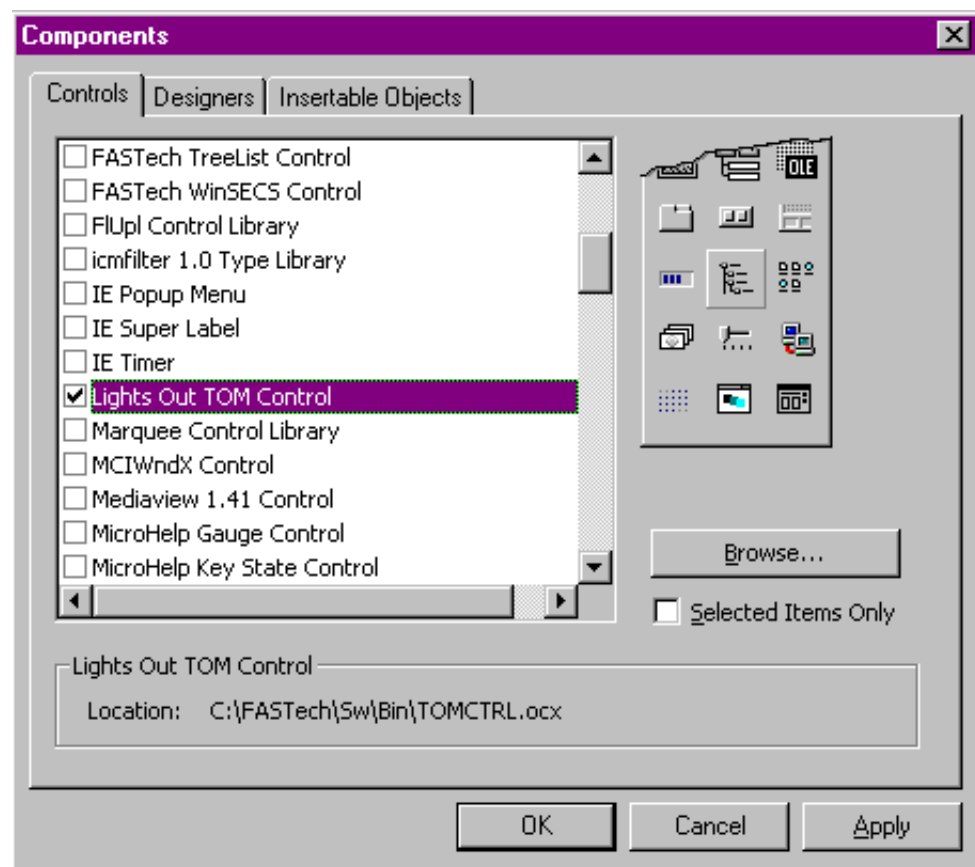
Creating the Visual Basic Project

When you first open Visual Basic and the New Project dialog appears, select `ActiveX EXE` for the project type and click `Open`.

Adding Custom Controls to Your Project

Your Visual Basic project must include some custom controls to work with the Tool Object Model (TOM). You add these controls:

1. Open the `Components` box (by selecting `Project => Components`).
2. Select the `Lights Out TOM Control`:



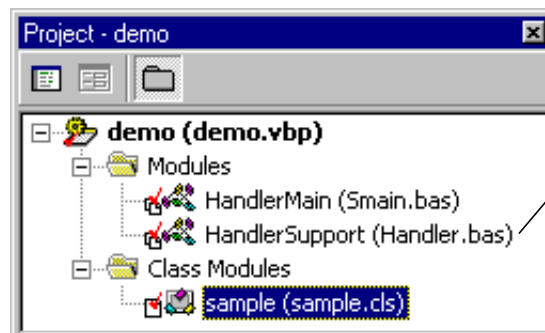
This control is required in all STATIONworks/TOM Services. This action adds the TOM control to the Visual Basic Toolbox in your project. It also makes the TOM control available for your Service to use.

3. If you want to use the secs handler support routines, you should also select the `FASTech WinSECS Control`.

Adding Required Files to Your Project

In addition to the custom controls you need to work with TOM, you also need particular files to ensure you have access to handler support routines. You find those files in the `\FASTech\STATIONworks\Dev\Services` directory:

1. To be able to work with the `srv` handler support routines from TOM, add the `Handler.bas` file provided with TOM to your project.



This file provides handler support routines

2. If you want to use the `secs` handler support routines from TOM, add the `SecsL1.bas` file to your Visual Basic project.

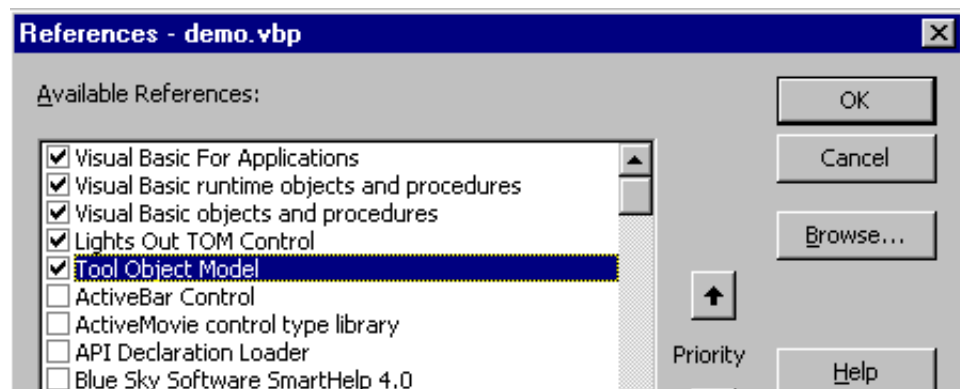
In addition, to be sure your project has the `Sub Main` routine:

3. Add the `Smain.bas` file provided with TOM to your Visual Basic project.

Creating References for Your Project

Your Visual Basic project must include some references to work with TOM. To create those references:

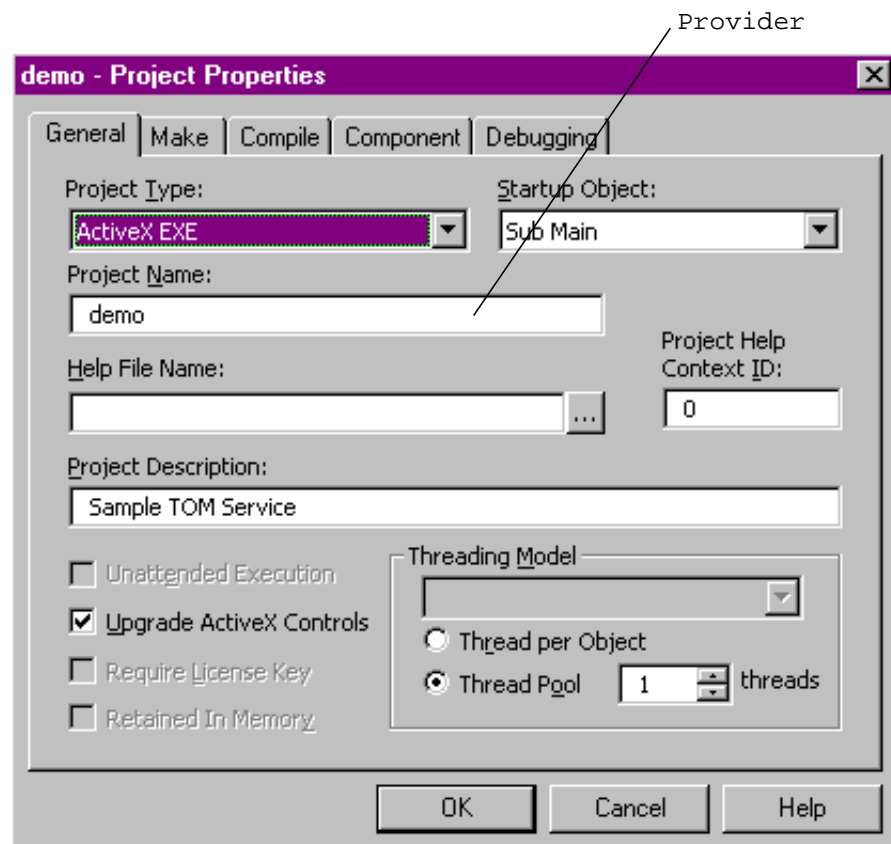
1. Open the `References` box (by selecting `Project => References`).
2. In all TOM Services, you must create references to the `LightsOut TOM Control` and the `Tool Object Model`.



Assigning the Project Name and Title

When you save the project and assign it a name, such as *demo.vbp*, you should also:

1. Open the Project Properties box (by selecting Project => <app> Properties).
2. Set the Project Name (under the General tab) to the same name as the .vbp file. Later, you use this name as the Service's Provider when you add the Service to the database.

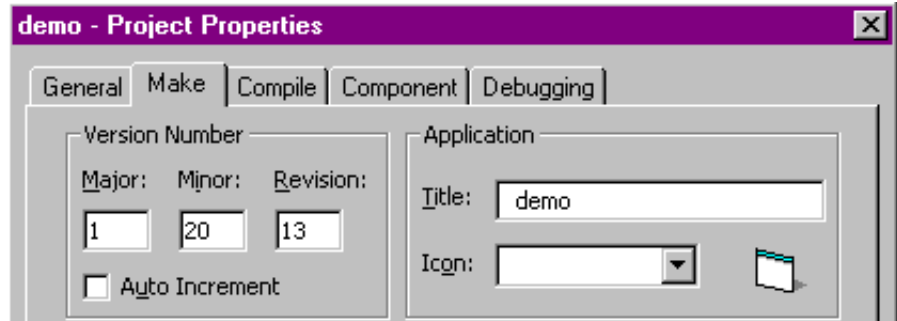


NOTE TOM Tip — Projects

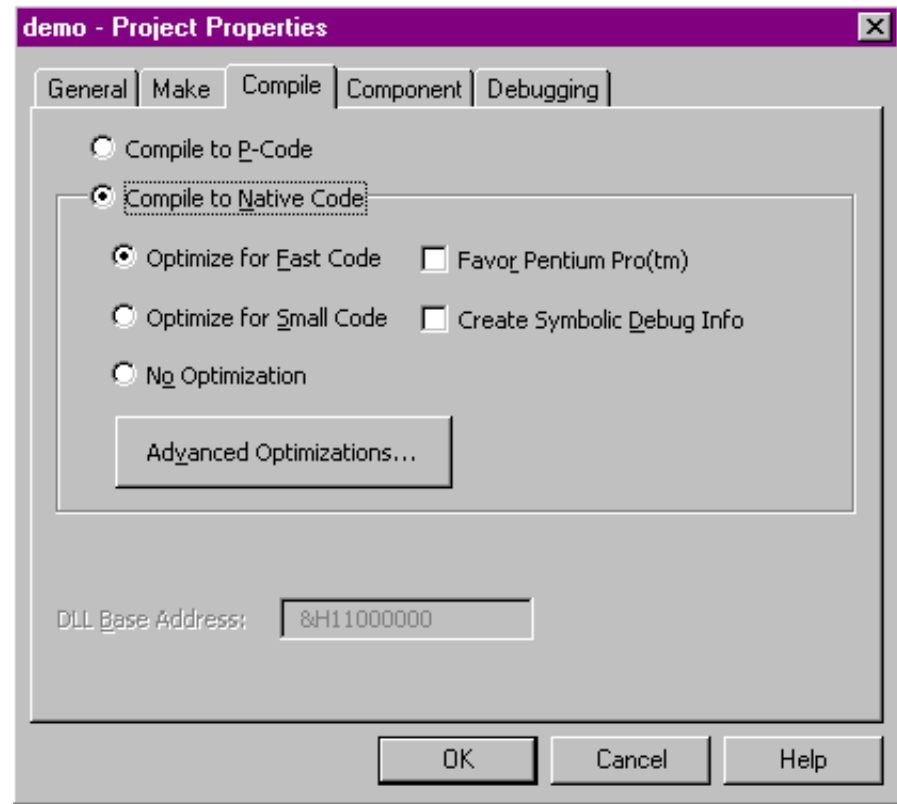
Add your assigned prefix to the name of the project (such as *MY* to produce *MYdemo.vbp*) to identify the project as yours. You should select two or three characters to be a unique prefix for your organization.

3. Note that the Startup Object is Sub Main, which you have access to because you included *Smain.bas* in the project.

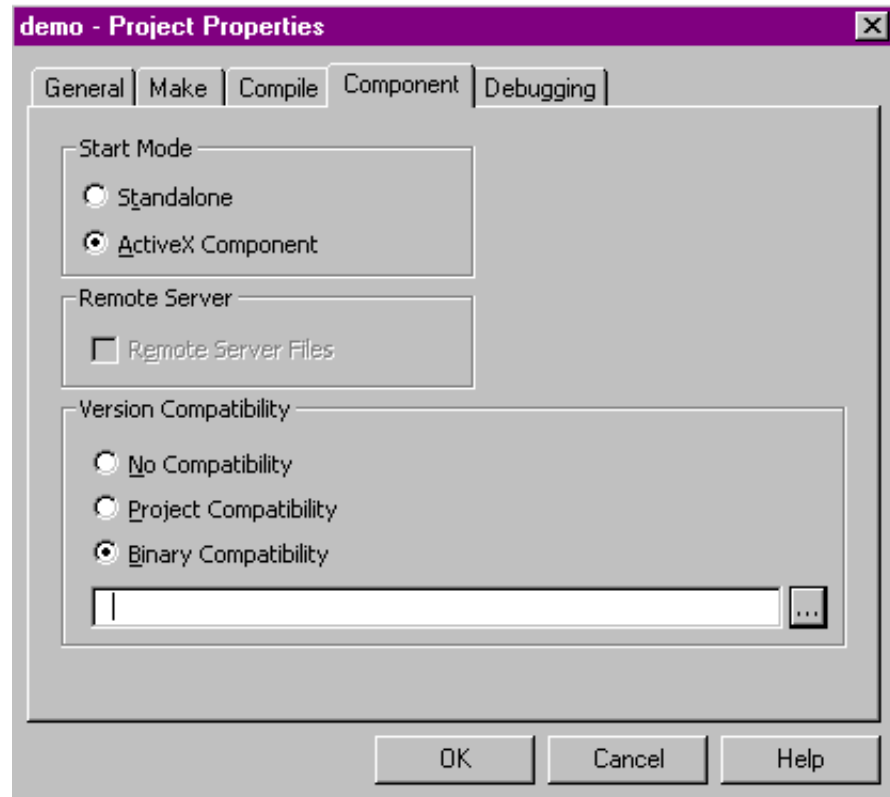
4. Later, you may want to add the name of the `Help File` and the `HelpContextID`.
5. Also, be sure the `Project Type` is `ActiveX EXE`.
6. Under the `Make` tab, be sure to set the `Title` to the name of the `.vbp` file also.



7. Under the `Compile` tab, you should select `Compile to Native Code` and `Optimize for Fast Code`.



- Under the Component tab, select ActiveX Component under Start Mode.



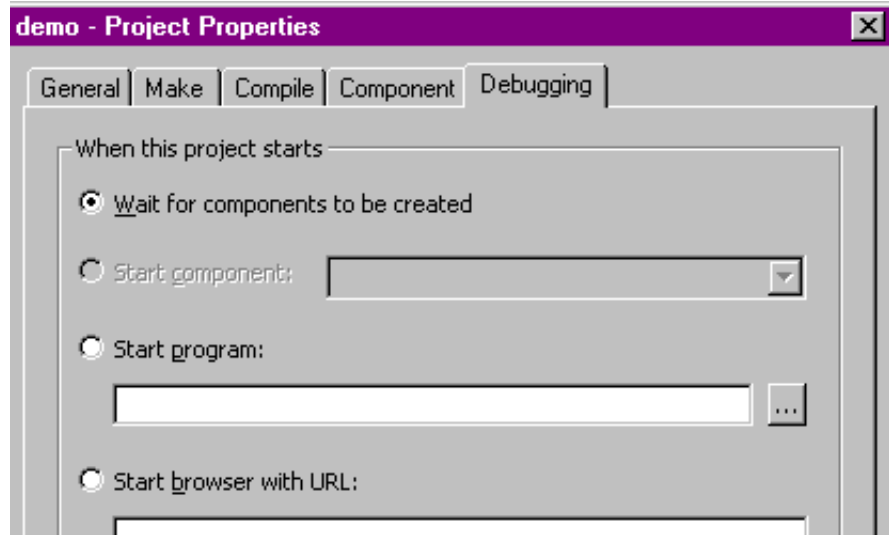
- Under Version Compatibility, select Binary Compatibility.



CAUTION

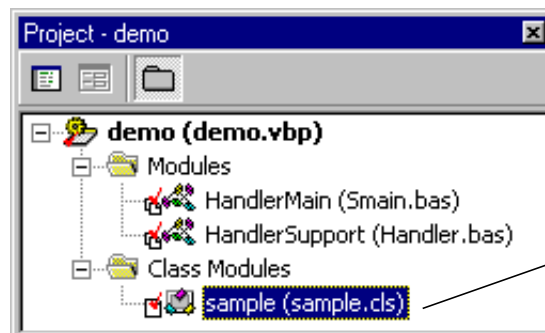
If you do not set Version Compatibility to Binary Compatibility, when you try to use your custom Service on another machine, you will not be able to successfully register the *.dll*.

10. Under the `Debugging` tab, for purposes of running the demo, select `Wait for components to be created`.



Creating a Class Module and Declaring Service Name

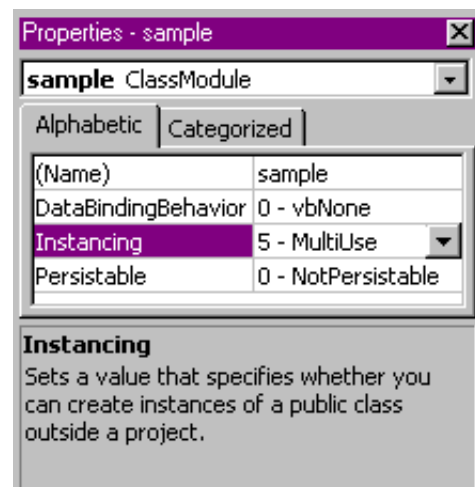
1. In Visual Basic, create a class module and assign it a name that you can easily identify it by. For instance, let's name the sample Service class *sample.cls*. *This name should be distinct from the project name.*



class file for
sample Service

2. Open the class properties by right clicking on the class name and selecting Properties from the menu.
3. In the Name property for the Visual Basic class module, enter the name of the class (refer to next illustration).

The value you assign to the Name property of the Visual Basic class becomes the Name property of the Service you develop.



CAUTION

When you assign names to new Services that do not replace any existing Service, you should add a unique prefix for your organization to ensure you do not inadvertently overwrite an existing Service.

4. When assigning the value of the `Instancing` property, be sure to select 5 - `MultiUse`.



CAUTION

If you are unable to execute the previous steps, it is probably because you are not using the Professional or Enterprise Edition of Visual Basic 5.0.

5. Under `General Declarations`, declare the Service name as a private constant so that the Service name later appears in the lists of Services the TOM Explorer and the TOM Builder display:

```
Private Const SERVICE_NAME = "filename.ExampleService"
```

The constant should be called `SERVICE_NAME` or something similar. Brooks recommends that the *filename* be the name of the Visual Basic project that contains your Services as well as the name assigned as the Service's `Provider` in the TOM database. You should be sure to use a prefix in front of the name of every file you produce to ensure its name is unique.

NOTE

TOM Tip—Constructing `SERVICE_NAME` in Code

If you want to later be able to change the name of the .DLL without having to make changes to your code, you can create a local variable called `SERVICE_NAME` and have a class level function named `Initialize` that sets the local variable as follows:

```
Private Sub Class_Initialize()  
    SERVICE_NAME = App.Title + TypeName(Me)  
End Sub
```

This technique saves you from recoding your Service if you change the project's name.

Understanding References, Variables, & Constants Required

In the *General Declarations* section of your class module, you create global constants, global variables, and references to TOM objects.

What Kinds of References Are Required?

Your class code should declare references to any TOM objects under General Declarations. For instance, it should create references to:

- (required) The Service object that “owns” this class
- (as needed) Any other Service this one uses
- (as needed) Other types of TOM objects, such as DataDefs and Attributes

What Kinds of Variables Are Required?

Your Service should have the following global variables:

- (recommended) A String for the name of your Service
- (recommended) A Boolean that indicates whether or not full verification is on
- (as needed) Variables for Attributes of your Service or Attributes of another Service that yours uses

What Kinds of Constants Are Required?

In the *General Declarations* section of your class module, you should create constants for the following in your Service:

- (recommended) The name of your Service, `SERVICE_NAME`. (see *Creating a Class Module and Declaring Service Name*, p. 2-10)
- (recommended) Names for the Methods, Events, and Attributes of your Service
- (recommended) Names of any other Services yours uses
- (recommended) Names for the Methods, Events, and Attributes of each Service you want to use within your Service
- DataDefs of this Service or another Service yours uses

NOTE

TOM Tip—Defining Constants Multiple Services Can Use

The Visual Basic project for a series of Services that would belong to a single .DLL should use a *def.bas* file to define global constants so that all those Services can access those constants. Add your unique prefix to the name of *def.bas* to identify the family of Services in that .DLL, such as *MYdef.bas*.

Creating References, Variables, and Constants

In your class module's *General Declarations* section, you create constants, variables, and references your Service needs:

1. (recommended) Create a global string to contain the Service's name:

```
Private SERVICE_NAME As String
```

2. Create a reference to the Service you are creating as a `tom.Service` type object:

```
Private m_oService As tom.Service
```

3. Create a reference to any other Service this Services uses and make each a `tom.Service` type object. For instance, to be able to work with the *SecsLoopbackDiagnostic* and *ProtocolSECS* Services, create references to them:

```
Private m_oLoopback As tom.Service  
Private m_oProtocolSECS As tom.Service
```

(Optional) You might also find it convenient to have constants for those Services:

```
Private Const SRV_LOOPBACK = "SecsLoopbackDiagnostic"  
Private Const SRV_PROTOCOLSECS = "ProtocolSECS"
```

4. (Optional) You might need a Boolean variable for saving the `FullVerification` argument later passed in by TOM indicating whether or not full verification is on; to use it in more than one handler method, you can create it as a global:

```
Private m_oFullVerification As Boolean
```

5. (Optional) Create global constants for referring to your Service's Methods:

```
Private Const METH_METHOD1 = "Method 1"  
Private Const METH_METHOD2 = "Method 2"  
Private Const METH_METHOD3 = "Method 3"  
Private Const EVENT_CONNECT = "Connect Event"
```

6. (optional) Create global constants for referring to your Service's DataDefs:

```
Private Const DD_DD1 = "DataDef1"  
Private Const DD_CHILDA = "ChildDataDefA"  
Private Const DD_CHILDB = "ChildDataDefB"  
Private Const DD_DD2 = "DataDef2"
```

7. Create global variables for your Service's Attributes or Attributes of another Service that yours uses:

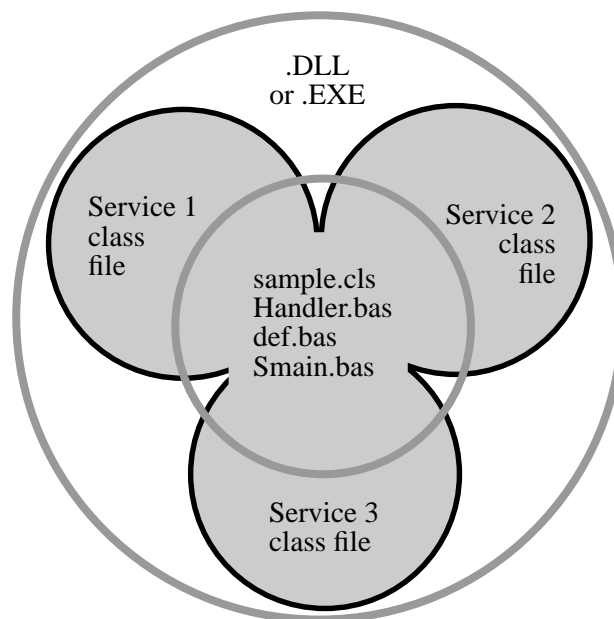
```
Private Const ATT_EVENT_ENABLED = "ToolEventEnable"
```

8. (Optional) The sample Service creates constants for use in sequencing Methods. You might find the need for such constants also:

```
Private Const CaseStep1 = 1
Private Const CaseStep2 = 2
Private Const CaseStep3 = 3
Private Const CaseEnd = 4
```

Fitting Services Together in Visual Basic Project

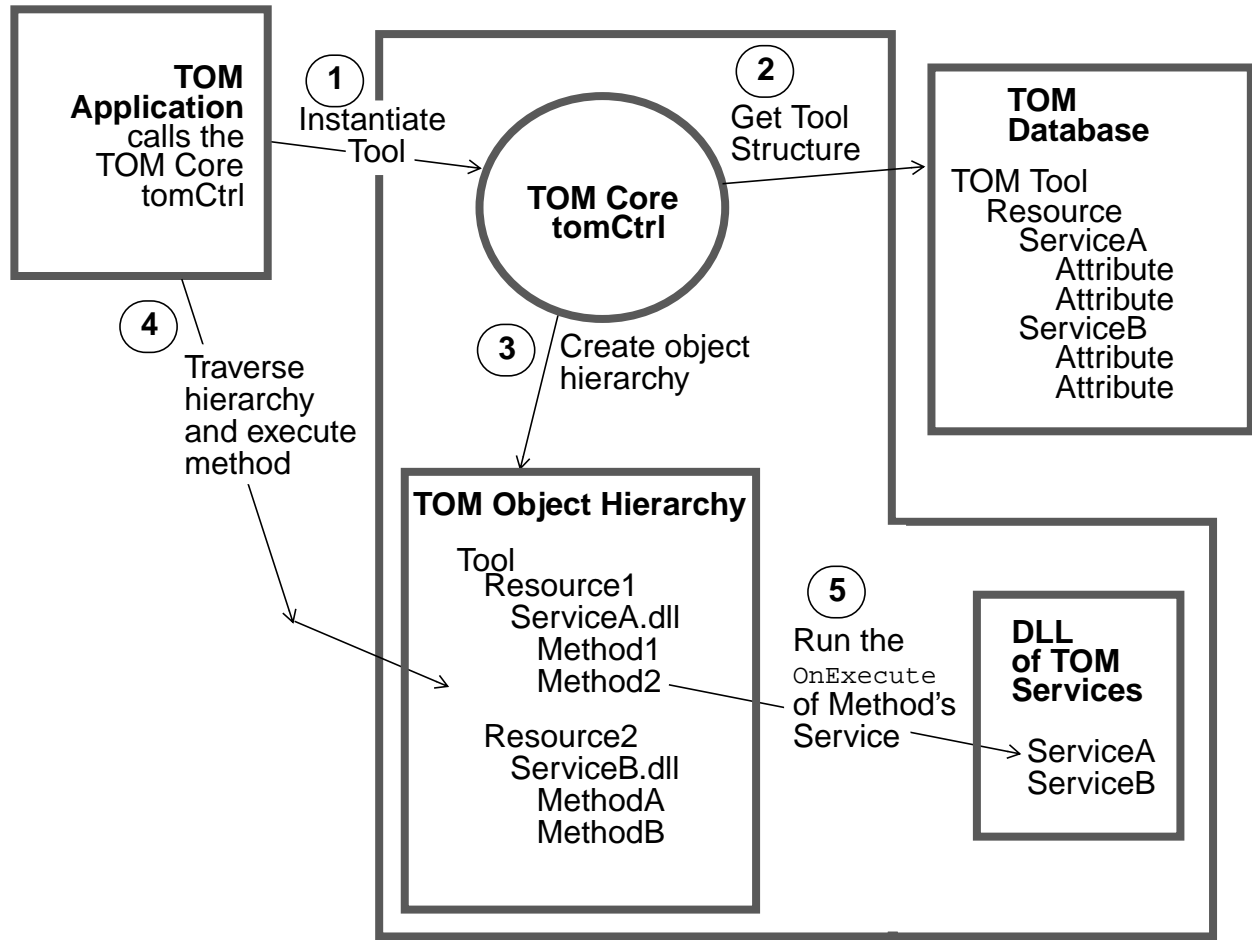
The illustration below shows how multiple Services and the other pieces of a Visual Basic project fit together in a .DLL or .EXE. A series of related Services might be in the same .DLL or .EXE file.



The .DLL or .EXE file contains a .cls file for each Service you have written. It also includes all the .bas files required:

- Any *def.bas* files you need.
- *Smain.bas*, which is required.
- *handler.bas*, which contains the handler support routines.

The next illustration shows how the TOM core is involved in running your Services. You use the Services in a TOM application.



1. When the application instantiates a Tool, it accesses the TOM Core tomCtrl.
2. The TOM Core determines the structure of the TOM Tool—the Resources and Services that make up that Tool—based on the information in the database.
3. TOM creates a copy of the object hierarchy and puts it in memory.
4. If your application tries to run a Method of a Service the Tool uses, it traverses the object hierarchy in memory and runs the Method.
5. TOM calls the `OnExecute` of that Service, one of the handler methods in each Service.

From there, the application proceeds. Steps 4 and 5 repeat every time the application executes a Method in a Service.

Introduction

Topics in This Chapter

- Writing Required Handler Methods That TOM Triggers, p. 3-2
- Understanding the OnCreate Handler Method, p. 3-5
- Writing the OnCreate Handler Method, p. 3-6
- Defining Method Objects for Your Service in OnCreate, p. 3-9
- Defining Event Objects for Your Service in OnCreate, p. 3-11
- Writing the LetAttribute Handler Method, p. 3-14
- Writing the GetAttribute Handler Method, p. 3-15
- Writing the OnInitialize Handler Method, p. 3-16
- Writing the OnExecute Handler Method, p. 3-20
- Executing Existing Methods in OnExecute, p. 3-22
- Writing the OnMethodCompleted Handler Method, p. 3-25
- Writing the OnSubscribedEvent Handler Method, p. 3-30
- Triggering Your Service Event, p. 3-31
- Writing the OnVerify Handler Method, p. 3-33
- Verifying a Service—The Nuts and Bolts, p. 3-36
- Writing the Version Handler Method, p. 3-40
- Writing the OnTerminate Handler Method, p. 3-41
- Writing a Terminate Class Method, p. 3-41

This chapter covers how to develop the actual Service code. It shows developing the handler methods TOM requires your Service to have. This chapter refers to the template Service (a dummy Service) provided with TOM under `\FASTech\TOM\Samples\Services\demo` and the Tool that accompanies it under `\FASTech\TOM\Samples\Services\Drivers`. The complete code for the Service's class is listed in Appendix A.

Writing Required Handler Methods That TOM Triggers

Now you are ready to start generating handler methods. Handler methods are those Visual Basic methods required in your Service code. They are distinct from TOM Methods (with a capital M) that your Services defines and/or executes.

In a TOM Service you are required to include several handler methods:

- `OnCreate`
- `OnInitialize`
- `OnExecute`
- `OnVerify`
- `OnTerminate`
- `Version`

The TOM control uses these handler methods when it executes your Service.

If your Service executes one or more Methods of another Service, it also requires the following handler method:

- `OnMethodCompleted`

If your Service has Attributes, you should also have two other handler methods:

- `GetAttribute`
- `LetAttribute`

If your Service subscribes to Events in other Services, you need another handler method :

- `OnSubscribedEvent`

If your Service is among a series of Services for a particular Tool, and you would like some actions to begin automatically when the Tool starts up, you should also have a handler method called:

- `OnStartup`

If your Service needs a timer you can create it with `srvCreateTimer` (a handler support routine); when you use such a timer, your Service must have another handler method called:

- `OnTimerEvent`

When Does TOM Execute Handler Methods?

How does TOM work with the handler methods? When you create a tool (in TOM Explorer or another application) that your Service is associated with, TOM runs your `OnCreate` and `OnInitialize` handler methods. TOM retrieves attributes from the database (or the registry) after running `OnCreate` and before running `OnInitialize`.

When all Services associated with the Tool have been initialized (TOM has executed their `OnCreate` and `OnInitialize` handler methods), TOM runs

this Service's `OnStartup` handler method (if one exists). By having `OnStartup`, a Service can automatically begin executing Methods of another Service rather than waiting for direction from a TOM application. However, `OnStartup` is not required and Brooks discourages indiscriminate use of it.

NOTE **Minimize Actions in `OnStartup` and Prominently Document Actions in `OnStartup`**

In general, you should code your Services so that after initialization, they take action when told to do so, rather than automatically. Since actions set up in `OnStartup` occur automatically after initialization, you should minimize use of `OnStartup` and *always prominently document* all actions that occur there.

When you (through, for example, TOM Explorer) try to execute a Method of your Service, TOM runs your `OnExecute` handler method.

Your Service can make a copy of another Service's Method and use it. In TOM copying an object is referred to as *cloning*. If your Service clones a Method of another Service (or one of its own Methods) and then executes it, when the Method completes, TOM calls your `OnMethodCompleted`.

If your Service has Attributes, when the application (TOM Explorer) reads the `Value` property of an Attribute object, it runs your `GetAttribute` handler method. When the application (TOM Explorer) needs to assign a value to the `Value` property of an Attribute of your Service, it calls your `LetAttribute` handler method.

If another Service's Event is triggered and you have subscribed to that Event in your Service, TOM calls your `OnSubscribedEvent` handler method when the Event triggers.

When you try to verify your Service, as you would in TOM Explorer by right clicking on the Service and selecting `Verify` from the pulldown menu that appears, TOM runs the `OnVerify` handler method.

When you remove a Tool by going to the `Object` menu, right clicking on the tool, and selecting `Remove`, TOM calls your `OnTerminate` method. TOM also calls `OnTerminate` when you exit TOM Explorer.

Order of TOM Calls to Multiple `OnCreates` and `OnInitializes`

When TOM calls the `OnCreate` handler method for multiple Services, it calls them by level, starting at level 0. TOM first calls `OnCreate` for each Service at level 0, then level 1, then level 2, and so on, through level 5. Within a given level, TOM does not call `OnCreate` in any particular order.

The same order applies to `OnInitialize`.

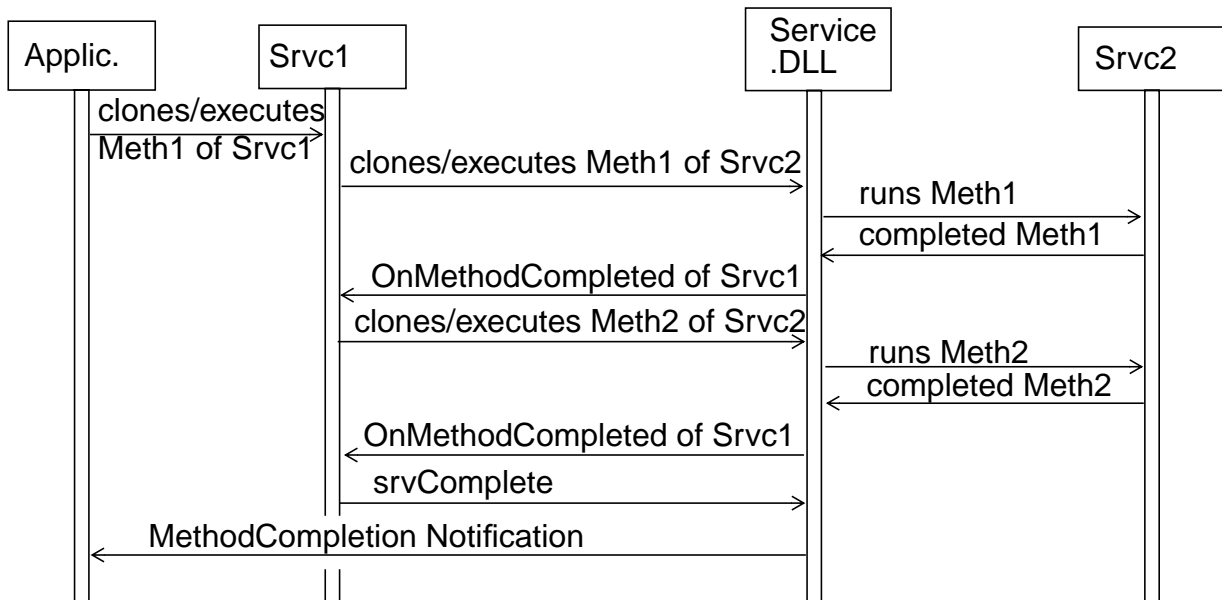
For information on Service levels, refer to the STATIONworks Help file or to the *STATIONworks Tool Deployment Guide*.

Relationship between OnExecute and OnMethodCompleted

What is the purpose of OnMethodCompleted? In TOM whenever your Service clones and executes a Method of another Service, it needs to know when the cloned Method completes (receive a notification).

When the cloned Method completes, TOM runs your Service's OnMethodCompleted. This means that every time your Service clones and executes a Method object, TOM calls your OnMethodCompleted afterwards. So if your Service clones and executes several Methods objects, TOM *cycles through* OnMethodCompleted for each cloned Method.

This handler method is designed to work with Methods you clone and execute. You should always clone a Method before executing it, regardless of where it is defined. The following illustration shows what happens when a Method of *Srvc1* clones and executes several Methods of *Srvc2*.



Terminating Service Action

When you try to exit your Service code, TOM calls your OnTerminate handler method.

Required Class Method

In addition to the handler methods, your Service should have a Class_terminate method to terminate the class when the OLE server stops running.

You see how to write this class method as well as the handler methods in the sections that follow.

Understanding the OnCreate Handler Method

TOM triggers the `OnCreate` handler method when an object of this Service's class is created.

Each time your program creates an instance of the object, after it executes `OnCreate`, TOM defines Attributes for the object instance. When it defines those Attributes, it gives them the values from the registry first. If there is no value for that attribute in the registry, TOM retrieves it from the database.

NOTE **Tip—Working with TOM Attributes**

Since TOM defines Attributes after it runs `OnCreate`, you should not attempt to work with Attributes in the `OnCreate` handler method. Save those actions for after `OnCreate` completes and take them in the `OnInitialize` handler method.

Every Service must have an `OnCreate` handler method. So, when you load a Tool, for every Resource of the Tool and every Service the tool uses, TOM executes an `OnCreate` handler method. Since each Resource normally has several Services, TOM usually executes several `OnCreate` handler methods for a single Resource, in random order. If your Service needs to work with another Service, that Service's object may not even have been created when your `OnCreate` handler method runs. So, the rule on working with other Services in `OnCreate` is:

NOTE **Tip—Working with Other Services**

Since other Services may not yet exist when your Service runs, do not take actions involving another Service in `OnCreate`. Save those actions for the `OnInitialize` handler method.

Do not unnecessarily postpone defining any objects you should generate in `OnCreate`. If an initialization does *not* require Attribute values from the database or access to another Service, you should not postpone it.

Writing the OnCreate Handler Method

In `OnCreate`, your Service should:

1. Receive a reference to your Service as an argument.
2. Save the reference to the Service.
3. Initialize any other data your Service requires.
4. (If required by your Service) Retrieve a reference to the Service Specific area in the Dictionary.
5. (If required by your Service) Load DataDefs from the Service Specific area into memory.
6. Create Methods for the Service.
7. Create Events for the Service.

The details on each step follow:

Pass Reference to Service to OnCreate

1. When you write the `OnCreate` handler method, you set it up to accept a single argument that is the TOM Service passed to it by value:

```
Public Sub OnCreate(ByVal Service As tom.Service)
```

Save a Reference to the Service Object

2. The `OnCreate` handler method should save a reference to the Service object passed to it:

```
' Save Service reference  
Set m_oService = Service
```

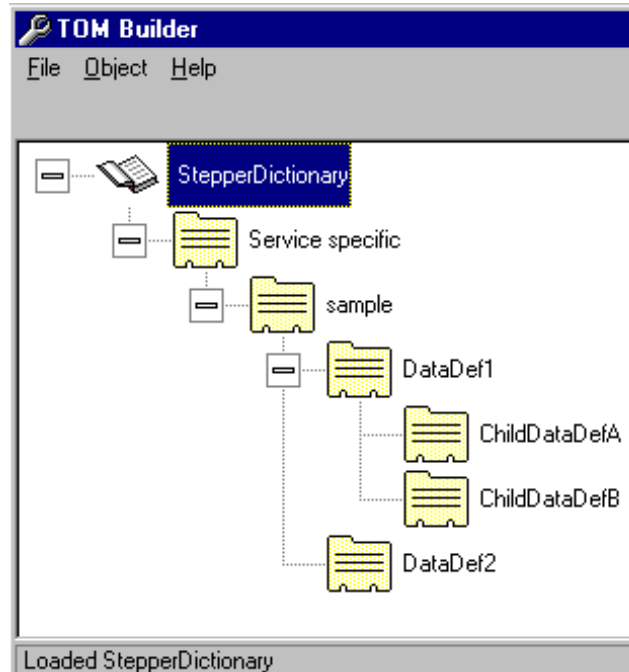
Initialize Other Objects

3. If you want to initialize any other data your Service requires (except Attributes), you should initialize it next.

Create Service Specific Area and DataDefs in Dictionary

- If you would like to retrieve the values of any DataDefs from the database, you must have created them into the Dictionary first. For more information refer to *Creating DataDefs*, p. 4-10.

You can use TOM DB Editor or TOM Builder to create them. The structure of the DataDefs used by the sample service appears in TOM Builder as shown below:



Load DataDefs from Service Specific Area

You load DataDefs into memory in `OnCreate` using two TOM handler support routines called `srvServiceDataDef` and `srvLoadDataDef`.

- You use the `srvServiceDataDef` routine to retrieve a reference to the Service Specific DataDef for your Service (whose name matches your Service's `ClassName` property).

To create the Service Specific area for your Service, pass the routine the reference to your Service:

```
ServiceSpecificArea = srvServiceDataDef (m_oService)
```

Your Service then owns the `tom.DataDef` object that `ServiceSpecificArea` references.

Create Child DataDefs in Service Specific Area

2. After you call `srvServiceDataDef`, you can then load child DataDefs that are in the `Service Specific` area into memory. You do that using `srvLoadDataDef`, which returns a `tom.DataDef` object.

The `srvLoadDataDef` routine takes three arguments:

- ◆ *Service*—Reference to the Service being developed.
- ◆ *Parent*—Parent DataDef of the collection of DataDefs being loaded.
- ◆ *DataDefName*—String containing the name of the DataDef to load.

Load DataDefs called `DataDef1` and `DataDef2` by passing the `ServiceSpecificArea` reference as the parent:

```
Set DataDef1 = srvLoadDataDef(m_oService,_  
srvServiceDataDef(m_oService), "DataDef1")
```

```
Set DataDef2 = srvLoadDataDef(m_oService,_  
srvServiceDataDef(m_oService), "DataDef2")
```

Then create child DataDefs of `DataDef1` by passing `DataDef1` as the parent:

```
Set ChildDataDefA = srvLoadDataDef(m_oService,_  
DataDef1, "ChildDataDefA")
```

```
Set ChildDataDefB = srvLoadDataDef(m_oService,_  
DataDef1, "ChildDataDefB")
```

NOTE

Tip—References to DataDefs

Do you need references to these DataDefs so that you can access them elsewhere in the code? Not always. If you later associate the DataDefs with the Methods that use them (in TOM Builder or the TOM DB Editor), when you clone the Method, TOM creates not only a copy of the Method, but a copy of its DataDefs as well. Under these conditions, you need only the reference to the clone of the Method.

Defining Method Objects for Your Service in onCreate

Another task you carry out in `onCreate` is defining methods that you want for your Service. Methods are commands your Service carries out. You use a handler support routine called `srvDefineMethod` to create a method. The routine returns a `tom.Method` object.

The `srvDefineMethod` routine takes three arguments:

- *Service*—Name of the Service being developed.
- *MethodName*—String containing the name of the Method (this is the name that appears in TOM Explorer).
- *Description*—String containing a description of the Method.

The routine returns a reference to the new Method object.

1. Create the Method 1 method and include a description for it:

```
' Define Methods
Set Method1 = srvDefineMethod(m_oService, METH_METHOD1, "A
Sample Method")
```

The second argument is the name of the Method as it later appears in TOM Explorer (contained in the constant here). The third argument, the description, becomes the Method's `Description` property setting.

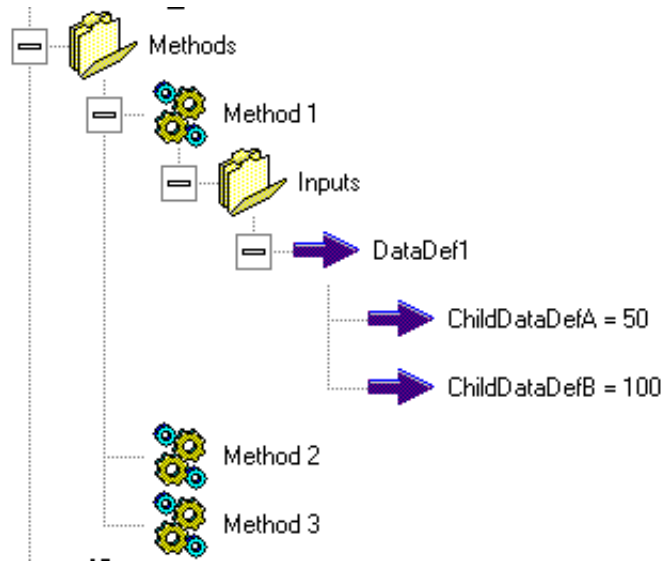
2. To create a `DataItem` for the Method, you use the `srvAddDataItem` routine. The `srvAddDataItem` routine, which returns a `tom.DataItem` object, takes four arguments:
 - ◆ *Service*—Name of the Service being developed.
 - ◆ *Parent*—`DataItem` that should be the parent of your `DataItem` (yours will be its child).
 - ◆ *DataDef*—A reference to the `DataDef` that defines the type of `DataItem` you are creating.
 - ◆ Optional *Children*—`True` if you want the new `DataItem` to be based on the definition of a child of the `DataDef`, `False` otherwise.

The routine returns a reference to the new `DataItem` object. If you are not interested in that returned value, you can add the data item without putting parentheses around the arguments; then, later you can retrieve the `DataItem` using the `MethodName.Inputs.Item(number)` technique.

You set the `DataItems` using a `DataDef` you created earlier:

```
Set DataItemInput = srvAddDataItem(m_oService, _
Method1.Inputs, ServiceSpecificDataDef.Item("DataDef1"))
```

Later, you see these DataItems under Methods for the Service in TOM Explorer:

**NOTE****TOM Tip—Generating Method Input Items**

To define an input item that uses information from Service Attributes, you must wait until you have Attributes, which TOM creates *after* it runs `OnCreate` and *before* it runs `OnInitialize`. Since you can't work with Attributes until they exist, you should *not* generate that input item in `OnCreate`, but instead in `OnInitialize`.

Although you can create the “empty shells” (DataDefs) in `OnCreate`, you need to delay creating any actual DataItems until `OnInitialize`, again, because you have no Attributes until after `OnCreate` runs.

Now you are ready to define Events for the Service.

Defining Event Objects for Your Service in OnCreate

An application that uses your Service might wait until a Service Event occurs before taking certain actions. How does an Event occur? Your Service fires the Event in response to an equipment initiated event (called a *collection event* in this manual). For instance, if you were establishing communications with a piece of equipment, your Service might set up communication parameters using a Method, but it would have an Event for TOM to execute when the communication status of the equipment changes. A Service fires the event in response to the equipment. Only a Service can fire TOM Events.

In the sample Service presented here `ToolEvent` is a Service Event. For your Service to fire Events, it must first create Event objects.

You define Event objects for your Service similarly to the way you defined methods, only you use `srvDefineEvent`. The `srvDefineEvent` routine takes three arguments:

- *Service*—Name of the Service being developed.
- *EventName*—String containing the name of the Event (this name appears in TOM Explorer).
- *Description*—String containing a description of the Event.

The routine returns a reference to the new Event object.

1. Create the `ToolEvent` Event and include a description for it:

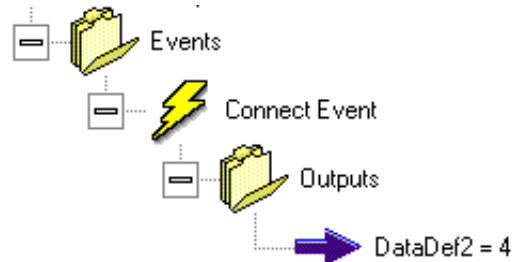
```
Set ToolEvent = srvDefineEvent(m_oService, "Tool Event", _  
"A sample event")
```

The second argument is the name of the Event. The third argument, the description, becomes the Event's `Description` property setting.

2. Create a `DataItem` for each `Output` that should result from the Event. You create the `DataItem` using the `srvAddDataItem` routine. You pass the routine the reference to the Service, the parent of the new `Output` (`DataItem`) you are creating, and the `DataDef` that defines the new `Output` (`DataItem`):

```
Set DataItemOutput = srvAddDataItem(m_oService, _  
ToolEvent.Outputs, ServiceSpecificDataDef.Item("DataDef2"))
```

Later, you see these DataItems under `Events` for the Service in TOM Explorer.



Restrictions in OnCreate

There are some actions you should never take in `OnCreate`:

NOTE

Tip—Actions Not Allowed in OnCreate

- If you raise an error in `OnCreate`, TOM terminates your Service object.
- Do not execute methods inside `OnCreate`. You create them here, but you execute them in another handler method.

After you have carried out the required tasks in your `OnCreate` handler method, you are ready to proceed to the next handler method.

Code of Sample OnCreate

The full code of `OnCreate` appears below:

```
Public Sub OnCreate(ByVal Service As tom.Service)
    Dim ServiceSpecificDataDef As tom.DataDef
    Dim ToolEvent As tom.Event
    Dim DataItemOutput As tom.DataItem
    Dim DataItemInput As tom.DataItem

    Dim DataDef1 As tom.DataDef
    Dim DataDef2 As tom.DataDef
    Dim ChildDataDefA As tom.DataDef
    Dim ChildDataDefB As tom.DataDef

    Dim Method1 As tom.Method
    Dim Method2 As tom.Method
    Dim Method3 As tom.Method
```



```
' Save Service reference
Set m_oService = Service
Debug.Print "Entering OnCreate"

' Retrieve your Service Specific area in the Dictionary
Set ServiceSpecificDataDef = srvServiceDataDef(m_oService)

' Here is an how to load child DataDefs into your Service Specific area
Set DataDef1 = srvLoadDataDef(m_oService, srvServiceDataDef(m_oService),_
"DataDef1")
Set ChildDataDefA = srvLoadDataDef(m_oService, DataDef1, "ChildDataDefA")
Set ChildDataDefB = srvLoadDataDef(m_oService, DataDef1, "ChildDataDefB")
Set DataDef2 = srvLoadDataDef(m_oService, srvServiceDataDef(m_oService),_
"DataDef2")

' Here is how to define a Method object
' This method is Method1
Set Method1 = srvDefineMethod(m_oService, METH_METHOD1, "A Sample Method")
Set DataItemInput = srvAddDataItem(m_oService, Method1.Inputs, _
ServiceSpecificDataDef.Item("DataDef1"))

' Here is a second Method object
' This method is Method2
Set Method2 = srvDefineMethod(m_oService, METH_METHOD2, "A Second Sample Meth

' Here is a third Method object
' This method is Method3
Set Method3 = srvDefineMethod(m_oService, METH_METHOD3, "A Third Sample Methc

' Here is how to define Event objects
Set ToolEvent = srvDefineEvent(m_oService, EVENT_CONNECT, "A sample event")
Set DataItemOutput = srvAddDataItem(m_oService, ToolEvent.Outputs, _
ServiceSpecificDataDef.Item("DataDef2"))

Debug.Print "Leaving OnCreate"
End Sub
```

Writing the LetAttribute Handler Method

The `LetAttribute` handler method should set any Attributes inside the Service. TOM calls this handler method after `OnCreate` and before `OnInitialize`, but this handler method is required only if your Service has Attributes.

TOM also calls `LetAttribute` whenever your Service assigns a value to an Attribute object by setting its `Value` property.

This handler method receives an Attribute name (in a string, of course) and an Attribute value (which could come from a lower level Service in TOM):

```
Public Sub LetAttribute(ByVal AttributeName As String, _
    ByVal NewValue As Variant)
```

Then `LetAttribute` might test to see if the Attribute name passed to it (`AttributeName`) matches an expected name. If it finds a match, the handler method can then set the Service's corresponding Attribute variable to the value passed to `LetAttribute` in `NewValue`, as shown below:

```
Select Case AttributeName
    Case ATT_EVENT_ENABLED
        Att_ToolEventEnable = NewValue
    Case Else
        Debug.Print "Cannot set ", AttributeName
        Debug.Print "Leaving GetAttribute"
End Select
```

The `NewValue` is `As Variant` because, since it often comes from the equipment (via a lower level Service), you can't be sure whether the value of the Attribute is a string or a number. If you want to know its type, you should check the type in your code. If you want to restrict the `NewValue` to not simply numeric, but a specific range of numbers, you must check for that level of compliance in your code.

If the `NewValue` passed does not fit the requirements for the Attribute value, you can have the code do one of the following:

- Set the value of the Attribute to a default value
- Leave the value at its previous setting
- Raise an error

Note that you cannot pass the `NewValue` by reference!

Raise an Error in LetAttribute

You can raise an error inside `LetAttribute`. Raising an error does not change the value of the Attributes that have been set. You find out more about raising an error in *Raising an Error*, p. 4-6.

Writing the GetAttribute Handler Method

`GetAttribute` retrieves the Attribute setting stored inside the Service. If Your Service has Attributes, you must have a `GetAttribute` handler method. As long as you have a `GetAttribute` handler method, another Service can also request an Attribute value from your Service.

You should create this handler method as a function that takes a string argument and returns a variant:

```
Public Function GetAttribute(ByVal AttName As String) _  
As Variant
```

Then have `GetAttribute` use the Attribute name TOM passes it and compare that name to the various possible Attribute names actually used by the Service. If the name of the Attribute you pass it matches one of the Attributes you were expecting, you then have the function return the value of that Attribute.

For instance, `GetAttribute` might contain a Case statement like the one shown below:

```
Public Function GetAttribute(ByVal AttName As String) _  
As Variant  
  
    Select Case AttributeName  
        Case ATT_EVENT_ENABLED  
            GetAttribute = Att_ToolEventEnable.Value  
        Case Else  
            Debug.Print "No such attribute exists ", AttributeName  
    End Select  
End Function
```

TOM returns the value that `GetAttribute` returns to the caller.

Writing the OnInitialize Handler Method

The `OnInitialize` handler method should complete set up of input items that require Attributes from the database or from other Services. TOM executes this handler method after it has executed `OnCreate` for each Service associated with the tool and has generated Attributes.

This handler method should :

1. Perform any initializations that must occur after Attributes have been set and/or other Services have started.
2. Check that no incompatible Services are running.
3. Verify that all required Services are present.
4. Subscribe to Events of other Services that your Service requires.
5. Set whether or not an application using your Service should receive notification on Events your Service subscribes to.
6. Generate References to other Services you want to work with.
7. Generate local storage for DataDefs your Service needs.

Create OnInitialize

Start by declaring `OnInitialize` as public. It has no arguments:

```
Public Sub OnInitialize()
```

Perform Any Initializations That Require Attributes

Now that TOM has retrieved the Attributes from the registry or database, you can initialize any values that depend on those Attribute settings.

Since TOM Services can never change values in the database, you set the Attributes to their initial values using TOM Builder or TOM DB Editor. For more information refer to *Creating Attributes*, p. 4-17.

Check That No Incompatible Services Are Running

In `OnInitialize`, you should always check to be sure that no Services are running that are incompatible with yours. For instance, if you are writing a special *CustomAlarms* Service, and it conflicts with the standard *GemAlarms* Service, you should not have the standard Service running.

To have TOM check for incompatible Services, call the `srvIncompatibleService` handler support routine. You pass the routine a reference to your Service, then the name of the Service that is not compatible with it:

```
srvIncompatibleService m_oService, ServiceName
```

(In the sample service, there are no incompatible Services specified, so this line of code is there, but commented out.)

You should call the routine once for each incompatible Service. If TOM finds an incompatible Service is running, it raises an error and TOM handles the

error by having the initialization of the tool fail. You wouldn't want the tool to be running if an incompatible Service were running!

If your handler method has an `On Error Goto` statement, the line the code goes to should use the `srvExtendError` routine to extend the error. (See *Deciding to Raise, Extend, or Trigger an Error*, p. 4-2.)

Verify That All Required Services Are Present

Your `OnInitialize` handler method should always verify that all required Services are running. You call the `srvRequiredService` handler support routine to verify required Services. You pass the routine a reference to your Service, then the name of the other Service that is required.

You should call the routine once for each required Service. For instance, if *ProtocolSECS* and *SecsLoopbackDiagnostic* are required, you would enter the routine once for each:

```
srvRequiredService m_oService, SRV_LOOPBACK
srvRequiredService m_oService, SRV_PROTOCOLSECS
```

If TOM finds a required Service is missing, it raises an error and TOM handles the error by having the initialization of the tool fail. You wouldn't want the tool to be running if any required Service is missing!

If your handler method has an `On Error Goto` statement, the line the code goes to should use the `srvExtendError` routine to extend the error. (See *Deciding to Raise, Extend, or Trigger an Error*, p. 4-2.)

Generate References to Other Services That Work with Yours

To actually work with the required Services, you generate references to those Services using the `srvGetService` handler support routine:

```
Set m_oLoopback = srvGetService(m_oService, SRV_LOOPBACK)
Set m_oProtocolSECS = srvGetService(m_oService, _
SRV_PROTOCOLSECS)
```

Subscribe to Events Your Service Requires

Your Service may need to take action when Events occur that another Service sets into motion. To ensure that your Service knows about those Events and can respond when they occur, you start by having your Service subscribe to those Events. You carry out the subscription in two steps:

1. You can have your Service subscribe to the other Service's events by calling the `srvSubscribeEvent` handler support routine. You pass it a reference to your Service, the name of the Service owning the Event, and the Event you want to subscribe to:

```
srvSubscribeEvent m_oService, SRV_PROTOCOLSECS, "Connect"
```

- Write another handler method for your Service called `OnSubscribedEvent`, which you learn more about later in this chapter.

NOTE**Tip — Subscribing to Another Service's Events**

If you call `srvSubscribeEvent` in your Service, when the Event occurs, in addition to sending news of the Event to the TOM applications (such as TOM Explorer, which receives notifications in its Event log), TOM triggers the `OnSubscribedEvent` handler method in your Service.

Once your Service calls `srvSubscribeEvent`, `OnSubscribedEvent` becomes a required handler method for your Service.

Set Whether or Not TOM App Receives Notification of Events Your Subscribed To

Once your Service has subscribed to another Service's Event, when the Event occurs, if your Service is handling that Event, you may not want the TOM application to take action. In such a situation, you would not want to notify the TOM application that is running your Service.

To cancel notification to a TOM application about an event you subscribe to, you can call `srvSetEventNotification`. You pass this routine several arguments:

- A reference to your Service
- The name of the Service whose event you've subscribed to
- The name of the event
- The value for the `Notify` property, either `tomNotifyNever`, `tomNotifyError` (sends notification when an error occurs), or `tomNotifyAlways`.

To cancel notifications to a TOM application when the `Connect` Event of *ProtocolSECS* occurs, you would pass the routine `tomNotifyNever`.

```
srvSetEventNotification m_oService, SRV_PROTOCOLSECS, _
"Connect", tomNotifyNever
```

In this case, you would want to notify the TOM application of the Event. To ensure that the application using your Service (such as TOM Explorer) receives notification, pass the routine `tomNotifyAlways`:

```
srvSetEventNotification m_oService, SRV_PROTOCOLSECS, _
"Connect", tomNotifyAlways
```

NOTE

When a Service Event occurs, TOM applications can receive notifications. For details, refer to the *Tool Object Model (TOM) Application Developer's Guide*.

Restrictions in OnInitialize

There are some actions you should never take in OnInitialize:

NOTE

Tip — Actions Not Allowed in OnInitialize

- When you raise an error in OnInitialize, TOM terminates your Service object. Do not raise an error in OnInitialize unless you want to terminate the Service.
- Do not execute methods inside OnInitialize.

Code of Sample OnInitialize

The full code of OnInitialize appears below:

```
Public Sub OnInitialize()
    Dim localAttribute As String

    Debug.Print "Entering OnInitialize"

    ' Perform initialization that must happen after Attributes are
    ' set and/or other services are started.

    ' Here is how to check to be sure a required service is present
    ' If the service is present, it is registered in the NT registry
    srvRequiredService m_oService, SRV_LOOPBACK
    srvRequiredService m_oService, SRV_PROTOCOLSECS

    ' Generate References to other services this service works with
    Set m_oLoopback = srvGetService(m_oService, SRV_LOOPBACK)
    Set m_oProtocolSECS = srvGetService(m_oService, SRV_PROTOCOLSECS)

    ' Check that no incompatible services are running
    '   srvIncompatibleService m_oService, ANYSERVICECONSTANT

    ' Subscribe to events your service requires
    srvSubscribeEvent m_oService, SRV_PROTOCOLSECS, "Connect"

    ' Set whether or not other services require notification
    ' Pass this handler support routine tomNotifyAlways or tomNotifyNever
    srvSetEventNotification m_oService, SRV_PROTOCOLSECS, "Connect", tomNotifyAlv
    'Make use of an attribute in OnInitialize rather than in OnCreate
    Debug.Print "Leaving OnInitialize"
End Sub
```

Writing the OnExecute Handler Method

The `OnExecute` handler method triggers when TOM executes a `Method` object of this `Service`. You must have this handler method if your `Service` has any `Method` objects.

For every `Method` you defined in `OnCreate`, you need to write the Visual Basic code that implements the `Method` actions. You write that code in `OnExecute`.

Accept a TOM Method as an Argument

Start by declaring `OnExecute` as `public` and setting it up to accept a TOM `Method` object passed to it by value:

```
Public Sub OnExecute(ByVal ExecuteMethod As tom.Method)
```

Trap Any Errors

Before taking any other action, you should direct the handler method to the `ErrorTrap` label when an error occurs:

```
On Error GoTo ErrorTrap
```

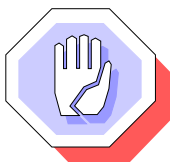
For further information on how to deal with errors, refer to *Deciding to Raise, Extend, or Trigger an Error*, p. 4-2.

Determine Method to Execute

Remember that TOM runs `OnExecute` each time it needs to execute a `Method` of your `Service`. So, each time TOM runs `OnExecute`, it starts at the beginning of the handler method again and must, based on the existing conditions at that time (such as variables you have set in the `Service` or `Properties` of the `Method`), execute the appropriate `Method`. In the sample `Service`, when TOM runs `OnExecute`, one of the three `Methods` can be the `Method` to execute:

- Method 1
- Method 2
- Method 3

A `Case` statement can determine the `Method` that the object model should execute for the particular run of `OnExecute` by checking the `Name` property of the `Method` passed to the routine. Which `Method` was passed depends on what the application or higher level `Service` is trying to do with your `Service`.



CAUTION

Since `Methods` can execute concurrently, do not use global variables to maintain state information.

Never use the `Tag` property of the `Method` passed to `OnExecute` in the code inside the `OnExecute`. The `Tag` property is reserved for the calling application's use.

Code Method Action

In the sample Service, the Methods do not take any action other than printing to the Debug window. You need to determine the actions your methods are going to take and code that action. You would fit the code inside the Case statement, as in the following example:

*Called in OnExecute only if all your custom method action occurs here—because the method does **not** execute another Service's Methods.*

```
Select Case ExecuteMethod.Name
    Case METH_METHOD1
        Debug.Print "Method 1 Executing"
        Debug.Print "  ChildDataDefA: " & _
            ExecuteMethod.Inputs.Item("DataDef1").Item_
                ("ChildDataDefA").Value
        Debug.Print "  ChildDataDefB: " & _
            ExecuteMethod.Inputs.Item("DataDef1").Item_
                ("ChildDataDefB").Value
        srvCompleted ExecuteMethod
    Case METH_METHOD2
        Debug.Print "Method 2 Executing"
        srvCompleted ExecuteMethod
    Case METH_METHOD3
        Debug.Print "Method 3 Executing"
        srvCompleted ExecuteMethod
End Select
```

Handle Any Errors

You should always have an error trap set up in OnExecute to handle any errors that arise. In the ErrorTrap section, in addition to any error handling required for your Service, if you created a new object in this handler method, you should always include the exact code that follows, only you should substitute the appropriate variable for ExecuteMethod:

```
ErrorTrap:
    Dim ErrorState As t_ErrorState
    srvSaveErrorState ErrorState
    `insert custom error handling code here
    Set ExecuteMethod = Nothing
    srvRestoreErrorState ErrorState
    srvExtendError "OnExecute"
```

For details on how this code handles the error, refer to *Extending an Error*, p. 4-3.

Executing Existing Methods in OnExecute

Let's suppose that inside `Method 1` you want to run a Method from another Service. You would take the following actions:

1. Clone the Method
2. Set any Inputs of the Method
3. Execute the Method with `srvExecute`

Clone a Method

To run a Method from another Service, you clone the Method and store a reference to the clone locally in a variable. You clone it using `srvCloneMethod`. To use this routine, you pass it a reference to the Service that owns the Method to clone and the name of the particular Method. For instance, to execute the `Test` Method from the *SecsLoopbackDiagnostic* Service:

```
Set MethodToExec = srvCloneMethod(m_oLoopback, "Test")
```

Once you have cloned the Method, you have not only a copy of the Method in memory, but copies of the associated Inputs and Outputs of the Method—its `DataDefs`. So, now you can set the cloned Method's Inputs using information from Attributes or other `DataDefs`:

```
MethodToExec.Inputs.Item("ABS").Value = _
ExecuteMethod.Inputs.Item("DataDef1").Item_
("ChildDataDefA").Value
```

Execute the Cloned Method

After you clone a Method, you execute it using the `srvExecute` handler support routine. You pass the routine a reference to the Method to execute, a reference to your Service, and a reference to the invoking Method, if applicable. If you have `Method 1` execute the `Test` Method of *SecsLoopbackDiagnostic*, the `Test` Method is the one being executed and `Method 1` is the invoking Method.

To keep the code readable, since the `ExecuteMethod` (in this case `Method 1`) is going to invoke another Method, let's store it in `InvokingMethod`:

```
Set InvokingMethod = ExecuteMethod
```

Now that the `Test` Method is in `MethodToExecute` and `Method 1` is in `InvokingMethod`, here is the call to `srvExecute`:

```
srvExecute MethodToExec, m_oService, InvokingMethod
```

If no other Method is invoking this one, you pass it the Method to execute as the invoking Method, too.

The invoking method is the one returned to `OnMethodCompleted` later.

When the method completes, the Service throws program control into `OnMethodCompleted` and you take all subsequent action there. Let's jump to that handler method (next section) and see how it should complete the Methods.

NOTE**Tip—Always Clone Method before Executing**

Always clone a Method before executing it, even if it is a Method in your own Service. This action ensures that, since each Service has its own copy, no other Service can execute the same Method object while your Service is executing it.

Being able to clone Methods of other Services gives you access to the Methods of all existing Services associated with the Resource. Once you have a clone of a Method, you can alter the clone's `DataDefs`.

Once you execute the Method clone, TOM Core keeps a reference to it as long as it's executing. Once it finishes executing and TOM passes the clone to `OnMethodCompleted`, it is up to you to save a reference to the clone or dispose of it.

Restrictions in OnExecute

There are some actions you should never take in `OnExecute`, delineated below:

NOTE**Tip—Actions Not Allowed in OnExecute**

- Although you must call the `srvCompleted` handler support routine when you have finished processing each Method object, you do not usually call it in `OnExecute`. You usually call it in `OnMethodCompleted`, when the Method you have called has completed. If you do not call `srvCompleted`, the Method hangs and TOM does not send a completion notification to the invoking Method.
 - ◆ For synchronous operations, you usually call `srvCompleted` from `OnExecute`.
 - ◆ For asynchronous operations, you usually call `srvCompleted` from `OnMethodCompleted`.
- Since TOM could call your `OnExecute` again while you are still processing a previous call of `OnExecute`, do not create a global reference or global variable (by creating it in `General Declarations`), then expect to use that reference or variable in the `OnMethodCompleted` routine. The reference or variable could be overwritten by the new call to `OnExecute` (and thus invalidated) before TOM calls `OnMethodCompleted`.

Code of Sample OnExecute

The full code of OnExecute from the sample service shows Method 2 cloning and executing Method 3:

```
Public Sub OnExecute(ByVal ExecuteMethod As tom.Method)
    Dim MethodToExec As tom.Method
    Dim InvokingMethod As tom.Method
    Debug.Print "Entering OnExecute"
    On Error GoTo ErrorTrap

    Select Case ExecuteMethod.Name
        Case METH_METHOD1
            Debug.Print "Method 1 Executing"
            Debug.Print " ChildDataDefA: " & ExecuteMethod.Inputs.Item("DataDef1").
                Item("ChildDataDefA").Value
            Debug.Print " ChildDataDefB: " & ExecuteMethod.Inputs.Item("DataDef1")
                Item("ChildDataDefB").Value
            Set MethodToExec = srvCloneMethod(m_oLoopback, "Test")
            MethodToExec.Inputs.Item("ABS").Value = _
                ExecuteMethod.Inputs.Item("DataDef1").Item("ChildDataDefA").Value
            Set InvokingMethod = ExecuteMethod
            srvExecute MethodToExec, m_oService, InvokingMethod

        Case METH_METHOD2
            Debug.Print "Method 2 Executing"
            Set MethodToExec = srvCloneMethod(m_oService, METH_METHOD3)
            Set InvokingMethod = ExecuteMethod
            srvExecute MethodToExec, m_oService, InvokingMethod
            srvCompleted ExecuteMethod

        Case METH_METHOD3
            Debug.Print "Method 3 Executing"
            srvCompleted ExecuteMethod
    End Select
    Debug.Print "Leaving OnExecute"
    Exit Sub

ErrorTrap
    ` Standard ErrorTrap code goes here
End Sub
```

Writing the OnMethodCompleted Handler Method

The `OnMethodCompleted` handler method triggers when any Service method executed with `srvExecute` finishes executing.

Every TOM method you invoke with `srvExecute` (in `OnExecute`), you must also have completion instructions in `OnMethodCompleted`.

For every time a calling Service or application invokes one of your Service methods using `srvExecute`, your Service must eventually call `srvCompleted`. If for some reason, you take an exit path out of a routine, such as by raising an error, TOM calls `srvCompleted` for you.

Steps you should take in `OnMethodCompleted` are:

1. Determine whether you are completing `OnVerify` or another Method and branch to separate actions for these two major options.

Inside the branch that responds to an executed Method:

2. Determine whether or not errors have occurred and handle them.
3. Determine Method that is completing.
4. If executing several Methods in sequence, clone and execute the next Method in the sequence.
5. Fill in values for any `DataItems` in the invoking Method's `Outputs` collection.
6. If all required Methods have been executed, complete the action by calling `srvCompleted` on the invoking method.

Accept Completed Method and Invoking Method as Arguments

Let's start by declaring `OnMethodCompleted` as public and setting it up to accept the name of two TOM methods passed to it by value, the first the method that was executing and the second the method that invoked the first one:

```
Public Sub OnMethodCompleted(ByVal CompletedMethod As_  
tom.Method, ByVal InvokingMethod As tom.Method)
```

You are required to have this handler method take these two arguments.

The invoking method is always the same invoking Method you passed to `srvExecute`, unless you did not pass it an invoking Method. If you did not pass an invoking Method to `srvExecute`, then that argument became `Nothing` by default; your Service then passed `Nothing` as the argument to `OnMethodCompleted`.

Determine the Method Being Completed and Set Up Major Code Branches

When you enter `OnMethodCompleted`, the handler method must determine why you are here.

Which Service method is being completed? It can be any Method you have cloned/executed either in `OnExecute` or, to verify it, in the `OnVerify` handler method. Those two possibilities should form the two major branches of code in this handler method.

Set Up Major Code Branches

You must always begin `OnMethodCompleted` by determining which path lead to it:

- `OnVerify`
- `OnExecute`

If `OnVerify` lead to `OnMethodCompleted`, then the invoking method is equal to `Nothing`. So you can check the value of the invoking method to determine you should take the `OnVerify` completion path. Otherwise, you always take the other Method completion path:

```
If InvokingMethod Is Nothing Then
    ' Verification path
    srvCompleted Method
    lVerify (Method.Tag)
Else
    ' Second layer of Service method tasks
    lCompleted CompletedMethod, InvokingMethod
End If
```

It is best if you set up these major branches and have them each call private functions to carry out the details of the verification and completion paths. For instance, in the verification path, you execute first `srvCompleted` (a handler support routine), then the private function `lVerify`.

NOTE

Tip—Calling `srvCompleted`

For every time another Service or an application invokes one of your Service methods using `srvExecute`, your Service must eventually call `srvCompleted`. If for some reason, you take an exit path out of a routine, such as by raising an error, TOM carries out the `srvCompleted` for you.

What does `srvCompleted` do? It tells TOM the Method object has finished executing. If you do not run `srvCompleted`, and thereby inform TOM the method is finished, the Method never completes and TOM does not send a completion notification to the invoking Method.

In the alternative path, you execute the private function called `lCompleted`. Eventually this private function calls `srvCompleted` for all methods that complete.

Let's focus on the completion of Service methods other than `OnVerify`. (You see how `OnVerify` works later under *Writing the OnVerify Handler Method*, p. 3-33.)

Create Branch to Complete the Method Action

The `lCompleted` private function should check to see which Service method executed last and determine which Service method to execute next.

Using this approach is one technique for cycling through `OnExecute`, then `OnMethodCompleted`, and always knowing where in its sequence the process of setting up events is.

Let's see how this private function should be constructed.

Accept Completed Method and Invoking Method as Arguments

This method should always be private and take the same arguments that `OnMethodCompleted` takes, the method being executed and the invoking method:

```
Private Sub lCompleted(ByVal CompletedMethod As tom.Method, _
    ByVal InvokingMethod As tom.Method)
```

The Method being executed can be a method you clone from another Service. Under those conditions, the invoking method would be one in the Service you are writing.

Determine Whether or Not Errors Have Occurred

Before you proceed, you should establish that no errors have occurred up to this point in the process. So, you can start by checking to see whether or not the `Error.ErrorCode` Property of the Method is zero. If it is *not* zero, an error has occurred. Under those conditions, the first action you should take is to call `srvCompleted` on the invoking Method:

```
If (CompletedMethod.Error.ErrorCode <> 0) Then
    srvCompleted InvokingMethod, FailedMethod:=CompletedMethod
    FinishedSteps = False
    Debug.Print "Method Failed: ", InvokingMethod.Name
Else
    ...
End If
```

For more information on how to deal with errors, refer to *Deciding to Raise, Extend, or Trigger an Error*, p. 4-2.

Use Name Property to Branch

Based on the setting of the `Name` property of the method, you can branch to the various possible actions. For instance, if your Service invoked not only `Test`, but its own `Method 3 Method`, then you would have each of them as possible Methods being completed:

```
Select Case CompletedMethod.Name
    Case "Test"
        ...
    Case "Method 3"
        ...
End Select
```

Determine the Method That Is Completing and Prepare to Proceed

Inside each case, you check to see what the `CompletedMethod.Name` is set to—that should be the name of the Method being completed. Based on which Method that is, you can prepare to proceed to the next action.

The sequence of execution for the Methods depends on your goal. If you want to execute the Methods in a particular sequence, as part of the action in `OnMethodCompleted`, you might want to clone the Method that should execute next.

You can then choose to execute another Method using `srvExecute`. After your code calls `srvExecute`, TOM throws program control into `OnMethodCompleted` again, so the action starts at the top of this handler method. The handler method determines that a method has executed and calls `lComplete`.

After you have taken any such “end actions” for the Method, you should call `srvCompleted` on the invoking Method:

```
Select Case CompletedMethod.Name
    Case "Test"
        Debug.Print "Completing Test"
        srvCompleted InvokingMethod
    Case "Method 3"
        Debug.Print "Completing Method 3"
        srvCompleted InvokingMethod
End Select
```


**Code of Sample
OnMethodCompleted**

The full code of the sample OnMethodCompleted routine follows:

```
Public Sub OnMethodCompleted(ByVal CompletedMethod As tom.Method, _
    ByVal InvokingMethod As tom.Method)
    Debug.Print "Entering OnMethodCompleted", CompletedMethod.Name
    If InvokingMethod Is Nothing Then
        ' Do Verification
        lVerify CompletedMethod.Tag
    Else
        ' Take actions that should occur after method completes
        lCompleted CompletedMethod, InvokingMethod
    End If
    Debug.Print "Leaving OnMethodCompleted"
End Sub
```

**Code of Sample
lCompleted**

The full code of the sample lCompleted routine follows:

```
Private Sub lCompleted(ByVal CompletedMethod As tom.Method, _
    ByVal InvokingMethod As tom.Method)
    Dim FinishedSteps As Boolean
    Dim ExecuteMethod As tom.Method

    If (CompletedMethod.Error.ErrorCode <> 0) Then
        srvCompleted InvokingMethod, FailedMethod:=CompletedMethod
        FinishedSteps = False
        Debug.Print "Method Failed: ", InvokingMethod.Name
    Else
        Select Case CompletedMethod.Name
            Case "Test"
                Debug.Print "Completing Test"
                srvCompleted InvokingMethod
            Case "Method 3"
                Debug.Print "Completing Method 3"
                srvCompleted InvokingMethod
        End Select
    End If
End Sub
```

Writing the OnSubscribedEvent Handler Method

If your Service subscribes to another Service's Events, you must have a handler method called `OnSubscribedEvent`. In this handler method, you take the following steps:

1. Receive the Event TOM passes to the handler method so that you can work with the Event.
2. Retrieve any Output DataItems of the Event.
3. Take other action.

Accept TOM Event as Argument

Let's start by declaring `OnSubscribedEvent` as public and having it receive a single argument of a TOM Event passed to it by value. This Event is the one your Service subscribes to:

```
Public Sub OnSubscribedEvent(ByVal TomEvent As tom.Event)
```

Trap Any Errors That Occur

Before you generate any other code, at the top of the handler method, you can have an `On Error Goto` statement that sends program control to an `ErrorTrap` section when it encounters an error while this handler method is running:

```
On Error GoTo ErrorTrap
```

Retrieve Any Output DataItems

You verify the Event that occurred. If the Event name matches the one you expected, you can retrieve any of its Output DataItems. In this situation, let's store the value of the received Event's first DataItem in a local variable:

```
If TomEvent.Name = "Connect" Then
    m_bConnected = True
    m_bConnectedData = Event.Outputs.Item(1).Value
End If
```

Take Other Action

You can take any other action you want to take when notified about the Event. Usually the Event that is passed to your Service is already a clone, so you can often treat it as a clone; however, it may not be a clone, so be aware of that possibility.

The sample code clones its own Event and stores its reference in `NewEvent`. It then sets `NewEvent's DataDef2` to the received Event's `Description` Property:

```
NewEvent.Outputs.Item("DataDef2").Value = TomEvent.Description
```

Handle Any Errors

In the `ErrorTrap` section, in addition to any error handling required for your Service, if you created a new object in this handler method, you should always include the exact code that follows, only you should substitute the appropriate variable for `NewEvent`:

```
ErrorTrap:
    Dim ErrorState As t_ErrorState
    srvSaveErrorState ErrorState
    `insert custom error handling code here
    Set NewEvent = Nothing
    srvRestoreErrorState ErrorState
    srvExtendError "OnSubscribedEvent"
```

For details on how this code handles the error, refer to *Extending an Error*, p. 4-3. Next, you see how to trigger your own Service Event.

Triggering Your Service Event

In addition to subscribing to other Service's Events, your Service can have Events of its own that it triggers.

To trigger your own Service Event:

1. Clone the Event within your Service that you want to trigger.
2. Set any Output DataItems of the Event.
3. Trigger the Event for any subscribing higher-level Service or application.

Clone Your Service Event

Before you can trigger your Service's Event, you must clone the Event. Let's clone the `ToolEvent` defined earlier in `OnCreate`:

```
Set NewEvent = srvCloneEvent(m_oService, EVENT_CONNECT)
```

Set Any Event DataItems

Once you have the clone, you then set the `NewEvent`'s Output DataItem Property:

```
NewEvent.Outputs.Item("DataDef2").Value = "First ToolEvent"
```

Trigger the Event for Application

Next, you need to trigger the Event for any application or higher level Service that receives notifications about/subscribes to your Service's Event. To trigger the Event, you use `srvTriggerEvent`. You pass this routine the clone of the Event stored in `NewEvent`:

```
srvTriggerEvent NewEvent
```

When you call this routine, it triggers the `OnSubscribedEvent` of the subscribing Service or the `tomCtrl_EventNotification` routine in the application.

You may want to trigger your own Event in response to another Service's Event occurring.

For more details on how an application receives notifications of Service Events, refer to the *TOM Application Developer's Guide*.

Code of Sample OnSubscribedEvent

The full code of the sample `OnSubscribedEvent` routine follows. This code receives a subscribed Event and triggers its own Event in response, a relatively common scenario.

In this code, after receiving notification that an Event you subscribed to occurred, you check to see if the `Value` of the `ToolEnabled` attribute of your Service is `True`. If it is, you clone your own Event (`EVENT_CONNECT`) in response. Next, you check to see if the Event that occurred was the `Connect` Event of *ProtocolSECS*; if so you then can take the `Description` Property of the `Connect` Event and use it to set `DataDef2` of your Event. Finally, you trigger the clone of your own Event for the application using your Service.

```
Public Sub OnSubscribedEvent(ByVal TomEvent As tom.Event)
    Debug.Print "Entering OnSubscribedEvent"
    On Error GoTo ErrorTrap
    Dim NewEvent As tom.Event

    If m_oService.Attributes.Item(ATT_EVENT_ENABLED).Value = "True" Then
        Set NewEvent = srvCloneEvent(m_oService, EVENT_CONNECT)
        Debug.Print NewEvent.Name
        If TomEvent.Name = "Connect" Then
            NewEvent.Outputs.Item("DataDef2").Value = TomEvent.Description
            srvTriggerEvent NewEvent
        End If
    Else
        Debug.Print "ToolEventEnable is False"
    End If
    Debug.Print "Leaving OnSubscribedEvent"
    Exit Sub

ErrorTrap:
    Dim ErrorState As t_ErrorState
    srvSaveErrorState ErrorState
    Set NewEvent = Nothing
    ErrorState
    srvExtendError "OnSubscribedEvent"
End Sub
```

Writing the OnVerify Handler Method

The `OnVerify` handler method triggers when an object of this Service's class is being verified. In this handler method, you must check that all of your Service's capabilities work correctly in the current environment. This handler method is required.

To verify the Service, your Service's `OnVerify` handler method should:

1. Execute each Method object it generates.
2. Ideally, the verification process should test each Event of the Service as well, or at least those events it can force the equipment to trigger. To complete testing of other events, you should have an operator or manufacturing engineer set up the equipment to generate each remaining event.
3. Send a notification to TOM that the verification is complete.

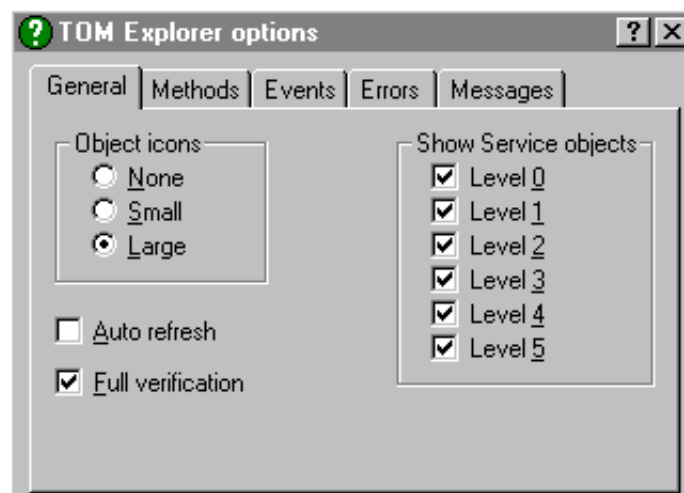
Accept a Boolean as an Argument

Let's start creating this handler method by declaring it as public:

```
Public Sub OnVerify(ByVal FullVerification As Boolean)
```

TOM passes the function a `FullVerification` variable that you should assign to the corresponding variable you declared in `General Declarations` for the class. The argument is a `Boolean` variable.

How does your Service determine whether or not Full Verification is on? It receives this information from an application. If you are using TOM Explorer to run your Service, you can set an option for Full Verification (select `View => Options` and click on the `General` tab), as shown below:



Prepare to Handle Any Errors

Before you take any action in the `OnVerify` handler method, you first need to send control to an error handler for any error that may have occurred in the verification process:

```
On Error GoTo ErrorVerify
```

Carry Out the Verification Process

To start verification, store the setting of `FullVerification` (a Boolean) in the local variable named `m_fFullVerification` that you created earlier:

```
m_fFullVerification = FullVerification
```

Full Verification

If `FullVerification` is `True`, you should carry out an exhaustive verification process for the `Service`, which thoroughly tests every `Method` in the `Service`. As long as the `FullVerification` Boolean variable is `True`, your `Methods` can and should change the state of the physical equipment to ensure they interact with the tool correctly. Ideally, the full verification process should also test `TOM` Events using your equipment, at least those Events you can force the equipment to trigger. Other Events may not be testable through the verification process.

Partial Verification

If `FullVerification` is `False`, you can still verify the tool, but you should perform only those tests that do not modify the physical state of the equipment when they execute. In this case, you must leave the tool in exactly the state you found it in at the start of the verification process. So, you can change the state of the physical equipment *temporarily*, as long as you restore it to its original setting afterwards.

You can call a local handler method to carry out the full verification. In this case, let's call `lVerify` and pass it the `Tag` of the first method to verify for the index into the verification sequence:

```
If InvokingMethod Is Nothing Then
    ' Do Verification
    lVerify CompletedMethod.Tag
Else
    ' Take actions that should occur after method completes
    lCompleted CompletedMethod, InvokingMethod
End If
```

In `lVerify`, you carry out the actual verification, as outlined in the section on *Verifying a Service—The Nuts and Bolts*, p. 3-36. After the verification is complete, you must send a notification to `TOM`.

Send Notification to TOM

Once the verification process is complete, you need to call the `srvVerified` handler support routine to let TOM know that the Service has completed its verification process:

```
srvVerified m_oService
```

You must call `srvVerified` to notify TOM. If you do not, the verification process hangs because it is waiting for notification from your Service.

Handle Any Errors

In this handler method, as in `OnExecute`, you need to have a section that `OnError Goto` sends program control to. In the `ErrorTrap` section, in addition to any error handling required for your Service, if you created a new object in this handler method, you should always include the exact code that follows, only you should substitute the appropriate variable for `VMethod`:

```
ErrorVerify:  
    Dim ErrorState As t_ErrorState  
    srvSaveErrorState ErrorState  
    'insert custom error handling code here  
    Set VMethod = Nothing  
    srvRestoreErrorState ErrorState  
    srvExtendError "OnVerify"
```

See also *Extending an Error*, p. 4-3.

Issues in OnVerify

You need not have `OnVerify` test all possible settings of Attributes that establish configuration information:

NOTE **TOM Tip—Testing Configuration Attributes in OnVerify**

If your Service uses Attributes to establish configuration information, you are not responsible for testing every possible setting of such Attributes in the `OnVerify` handler method. Instead, you can test your Service by assuming these Attributes retain the default values assigned during initialization.

NOTE **Tip—Using the Tag Property of a Method**

If you execute multiple Methods within your Service, you can use the `Tag` Property to indicate the last one run.

Verifying a Service—The Nuts and Bolts

How do you verify the Service? You execute the Methods and Events of the Service.

Execute the Methods

You execute each Method object it generates, by carrying out the following steps for each:

1. Use `srvCloneMethod` to create a clone of the Method.
2. Set up `DataItem` objects for the methods using `srvAddDataItem`.
3. Execute the Method clone using `srvExecute`.

NOTE When you run `srvExecute` in this handler method, you must not pass it the *InvokingMethod* argument.

For more details on executing a Method, refer to *Executing Existing Methods in OnExecute*, p. 3-22.

Trigger the Events

In addition to testing each Method, ideally, the verification process should test each Event of the Service as well, or at least those Events that it can force the equipment to trigger. To complete testing of other Events, you should have an operator or manufacturing engineer set up the equipment to trigger each remaining Event.

1. Use `srvCloneEvent` to create a clone of the Event.
2. Set up Output `DataItem` objects for the methods using `srvAddDataItem`.
3. Trigger the Event for the subscribing higher-level Service or Application using `srvTriggerEvent`.

To test Events, go to the lower level Service (level 0) and execute the Events to force them to occur. You can subscribe to the protocol level Events. Then you verify the protocol level Service from TOM Explorer and that tests the corresponding events in your Service.

For more details on triggering an Event, refer to *Triggering Your Service Event*, p. 3-31.

Send Notification to TOM

You are always required to send a notification to TOM when you have verified a Service. You must use `srvVerified`. In TOM Explorer, you can see some Properties of a Service that indicate the verify status:

- `Verified`—Set to True when all of your Methods and Events have been run at least once.
- `Verification Completed`—True when you have done the `srvVerified` on the Service.

Take a Closer Look at Sample Verification Process

The sample Service's `lVerify` function illustrates how the complete verification process occurs in any Service.

When the `OnVerify` handler method executes, it determines whether to run `lVerify` or `lCompleted` and passes `lVerify` the name of the first method to verify. The name is stored in `CaseStep1`, so that the routine proceeds to execute that case. For each case, it clones the next Method to verify and stores it in `VerifyingMethod`, then sets that method's `Tag` property to the name of the next Method to verify:

```
Set VerifyingMethod = srvCloneMethod(m_oService, METH_METHOD1)
VerifyingMethod.Tag = CaseStep2
```

After setting up the action this way, `lVerify` calls `srvExecute` on the `VerifyingMethod`, passing it `Nothing` as the invoking Method, which is a clue to `OnMethodCompleted` that the verification process executed the Method:

```
srvExecute VerifyingMethod, m_oService, Nothing
```

As with all other calls of `srvExecute`, after it executes the Method, TOM sends program control into `OnMethodCompleted`. `OnMethodCompleted` retrieves the `Tag` from the Method and uses it as the next `Index` into `lVerify`:

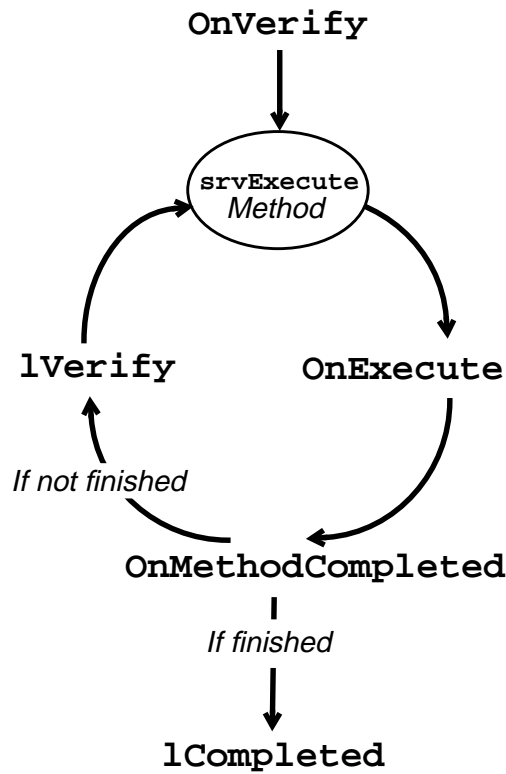
```
lVerify CompletedMethod.Tag
```

This process continues until `lVerify` receives `CaseEnd` as the index. At this point, `lVerify` calls `srvVerified` to indicate the Service has been verified:

```
Case CaseEnd
    srvVerified m_oService
Exit Sub
```

To carry out a partial verify, the sample code also illustrates checking `m_oFullVerification` and, if it is not `True`, setting the `Tag` to `CaseEnd` at that point to cut the verification process short.

The following illustrates the flow of the code, which is a typical flow for a verification process. When `OnVerify` executes a `Method`, TOM goes to `OnExecute` to know what action to take. When TOM throws program control into `OnMethodCompleted`, the verification process is either finished or not finished. The circle continues as long as the verification is not complete:



**Code of Sample
IVerify**

The full code of the sample `IVerify` routine follows:

```
Private Sub IVerify(Index As Variant)
    Dim VerifyingMethod As tom.Method
    Dim ExecuteMethod As tom.Method
    On Error GoTo ErrorTrap

    Select Case Index
        Case CaseStep1
            Set VerifyingMethod = srvCloneMethod(m_oService, METH_METHOD1)
            VerifyingMethod.Tag = CaseStep2

        Case CaseStep2
            Set VerifyingMethod = srvCloneMethod(m_oService, METH_METHOD2)
            If m_oFullVerification Then
                VerifyingMethod.Tag = CaseStep3
            Else
                VerifyingMethod.Tag = CaseEnd
            End If

        Case CaseStep3
            Set VerifyingMethod = srvCloneMethod(m_oService, METH_METHOD3)
            VerifyingMethod.Tag = CaseEnd

        Case CaseEnd
            srvVerified m_oService
            Exit Sub
    End Select

    srvExecute VerifyingMethod, m_oService, Nothing

    'Standard ErrorTrap code goes here

End Sub
```

Writing the Version Handler Method

The `Version` handler method triggers when a TOM application or Service calls the `Version` property this Service. For instance, you can access the `Version` property of any Service from TOM Explorer.

The handler method must always call the `srvVersion` handler support routine, which takes no arguments. `srvVersion` returns a string that contains the following properties of the `App` object (the Visual Basic project):

- Major
- Minor
- Revision

In the string it returns, `srvVersion` concatenates the three values and puts dots between them, so the resulting string for a `Major` value of 1, `Minor` value of 0, and `Revision` value of 2 would contain the following:

```
1.00.0002
```

The code to the `Version` function should always appear as follows:

```
Public Function Version() As String
    Version = srvVersion
End Function
```

Writing the OnTerminate Handler Method

You are required to have an `OnTerminate` handler method for your Service. TOM calls the `OnTerminate` handler method before the Service terminates.



CAUTION

This handler method should clean up memory by removing any objects your Service has created. You remove the objects by setting the references to them equal to `Nothing`.

For example, in the sample Service, several objects are cleaned up in this handler method:

```
Public Sub OnTerminate()  
    Set m_oService = Nothing  
    Set m_oLoopback = Nothing  
    Set m_oProtocol= Nothing  
End Sub
```

Writing a Terminate Class Method

The final *Method* you must generate is not a handler method, but a *class* method. It is a `Terminate` method that terminates the class when the OLE server stops running. This class method enters the terminate process by calling `OnTerminate`:

```
Private Sub Class_Terminate()  
    Me.OnTerminate  
End Sub
```


Creating a Tool for Your Service

4

Introduction

Topics in This Chapter

Every Service requires a Tool, even if it is a higher level Service that does not work with equipment. This chapter covers how to create a conceptual Tool to work with a level 4 Service like the one illustrated in this manual. You take the following steps using TOMBuilder:

Working with TOM Builder, p. 4-3

Creating a New Tool, p. 4-4

Creating a New Resource, p. 4-5

Adding Resources to the Tool, p. 4-6

Adding Your Custom Service to Database, p. 4-8

Assigning Services to Tool Resources, p. 4-11

Creating a New Service Dictionary, p.4-13

Assigning the Dictionary to a Service, p.4-16

Creating a New Resource Dictionary, p.4-17

Assigning the Dictionary to Resources, p.4-20

Creating DataDefs, p.4-21

Cloning DataDefs, p.4-27

Creating Attributes, p.4-28

Finalizing Tool by Releasing It, p.4-31

Building TOM Database (Containing New Tool), p.4-32

NOTE

Before you create a new Tool, you should have your own copy of the database to add the Tool to. You can create a new database as described in Chapter 1 under *Establishing Database Components*, p. 1-3 and *Building a Database of Sample Tools*, p. 1-5, also described in the *TOM Builder User's Guide* Help file.

Working with TOM Builder

TOM Builder is an editor for the Tool Object Model (TOM) database. You use TOM Builder to create and modify databases for TOM.

When you originally receive TOM, the database includes a series of component files with `.tbf` extensions and a built database file with an `.mdb` extension. Each `.tbf` file contains the information for a TOM component, such as a Tool, Resource, Manufacturer, Dictionary, or Service. With the TOM Builder you can edit the separate files (you make the changes in the GUI and TOM Builder takes care of the separate component files for you). After you edit the files, you then use TOM Builder to build a STATIONworks database from those same files.

Use TOM Builder Windows

When you first see TOM Builder, you see two windows side-by-side. The left window is called the `Object View`, the right the `Component View`.

To work with a TOM object, such as a Tool, you must first put that object in the `Object View`. You put the object there by taking these steps:

1. Click on the tab for the object type, such as the `Tools` tab.
2. When a list of the objects appears, double click on the particular object you want to modify, edit, copy, or take other action on.
3. The object should appear in the `Object View`.

If you are displaying a Tool in the `Object View`, you can expand it to see its Resources, too; however, to modify, edit, copy, or take other action on the Resources, you must repeat the steps above for the Resources object.

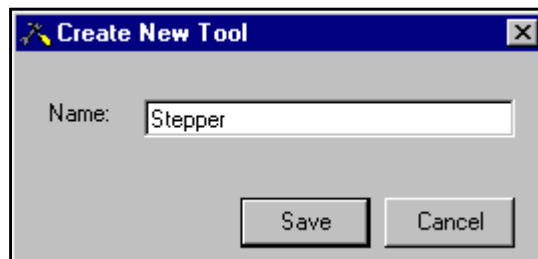
The object you take action on is always the *topmost* object displaying in the `Object View`.

Now, let's take a look at how to form a new Tool.

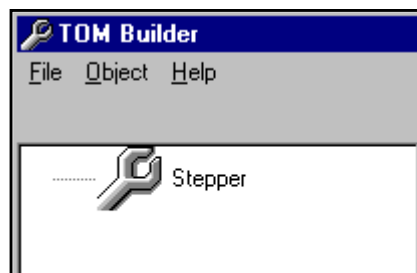
Creating a New Tool

For the Service in this manual, let's create a Tool with two Resources. First, you create the Tool:

1. Click on the Tools tab.
2. From the menu bar, select File => Create new Tool.
3. When the Create New Tool dialog appears, enter the name of the new Tool exactly as you want it to appear in the database.



4. After you click Save, the Determine Tool Manufacturer dialog appears. You can select a manufacturer from the list or create a new manufacturer.
5. After you enter a new manufacturer or select an existing one and click Save, the Determine Tool Manufacturer dialog appears. You can select a manufacturer from the list or, to enter a new manufacturer, click the Create New Manufacturer check box and enter the name next to New Name. Later, the manufacturer and developer display as properties of the Tool in TOM Explorer.
6. Now, when you click Save, the Tool appears in the list under the Tools tab. To display an icon for the Tool in the Object View, list the Tools in the Component View and double click on the Tool's name in the list.



Now you are ready to create some Resources for the Tool. Resources are devices that make up the Tool, such as tubes that make up a furnace. If you are creating a new Tool that does not use existing Resources, you need to add

Resources to the database. For the sample Tool, you can also have conceptual Resources.

Creating a New Resource

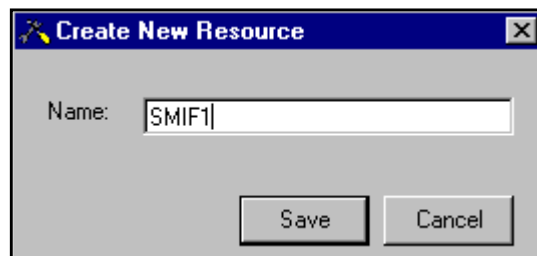
Resources are the physical or conceptual components of a Tool that are each separately programmed to generate a model of the Tool.

For instance, a stepper is a Tool; its input POD and output POD are each distinct Resources, since they behave differently.

Rule of thumb—if the equipment has a SECS Resource ID, then it is a Resource rather than a Tool (although it can be both).

For the sample Tool, let's create SMIF1 and SMIF2 as Resources:

1. In the **Component View**, click the tab for the type of component you want to create (Resources). Select the **File => Create new...**
2. A dialog pops up where you enter the name of the new Resource.



3. Once you have created a new component file you can edit it by double-clicking it in the **Component View**.

Creating a new component makes a new file in the corresponding component directory. The file has a `.tbf` extension. You can see a list of the existing components of a particular type by clicking on the tab for that component in the **Component View**.

Logical Resources

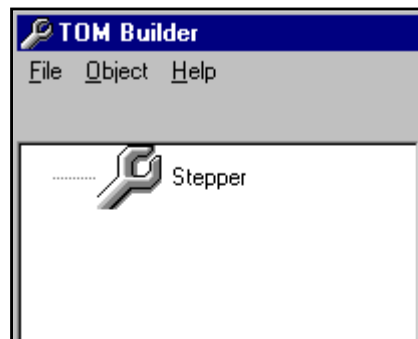
You can also have logical Resources. How do you know a Resource as logical? For instance, if you have a stepper with two SMIF arms, each a Resource, you could make a single logical Resource that represents the three as a unit. This logical Resource is not associated with any physical equipment. You could then attach Services to this “super stepper” logical Resource. Why not make the “super stepper” a Tool? Because if it were a Tool, you could not have Services for it. You must associate Services with a Resource.

Now, you are ready to add Resources to the Tool (what actually happens is that the Tool references the Resources).

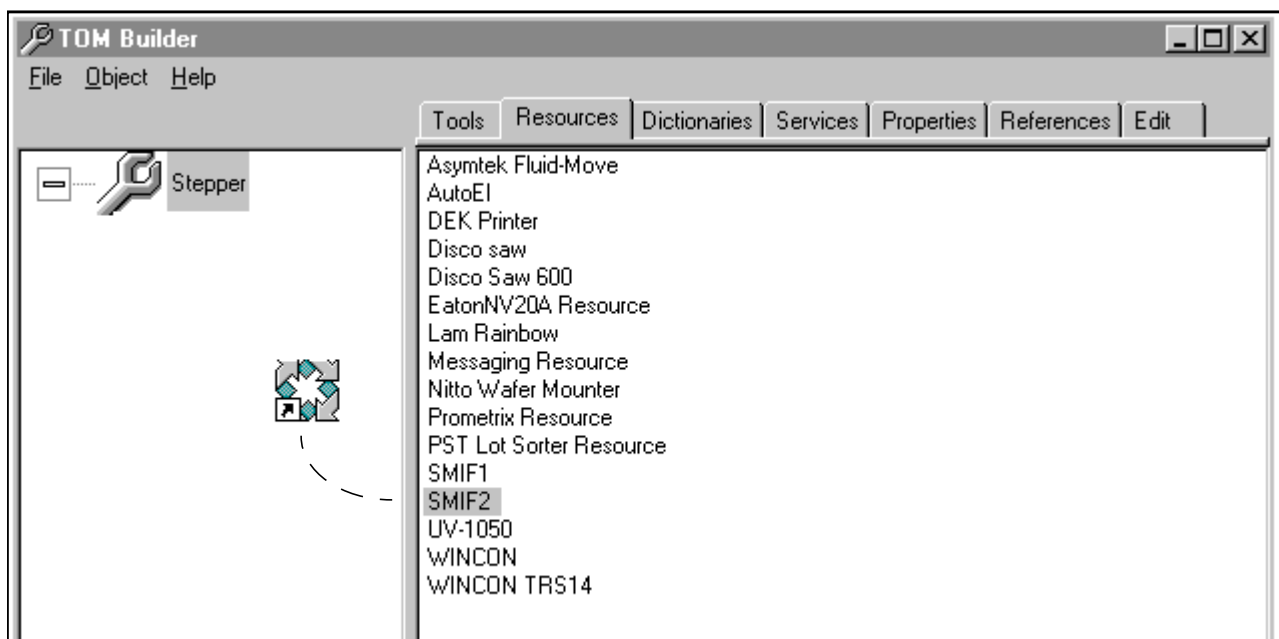
Adding Resources to the Tool

Once you have created a new Tool and created Resources for it, you can assign the Resources to the Tool. You can also follow this procedure to assign more Resources to an established Tool. Take this action for each Resource:

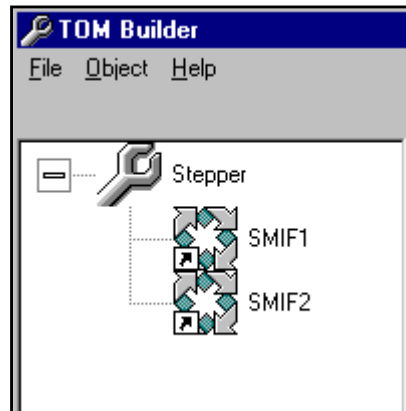
1. To display an icon for your Tool in the `Object View`, first click on the `Tools` tab in the `Component View`, then double click on the Tool's name in the `Tools` list.



2. Now, to add Resources to the Tool, click on the `Resources` tab in the `Component View`.
3. When you see a list of Resources, click on the one you want added to this Tool and hold down the mouse button.
4. Drag and drop the Resource icon onto the Tool icon in the `Object View`.



You should see the Resources appear under the Tool in the `Object View`.

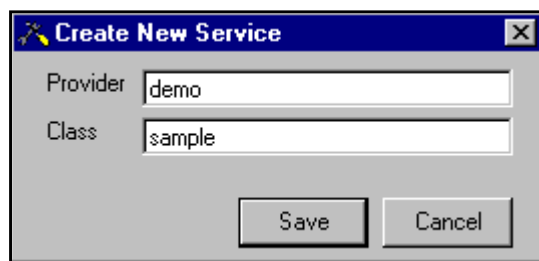


Next, you can select existing Services that match the messages of your Tool.
If you have a custom Service you want to add to the Tool, proceed to the next section.

Adding Your Custom Service to Database

You can add your own Service to the database before it is successfully compiled; however, you cannot use the Service until you have compiled it (Chapter 5 takes you through compiling and debugging the Service):

1. Click on the Services tab in the Component View.
2. For `Provider` enter the root name of the Visual Basic project file.
3. For `Class` enter the root name of the `.cls` file in the Visual Basic project.



(Together `Provider` and `Class` make up the name of the component file.) The illustration shows entering the name of the sample Service just to illustrate how the provider and class fit together to form the name `demo.sample`.

4. Click `Save`. The name of the Service then appears in the list of Services. Next, you must set Properties of the Service.

Setting Properties of Your Service

To set the Properties of a Service, first be sure the list of Services is displaying by clicking the Services tab. Then:

1. Double click on the Service name. The Service appears in the Object View.
2. Click the `Edit` tab to edit the Properties of the Service.

Property	Value
CanClone	True
Comments	
Description	
DictionaryName	
HelpContext	0
HelpFile	
Level	4
Name	sample

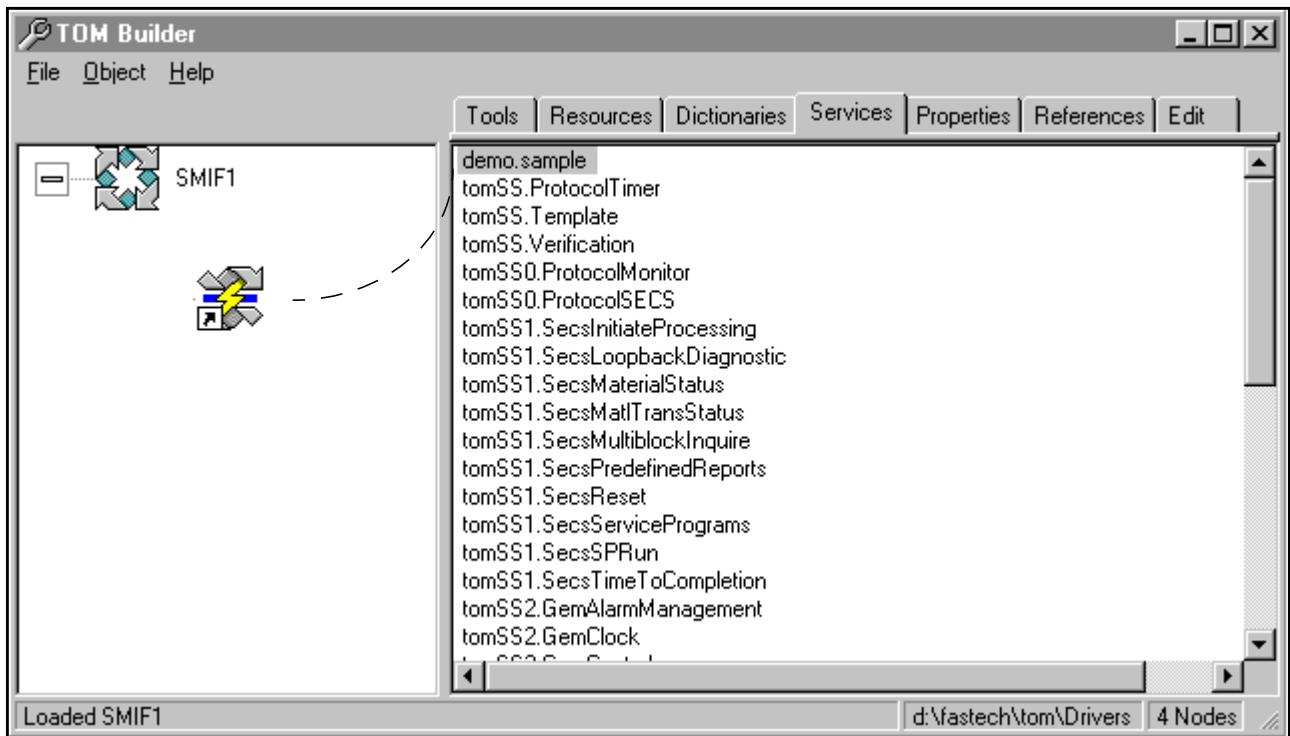
3. The `Name` property (at the bottom) is required. Set the `Name` to the root name of the `.cls` file in the Service's Visual Basic project. For the `demo.sample` Service, `sample` alone is the name. The name of a standard Service should not change.
4. `CanClone` is optional, but defaults to `False`. Set `CanClone` to `True` if you want other Services or Applications to be able to clone this Service. Otherwise, leave it `False`.
5. `Comments` are optional. Enter any comments you want to make in the `Comments` field. To enter a long comment, double click in the field and an `Cell Editor` dialog appears, where you can see more characters.
6. The `Description` is optional. This description later appears in TOM Explorer. To see a larger area, double click in the field and an `Cell Editor` dialog appears, where you can see more characters.
7. The `HelpContext` is the Help context ID to index into the Help file for the Service.
8. `HelpFile` is the name of the Help file, including its `.HLP` extension.

9. `Level` is the level of the Service. The default is 0, but your Service is unlikely to be talking directly to the equipment, as a Level 0 Service does. For details on the meanings of the Service levels, refer to [Service Levels](#).
10. Right click on the Service icon in the `Object View` and select `Save`.
The Service is not ready to use until you assign it to a Resource.

Assigning Services to Tool Resources

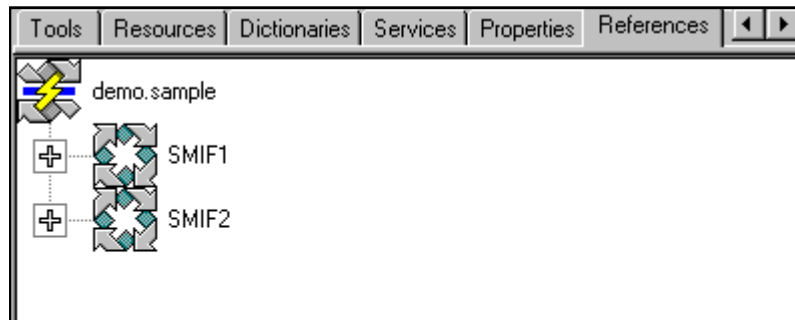
You always assign Services to the Resources of a Tool, rather than to the Tool itself. To assign a Service to a Resource:

1. Click the Resources tab in the Component View.
2. Double click on the Resource you want to assign the Service to. The Resource should appear in the Object View.
3. Click on the Services tab. Here you should find your Service in the list of Services.
4. Select your Service's name and hold down the mouse button.
5. Drag and drop the Service icon onto the Resource icon in the Object View.



1. Click on the icon for the Service in the Object View.

2. Click the `References` tab to see the `Resources` appear under the `Service` in the `Object View`. Under this tab, you now see all the `Resources` that reference this `Service`.



Your custom `Service` is still not ready to use until you have added all the `Attributes` it requires.

NOTE You cannot add `Attributes` to a standard `Service`, only to a custom `Service`.

Creating a New Service Dictionary

There are two types of Dictionaries in TOM Builder—Service Dictionaries and Resource Dictionaries. They each have a `ServiceDictionary` Property that is either `True` or `False`. You establish this Property's setting when you first create the Dictionary.

A Service Dictionary is so named because it is a Dictionary for a set (a single DLL) of Services. For instance, the SECS Standard Dictionary is a Service Dictionary for the SECS Services provided with TOM.

A Service Dictionary defines the `DataDefs` used by a collection of related Services. For instance, the SECS Standard Dictionary defines the `DataDefs` used by SECS Services at levels 0 through 3.

You need to create a new Dictionary for your custom Tool. If your Tool has `DataDefs` that are not defined in any other existing Dictionary (usually not defined in SECS Standard Dictionary), you must create a Dictionary that contains those `DataDefs`. You can have that Dictionary be a copy of the SECS Standard Dictionary that you add your specific `DataDefs` to. The `DataDefs` you create that are for your Service only are called *Service Specific DataDefs*.

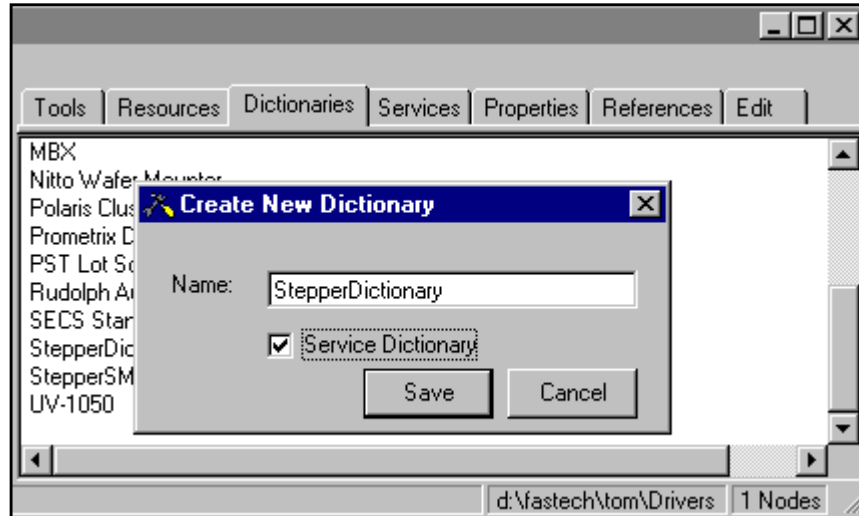
NOTE Developing Service Dictionaries is a specialized field. You should never modify the Service Dictionaries provided with TOM. If, however, you are developing of a new collection of TOM Services, you can copy the original Dictionaries and alter the copies.

Keep in mind that a Service Dictionary is always self-contained—it cannot reference other Dictionaries. Another type of Dictionary in TOM, the Resource Dictionary, can reference other Dictionaries.

Before you can associate a new Dictionary with a Service, you first create the Dictionary:

1. Click on the `Dictionaries` tab in the `Component View`.

2. Select File =>Create New Dictionary and fill in the name of the dictionary.



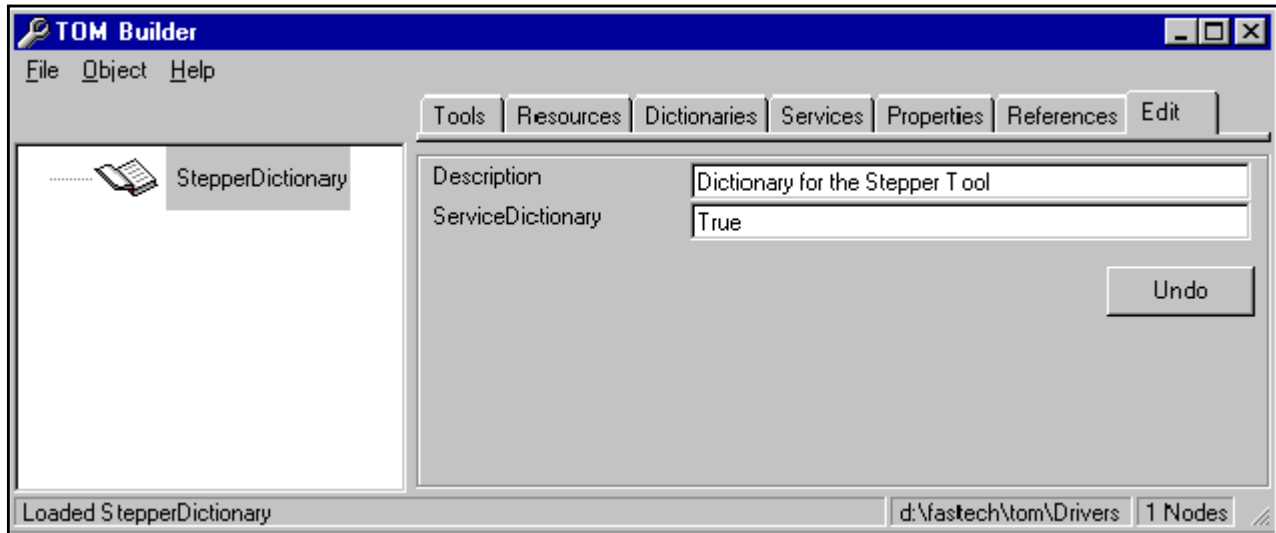
3. If the dictionary is associated with a particular Service, click in the Service Dictionary check box to set this property to True. Once you make it a *Service* dictionary, you cannot assign it to a Resource.

Add Description of Dictionary

To add a description of the dictionary:

1. Click on the Dictionaries tab.
2. Double click on the Dictionary name in the list. It should appear in the Object View.

3. Click the `Edit` tab and fill in the `Description` property. If you decide you want to change the setting of `ServiceDictionary` property (True or False), change it here.



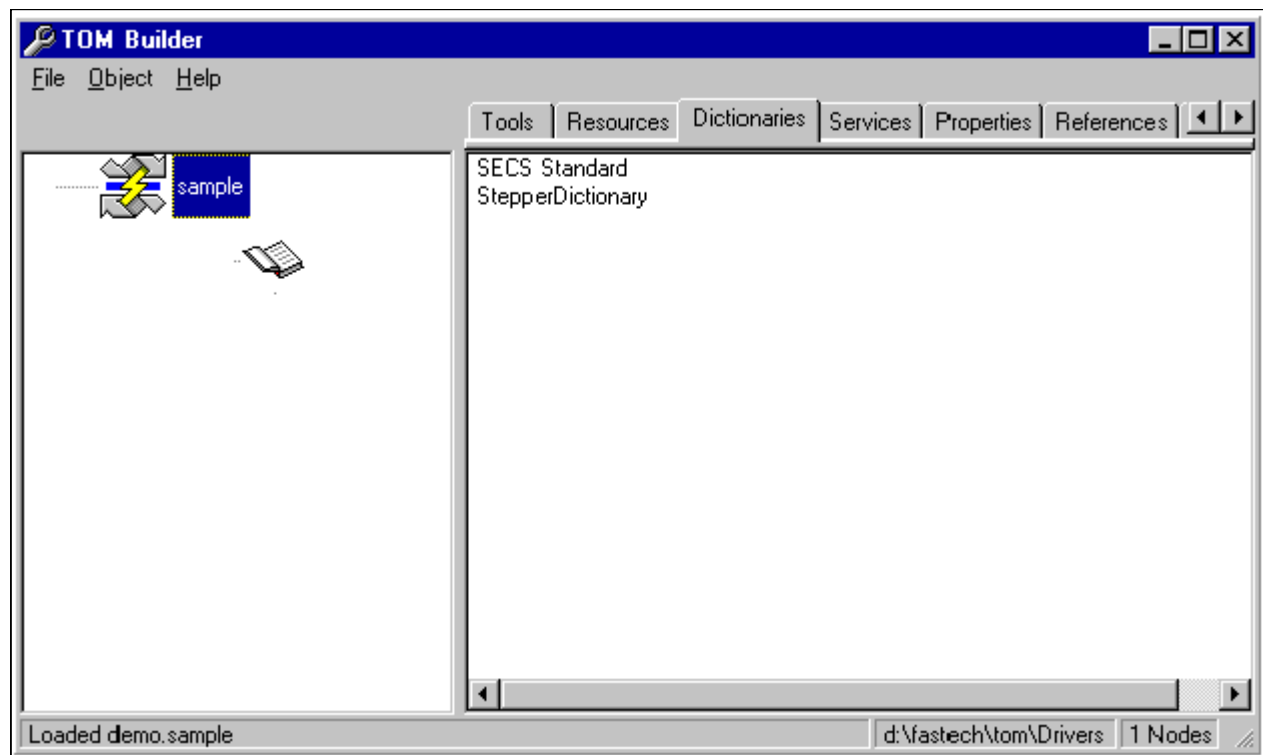
4. Go back to the `Object` view and right click. Select `Save` from the pulldown menu that appears.

Next, you assign the dictionary to the `Service` that later uses it.

Assigning the Dictionary to a Service

To assign a Dictionary to a Service, first be sure it is a Service Dictionary, then:

1. Click on the Services tab in the Component View.
2. When the list of Services appears, double click on the Service you want to assign the Dictionary to.
3. When the Service's icon appears in the Object View, go to the Component View and click on the Dictionary tab. A list of Dictionaries appears.
4. Select a Dictionary in the list and drag and drop the dictionary icon onto the Services icon in the Object View.



Next, you must create a Resource Dictionary.

Creating a New Resource Dictionary

A Resource Dictionary is quite different from a Service Dictionary. It contains all Dictionary information required for a particular Resource of a Tool. Each Resource has one and only one Resource Dictionary. That Resource Dictionary can reference multiple Service Dictionaries. For instance, if the Resource receives SECS messages, then its Resource Dictionary should reference the SECS Standard Dictionary. If the Resource also uses, for instance, the MES Services, since that set of Services use another Dictionary, the Resource Dictionary should also reference the Service Dictionary for MES Services.

In addition, you may have custom Service Dictionaries that work with your own set of Services. The Resource Dictionary may also reference one or more of those Service Dictionaries.

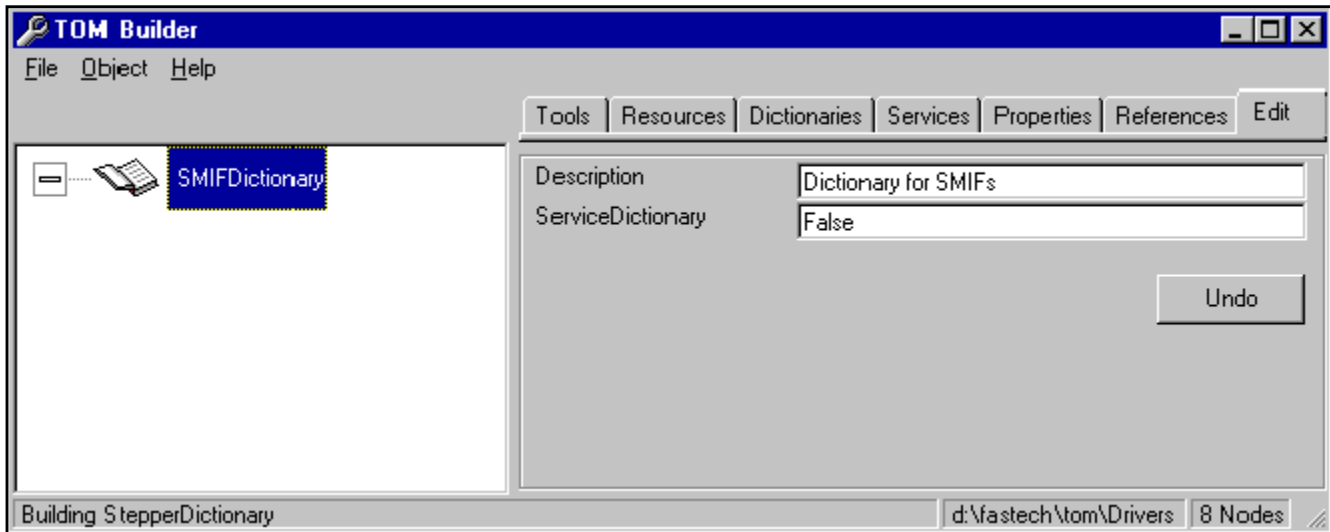
So that you can associate one or more Service Dictionaries with a Resource, you first create the Resource Dictionary:

1. Click on the `Dictionaries` tab in the `Component View`.
2. Select `File =>Create New Dictionary` and fill in the name of the dictionary.



Do **not** click in the `Service Dictionary` check box. If you make it a *Service Dictionary*, you **cannot** assign it to a Resource; also, once it is a *Service Dictionary*, you cannot make it a Resource Dictionary.

After you click *Save*, you can see the *Properties* of the *Service* by double clicking on it until its icon appears in the *Object View*, then clicking the *Edit* tab.

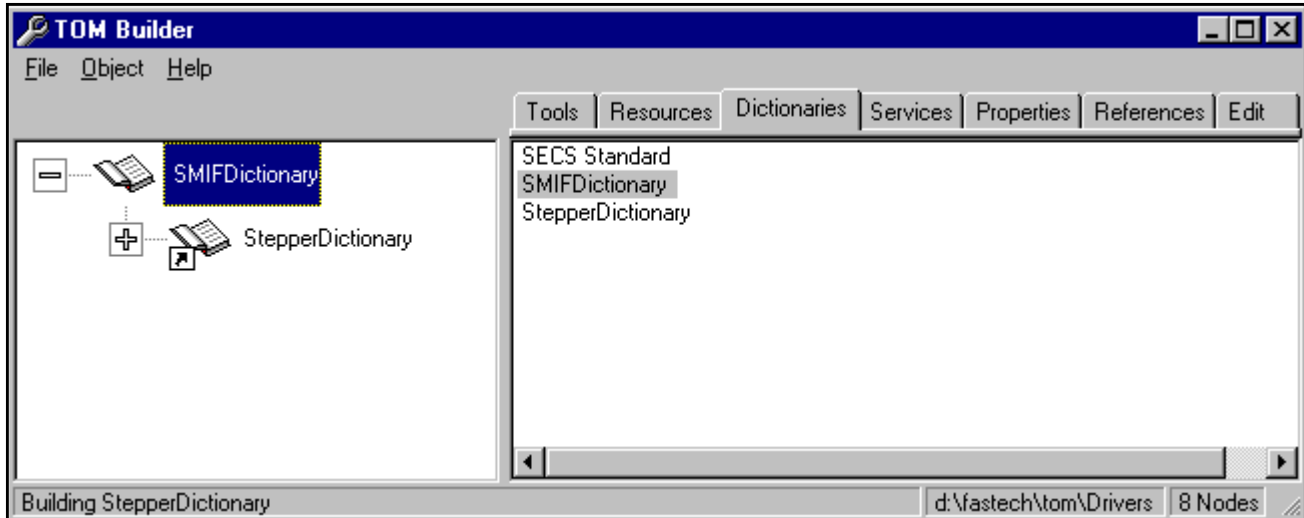


If you want the option of modifying *DataDefs* in a *Service Dictionary*, then you should create a reference to that *Service Dictionary* in the *Resource Dictionary*.

You create a reference to any *Service Dictionary* required by the *Service* the *Resource* uses. Usually the only one you need to reference is the *SECS Standard Dictionary*. However, in the *demo.sample* *Service* you set the *DataDefs* in the *Service Dictionary*, so you need to have the *Resource Dictionary* reference the *Service Dictionary*:

1. Once the *Resource Dictionary* appears in the list of *Dictionaries*, double click on its name until its icon appears in the *Object View*.
2. Click on the *Dictionary* icon in the *Object View*.
3. Find the *Service Dictionary* in the list of *Dictionaries* under the *Dictionaries* tab; then select it and hold down the mouse button.
4. Drag and drop the *Service Dictionary* onto the *Resource Dictionary*.
5. Repeat the previous step for each *Service Dictionary* this *Resource Dictionary* references.
6. Your *Resources* may also require the *SECS Standard Dictionary* if the *Resource* uses a *SECS/GEM/VFEI Service*. If your *Service* clones and executes a method of a *SECS, GEM, or VFEI Service*, or adds *DataDefs* to or modifies *DataDefs* of one of those *Services*, you must include the *SECS Standard Dictionary* in your *Resource Dictionary*. You should add other

Service Dictionaries to the Resource only if the Service documentation specifically states that the Dictionary is required.



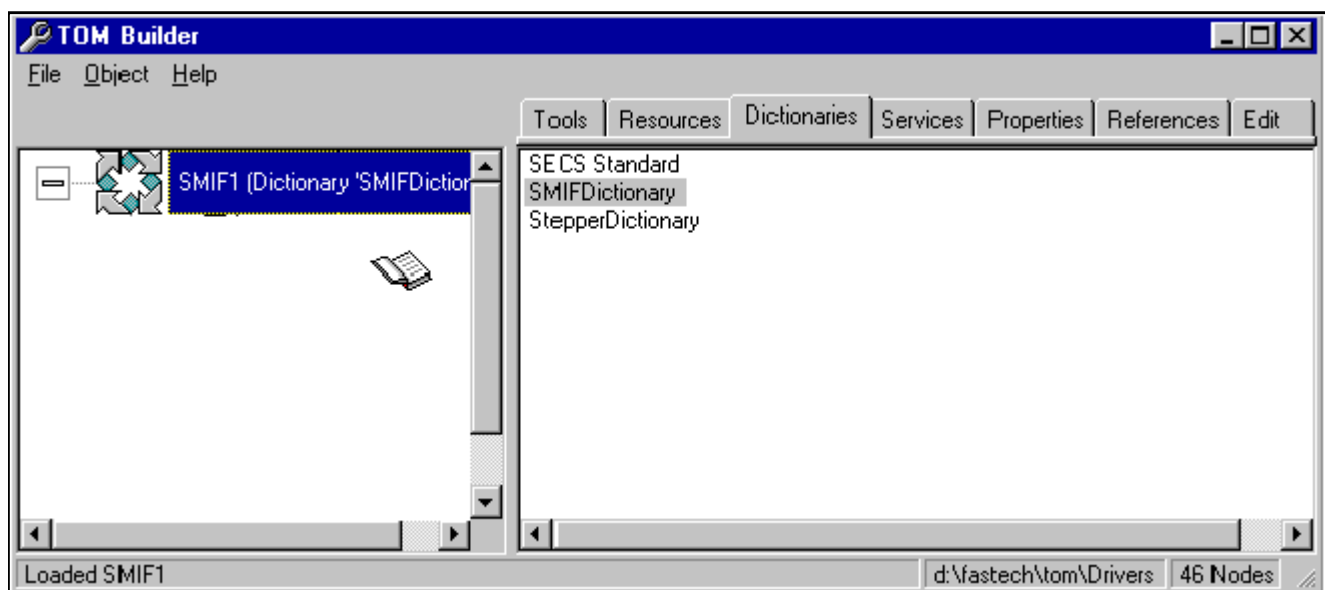
7. Go back to the Object view and right click. Select Save from the pulldown menu that appears.

After you create a Resource Dictionary, you assign the Resource Dictionary to Resources.

Assigning the Dictionary to Resources

To assign a Dictionary to particular Resources, first be sure it is not a Service Dictionary, then:

1. Click on the Resources tab in the Component View.
2. When the list of Resources appears, double click on the Resource you want to assign the Dictionary to.
3. When the Resource's icon appears in the Object View, go to the Component View and click on the Dictionary tab. A list of Dictionaries appears.
4. Select a Dictionary in the list and drag and drop the dictionary icon onto the Resources icon in the Object View.
5. If you are creating a custom Dictionary for a piece of SECS or GEM equipment, be sure to assign the SECS Standard Dictionary to its resource or resources.



6. In the Object View and right click. Select Save from the pulldown menu that appears.

Now that you have some Dictionaries, you need to put DataDefs into them.

Creating DataDefs

A DataDef is like a template. Once you have defined a DataDef you can load it into the Service specific area of the Service for customized use. You can use DataDefs from other Services that your Service is working with as well as DataDefs defined specifically for your Service in its own Dictionary.

What Can Your Service Do with TOM DataDefs?

Your Service can use DataDefs as templates for DataItems it needs. Your Service may:

- Use a subset of DataDefs from its own Dictionary
- Add DataDefs from another Service's Dictionary

For example, your Service might copy the DataDefs under `Equipment Constants` from the SECS Standard Dictionary by “loading” them. The Service can then use those constants.

Another example might be to use the `ALID` DataDef (an input to the `Enable Method` of the *GemAlarmManagement* Service) to set up a particular list of Alarms for that Service from within your Service.

Create DataDefs in Your Service

To create DataDefs for your Service:

1. Create a constant for any DataDef for the Service (which you later add to the database) or any DataDef the Service uses from a dictionary already existing in TOM. Some examples of constants in the sample Service:

```
Private Const DD_DD1 = "DataDef1"
Private Const DD_CHILDA = "ChildDataDefA"
Private Const DD_CHILDB = "ChildDataDefB"
Private Const DD_DD2 = "DataDef2"
```

2. Later, in `OnInitialize`, you load the DataDefs into the reference:

```
Set m_oDataDef1 = srvLoadDataDef(m_oService, Nothing, DD_DD1)

Set Method = srvDefineMethod(m_oService, METHODNAME, "Text")
srvAddDataItem m_oService, Method.Inputs, m_oDataDef1
```

3. If you have not already defined the DataDef in the database, you should add it to the Dictionary associated with this Service using either the TOM DB Editor or the TOM Builder. Refer to the next section (or the TOM Builder Help file) for details.

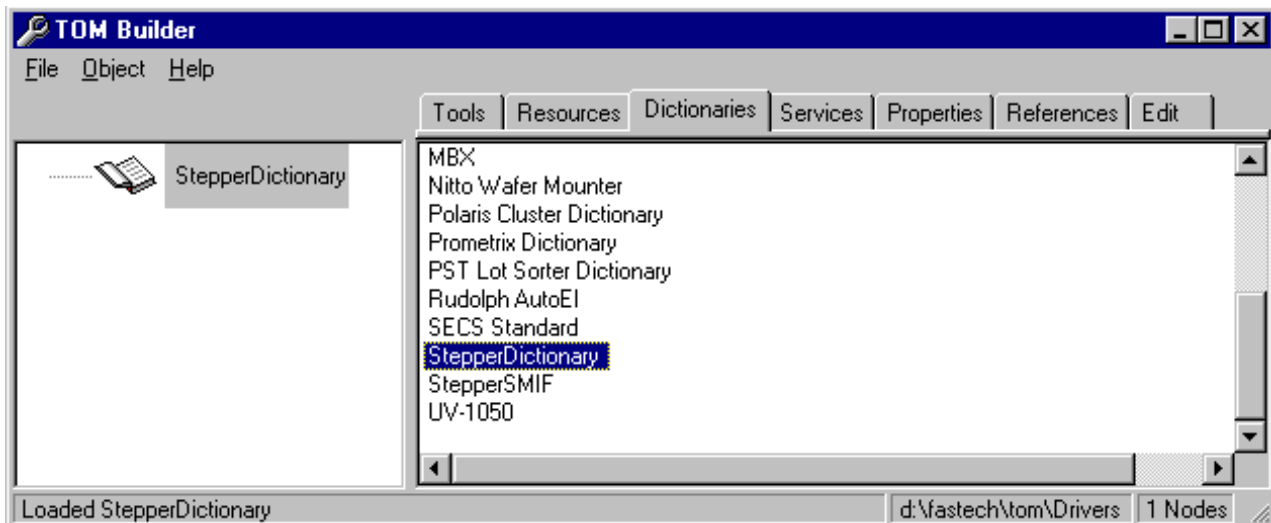
Create DataDefs in the Dictionary

If the Dictionary is a Service Dictionary, it must be self-contained—the parent of a given DataDef must be defined within the same Dictionary if it is a Service Dictionary.

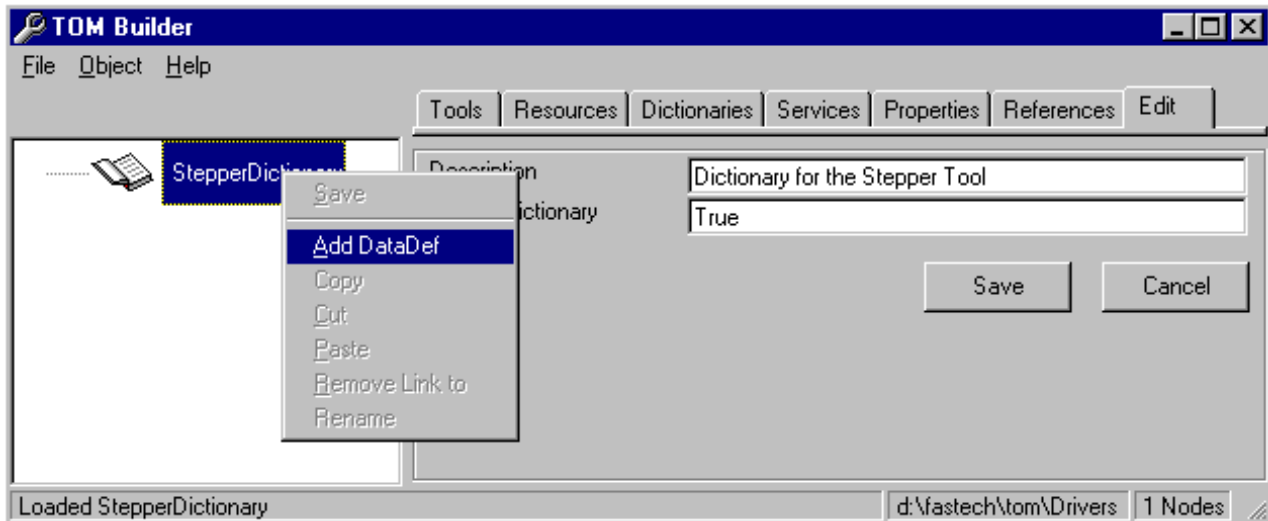
By comparison, Resource Dictionaries are not self-contained. The parent of a DataDef in a Resource Dictionary may be defined in the same Dictionary or in a Service Dictionary.

This structure is designed to help you, the Service/Driver developer, create a skeletal Dictionary that contains a structure that makes sense for the Service. By forming the skeletal structure, you create rules that TOM Driver developers can follow when creating Dictionaries for Service. More than one Service can (and usually does) reference the same Service Dictionary. To add the Service specific DataDef to the Dictionary:

1. Click on the Dictionaries tab in the Component View to see a list of Dictionaries.
2. Double click on the Dictionary in the list that you want to add the DataDefs to. The Dictionary should appear in the Object View.



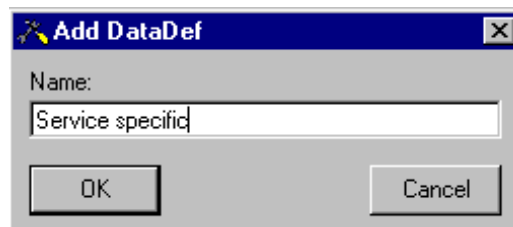
3. In the Object View, right click on the Dictionary name and select Add DataDef.



NOTE For New Services

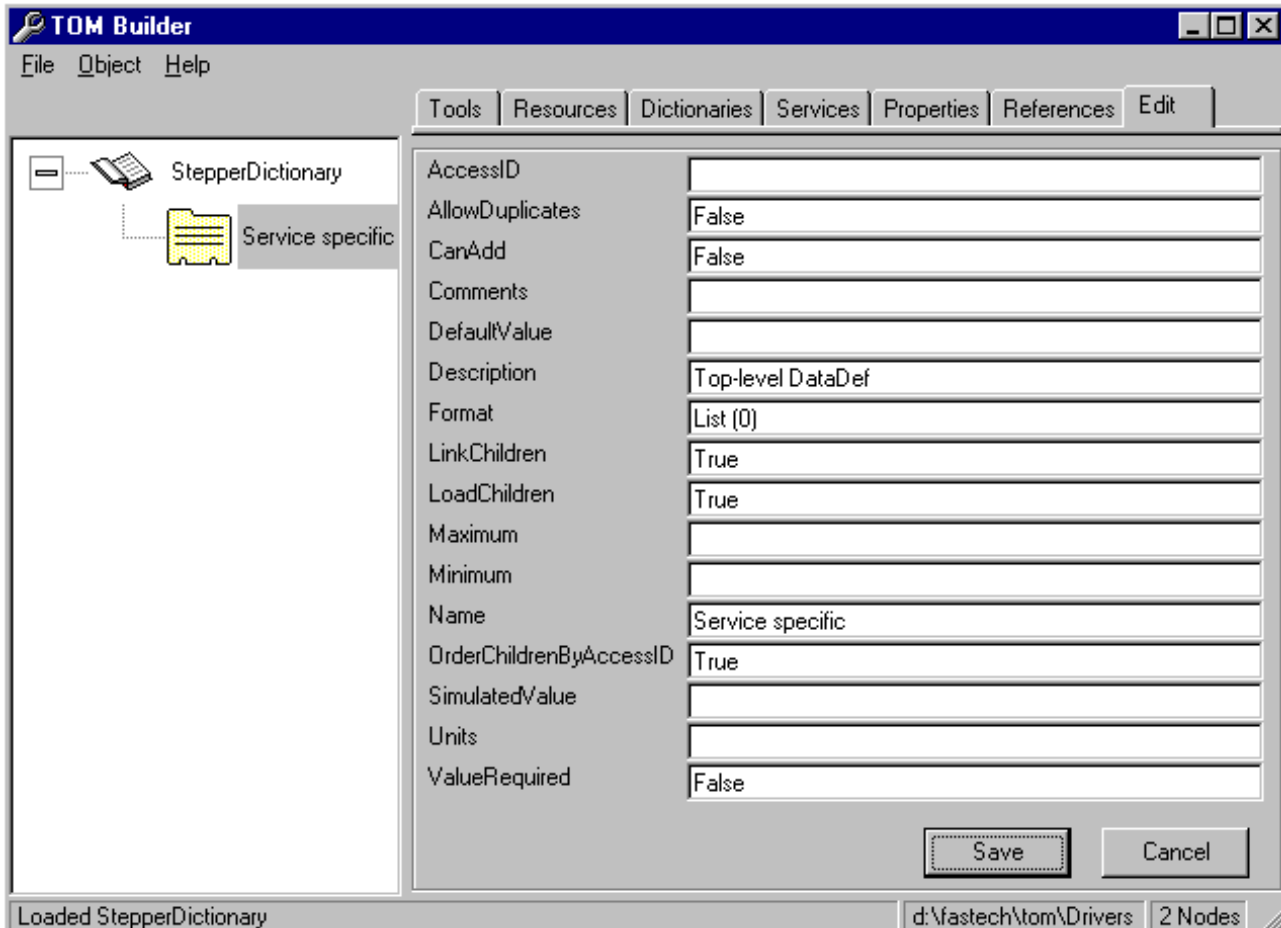
For a new Service, you must add the Service specific DataDef.

4. When the Add DataDef dialog appears, to create the Service specific DataDef, be sure you enter it exactly as shown below, with a space between the words and a lowercase S starting the word *specific*.



5. After you click on OK, an icon for the DataDef should appear in the Object View.

- Click on the DataDef's icon; then go to the Component View and click on the Edit tab.



The editable properties of a DataDef appear in the Component View. For the Service specific DataDef or any other parent DataDef, you must set the required Properties:

- ◆ **Format**—Must be `List (0)` for any top-level DataDef that has children.
- ◆ **LinkChildren**—Must be `True` for any parent DataDef, including the Service specific DataDef. When it is `True`, TOM automatically creates links between the parent DataDef and its children when you instantiate the Tool that uses this DataDef.
- ◆ **LoadChildren**—Must be `True` for any parent DataDef, including the Service specific DataDef. When it is `True`, TOM automatically loads the children when you load the parent DataDef.

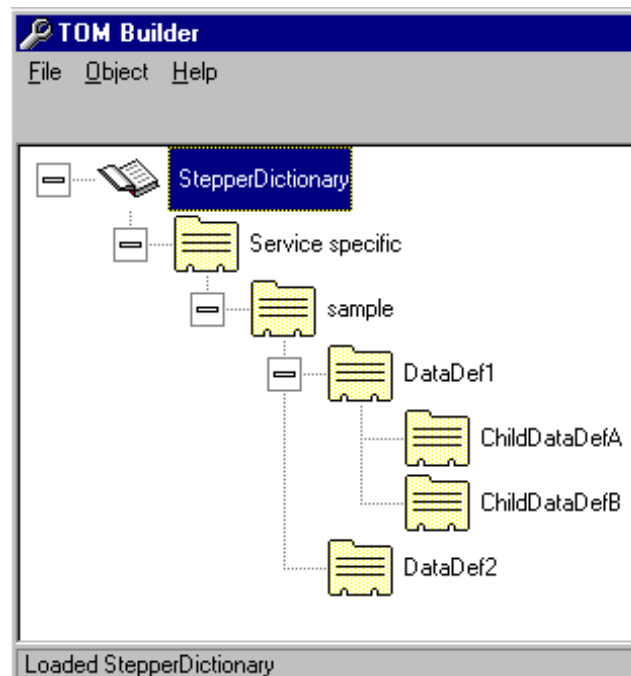
- ◆ **Maximum**—For a parent DataDef, set to the maximum number of children the DataDef can have. For a child DataDef, set to the maximum value it can have.
- ◆ **Minimum**—For a parent DataDef, set to the minimum number of children the DataDef must have. For a child DataDef, set to the minimum value it must have.
- ◆ **Name**—Set to the name of the DataDef with the exact spacing and capitalization you used when you created the DataDef.

Be sure to save the DataDef:

1. Click Save in the Component View.
2. Right click on the DataDef icon in the Object View and select Save.

Add Children to the Database

When you add child DataDefs, for the Service specific DataDef, you must have a child DataDef that has the name of the class. For the *demo.sample* Service, the child must be named *sample* (see below). This DataDef must also be a parent, so it has the `List(0)` format and other settings appropriate for a parent DataDef. Below this DataDef, you add the entire hierarchy of other DataDefs that are specific to your Service.



Loading DataDefs in Your Service

1. In `OnCreate`, you can load the DataDefs using the `srvLoadDataDef` handler support routine.

Load a Top-Level DataDef from Dictionary

If the DataDef you are loading is immediately under a top-level Dictionary object, you can pass `Nothing` as the parent.

The routine returns a DataDef object that belongs to this Service, of type `tom.DataDef`.

For example, to load the "SECS elements", a top level DataDef in the Standard SECS Dictionary, into the `SECSElements` variable, you enter:

```
Set SECSElements = srvServiceDictionaryRoot(m_oService)._
Item("SECS elements")
```

`srvServiceDictionaryRoot` returns the parent DataDef from the top level under the Dictionary. Later this entire branch of the SECS Standard Dictionary appears in your Tool's Dictionary in TOM Explorer.

Cloning DataDefs

You can create clones of DataDefs from the Dictionary to use in your Service. You create a clone using `srvCloneDataDef` inside your `OnCreate` handler method.

The `srvCloneDataDef` routine takes these arguments:

- *ToParent*—Name of the service being developed.
- *FromDataDef*—Parent DataDef of the collection of DataDefs being loaded.
- **NewName**—Optional. Set equal to a string containing the name of the new DataDef. If you leave out this argument, the routine uses the name in *FromDataDef*.
- **Children**—Optional. Set equal to a `True` if you want children defined, `False` if not.

The routine returns a reference to a new DataDef object.

To make clones of the SECS elements in the Standard SECS Dictionary, you first create a variable to receive the reference to the new DataDef object, then use `srvCloneDataDef` to clone the entire set of DataDefs:

```
Set m_oLocSECS = srvCloneDataDef(srvServiceDataDef(m_oService), _  
SECSElements.Item("SECS elements"), NewName:="My SECS elements", Children:=T:
```

You later see the clones as `DataItems` in the Dictionary under the Service Specific area for your service.

Once you have created the clones in `OnCreate`, you have set up the “empty shells” for DataDefs, but the actual DataDefs do not yet exist. Later, when you clone the Method that uses these DataDefs, you actually fill in the DataDefs and assign them `DataItems`.

Now, you can use the DataDef clones within your Service.

Creating Attributes

An Attribute is a piece of information about the Tool or about a lower level Service.

What Can Your Service Do with TOM Attributes?

Your Service can do the following with Attributes:

- Read values of Attributes from a lower level Service. For instance, you can determine the comm port the tool is connected to over RS-232 cable by retrieving the `PortID` Attribute of the *ProtocolSECS* Service.
- Set Attributes of a lower level Service to change its behavior. For instance, it can set the value of the Tool's `IPAddress`, which is an Attribute of the *ProtocolSECS* Service.
- Have its own Attributes, which you must add to the database using either TOM Builder or TOM DB Editor.

Create Attributes in Your Service

Let's take a look at how to create Attributes in your Service. For the sample Service, you could create an Attribute that you set to enable or disable the `ToolEvent` Event:

1. To define the attribute inside the class, you can define constants and variables to represent the attribute:

```
Private Const TOOLEVENTENABLE = "ToolEventEnable"  
Private Att_ToolEventEnable As Boolean
```

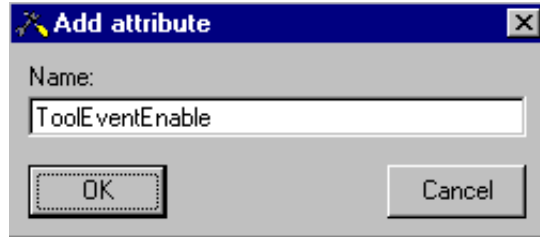
2. Define `ToolEventEnable` in the database using TOM DB Editor or TOM Builder. When you define the attribute in the database, you also assign it a default value, which becomes the attribute's value after it is initialized. For more detail, refer to the next section.

Add Attributes to the Database

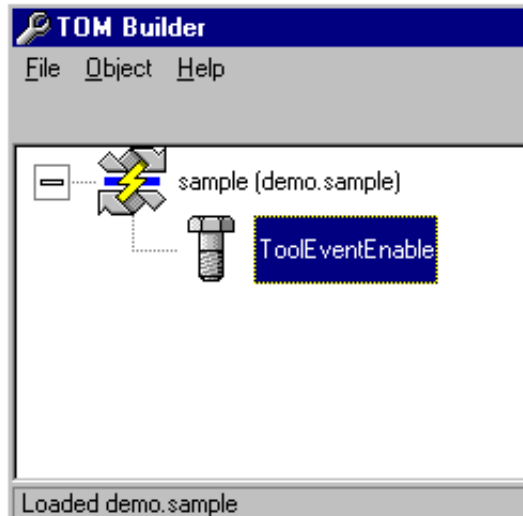
To add an Attribute of the Service to the database:

1. Click on the `Services` tab in the `Component View`.
2. In the list of `Services` that appears in the `Component View`, double click on the Service you want to add the Attribute to.

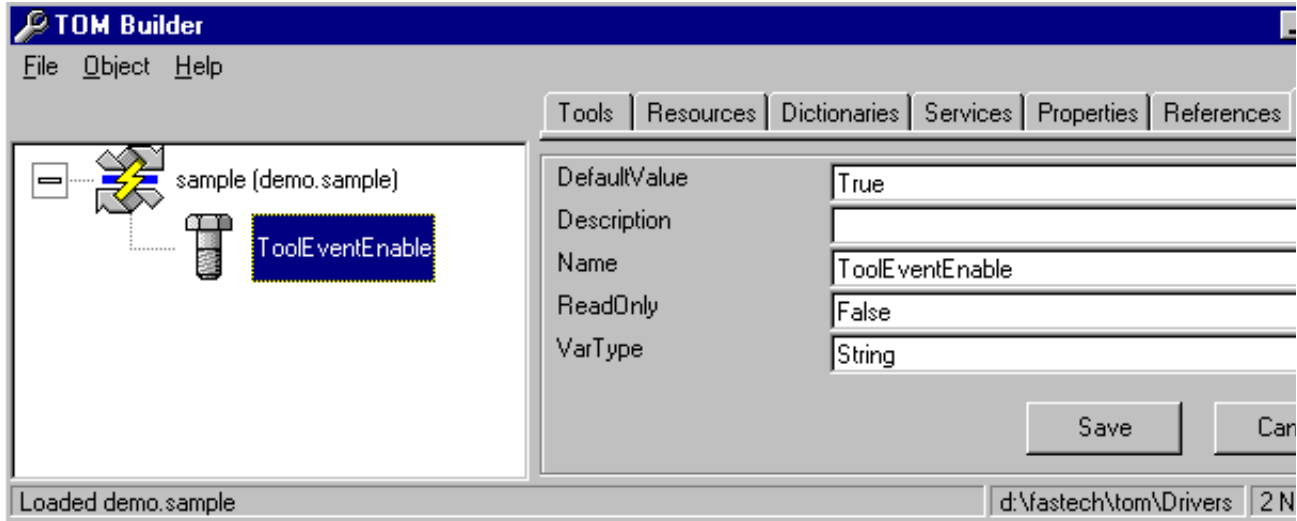
- When the Service appears in the Object View, right click on the Service icon and select Add Attribute from the pulldown menu.



- The Attribute icon should appear below the Service icon in the Object View. When it does, you can click the Edit tab to edit the properties of the Attribute.



- Set the DefaultValue, enter the Name, and set ReadOnly to True or False, and set the data type in VarType to a Visual Basic data type, such as String, Integer, or Variant.



- Click Save in the Component View.
- Right click on the Service icon in the Object View and select Save.

After you have taken all steps in this section, ToolEventEnable is an Attribute of your Service.

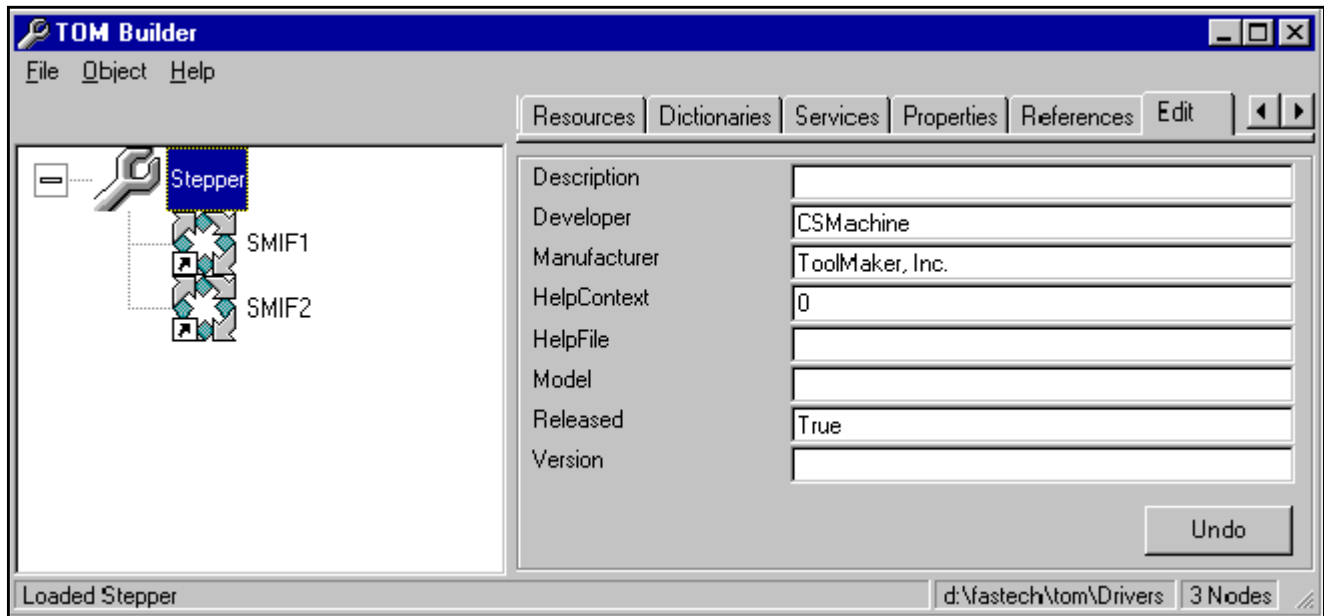
You can see the Attributes in TOM Explorer. For example, below you see the Attribute from the sample Service.



Finalizing Tool by Releasing It

When you have finished creating the Tool, you cannot use the Tool until you release it, as follows:

1. Click on the `Tools` tab.
2. In the list of Tools that appears in the `Component View`, double click on the Tool you want to release. An icon for the Tool should appear in the `Object View`.
3. Click on the icon for the Tool and then click on the `Edit` tab in the `Component View`.
4. When the Properties for the Tool appear in the `Component View`, change the `Released` property to `True`.



5. Right click on the Service icon in the `Object View` and select `Save`.

If you have carried out all the preceding steps and released your Tool, you cannot use it until you build the database.

Building TOM Database (Containing New Tool)

The Build operation creates a new STATIONworks Database. The component files are inserted into the database.

STATIONworks cannot use the component files of the TOM Builder directly. You must build a database before you can test it with STATIONworks.

Before you proceed to build, you can remove any excess Services or Dictionaries that you are not using. Select `File => Delete` from the menu to delete any object. This action pares the database down to only the necessary components.

To build a TOM database:

1. Select `File => Build Database...` from the menu bar.
2. In the dialog box that displays, enter the name of the TOM database to build. The default database name that appears is the one Brooks provides. ***Be sure to assign your own database name. Do not overwrite the default database.***

NOTE You can rebuild the database this same way after adding a single Tool or several Tools. Brooks advises that you test each new Tool before checking in your database.

3. To test each Tool, you should open the Tool in TOM Explorer and run the Verification Service on each Service for each Resource of the Tool.

Once you have built the database and have tested each Tool, you are ready to add the `.tbf` files TOM Builder has created to the revision control system.

To alter the database, check out the `.tbf` files, edit the database, and then rebuild it.

Introduction

Topics in This Chapter

Preparing to Use Your Service in TOM Explorer, p. 5-2

Running Your Service in Debug Mode, p. 5-4

Executing Methods through TOM Explorer, p. 5-8

Verifying the Service from TOM Explorer, p. 5-11

Exiting TOM Explorer, p. 5-14

Compiling Your Service—Final Compile, p. 5-15

Testing Your Service, p. 5-15

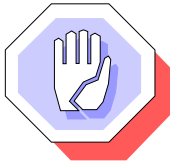
Using Your Service in an Application, p. 5-15

After you have completed the handler methods of your Service and it has no syntax errors, you are ready to Debug it in runtime mode. In this chapter, you see how to use TOM Explorer for debugging.

Preparing to Use Your Service in TOM Explorer

Before you can use your Service in TOM Explorer, you must add the following information about your service to the database using either the DB Editor or TOM Builder:

- `Service ClassName`— Set to the `Name` property of the Visual Basic class module. This property applies to only a single service within your project.



CAUTION

Remember to place your unique prefix at the beginning of your DLL and your Service name.

- `ServiceProvider`—The `Project Name` you assigned to the Visual Basic project earlier. Brooks recommends that the .DLL's root name be the same as the name of the Visual Basic project. The `ServiceProvider` *must* be the same as the Visual Basic `Project Name`.

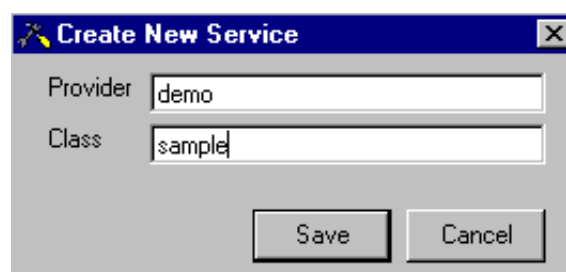
The same `Provider` applies to all the services in your project.

You put this information into the database using either the TOM Builder or the TOM DB Editor.

Make Service Available to TOM Explorer with TOM Builder

To make it available to TOM Explorer, you must add your service to the TOM database. To add the service:

1. Click the Services tab.
2. Go to the menu bar and select `File => Create New Service`.
3. When the Create New Service dialog appears, enter the `Provider` and `Class` in the edit boxes, as shown below:



The name of your `Provider` should contain the company prefix (*MY*) followed by a code that identifies the type of `Service` or `Services` in the file.

The `Class` is simply the root name of the `.cls` files for the service.

4. Click `Save` to add the `Service` to the database.

Follow the instructions provided with TOM Builder to associate `Tools` and `Resources` with your `Service`.

Set Required Attributes in Database

Be sure you have set `Attributes` your `Service` requires in the database using either TOM Builder or TOM DB Editor.

For the sample `Service` the `ToolEventEnable` attribute should already be set to `True`.

Running Your Service in Debug Mode

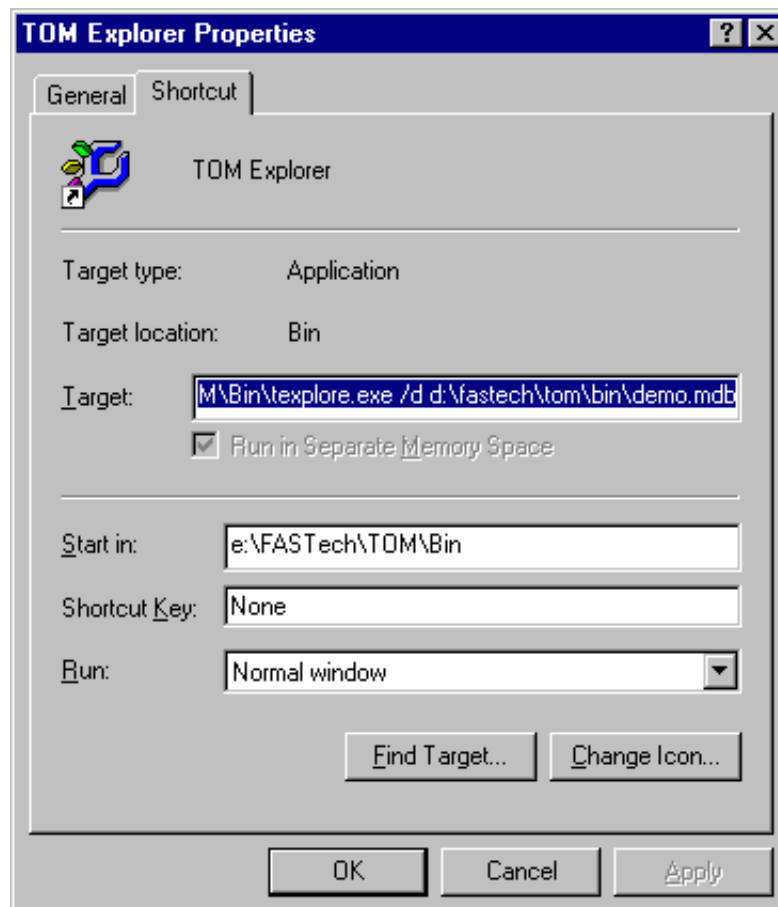
Debugging a Service is not exactly the same as debugging any other Visual Basic program. What is different about it? Well, first, you must fully compile the Service to make it available to the Tools you have associated it with in the database.

Then your Service doesn't really run unless your Tool is using it.

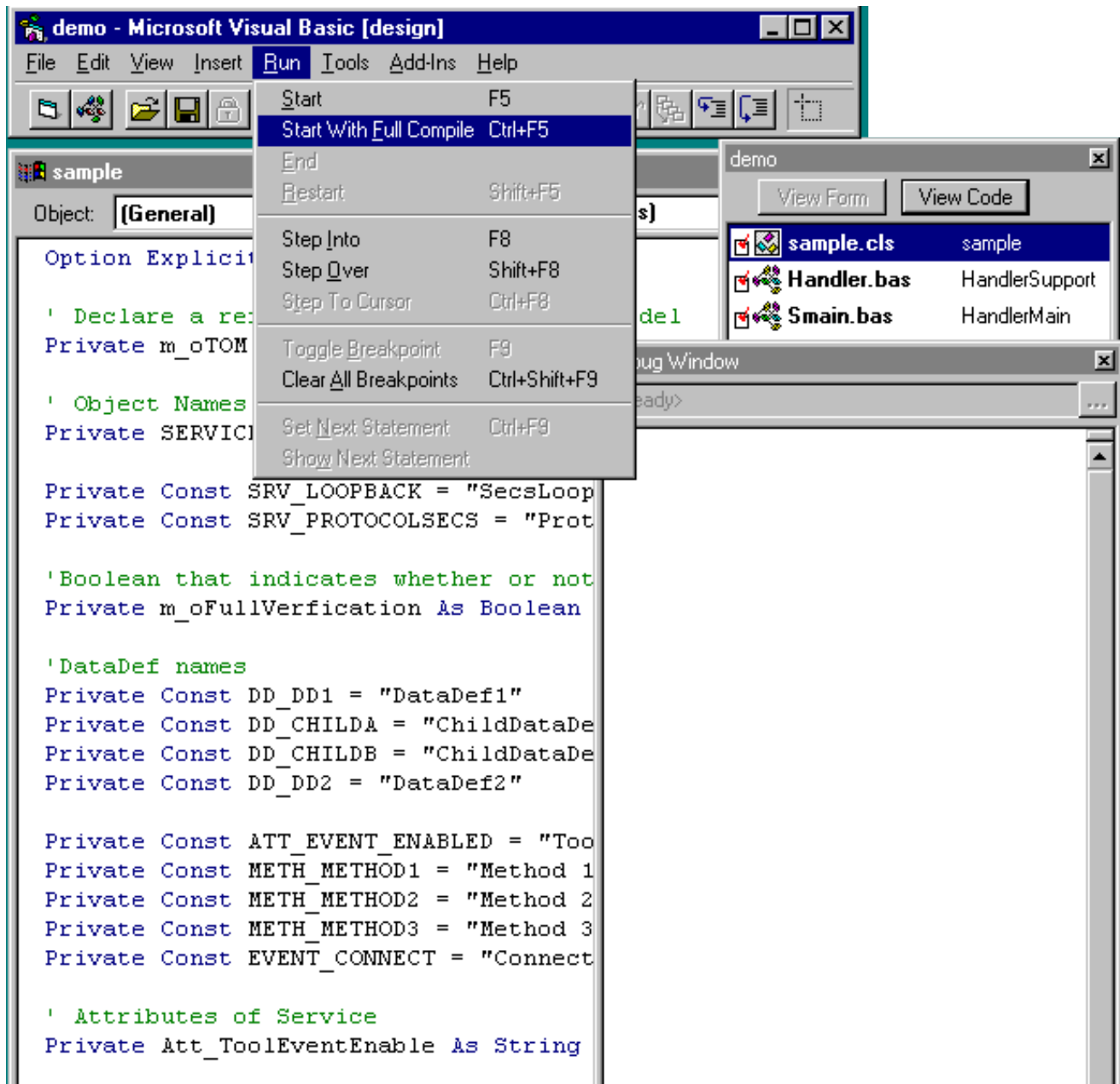
So, how do you debug the Service? You carry out a series of steps:

1. Begin by creating a shortcut to TOM Explorer and setting it to run the database containing your Tool. Since that may or may not be the same as the standard database, be sure to set the path to the database using the /d option and following it with the full path to the database (or the local file name if it is in the same directory as TOM Explorer):

```
C:\FASTech\TOM\bin\texplorer.exe /d tomDB.mdb
```



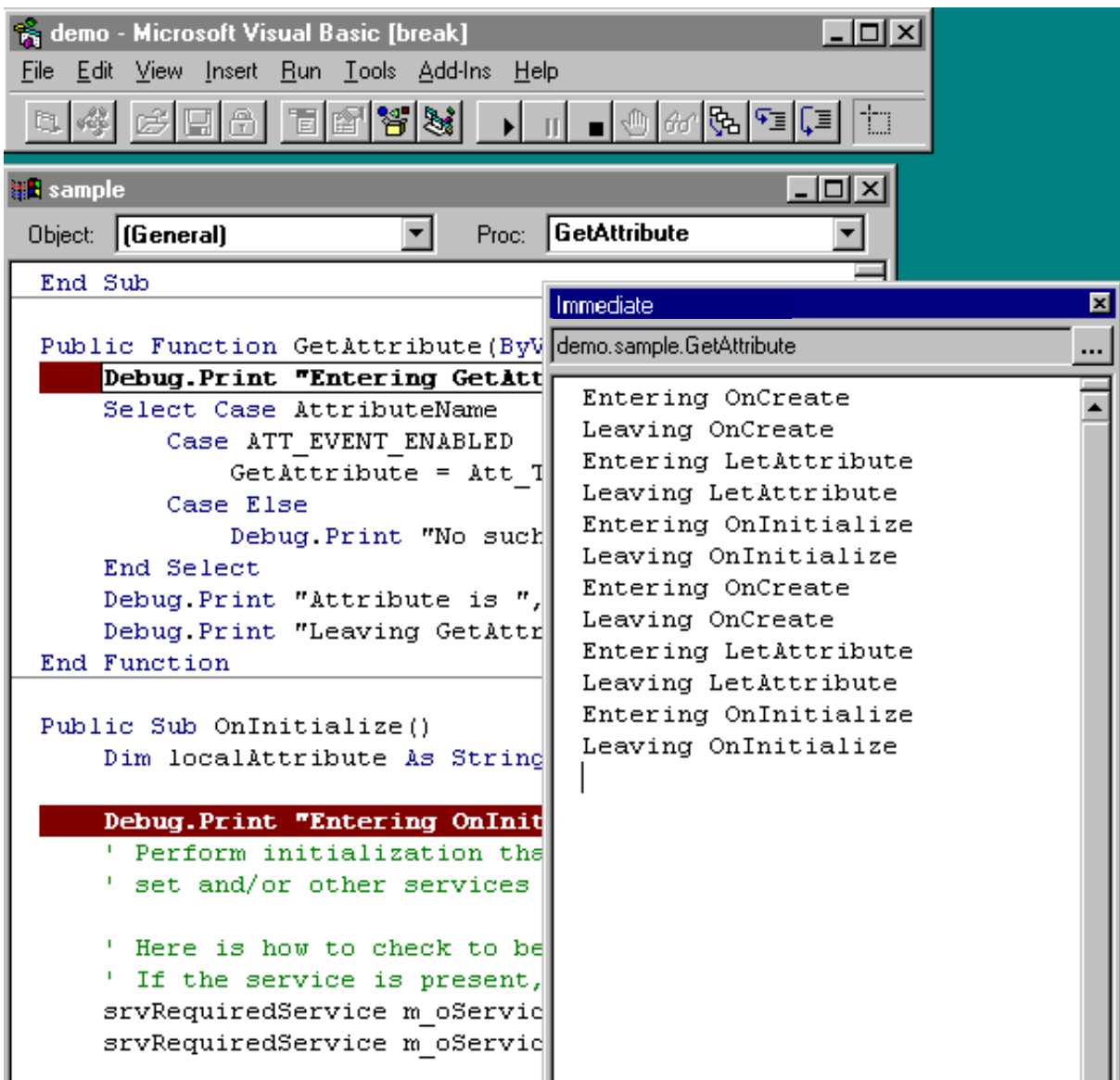
2. In your class code, set some breakpoints in Visual Basic. It's a good idea to put a breakpoint at the beginning of each handler method and at key lines inside them.
3. Go to the Visual Basic menu bar and select Run => Start with Full Compile. This way, your Service is completely compiled for your Tool to use.



4. Notice that nothing appears to be happening. To see action in the Immediate window, start TOM Explorer and open the Immediate

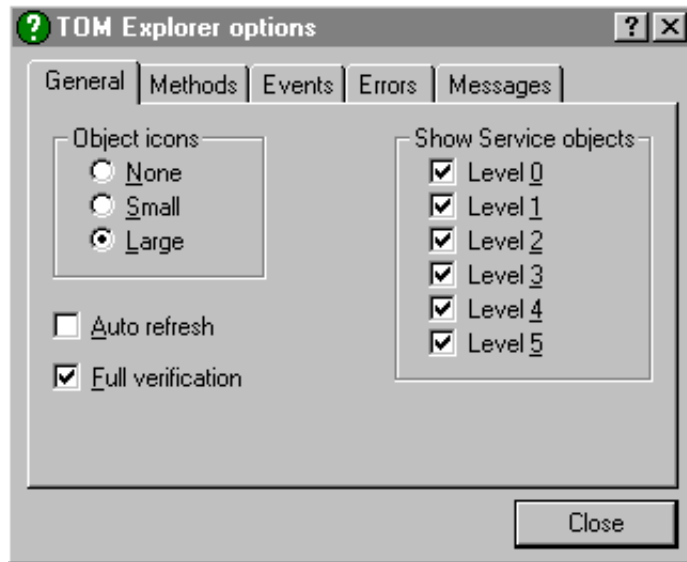
window. In a moment, the Create Tool Object window appears and your Tool should be in the list of Tools. Select your Tool. To run the *demo.sample* service, select the Stepper Tool.

- When the Visual Basic debugger stops on the first breakpoint in your code, start stepping through the code. You'll step through `OnCreate`, `LetAttribute`, and then `OnInitialize`, in that order. Then you'll step through them again—why? Because the TOM Explorer runs them for each Resource associated with the Tool and the Stepper Tool has two resources—SMIF1 and SMIF2.

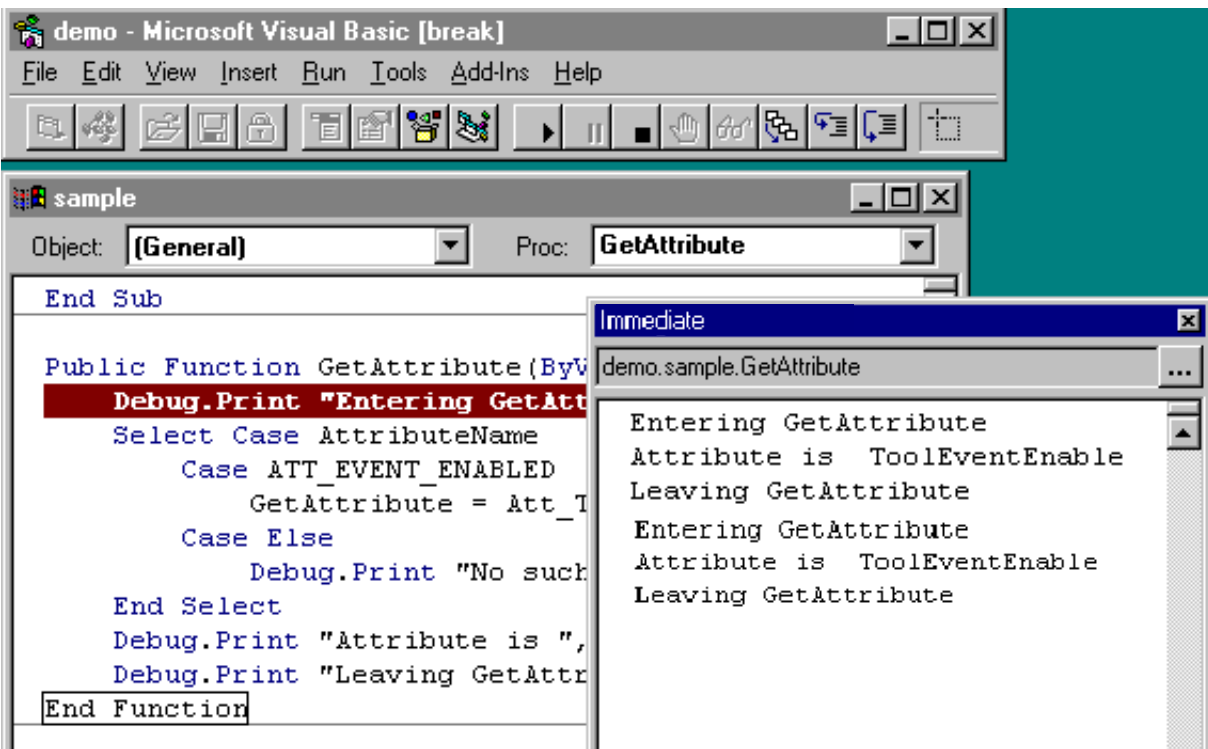


- You may see some of the text printed to the `Immediate` window print more than once. If you do, it is because TOM Explorer's `Auto Refresh`

option is on. You can turn it off by selecting View => Options from the menu bar, going to the General tab, and then toggling off the Auto Refresh check box.

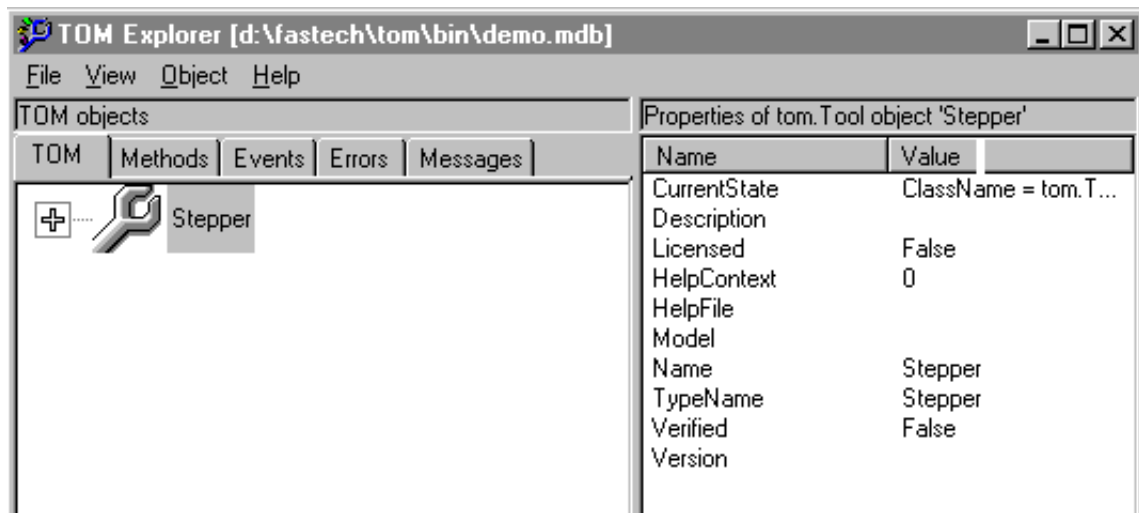


7. Continue to step through the code and you see TOM run GetAttribute twice, once for each Resource.

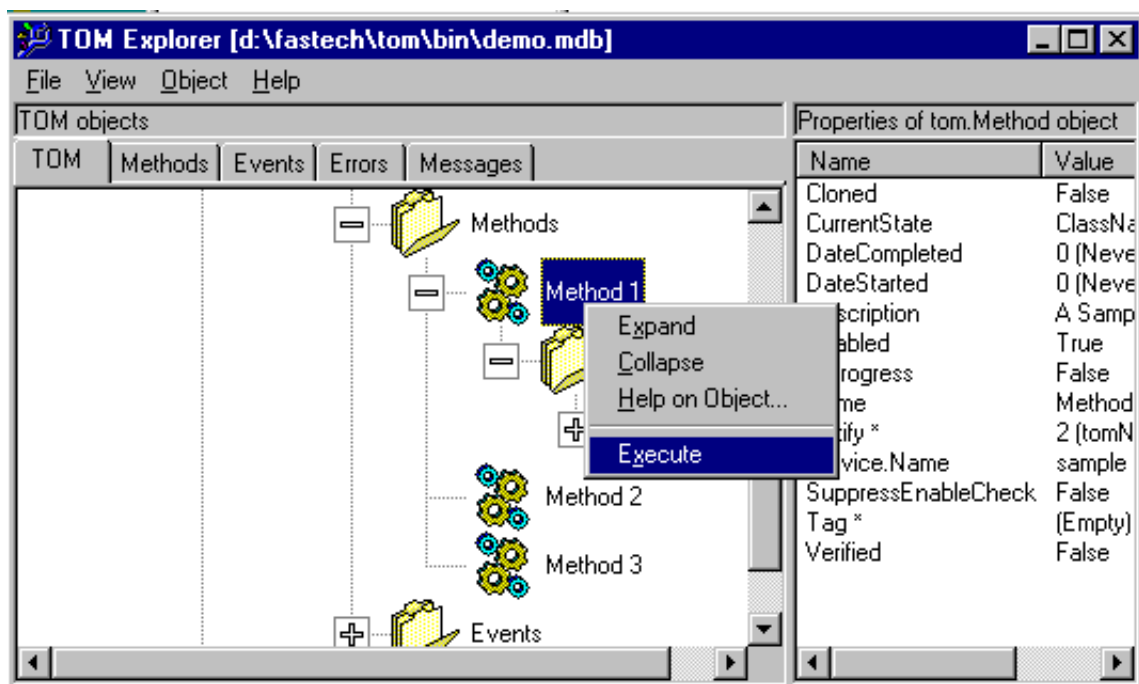


Executing Methods through TOM Explorer

After TOM has retrieved the Attribute settings, it waits for you to take action. The Tool displays in TOM Explorer.

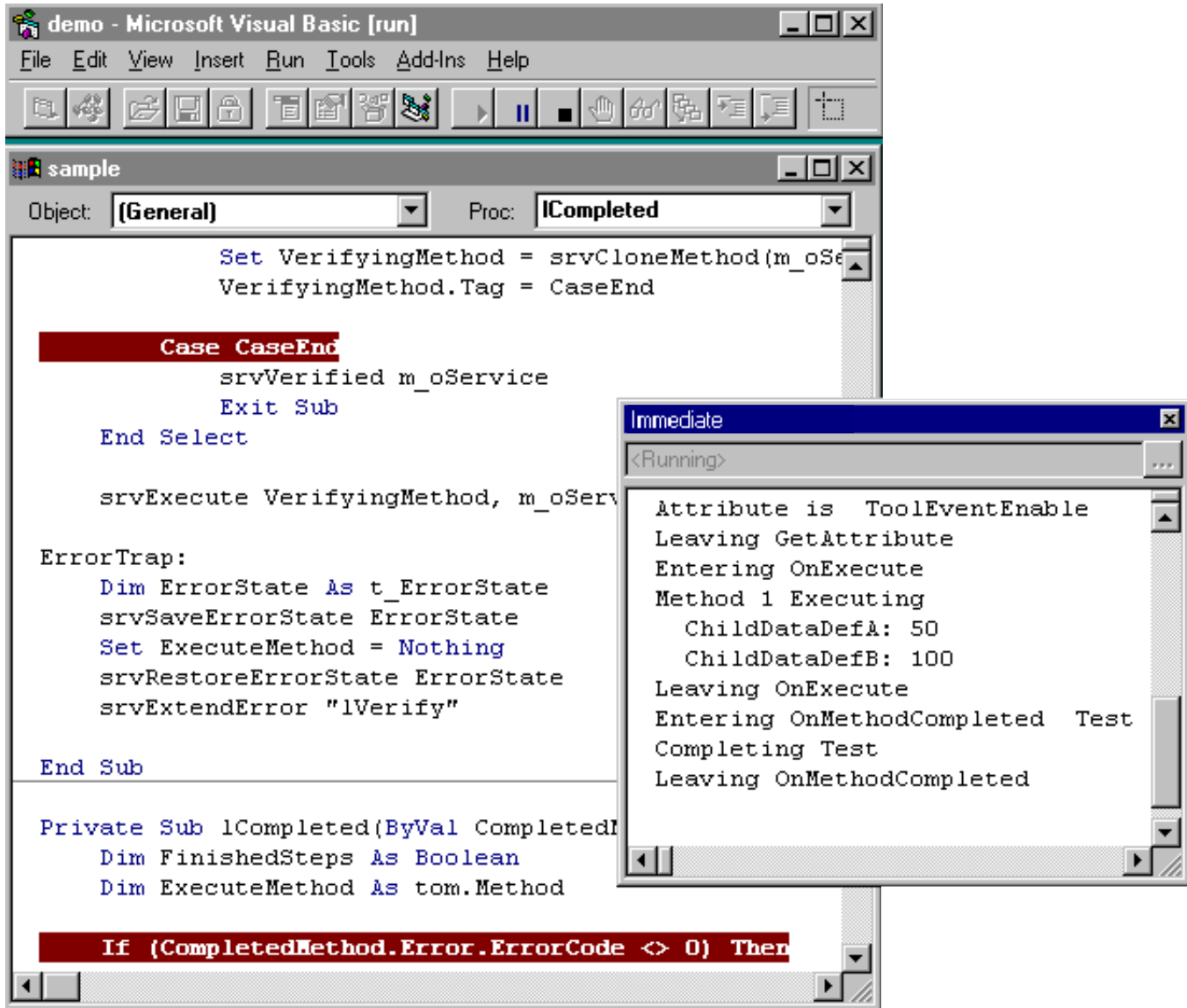


1. Now, try expanding the Tool. Find the Methods under one of the Resources. Right click on the Method and select **Execute** from the pull-down menu.



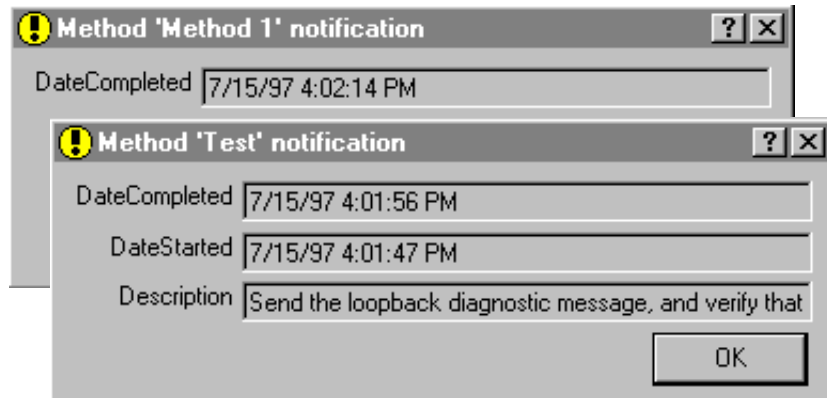
2. If your code stops at a breakpoint, click on the Continue icon in the Visual Basic debugger.

3. You should be able to see the Service proceeding in the code and printing statements about where it is into the Immediate window.

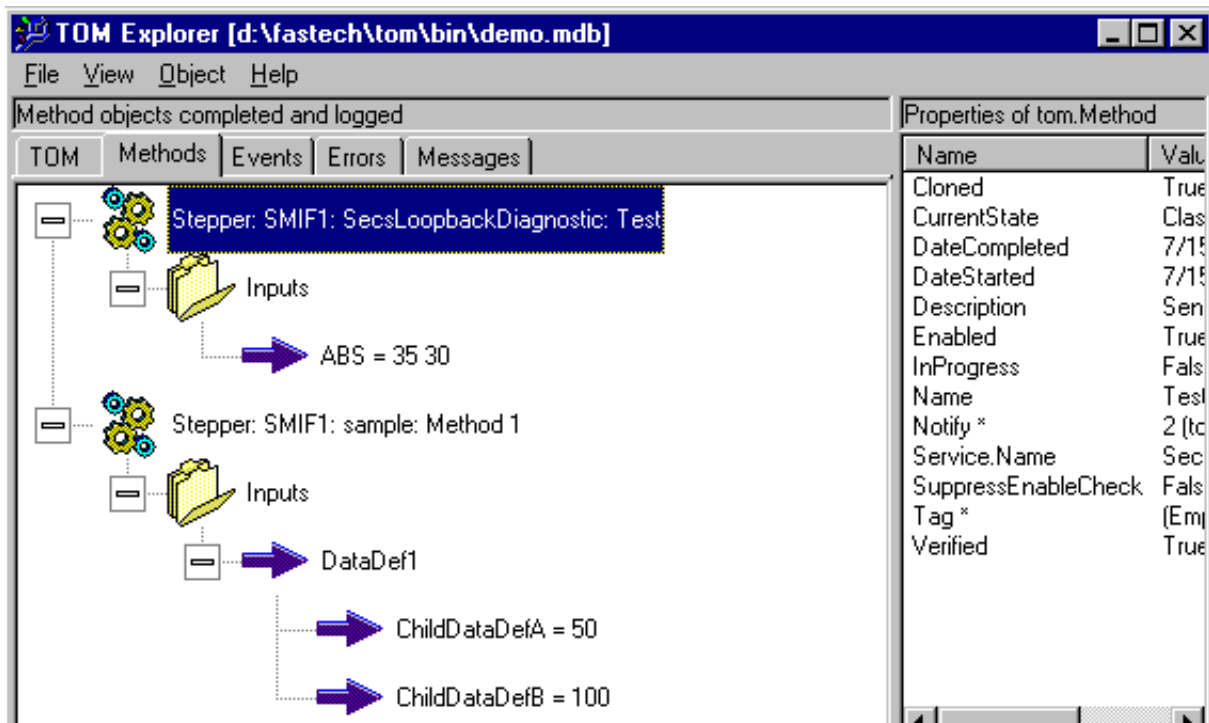


You can see that when Method1 executes the Test method, TOM sends program control into OnMethodCompleted for Test. OnMethodCompleted then carries out end tasks for Test.

- As each Method completes, TOM Explorer pops up a Method Notification to show it has completed, first for Method1, then after you acknowledge that one by clicking OK, another for Test.



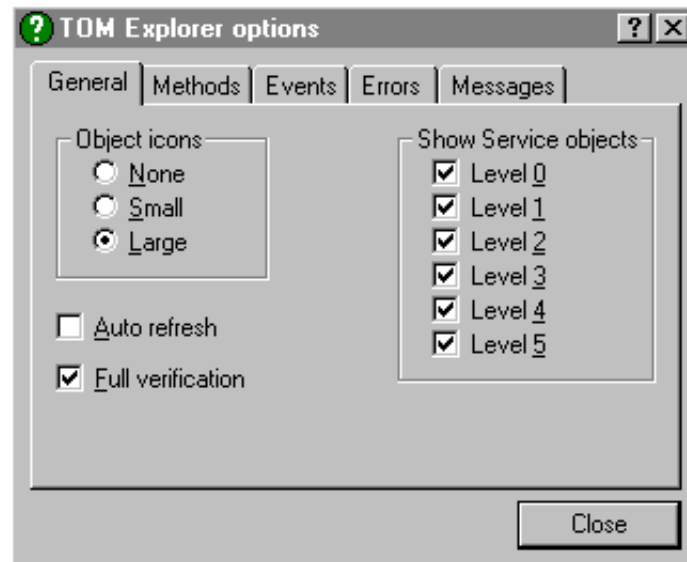
- To see the results of running Method1, you can click on the Methods tab and expand each Method shown there.



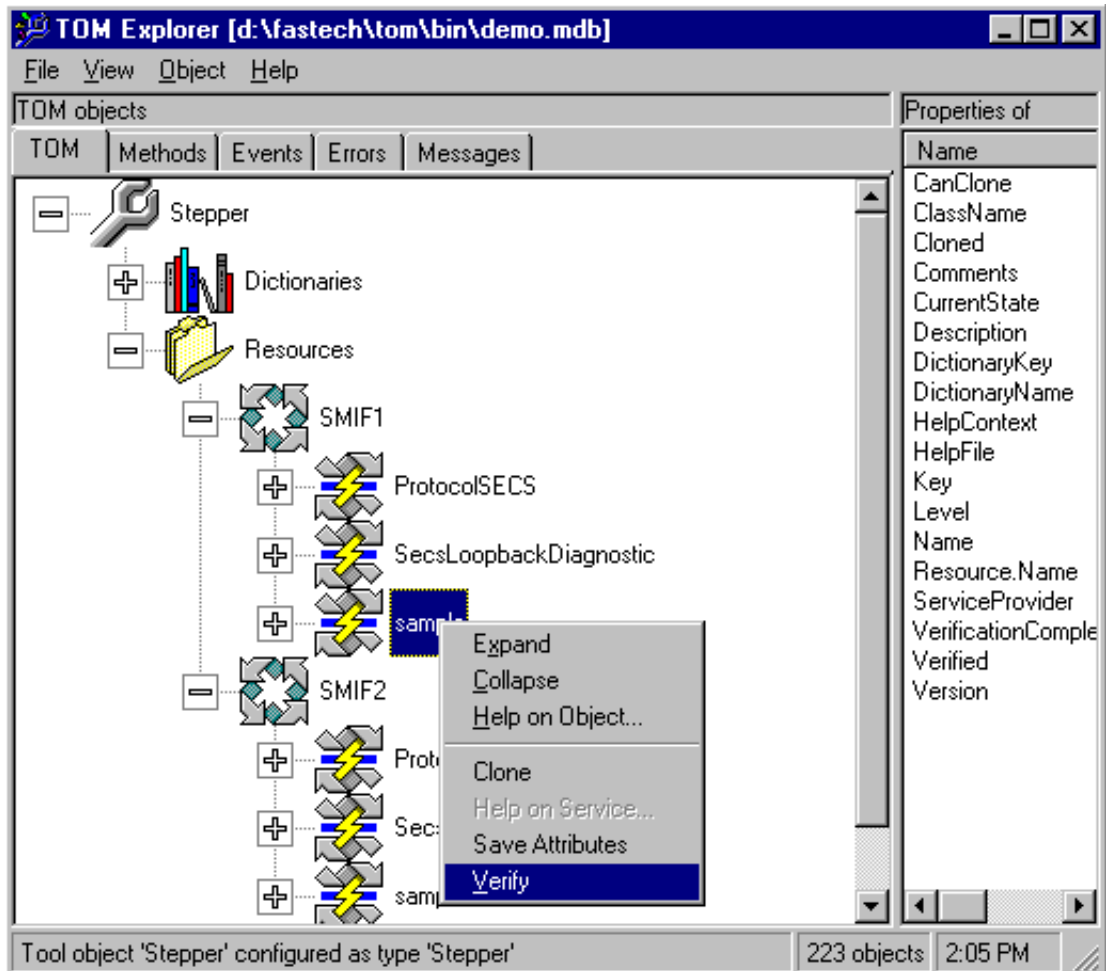
- Execute the other Methods to see the messages that appear in the Immediate window.
- If you have equipment connected, you may want to test Events in your Service by forcing the equipment to trigger one.

Verifying the Service from TOM Explorer

Before you proceed to verify the Service, you should decide whether or not you want to run a full verification. You can toggle this option in TOM Explorer by selecting `View => Options` from the menu bar, going to the `General` tab. Toggle on or off the `Full Verification` check box.

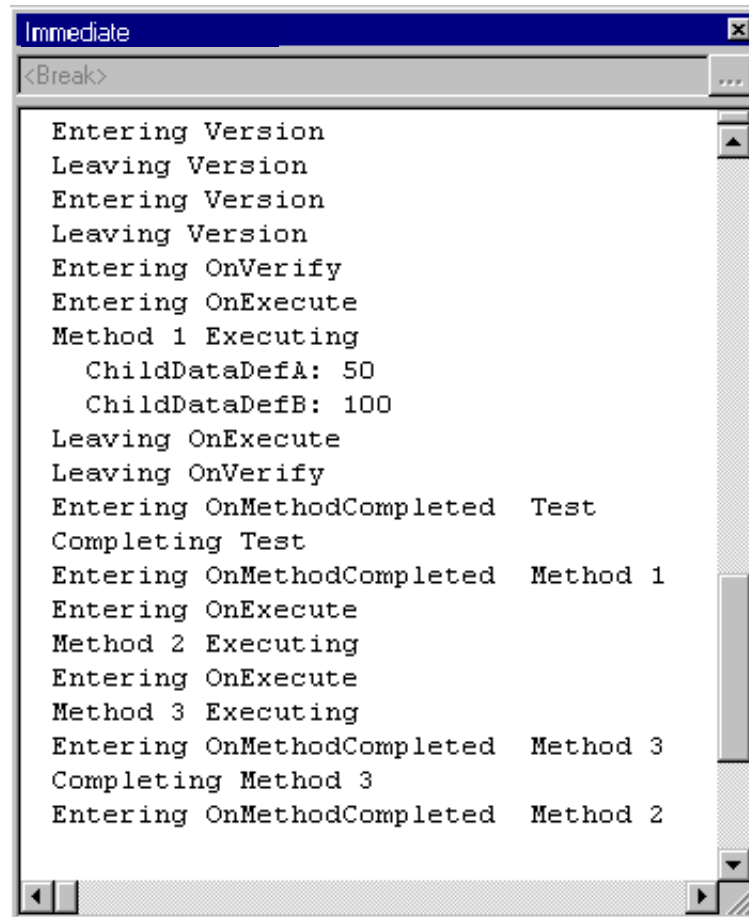


1. Expand the Resource and find your Service under it.



2. Right click on the Service and select `verify` from the pulldown menu.

- When the Visual Basic Debugger stops on your first breakpoint, it will be in `OnVersion`, which it runs first. It then proceeds to `OnVerify`. You can see the trace statements in your Immediate window.



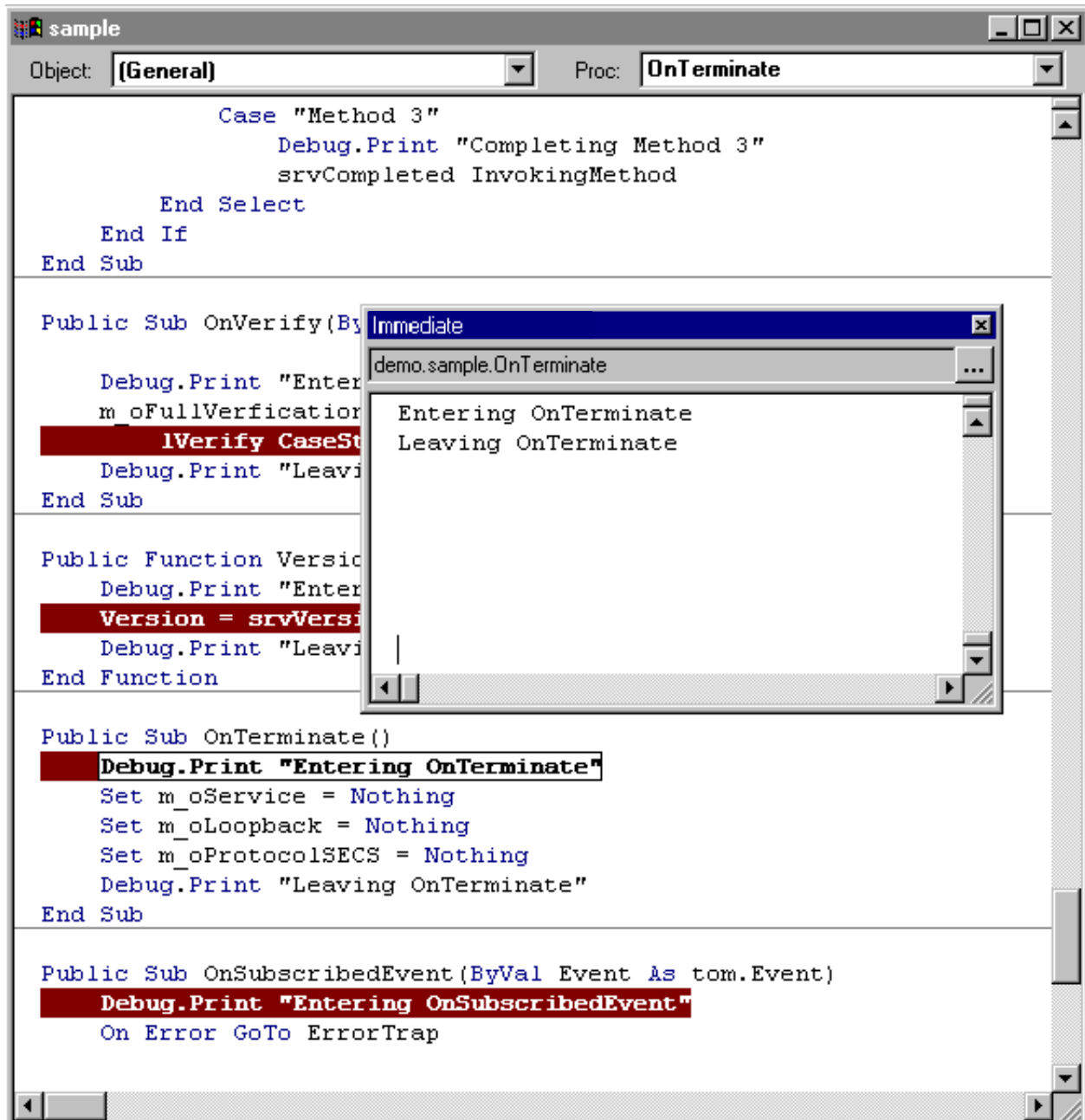
```
Immediate
<Break>
Entering Version
Leaving Version
Entering Version
Leaving Version
Entering OnVerify
Entering OnExecute
Method 1 Executing
  ChildDataDefA: 50
  ChildDataDefB: 100
Leaving OnExecute
Leaving OnVerify
Entering OnMethodCompleted Test
Completing Test
Entering OnMethodCompleted Method 1
Entering OnExecute
Method 2 Executing
Entering OnExecute
Method 3 Executing
Entering OnMethodCompleted Method 3
Completing Method 3
Entering OnMethodCompleted Method 2
```

Notice that the verify process executes each method and when a method has been cloned, then executed, it runs `OnMethodCompleted`.

After it verifies the first Method, notice that TOM leaves `OnVerify` and takes all other verification action in `OnMethodCompleted`. In the sample Service, `OnMethodCompleted` calls `lVerify` to continue the verification process. The messages shown in the Immediate window confirm that the Service operates as intended.

Exiting TOM Explorer

When you are ready to exit TOM Explorer, select `File => Exit` from its menu bar and watch as the Service jumps into `OnTerminate`.



Step through the remainder of your code and when `OnTerminate` completes, TOM Explorer terminates.

Compiling Your Service—Final Compile

You can compile your Service as an in-process OLE server or an out-of-process OLE server.

The differences between the two types of OLE servers are delineated in Microsoft's *Visual Basic Programmer's Guide* and *Visual Basic Professional Features* guide.

Generate DLL

To generate the DLL for the Service, go to your Visual Basic project and select `File => Make demo.dll`.

The DLL file name should use the unique prefix you chose for your organization. It should ideally be the same as the project name in the `Options` window in Visual Basic.

NOTE

When you compile your service DLL, the compiler automatically registers it on the machine you compile it on. However, if you want to use that DLL on another machine, you must be sure you register it by hand using *regsvr32.exe*.

If you did not set `Version Compatibility` to `Binary Compatibility` when you created your Visual Basic project (see *Creating References for Your Project*, p. 2-5), when you try to use your custom Service on another machine, you will not be able to successfully register the DLL.

Testing Your Service

To test your Service, you can:

- Use TOM Explorer—As shown in this chapter.
- Write your own application to run the Service. If you developed it for a particular tool, test it with that Tool.
- Write another Service that calls your Service (like an application), but that you run from TOM Explorer. This way, you can avoid writing an application; TOM Explorer becomes the application that runs your service.

Using Your Service in an Application

To use your service in an application, refer to the *Tool Object Model (TOM) Application Developer's Guide*.

Reusing Existing Services in Yours: Containment

6

Introduction

Topics in This Chapter

Choosing a Related Standard Service, p. 6-2

Writing the Container Service, p. 6-3

Writing Handler Methods for Low Level Services, p. 6-6

Here's the scenario: You've been using standard Services to create a driver, but now you find that your piece of equipment has a capability (or requires a SECS message) that does not have a corresponding TOM Service. You've considered writing that Service from scratch, but it is so similar to one of the standard Services that you want to use that standard Service—maybe simply modify it. You can do that rather easily by containing the standard Service within your custom Service code.

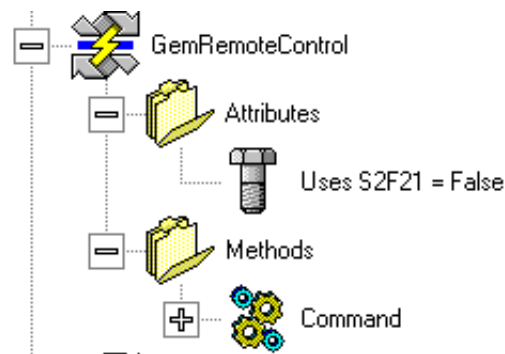
This chapter presents how to contain an existing Service inside a custom Service.

NOTE | The complete code for the sample container Service is included in Appendix B.

It refers to the container Service provided under `\FASTech\TOM\Dev\Samples\Contain\nv10` and the Tool that accompanies it under `\FASTech\TOM\Dev\Samples\Contain\Drivers`.

Choosing a Related Standard Service

Suppose your Tool requires a series of commands under a Remote Commands message. Most of the commands require an S2F21 message, so for them you could use the *GemRemoteControl* Service, which let's you set an Attribute called *Use S2F21* to *True* or *False* (see next illustration). You set this Attribute to *True* and you're all set, right? Not exactly. One command on the Tool, PP-SELECT, requires an S7F1 message, one without a corresponding standard TOM Service.



To determine the solution to this type of problem, ask yourself what you would do if the Tool were entirely compliant with the standard. If you would use a particular Service, then that is the Service you should try to contain.

In this case, what you really need is *GemRemoteControl* plus an additional command, a variation on *GemRemoteControl*.

So now what do you do? If only you could use *GemRemoteControl*...

But you actually can use it! You can contain it inside a custom Service and delegate particular aspects of your Service's capabilities to it, letting *GemRemoteControl* carry out the tasks it knows how to execute. The remainder of your Service need only handle additional tasks your equipment requires.

When choosing a related Service, you should identify:

- What Service would you use if the Tool were entirely SECS compliant?
- What message are you trying to find a Service for?
- Is there an existing Service that contains most of the capabilities you need?

If you find a Service that is *almost* what you need, you can then contain the Service whose capabilities you want to emulate inside your custom Service, as shown in the pages that follow. The resulting Service is essentially a container for the original Service.

Writing the Container Service

Let's see how you could write a Service that is a variation on *GemRemoteControl* for the piece of equipment presented in the previous section.

Create Service in the Database

Initially, you create the container Service the way you would any other:

- Create the Service in the database. Assign it the same name as the Service it is based on. In this case, the Service must be named *GemRemoteControl*.
- Assign a unique `Class` name for the Service, such as `NV10GemRemoteControl`. The name need only be unique within the particular Service DLL you are creating, so you can give it the original Service's name as long as it is in a separate DLL.
- Assign a unique `Provider` name for the Service, which should be the root name of the Visual Basic DLL, in this case `NV10`.
- Assign the Service to the Tool in the database.
- If the Service requires Attributes, create them in the database. For this container Service, you need a single Boolean Attribute called `UseS1F21`.

Create Required Dictionaries

Your container Service may also require access to existing Dictionaries or a unique Dictionary of its own. To handle this situation, you need to:

- Create your own Dictionaries if you need them—Service Dictionaries and Resource Dictionaries.
- Add `DataDefs` to the Dictionaries
- Assign Dictionaries to Services
- Assign Dictionaries to Resources

For details, refer to the *TOM Builder User's Guide* Help file.

Create Handler Methods for Service

You need to create the standard handler methods for a container Service:

1. To use the corresponding handler method from the original Service inside your Service, you would begin by referencing the original Service's DLL in the Visual Basic program by selecting `Tools => Reference` from the menu bar.
2. Declare a private object of a type based on the original Service. In this case, the object type is `GemRemoteControl`.

You can refer to the original Service as a base for the new Service, so let's name the object `m_oBase` and declare it using the full path to the Service in TOM, which is the `Provider`, dot, the `Class`:

```
' Below is the standard service that this one "contains"
Private m_oBase As New tomss2.GemRemoteControl
```

In a moment, you can use the `m_oBase` object to refer to the handler methods in the original Service.

3. You then declare the `SERVICE_NAME` and the command to be passed to the Method, `METHOD_COMMAND`:

```
' Object names
Private Const SERVICE_NAME = "NV10.NV10GemRemoteControl"
Private Const METHOD_COMMAND = "Command"
```

4. Create the usual Service reference:

```
' Objects referenced
Private m_oService As tom.Service ' Service owning Me
```

5. Inside the new Service's `OnCreate`, you need to take all of the actions that the original Service takes in its `OnCreate` and a few more. You use the `m_oBase` object to call the `OnCreate` handler method from the original Service, thus executing that handler method within the container Service:

```
m_oBase.OnCreate Service
```

6. You can then add custom code to the new Service to complete its `OnCreate` handler method.
7. For both `GetAttribute` and `LetAttribute` handler methods, since you are using only the Attribute from the original Service and no additional ones, every time your Service is retrieving or setting an Attribute, it simply needs to call the `GetAttribute` or `LetAttribute` handler method of the original Service. So each of these handler methods, shown below, calls the corresponding handler methods from the original Service using the `m_oBase` object:

```
Public Function GetAttribute(ByVal AttributeName As String)
As Variant
```

```
    GetAttribute = m_oBase.GetAttribute(AttributeName)
End Function
```

```
Public Sub LetAttribute(ByVal AttributeName As String,
NewValue As Variant)
```

```
    m_oBase.LetAttribute AttributeName, NewValue
End Sub
```

The same applies to the `OnMethodCompleted` and `OnVerify` handler methods, shown here:

```
Public Sub OnMethodCompleted(ByVal Method As tom.Method,
ByVal InvokingMethod As tom.Method)
```

```
    m_oBase.OnMethodCompleted Method, InvokingMethod
End Sub
```

```
Public Sub OnVerify(ByVal FullVerification As Boolean)
    m_oBase.OnVerify FullVerification
End Sub
```

None of these Services need do anything more.

8. Inside the sample container Service's `OnExecute`, you need to determine when to take action other than that in the original Service's `OnExecute`. In this example, if the operator is activating a resource, which requires the PP-SELECT command, then you need to take the special action that forced you to create a new Service. Otherwise, you can call the `OnExecute` of the original Service using the `m_oBase` object. The code for this `OnExecute` would be structured as follows:

```
Public Sub OnExecute(ByVal Method As tom.Method)
    ' If activating a resource, execute local function,
    ' otherwise delegate action to contained class.
    If Method.Inputs.Item("Commands").Item(1)._
        Name = "PP-SELECT" Then
        ExecutesS7F1 Method
    Else
        m_oBase.OnExecute Method
    End If
End Sub
```

The `ExecutesS7F1` handler method executes the custom actions in this container Service.

Writing Handler Methods for Low Level Services

If the container Service you are writing is a low level Service (Level 1 or 2), you may need to write or call the original Service's protocol level handler methods that interact with the *ProtocolSECS* Service:

- OnPrimaryIn
- OnPrimaryOutError
- OnSecondaryIn
- OnSecondaryInError

The sample container Service has an OnSecondaryIn handler method. For more information on this handler method, refer to *SECS Handler Methods Grouped by Function*, p. 5-2, in the *TOM Service Developer's Reference*.

Introduction

Topics in This Chapter

This chapter presents how to work with specific TOM objects inside your handler methods. For instance, it discusses:

Deciding to Raise, Extend, or Trigger an Error, p. 7-2

Extending an Error, p. 7-3

Raising an Error, p. 7-6

Triggering an Error, p. 7-8

Deciding to Raise, Extend, or Trigger an Error

If any other type of error occurs at any time in your Service, you send program control to an error handler. Your error handler can take care of the error in one of these ways:

- Extend the error
- Raise the error
- Trigger the error

Let's get a quick definition of each of these actions. It is important to realize that in most situations TOM is above your Service in the call stack, because usually TOM calls your Service handler methods (such as `OnCreate`):

- If you extend an error, you add information to it (as text) and pass that information along with the error up the call stack. You use this technique if a Service your Services uses finds the error. For more information see *Extending an Error*, p. 7-3.
- When you raise an error, you raise a Visual Basic runtime error to the next highest level, which in this case would be the TOM application using your Service. You raise an error if your Service is the first Service to find it (rather than a lower level Service passing it to yours). For more information, see *Raising an Error*, p. 7-6.

NOTE

Tip — Raising vs. Extending an Error

When should you raise an error instead of extending it? If you detect the problem within your own Service, you should raise the error. If you do not raise the error, Visual Basic assumes you have handled it.

If a lower level Service detects the problem, your Service should *not* raise it, but extend it—effectively passing the error up to the next level. If you do not percolate the error up, Visual Basic assumes you have handled it. If you do not handle it and Visual Basic ends up percolating the error up, it could generate issues later.

- If you trigger an error, you create an Error object and trigger it instead of letting TOM take its usual default action. For more information on when and why you would take such action, refer to *Triggering an Error*, p. 7-8.

Extending an Error

In most cases, you deal with an error by aborting your handler method and sending control to an error handler. The error handler then propagates the error up the call stack. You would do this with an error raised by another Service used in your Service. Since TOM is above your Service in the call stack, the error is extended to TOM. TOM traps the error and takes appropriate action.

Call srvExtendError

To extend an error, you call the `srvExtendError` handler support routine from inside your error handler:

```
On Error Goto ErrorTrap
    ...
ErrorTrap:
    srvExtendError "lComplete"
```

You use this routine to augment the Visual Basic `Err` object with custom information you want passed up the stack. The syntax for the routine is:

Public Sub `srvExtendError` (*RoutineName* **As String**, *Optional Number*, *Optional HelpFile*, *Optional HelpContext*, *Optional Description*, *Optional Params*)

- *RoutineName*—Name of routine or handler method extending the error.
- *Optional Number*—Numeric code of error.
- *Optional HelpFile*—Name of Help file containing help for the error.
- *Optional HelpContext*—Help context ID of help for the error.
- *Optional Description*—String containing description of the error or number containing the ID of the description in the resource file.
- *Optional Params*—The parameters to insert into the description. For more than one, pass them as an array.

Pass Your Handler Method Name as Argument

The first argument is required and should contain the name of your routine or handler method. The name gets added to the error description as the error is propagated up the stack, so you can trace the source of the error.

NOTE

Tip—Always Call `srvExtendError` in Your Error Handler

You can trace the source of any error in your Service as long as you consistently include an error handler in your handler method and call `srvExtendError` from that error handler.

Use Description Argument to Identify Error

All the other arguments are optional. The most commonly used optional parameter is `Description`. This parameter can be a string describing the error or a number used to locate the error string in the resource (*.rc*) file included in your Visual Basic project.

The description string can contain `%n` parameters (where *n* is a digit from 1 through 9). You pass the parameters to substitute for each `%n` in the `Params` argument, as shown in the example below:

```
ErrorTrap:
    srvExtendError "lComplete"

    Description:="The %1 Service found an error using %2 _
    resource while the %3 Method was invoking %4 Method"

    Params:=Array(m_oService.Name, m_oService.Resource.Name, _
    InvokingMethod.Name, Method.Name)
```

The parameters you pass with `Params` should always include your Service's name, the Resource name, the Method name, and any relevant `DataItems`.

Ensure Err Object Contains Correct Information

Because you are extending the error, it is important that the Visual Basic `Err` object contain the correct information when you extend it. You often find you need to take other action that could inadvertently clear the `Err` object and destroy the error information that you want to report. Visual Basic automatically clears `Err` when the object you have created is destroyed at the end of the handler method. The steps you should take inside the error handler to ensure the `Err` object remains available are:

1. Save the error state, stored in `Err`, so that later you can restore the error state. For instance, you could create a variable named `ErrorState` of the `t_ErrorState` type. You would pass this variable to the `srvSaveErrorState` handler support routine. The syntax for that routine is:

```
Public Sub srvSaveErrorState (ErrorState As t_ErrorState)
```

The single argument it takes is *ErrorState*, a `t_ErrorState` structure.

The error trap would start as follows:

```
ErrorTrap:
    Dim ErrorState As t_ErrorState
    srvSaveErrorState ErrorState
```

2. Destroy the object associated with the error state by setting it to `Nothing`:

```
Set TheMethod = Nothing
```

In your Service, the object you created with the `srvCloneMethod` or `srvCloneEvent` handler support routine is the object you must remove by setting it to `Nothing`. If you do not remove these objects, they hang

around in memory and could eventually cause problems. If you did not create any objects, then you can skip this step.

3. Restore the Err object in the `ErrorState` variable using `srvRestoreErrorState`:

```
srvRestoreErrorState ErrorState
```

4. Extend the error using `srvExtendError` and passing it the name of the handler method that the error came up in, such as `OnExecute`:

```
srvExtendError "OnExecute"
```

The complete code for the error handler should include the code that follows, only in yours you should replace `MyMethod` with the object you created using either `srvCloneMethod` or `srvCloneEvent`:

NOTE Your code should include the lines written below, only it should replace *TheMethod* with the appropriate object and *Routine* with the name of the handler method.

```
ErrorTrap:  
    Dim ErrorState As t_ErrorState  
    srvSaveErrorState ErrorState  
    'insert custom error handling code here  
    Set TheMethod = Nothing  
    srvRestoreErrorState ErrorState  
    srvExtendError "Routine"
```

Another issue to be aware of is code falling through to the `ErrorTrap` section. Naturally, you do not want this to occur. So, just before this section, you might want to have an `Exit Sub` statement to ensure that the code never gets here inadvertently.

Raising an Error

There are two ways of raising an error—you raise it yourself explicitly, or another entity raises an error in your code unexpectedly. The unexpected error is usually raised by Visual Basic or a Service that is using your Service.

You should raise an error when your Service detects an error. The major steps to raising an error are:

1. Disable the error handler
2. Call `srvRaiseError`
3. Exit the subroutine

Before you raise an error, you should always disable the error handler that `On Error Goto Label` has enabled at an earlier time in the program, so that you don't end up creating a duplicate stack trace entry for this routine in the error description. You disable the error handler with:

```
On Error Goto 0
```

Then to raise the error, you call the `srvRaiseError` handler support routine. The syntax for that routine is:

Public Sub `srvRaiseError` (*RoutineName As String, ByVal Number _ As Long, ByVal HelpContext As Long, Source As String, Description As _ Variant, Optional HelpFile, Optional Params*)

- *RoutineName*—Name of routine raising the error.
- *Number*—Numeric code of error.
- *HelpContext*—Help context ID of help for the error.
- *Source*—Name of the OLE server object raising the error.
- *Description*—String containing description of the error or number containing the ID of the description in the resource file.
- *HelpFile*—Name of Help file containing help for the error.
- *Params*—The parameters to insert into the description. For more than one, pass them as an array.

When you raise the error, you set the `RoutineName` to the name of the subroutine or function that detected the problem, such as `OnExecute`, and fill in the other arguments as appropriate. Remember the `SERVICE_NAME` constant you set when you first began creating the Service? Here, you set the `Source` argument to it:

```
' Error if the Event did not occur
srvRaiseError RoutineName:="OnExecute", _
Number:=999, _
HelpContext:=34000 + TOTAL, _
```

```
Source:=SERVICE_NAME, _  
Description:="The %1 Service found an error using %2 _  
resource while %3 method was invoking %4 method"  
HelpFile:=myHelpFile.hlp, _  
Params:=Array(m_oService.Name, m_oService.Resource.Name, _  
InvokingMethod.Name, Method.Name)  
Exit Sub
```

The parameters you pass with `Params` should always include your Service's name, the Resource name, the Method name, and any relevant DataItems.

After raising an error, you might want to exit the subroutine.

Triggering an Error

The last way to deal with an error is to create an Error object and trigger notification in a TOM application. Two situations where you might trigger an Error object are:

- An error comes in to your Service from the outside. In this situation, your Service is at the top of the call stack. In this case you must trap the error and, in most cases, you should trigger an Error object.
- When your Service encounters an error while executing a Method and you want to raise the error but do not want to terminate the Method.

Receiving an Error from the Outside

If an error comes from outside your Service, you should trigger the error for the application using your Service. For instance, suppose a piece of equipment sends up an error through an ActiveX control. Because this type of error occurs asynchronously, you cannot handle it in `OnExecute`, `OnSubscribedEvent`, or any other handler method TOM calls. TOM isn't active and isn't calling any handler methods, because only your Service knows about the error. In this situation, you set up a private routine in your Service that an ActiveX can call.

NOTE **Tip—From Top of Call Stack, You Must Handle Errors**

When an error occurs, and your Service is at the top of the call stack, you must trap these errors and not allow them to propagate upward. If an event handler raises an error and you do not trap it, the entire TOM application may terminate!

At the top of the private routine, you should have `On Error Resume Next` followed by a call to a special error handler, such as `!ErrorEvent`. This way you guarantee that the error is not propagated up the call stack because you do not go to an error trap label. Below is an example of how you call an event handler named `!ErrorEvent` for the `EquipCtrl` control:

```
Private Sub EquipCtrl_Error()  
    On Error Resume Next  
    !ErrorEvent  
Exit Sub
```

Inside the error handler, you generate an Error object and trigger the Error for the application using your Service (such as TOM Explorer). You use `srvTriggerError` to take both of those actions.

The syntax of this routine is as follows:

```
Public Sub srvTriggerError (Service As tom.Service, _
    ByVal ErrorCode As Long, _
    ByVal HelpContext As Long, _
    ByVal Source As String, _
    ByVal ErrorText As Variant, _
    Optional HelpFile, _
    Optional ErrorObject, _
    Optional Params)
```

You can pass parameters to the routine, just as you would with `srvRaiseError` or `srvExtendError` (see *Raising an Error*, p. 7-6, or *Extending an Error*, p. 7-3).

In the error handler, after you take any action you'd like to take in your `Service`, you call `srvTriggerError`:

```
Private Sub lErrorEvent()
    On Error Goto ErrorTrap
    'Deal with the error event here
Exit Sub
    srvTriggerError m_oService, _
        ErrorCode:=20039, _
        HelpContext:=70039, _
        Source:=SERVICE_NAME, _
        ErrorText:=10039, _
        HelpFile:="MyHelpFile.hlp"
End Sub
```

The TOM application's `EvrrorNotification` routine receives the error notification and receives the `Error` object as an argument.

For more information on the `ErrorNotification` routine, refer to the TOM Help file.

Trigger Error Your Service Encounters, but Resume Method Action

If your `Service` encounters an error, you would usually raise that error. Sometimes, however, raising an error can terminate a `Method` in progress.

To avoid terminating the `Method`, you can trigger the `Error` for the application using your `Service` and continue with the `Method` action.

In this case, you can also use `srvTriggerError`.

Introduction

Topics in This Chapter

Planning the Approach, p. 8-2
Create Constants and References in Declarations, p. 8-3
Creating Method Object in OnCreate, p. 8-3
Checking Required Services in OnInitialize, p. 8-3
Subscribing to Events in OnInitialize, p. 8-4
Setting Up TOM Notifications, p. 8-4
Starting the StartTool Method in OnExecute, p. 8-4
Continuing to Chain Methods in OnMethodCompleted, p. 8-5
Executing Last Method in OnSubscribedEvent, p. 8-6
Creating the Service DLL, p. 8-7
Creating Service, Tool, Dictionaries in Database, p. 8-7
Running Service in Visual Basic Debugger, p. 8-8

You want to start a Tool by taking the following actions:

- Start logging
- Open the port
- Establish communication with the Tool

This chapter presents how to write a Service that takes these actions using a single custom Method.

NOTE | The complete code for the sample container Service is included in Appendix F.

The Service is provided under `\FASTech\Sw\Dev\Samples\StartTool\init.vbp` and its Tool is under `\FASTech\Sw\Dev\Samples\StartTool\Drivers\GenTool`.

Planning the Approach

To create a new custom Service that would start a Tool, you need to identify the Services whose Methods you'll need to clone and execute to take those actions. In this case, the Services and corresponding Methods are commonly used ones:

- Start logging—*LOLogging* Service's `Stop` and `Start` Methods.
- Open the port—*ProtocolSECS* Service's `Close` and `Open` Methods.
- Establish communication with the Tool—*GemEstablishCommunications* Service's `Connect` Method.

Why do you need both `Stop` and `Start` from *LOLogging*? Because before you can start logging, you have to be sure logging is not already in progress, because if it is, you receive an error. So, before you start logging, you should stop logging. This technique circumvents the possibility of that error.

The same is true for the `Close` and `Open` Methods of *ProtocolSECS*. To be sure the port is not already open, you close it first, *then* open it.

Plan to Chain Methods

To clone and execute several Methods in sequence, you need to know which Method is to be executed next. The sequence you want to follow is:

- `Stop`
- `Start`
- `Close`
- `Open`
- `Connect`

When you clone and execute the first Method, you can set its `Tag` property to the clone of the next Method to execute. You write the code for this action later in the `OnExecute` and `OnMethodCompleted` handler methods.

Plan to Respond to Events

When you execute the `Open` Method, it generates a `Connect` Event from the Tool. Because you need to respond to this event, you don't need a section in `OnMethodCompleted` for when `Open` completes; instead, you need a section in `OnSubscribedEvent` for the `Connect` Event, which occurs after you execute `Open`, but not necessarily immediately after. The Event is asynchronous, so you must wait for the event before proceeding.

Similarly, when you execute the `Connect` Method of *GemEstablishCommunications*, you also must subsequently wait for an Event. Two possible Events can occur:

- `Established communications`
- `Changed`

The Changed Event occurs when the setting of the *Communicating* attribute of *GemEstablishCommunications* changes.

Create Constants and References in Declarations

So, in the Declarations section of your Visual Basic code, you would establish the constants for these Services as well as any of the Services they depend on, such as *GemIdentification* and *ProtocolTimer*:

```
Private Const SRV_PROTOCOLSECS = "ProtocolSECS"
Private Const SRV_GEMESTABCOMMS = "GemEstablishCommunications"
Private Const SRV_GEMIDENTIFICATION = "GemIdentification"
Private Const SRV_LOLOGGING = "LOLogging"
Private Const SRV_PROTTIMER = "ProtocolTimer"
```

In addition, you'll need a reference to your Service:

```
' References
Private m_oService As tom.Service 'Service that owns this class
```

Later, you need a global reference to the custom Method you are creating, so that you can refer to it inside the *OnSubscribedEvent* handler method. You create this reference in the Declarations also:

```
' Global reference to a custom Method
Private m_StartTool As tom.Method
```

Creating Method Object in OnCreate

In *OnCreate*, you need to create the Method object for your custom *StartTool* Method.

```
' Here is the StartTool Method Object
Set StartTool = srvDefineMethod(m_oService, METH_START,
"StartTool Method")
```

Checking Required Services in OnInitialize

In *OnInitialize*, you check to be sure each required service is present:

```
srvRequiredService m_oService, SRV_PROTOCOLSECS
srvRequiredService m_oService, SRV_GEMESTABCOMMS
srvRequiredService m_oService, SRV_GEMIDENTIFICATION
srvRequiredService m_oService, SRV_LOLOGGING
srvRequiredService m_oService, SRV_PROTTIMER
```

Subscribing to Events in OnInitialize

Also in `OnInitialize`, you subscribe to the events you plan to react to later in `OnSubscribedEvent`:

```
' Event occurs when you execute the Open Method of ProtocolSECS:
srvSubscribeEvent m_oService, SRV_PROTOCOLSECS, "Connect"

' One of two Events occur when you execute Connect Method of
' GemEstablishCommunications:
srvSubscribeEvent m_oService, SRV_GEMESTABCOMMS, "Established
communications"
srvSubscribeEvent m_oService, SRV_GEMESTABCOMMS, "Changed"
```

Setting Up TOM Notifications

To complete the `OnInitialize` handler method, you set whether or not TOM or other Services require notification by using `srvSetEventNotification` and passing either `tomNotifyAlways` or `tomNotifyNever` as an argument:

```
srvSetEventNotification m_oService, SRV_PROTOCOLSECS, "Connect",
tomNotifyAlways

srvSetEventNotification m_oService, SRV_GEMESTABCOMMS,
"Established communications", tomNotifyAlways

srvSetEventNotification m_oService, SRV_GEMESTABCOMMS,
"Changed", tomNotifyAlways
```

Starting the StartTool Method in OnExecute

In `OnExecute`, you start by constructing a Case statement that has code for each Method the operator can choose from the TOM Explorer or IDE Browser. In one case, you would create code for the `StartTool` Method, as explained in the remainder of this section.

Here, you begin the Method chaining process. Since you are going to begin by cloning and executing the `Stop` Method, you then clone and execute the `Start` Method and put it in the `Stop` Method object's `Tag` property:

```
Set MethodToExec = srvCloneMethod(m_oService, "Stop",
SRV_LOLOGGING)
Set MethodToExec.Tag = srvCloneMethod(m_oService, "Start",
SRV_LOLOGGING)
```

After these steps, you need to set the `InvokingMethod`, which is `StartTool`. TOM passes this Method to `OnMethodCompleted`, so that it knows `StartTool` was the Method from the TOM Explorer or IDE Browser that invoked `Stop`:

```
Set InvokingMethod = ExecuteMethod
```

Although TOM passes the `InvokingMethod` to `OnMethodCompleted`, since it doesn't pass that same information to `OnSubscribedEvent`, you need to set the global `Method` object you created for `StartTool` to the invoking method. Later, `OnSubscribedEvent` can use this `Method` object to determine what the invoking `Method` is:

```
Set m_StartTool = InvokingMethod 'for use by OnSubscribedEvent
```

Finally, the last step in `OnExecute` is to execute the `Stop Method`:

```
srvExecute MethodToExec, m_oService, InvokingMethod
```

This `Method` is the only one you start in `OnExecute`. When the `Stop Method` completes, TOM jumps in to `OnMethodCompleted`, where it can then execute the next `Method`.

Continuing to Chain Methods in `OnMethodCompleted`

In `OnMethodCompleted`, you can continue the `Method` chaining you began in `OnExecute`. Again, you generate a `Case` statement for each `Method` that can be completing:

```
Case "Stop"
'This is completion of the LLogging Stop Method to ensure no
'error on initiating logging.

  Set MethodToExec = CompletedMethod.Tag
  'Gets Start method from tag of Stop method

  Set MethodToExec.Tag = srvCloneMethod(m_oService, "Close",
  SRV_PROTOCOLSECS) 'Sets tag to the next method, Close
  srvExecute MethodToExec, m_oService, InvokingMethod

Case "Start"
'This is completion of the LLogging Start Method.

  Set MethodToExec = CompletedMethod.Tag
  'Gets Close method from tag of Start method

  Set MethodToExec.Tag = srvCloneMethod(m_oService, "Open"),
  SRV_PROTOCOLSECS) 'Sets tag to the next method, Open
  srvExecute MethodToExec, m_oService, InvokingMethod

Case "Close"
'This is completion of the ProtocolSECS Close method to ensure
no 'error when executing the Open method.

  Set MethodToExec = CompletedMethod.Tag
  'Gets Open method from tag of Close method

  srvExecute MethodToExec, m_oService, InvokingMethod
```

For the case of `Close`, you get the `Open Method` from the tag and then execute it. You do not set the tag to the next method to execute, because the `Service` needs to execute that method in `OnSubscribedEvent` rather than in `OnMethodCompleted`.

For the cases of `Open` and `Connect`, you can take all action when the associated event occurs, so you need not handle the completion of these Methods in `OnMethodCompleted`, but you can have markers for them that indicate what is going on:

```
Case "Open"
'After the Open Method, Service waits for Connect Event.

Case "Connect"
'After the Connect Method, Service waits for the Established
communications or Changed Event.
```

Executing Last Method in `OnSubscribedEvent`

When the `Connect` Event occurs in response to opening the port, you can take action in response to that Event that executes the final Method, the `Connect` Method of *GemEstablishCommunications*:

```
Case "Connect"

'Received notification of Connect Event from ProtocolSECS
'Completing StartTool Method's Opening of Port

    srv.GetService(m_oService,
    SRV_GEMESTABCOMMS).Attributes.Item("Interval").Value = 5
    Set MethodToExec = srv.CloneMethod(m_oService, "Connect",
    GEMESTABCOMMS)
    srv.Execute MethodToExec, m_oService, m_StartTool
```

You need to set the `Interval` Attribute of the *GemEstablishCommunications* Service to a number other than 0 so that the Service refreshes the values of the other Attributes, especially the `Communicating` Attribute. Otherwise, if that Attribute changes, it may not evoke the `Changed` Event.

Once you have executed the `Connect` Method, since you have subscribed to the possible Events it can evoke, you should establish a single case in response to either one of them occurring:

```
Case "Established communications", "Changed"

'Received notification that Communication with Tool
'established. StartTool Method has communicated with Tool
```

Since `Connect` is the last Method to execute, after one of the events it evokes occurs, the `StartTool` Method is complete and you need to execute `srvComplete` on it using the global Method object you established for it, `m_StartTool`:

```
If Not m_StartTool Is Nothing Then
    srvCompleted m_StartTool
    Set m_StartTool = Nothing
```

```
End If
```

It is a bit different to be executing `srvComplete` in `OnSubscribedEvent` rather than in `OnMethodCompleted`. Remember that to execute `srvComplete` here, you still have to pass it the invoking Method, but to have that Method available, you must have created a global Method object for it, just as this example does for `StartTool`.

Creating the Service DLL

Create the Service's `.dll` so that its `.tbf` file appears in the list of Services in TOM Builder and you can easily assign it to the Tool.

Before you can use the Service, you must create the Service, its Tool, and the associated Dictionaries in the database, in the next section.

Creating Service, Tool, Dictionaries in Database

For this new Service, you need a new Tool. The Tool is provided under the *Drivers* directory for this Service. It is called *GenTool.tbf* and its Resource is called *GenRes.tbf*. This Tool already has a `StartTool` Method, so you do not have to create a Tool from scratch unless you want to practice creating a Tool.

You should, however, build the database that contains the *GenTool* and use that database when you run this sample Service.

You can create a Tool in TOM Builder, just as you created the Stepper Tool earlier in this manual:

- Create a new Tool.
- Create a new Resource and assign it to the Tool.
- Create the Service.
- Create your own Dictionaries if you need them—Service Dictionaries and Resource Dictionaries.
- Add `DataDefs` to the Dictionaries.
- Assign Dictionaries to Services.
- Assign Dictionaries to Resources.

For details, refer to the chapter on *Creating a Tool for Your Service* or the *TOM Builder User's Guide* Help file.

After you create the Tool, you can run an instance of it in TOM Explorer.

Running Service in Visual Basic Debugger

You can run the Service in the Visual Basic debugger just as you did for the *demo.sample* Service earlier in this manual:

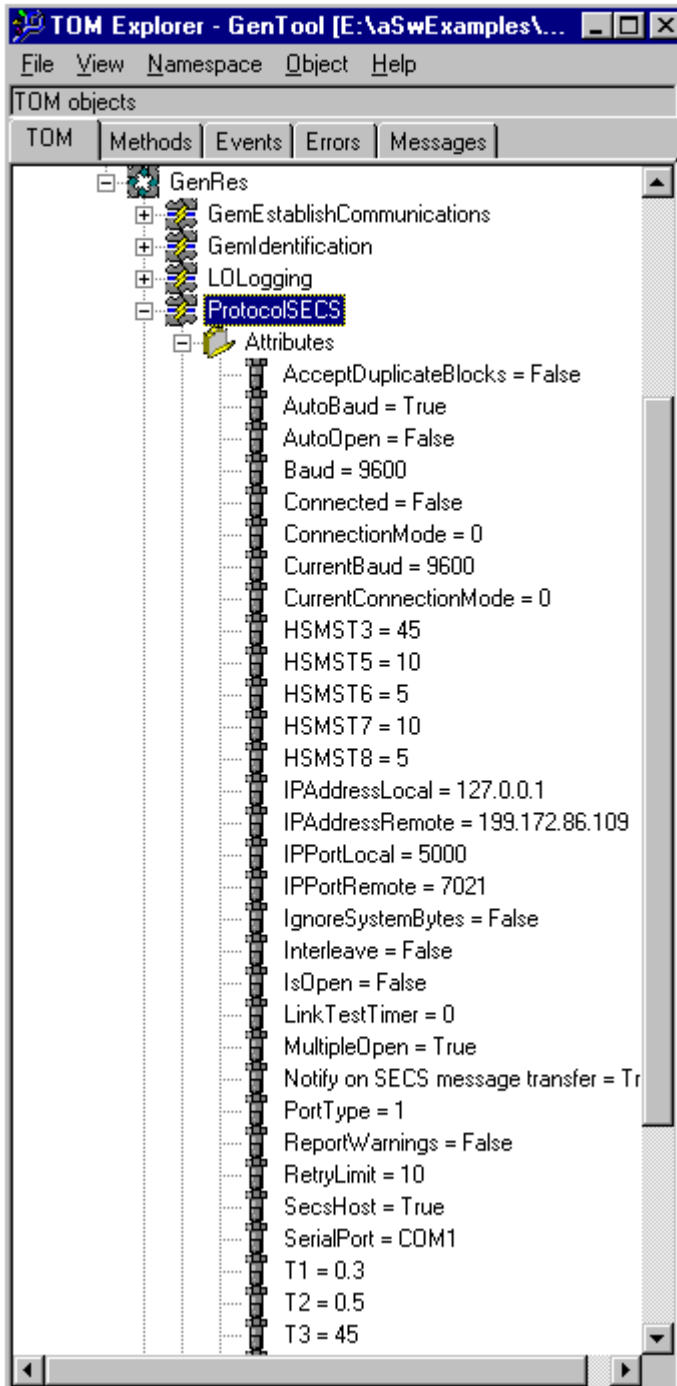
1. Create a shortcut to TOM Explorer and set it operate on the database containing the generic tool for this Service.
2. Go to the Visual Basic menu bar and select `Run => Start with Full Compile` to compile the Service.
3. Notice that nothing seems to be happening. To see the `Immediate` window appear, start TOM Explorer from the shortcut you created.
4. In the TOM Explorer menu bar, select `File => Create Tool object . . .`. To run the *init.sample2* Service, select `GenTool` from the list of Tools that appears.
5. To see communication with the Tool actually be established, you need to have a simulator or actual Tool running and set the following Attributes of the *ProtocolSECS* Service to match those of the appropriate settings in the simulator/Tool:

(The settings will vary depending on your simulator. The settings given here are for an HSMS port type, which requires HSMS related attribute settings.)

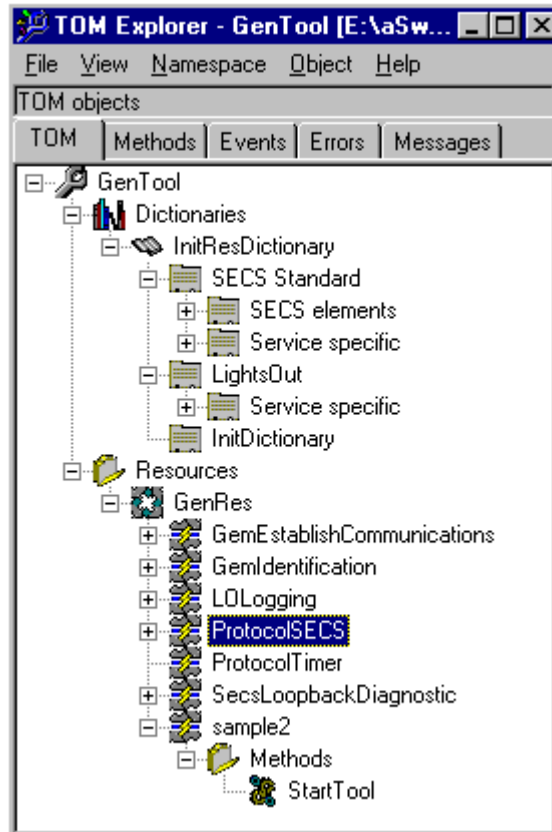
- ◆ `AutoOpen = False`
- ◆ `ConnectionMode = 0`
- ◆ `IPAddressLocal`
- ◆ `IPAddressRemote`
- ◆ `IPPortLocal`
- ◆ `IPPortRemote`

For an HSMS port type:

- ◆ `HSMST3`
- ◆ `HSMST5`
- ◆ `HSMST6`
- ◆ `HSMST7`
- ◆ `HSMST8`
- ◆ `PortType = 1`



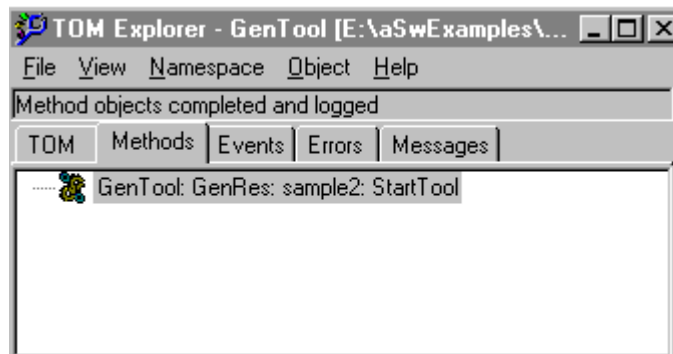
- Expand the *sample2* Service under the *GenRes* Resource and you see the *StartTool* Method under *Methods*.



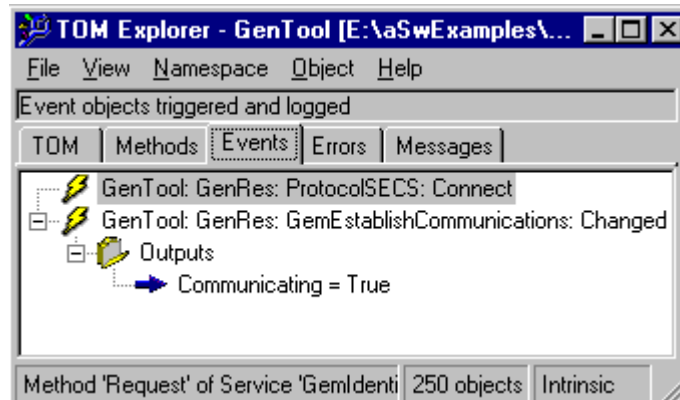
- Right click on and execute the *StartTool* Method.

Look in the lower left corner of the TOM Explorer at the status bar and you see messages indicating it is cloning and executing methods of various Services.

- Click the *Methods* tab of TOM Explorer to see the *StartTool* Method has been executed.



- Click the `Events` tab to see the TOM Events that have occurred.



- You can also see some indication of what is happening by looking in the Immediate window, where the debug messages print.

```

Immediate
Entering OnCreate
Leaving OnCreate
Entering OnInitialize
Leaving OnInitialize
Entering OnExecute
StartTool Method Executing
Entering OnMethodCompleted Stop
StartTool Executing Start Method of LOLogging Service
Entering OnMethodCompleted Start
StartTool Executing Close Method of ProtocolSECS Service
Entering OnMethodCompleted Close
StartTool Executing Open Method of ProtocolSECS Service
Entering OnMethodCompleted Open
Leaving OnMethodCompleted
Leaving OnMethodCompleted
Leaving OnMethodCompleted
Leaving OnMethodCompleted
Leaving OnExecute
Entering OnSubscribedEvent
Received notification of Connect Event from ProtocolSECS
Completing StartTool Method's Opening of Port
Leaving OnSubscribedEvent
Entering OnMethodCompleted Connect
Leaving OnMethodCompleted
Entering OnSubscribedEvent
Received notification that Communication with Tool has been established
StartTool Method has communicated with Tool
Leaving OnSubscribedEvent

```


Template/Sample Service Code

A

Introduction

This appendix lists the code for the template Service's class, from the *sample.cls* file.

Complete Code for the Service

```
Option Explicit
' Object Names
Private SERVICE_NAME As String
Private Const SRV_LOOPBACK = "SecsLoopbackDiagnostic"
Private Const SRV_PROTOCOLSECS = "ProtocolSECS"
' Boolean that indicates whether or not Full Verification is on
Private m_oFullVerification As Boolean
' DataDef names
Private Const DD_DD1 = "DataDef1"
Private Const DD_CHILDA = "ChildDataDefA"
Private Const DD_CHILDB = "ChildDataDefB"
Private Const DD_DD2 = "DataDef2"
Private Const ATT_EVENT_ENABLED = "ToolEventEnable"
Private Const METH_METHOD1 = "Method 1"
Private Const METH_METHOD2 = "Method 2"
Private Const METH_METHOD3 = "Method 3"
Private Const EVENT_CONNECT = "Connect Event"
' Attributes of Service
Private Att_ToolEventEnable As String
' Constants for sequence of verification
Private Const CaseStep1 = 1
Private Const CaseStep2 = 2
Private Const CaseStep3 = 3
Private Const CaseEnd = 4
' References
Private m_oService As tom.Service 'Service that owns this class
Private m_oLoopback As tom.Service 'Another Service this one accesses
Private m_oProtocolSECS As tom.Service

Public Sub OnCreate(ByVal Service As tom.Service)
    Dim ServiceSpecificDataDef As tom.DataDef
    Dim ToolEvent As tom.Event
    Dim DataItemOutput As tom.DataItem
    Dim DataItemInput As tom.DataItem

    Dim DataDef1 As tom.DataDef
    Dim DataDef2 As tom.DataDef
    Dim ChildDataDefA As tom.DataDef
    Dim ChildDataDefB As tom.DataDef

    Dim Method1 As tom.Method
```

```

Dim Method2 As tom.Method
Dim Method3 As tom.Method

' Save Service reference
Set m_oService = Service
Debug.Print "Entering OnCreate"

' Retrieve your Service Specific area in the Dictionary
Set ServiceSpecificDataDef = srvServiceDataDef(m_oService)

' Here is an how to load child DataDefs into your Service Specific area
Set DataDef1 = srvLoadDataDef(m_oService, srvServiceDataDef(m_oService),_
"DataDef1")
Set ChildDataDefA = srvLoadDataDef(m_oService, DataDef1, "ChildDataDefA")
Set ChildDataDefB = srvLoadDataDef(m_oService, DataDef1, "ChildDataDefB")
Set DataDef2 = srvLoadDataDef(m_oService, srvServiceDataDef(m_oService),_
"DataDef2")

' Here is how to define a Method object
' This method is Method1
Set Method1 = srvDefineMethod(m_oService, METH_METHOD1, "A Sample Method")
Set DataItemInput = srvAddDataItem(m_oService, Method1.Inputs,_
ServiceSpecificDataDef.Item("DataDef1"))

' Here is a second Method object
' This method is Method2
Set Method2 = srvDefineMethod(m_oService, METH_METHOD2, "A Second Sample Method")

' Here is a third Method object
' This method is Method3
Set Method3 = srvDefineMethod(m_oService, METH_METHOD3, "A Third Sample Method")

' Here is how to define Event objects
Set ToolEvent = srvDefineEvent(m_oService, EVENT_CONNECT, "A sample event")
Set DataItemOutput = srvAddDataItem(m_oService, ToolEvent.Outputs, _
ServiceSpecificDataDef.Item("DataDef2"))

Debug.Print "Leaving OnCreate"
End Sub

Private Sub Class_Initialize()
    SERVICE_NAME = App.Title + TypeName(Me)
End Sub

Public Sub LetAttribute(ByVal AttributeName As String, NewValue As Variant)
    Debug.Print "Entering LetAttribute"
    Select Case AttributeName
        Case ATT_EVENT_ENABLED
            Att_ToolEventEnable = NewValue
    
```

```

        Case Else
            Debug.Print "Cannot set ", AttributeName
        End Select
        Debug.Print "Leaving LetAttribute"
    End Sub

Public Function GetAttribute(ByVal AttributeName As String) As Variant
    Debug.Print "Entering GetAttribute"
    Select Case AttributeName
        Case ATT_EVENT_ENABLED
            GetAttribute = Att_ToolEventEnable
        Case Else
            Debug.Print "No such attribute exists, ", AttributeName
        End Select
        Debug.Print "Attribute is ", AttributeName
        Debug.Print "Leaving GetAttribute"
    End Function

Public Sub OnInitialize()
    Dim localAttribute As String

    Debug.Print "Entering OnInitialize"

    ' Perform initialization that must happen after Attributes are
    ' set and/or other services are started.

    ' Here is how to check to be sure a required service is present
    ' If the service is present, it is registered in the NT registry
    srvRequiredService m_oService, SRV_LOOPBACK
    srvRequiredService m_oService, SRV_PROTOCOLSECS

    ' Generate References to other services this service works with
    Set m_oLoopback = srvGetService(m_oService, SRV_LOOPBACK)
    Set m_oProtocolSECS = srvGetService(m_oService, SRV_PROTOCOLSECS)

    ' Check that no incompatible services are running
    '   srvIncompatibleService m_oService, ANYSERVICECONSTANT

    ' Subscribe to events your service requires
    srvSubscribeEvent m_oService, SRV_PROTOCOLSECS, "Connect"

    ' Set whether or not other services require notification
    ' Pass this handler support routine tomNotifyAlways or tomNotifyNever
    srvSetEventNotification m_oService, SRV_PROTOCOLSECS, "Connect", tomNotifyAlways
    'Make use of an attribute in OnInitialize rather than in OnCreate
    Debug.Print "Leaving OnInitialize"
End Sub

```

```

Public Sub OnExecute(ByVal ExecuteMethod As tom.Method)
    Dim MethodToExec As tom.Method
    Dim InvokingMethod As tom.Method
    Debug.Print "Entering OnExecute"
    On Error GoTo ErrorTrap

    Select Case ExecuteMethod.Name
        Case METH_METHOD1
            Debug.Print "Method 1 Executing"
            Debug.Print " ChildDataDefA: " & ExecuteMethod.Inputs.Item("DataDef1")._
                Item("ChildDataDefA").Value
            Debug.Print " ChildDataDefB: " & ExecuteMethod.Inputs.Item("DataDef1")._
                Item("ChildDataDefB").Value
            Set MethodToExec = srvCloneMethod(m_oLoopback, "Test")
            MethodToExec.Inputs.Item("ABS").Value = _
                ExecuteMethod.Inputs.Item("DataDef1").Item("ChildDataDefA").Value
            Set InvokingMethod = ExecuteMethod
            srvExecute MethodToExec, m_oService, InvokingMethod

        Case METH_METHOD2
            Debug.Print "Method 2 Executing"
            Set MethodToExec = srvCloneMethod(m_oService, METH_METHOD3)
            Set InvokingMethod = ExecuteMethod
            srvExecute MethodToExec, m_oService, InvokingMethod

        Case METH_METHOD3
            Debug.Print "Method 3 Executing"
            srvCompleted ExecuteMethod
    End Select
    Debug.Print "Leaving OnExecute"
    Exit Sub

ErrorTrap:
    Dim ErrorState As t_ErrorState
    srvSaveErrorState ErrorState
    Set ExecuteMethod = Nothing
    srvRestoreErrorState ErrorState
    srvExtendError "OnExecute"

End Sub

Public Sub OnMethodCompleted(ByVal CompletedMethod As tom.Method, ByVal InvokingMethod
As tom.Method)
    Debug.Print "Entering OnMethodCompleted", CompletedMethod.Name
    If InvokingMethod Is Nothing Then
        ' Do Verification
        lVerify CompletedMethod.Tag
    End If
End Sub

```

```

Else
    ' Take actions that should occur after method completes
    lCompleted CompletedMethod, InvokingMethod
End If
Debug.Print "Leaving OnMethodCompleted"
End Sub

Private Sub lVerify(Index As Variant)
    Dim VerifyingMethod As tom.Method
    Dim ExecuteMethod As tom.Method
    On Error GoTo ErrorTrap
    Select Case Index
        Case CaseStep1
            Set VerifyingMethod = srvCloneMethod(m_oService, METH_METHOD1)
            VerifyingMethod.Tag = CaseStep2

        Case CaseStep2
            Set VerifyingMethod = srvCloneMethod(m_oService, METH_METHOD2)
            If m_oFullVerification Then
                VerifyingMethod.Tag = CaseStep3
            Else
                VerifyingMethod.Tag = CaseEnd
            End If

        Case CaseStep3
            Set VerifyingMethod = srvCloneMethod(m_oService, METH_METHOD3)
            VerifyingMethod.Tag = CaseEnd

        Case CaseEnd
            srvVerified m_oService
            Exit Sub
    End Select

    srvExecute VerifyingMethod, m_oService, Nothing

ErrorTrap:
    Dim ErrorState As t_ErrorState
    srvSaveErrorState ErrorState
    Set ExecuteMethod = Nothing
    srvRestoreErrorState ErrorState
    srvExtendError "lVerify"
End Sub

```



```
Private Sub lCompleted(ByVal CompletedMethod As tom.Method, ByVal InvokingMethod As tom.Method)

    Dim FinishedSteps As Boolean
    Dim ExecuteMethod As tom.Method

    If (CompletedMethod.Error.ErrorCode <> 0) Then
        srvCompleted InvokingMethod, FailedMethod:=CompletedMethod
        FinishedSteps = False
        Debug.Print "Method Failed: ", InvokingMethod.Name
    Else
        Select Case CompletedMethod.Name
            Case "Test"
                Debug.Print "Completing Test"
                srvCompleted InvokingMethod
            Case "Method 3"
                Debug.Print "Completing Method 3"
                srvCompleted InvokingMethod
        End Select
    End If
End Sub

Public Sub OnVerify(ByVal FullVerification As Boolean)
    Debug.Print "Entering OnVerify"
    m_oFullVerification = FullVerification
    lVerify CaseStep1
    Debug.Print "Leaving OnVerify"
End Sub

Public Function Version() As String
    Debug.Print "Entering Version"
    Version = srvVersion
    Debug.Print "Leaving Version"
End Function

Public Sub OnTerminate()
    Debug.Print "Entering OnTerminate"
    Set m_oService = Nothing
    Set m_oLoopback = Nothing
    Set m_oProtocolSECS = Nothing
    Debug.Print "Leaving OnTerminate"
End Sub

Public Sub OnSubscribedEvent(ByVal TomEvent As tom.Event)
    Debug.Print "Entering OnSubscribedEvent"
    On Error GoTo ErrorTrap
End Sub
```

```
Dim NewEvent As tom.Event

If m_oService.Attributes.Item(ATT_EVENT_ENABLED).Value = "True" Then
    Set NewEvent = srvCloneEvent(m_oService, EVENT_CONNECT)
    Debug.Print NewEvent.Name
    If TomEvent.Name = "Connect" Then
        NewEvent.Outputs.Item("DataDef2").Value = TomEvent.Description
        srvTriggerEvent NewEvent
    End If
Else
    Debug.Print "ToolEventEnable is False"
End If
Debug.Print "Leaving OnSubscribedEvent"
Exit Sub

ErrorTrap:
Dim ErrorState As t_ErrorState
srvSaveErrorState ErrorState
Set NewEvent = Nothing
srvRestoreErrorState ErrorState
srvExtendError "OnSubscribedEvent"
End Sub

Private Sub Class_Terminate()
    Me.OnTerminate
End Sub
```

Container Service Code

B

Introduction

This appendix lists the code for the container Service, from the *nv10.vbp* file, which contains the *GemRemoteControl* container Service presented in Chapter 6.

Complete Code for Container Service

```

Option Explicit

' NV10
' This equipment handles resource-active differently.
' Instead of using remote commands, the equipment uses S7S1.
' This service "contains" a standard tomss2.GemRemoteControl
' object, and delegate most of functionality to it.
' it intercepts the OnExecute method, and uses S7S1 if
' the user is trying to active a resource. Otherwise, it
' delegates to GemRemoteControl, which uses S2F41 to send remote
' commands.

' Below is the standard service that this one "contains"
Private m_oBase As New tomss2.GemRemoteControl

' Object names
Private Const SERVICE_NAME = "NV10.NV10GemRemoteControl"
Private Const METHOD_COMMAND = "Command"

' Objects referenced
Private m_oService As tom.Service ' Service owning Me
Private m_oProtocol As Object ' SECS protocol handler

Private Sub Class_Terminate()
    Me.OnTerminate
End Sub

Public Sub OnTerminate()
    Set m_oService = Nothing
    Set m_oProtocol = Nothing
    Set m_oBase = Nothing
End Sub

Public Function Version() As String
    Version = srvVersion
End Function

Public Sub OnCreate(ByVal Service As tom.Service)
    Dim Trans As SecsTransaction
    Dim List As SecsItem

    ' This is where you create your "contained class"
    ' by executing the OnCreate handler method of that class
    m_oBase.OnCreate Service

```

```

' Save Service reference
Set m_oService = Service

' Get our protocol Handler
Set m_oProtocol = srvGetHandler(m_oService, "ProtocolSECS")

'   srvLoadDataDef m_oService, Nothing, "SECS elements"

Set Trans = secsDefineTransaction(m_oService, m_oProtocol, 7, 1)
Trans.Primary.Description = "Host command send"
Set List = secsAppendList(Trans.Primary.Root)
secsAppendItem m_oService, List, "RCMD"
Set List = secsAppendList(List)
Set List = secsAppendList(List, Name:="Parameter")
secsAppendItem m_oService, List, "CPNAME"
secsAppendItem m_oService, List, "CPVAL"
Trans.Secondary.Description = "Host command acknowledge"
Set List = secsAppendList(Trans.Secondary.Root)
secsAppendItem m_oService, List, "HACK", Value:=CByte(0)
Set List = secsAppendList(List)
Set List = secsAppendList(List, Name:="Parameter")
secsAppendItem m_oService, List, "CPNAME"
secsAppendItem m_oService, List, "CPACK"
End Sub

Public Function GetAttribute(ByVal AttributeName As String) As Variant
    GetAttribute = m_oBase.GetAttribute(AttributeName)
End Function

Public Sub LetAttribute(ByVal AttributeName As String, NewValue As Variant)
    m_oBase.LetAttribute AttributeName, NewValue
End Sub

Public Sub OnMethodCompleted(ByVal Method As tom.Method, ByVal InvokingMethod As tom.Method)
    m_oBase.OnMethodCompleted Method, InvokingMethod
End Sub

Public Sub OnVerify(ByVal FullVerification As Boolean)
    m_oBase.OnVerify FullVerification
End Sub

Public Sub OnExecute(ByVal Method As tom.Method)
    ' If user is activating a resource, then switch to a local function,
    ' otherwise delegate to base class.
    If Method.Inputs.Item("Commands").Item(1).Name = "PP-SELECT" Then
        Executes7F1 Method
    End If
End Sub

```

```

Else
    m_oBase.OnExecute Method
End If
End Sub

Public Sub ExecutesS7F1(Method As tom.Method)
    Dim Trans As SecsTransaction
    Dim i As Long
    Dim CPs As tom.DataItem
    Dim CP As SecsItem

    Set Trans = secsNewTransaction(m_oService, m_oProtocol, 7, 1)
    Set CPs = Method.Inputs.Item(1).Item(1)
    Trans.Primary.Item("RCMD") = CPs.DataDef.AccessID
    If CPs.Count = 0 Then
        Trans.Primary.Item("Parameter").Delete
    Else
        For i = 2 To CPs.Count
            Trans.Primary.Item("Parameter").Duplicate
        Next i
        For i = 1 To CPs.Count
            Set CP = Trans.Primary.Item("Parameter", i)
            CP.Item("CPNAME").Value = CPs.Item(i).DataDef.AccessID
            lSetCPVal CP.Item("CPVAL"), CPs.Item(i)
        Next i
    End If

    If secsSimulate(m_oProtocol) Then
        Trans.Secondary.Item("Parameter").Delete
    End If

    secsSend m_oService, m_oProtocol, Method, Trans
End Sub

Public Sub OnSecondaryIn(ByVal Method As tom.Method, ByVal Trans As SecsTransaction)
    Dim ACK As Long

    If Trans.Primary.Function = 1 Then
        ACK = secsItemAsLong(secsGetItem(Trans.Secondary, "HACK"))
        ' Ack code 4 is OK for S2F41
        If (ACK = 0) Or (ACK = 4) Then
            srvCompleted Method
        Else
            secsCompletedWithAck Method, Trans.Secondary, secsGetItem(Trans.Secondary, "HACK")
        End If
    End If

```

```
End If
    srvCompleted Method
End Sub

Private Sub lSetCPVal(SItem As SecsItem, DItem As tom.DataItem)
    On Error GoTo ErrorTrap

    Dim i As Long
    Dim ChildSItem As SecsItem

    SItem.Value = DItem.Value
    SItem.Format = secsFormat(DItem.DataDef)
    For i = 1 To DItem.Count
        Set ChildSItem = SItem.AddNew(SItem.ItemCount + 1)
        ChildSItem.Name = "CPVAL"
        lSetCPVal ChildSItem, DItem.Item(i)
    Next i
    Exit Sub

ErrorTrap:
    srvExtendError "lSetCPVal"
End Sub
```


Developing Equipment Services Using Sample Services

C

Introduction

This appendix presents some basic information about the examples included in the *FASTech\Sw\Dev\Samples\replace* directory. These samples show how to develop Level 1, Level2, and Level 3 Services from scratch and you can use these files as foundations for your own Services.

Level 1 Services are SECS Services, Level 2 Services are GEM Services, and Level 3 Services are VFEI Services.

For more information about SECS, GEM, and VFEI equipment standards, refer to the *STATIONworks Installation Guide*.

To test sample Services, you can use the FASTsim equipment simulation program (*FASTsim.exe*) provided with STATIONworks under *FASTech\Sw\Dev\Samples\winsecs\FASTsim*. You use this program to simulate equipment talking to a STATIONworks Tool. For further information on FASTsim, refer to the WinSECS Help file or the *WinSECS Reference* manual.

Finding Sample Equipment Services/Tools

The locations of sample replacement Services documented in this appendix and their corresponding Tools are indicated in the table that follows.

Sample Replacement Services Documented in Appendix

Sample Code, Manual, & Directory Location	Associated Tools & Location
Level 1 Service (SECS) <i>\FASTech\Sw\Dev\Samples\replace\replss1\replss1.vbp</i>	smp11 - new SecsLoopBack <i>FASTech\Sw\Dev\Samples\replace\Drivers</i>
Level 2 Service (GEM) <i>\FASTech\Sw\Dev\Samples\replace\replss2\replss2.vbp</i>	smp12 - new GemClock smp13 - new GemAlarm <i>FASTech\Sw\Dev\Samples\replace\Drivers</i>
Level 3 Service (VFEI) <i>\FASTech\Sw\Dev\Samples\replace\replss3\replss3.vbp</i>	smp15 - new VFEIAlarm using Old GClock and GAlarm smp16 - new VFEIAlarm using new GClock & new GAlarm <i>FASTech\Sw\Dev\Samples\replace\Drivers</i>

Help File for Sample Services

The Help file for these equipment Services is called *Replace.hlp*. You can find the Help file in *FASTech\Sw\Dev\Samples\bin*.

Building Replacement Services Tool Database

Build the replacment Services database just as you built the sample Tool database in Chapter 1, only this time, use the *.tbf* files from under *FASTech\Sw\Dev\Samples\replace\Drivers*.

Using Sample (Replacement) Equipment Services

To use the sample equipment Services, you do not need to build them. They are already built for you and their DLLs, executables, Help files, and data files are included in the *FASTech\Sw\Dev\Samples\bin* directory. You must, however, recompile any sample Services that you further modify.

These Services are all referred to as *replacement* Services, because they replace existing Services in a situation where the equipment is not quite standard.

To use the Services:

1. Register the replacement sample DLL OLE servers on your machine by running *register.bat* (in the *FASTech\Sw\Dev\Samples\bin* directory).
2. Set the Target in the shortcut to TOM Explorer to
`D:\FASTech\Sw\Bin\texplorer.exe /d replace.mdb` (where D is the drive TOM Explorer is installed on) and run TOM Explorer.

OR

Set the Target in the shortcut to the TOM DB Editor to

`D:\FASTech\Sw\Bin\tomdb.exe /d replace.mdb` (where D is the drive TOM Explorer is installed on) and run the TOM DB Editor.

You can refer to the Help for the replacement Services by executing the *replace.hlp* file in the *FASTech\Sw\Dev\Samples\bin* directory or by right clicking on the Service name in TOM Explorer.

Examining Sample Level 1 Service

Open the *replss1.vbp* project. This project produces the *replss1.dll*. It contains only one Service class module named *1s2f25.cls*, a replacement for the standard Level 1 *SecsLoopbackDiagnostic* Service. This Service offers the same features as the Service it is replacing.

If you have not already compiled the Service, compile it in Visual Basic and then open the `smp11 - new SecsLoopBack Tool` in TOM Explorer.

Examining Sample Level 2 Service

Open the *replss2.vbp* project. This project produces the *replss2.dll*. It contains two Service class modules named *2clock.cls*, *SampleNewGemClock*, a replacement for the standard Level 2 *GemClock* Service. This Service offers not only the features of the Service it is replacing, but a new Attribute and Event as well. The code for the new Attribute and Event is highlighted. When you open the project, pay special attention to the sections marked “- - - new - - -”.

If you have not already compiled the Service, compile it in Visual Basic and then open the `smp12 - new GemClock Tool` in TOM Explorer. Then try opening the `smp13 - new GemAlarm Tool`.

Examining Sample Level 3 Service

Open the *replss3.vbp* project. This project produces the *replss3.dll*. It contains a Service class module named *3vfeialm.cls*, *SampleNewVFEIAlarmAManagement*, a replacement for the standard Level 3 *VFEIAlarmManagement* Service that enhances the original code.

For a detailed description of the enhancements, refer to the comments in the code.

If you have not already compiled the Service, compile it in Visual Basic and then open the *smpl5 - new VFEIAlarm* using old *GClock* & old *GAlarm* in TOM Explorer. Then try opening the *smpl6 - new VFEIAlarm* using new *GClock* & new *GAlarm* Tool.

Removing Samples

When you have finished using the replacement sample Services, save any you would like to keep in a new location.

If you registered the DLLs for the replacement Services, you should also unregister those DLLs by executing *UnRegister.bat* under *FASTech\Sw*.

If you compiled the samples, to remove both the source and the compiled files from your machine, execute the *cleanup_all.bat* script located in *FASTech\Sw\Dev\Samples*.

Developing Help Files for Services Documentation Kit

D

Introduction

This appendix presents some basic information about the sample Help file included in the *FASTech\Sw\Dev\Samples\DocKit* directory. This sample shows how to develop Help files for your own Services using RoboHelp.

Writing Help Files for Your Custom Services

When you develop custom Services you would like Brooks to distribute, you should develop Help files to accompany them.

To develop Help files that meet Brooks's documentation standards, you should work with a Help-authoring product called RoboHelp, available from:

Blue Sky Software
7777 Fay Avenue Suite 201
La Jolla, CA 92037
www.blue-sky.com
800/455-5132

For some sample Help files to work with, refer to the HLP file in *FASTech\STATIONworks\Dev\Samples\DocKit*.

The *swdockit.hlp* file explains how to write a Help file and is itself a sample. The components that went into building it are also included in the DocKit directory:

- *swdockit.hpj*—The Help project file
- *swdockit.cnt*—Help contents file
- *swdockit.doc* Source file that contain the text of the Help
- *Robohelp.dot*—Main Help template file
- *Robortf.dot*—Template file to be used by subordinate files in the sample Help project that are developed on different machines.

For further information, execute the *swdockit.hlp* file and read it.

Using Testing Services

E

Introduction

Topics in This Chapter

FTIAttributeForms, p. E-2

FTISizeInfo, p. E-8

This appendix does not present all testing Service samples available with the product. For information on additional Services, refer to the *samples.htm* file available in the *FASTech\Sw\Dev\Samples* directory of STATIONworks.

The Services in this appendix are to facilitate the following actions in a Service or Tool driver development environment:

- Setting TOM attributes (*FTIAttributeForms*)
- Gathering sizing information (*FTISizeInfo*)

You use these Services while developing or testing Services or drivers, where you may need to alter connections or change attribute settings frequently.

To make use of the Services you must:

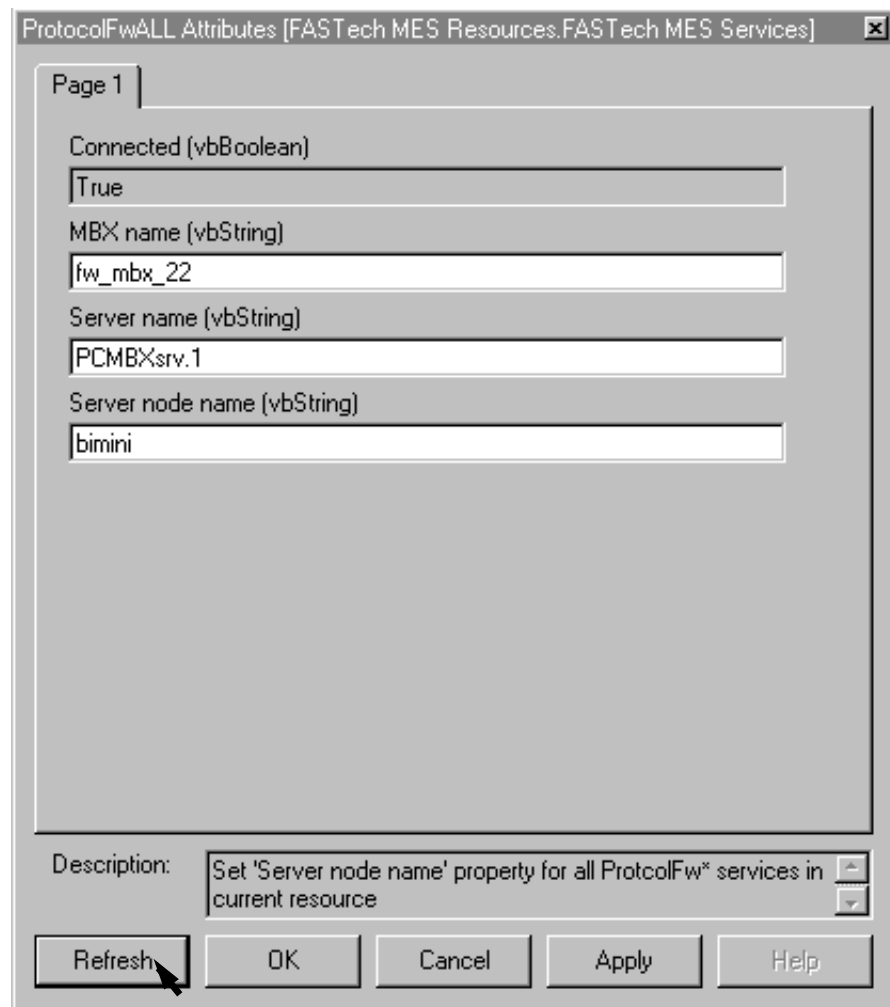
- Add the related *.tbf* files under *FASTech\Sw\Dev\Samples\Misc\TBFs* to the *Services* subdirectory in your database before building the TOM database.
- Register the *FTIdev5.dll* and *FrmServ.exe* files. You register these files by going to the *FASTech\Sw\Dev\Samples\Misc\Servers* directory, finding the *register.bat* file, and executing the file.

FTIAttributeForms

The *FTIAttributeForms* service provides three forms for viewing and/or setting Service Attributes. One for *ProtocolSECS*, another for *ProtocolMBX*, and a third generic form for all other Services.

Generic Form

An example of the generic form provided for all Services other than *ProtocolSECS* and *ProtocolMBX* is shown below.



The screenshot shows a dialog box titled "ProtocolFwALL Attributes [FASTech MES Resources.FASTech MES Services]". It contains a "Page 1" tab and a form with the following fields:

- Connected (vbBoolean): True
- MBX name (vbString): fw_mbx_22
- Server name (vbString): PCMBXsrv.1
- Server node name (vbString): bimini

At the bottom, there is a "Description:" field with the text "Set 'Server node name' property for all ProtocolFw* services in current resource". Below the description are five buttons: Refresh, OK, Cancel, Apply, and Help. A mouse cursor is pointing at the Refresh button.

The *FTIAttributeForms* Service dynamically generates this form for each Service whose attributes you want to set. If the Service has more Attributes than can be displayed in a single page, a Page 2 tab appears next to the Page 1 tab. Naturally, each Service has different Attributes,

but the forms all have several common elements presented under *Common Elements of All Three Forms*, p. E-4.

ProtocolSECS Form

For details on the ProtocolSECS form, refer to the samples.htm file available in the samples directory.

This form has several elements in common with the other *Attributes* forms presented under *Common Elements of All Three Forms*, p. E-4.

ProtocolMBX Form

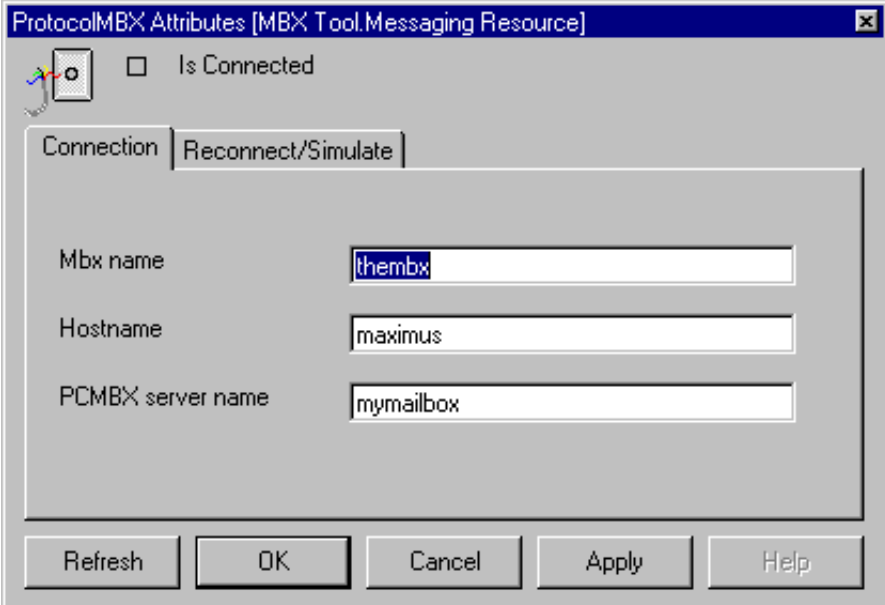
The *Attributes* form for *ProtocolMBX* has two “pages,” each with a tab you use to display them:

- Connection
- Reconnect/Simulate

Connection

Under the *Connection* tab, you set the *Attributes* required to connect STATIONworks to FACTORYworks:

- MBX name
- Hostname
- PCMBX server name



Reconnect/Simulate

Under the *Reconnect/Simulate* tab, you set these *Attributes*:

- `Connect at startup`—An automatic connect option. Toggle between `True` and `False`. `True` when the check mark displays.
- `Automatic reconnect`—An automatic reconnect when the MBX connection is lost. Toggle between `True` and `False`. `True` when the check mark displays.
- `Automatic reconnect interval`—Number of milliseconds between reconnect attempts.
- `Simulate`—Simulate mode option. Toggle between `True` and `False`. `True` when the check mark displays.
- `Simulated reply default`—The reply to send to the mailbox when in `Simulate` mode.

This form has several elements in common with the other `Attributes` forms presented under *Common Elements of All Three Forms*, p. E-4.

Common Elements of All Three Forms

Attributes, Values, Types

Read-only `Attributes` display in a grayed text box that you cannot edit.

The Visual Basic `VarType` of each `Attribute` appears in parentheses next to its name. If the `VarType` is `vbString`, then by double clicking on the text box, you expand the box to display multiple lines that you can edit. In this Multiline Edit/View mode, you can enter a multiline string with carriage returns and all buttons at the bottom of the form becomes disabled except `OK`, `Cancel`, and `Help`.

Clicking `OK` in this mode sets the new value in the form, but not in the Service.

Description

A `Description` block that displays the description of the Attribute whose value field you have the cursor in. If the description exceeds two lines, scrollbars become available.

Buttons

All three forms have the same buttons (corresponding keystrokes):

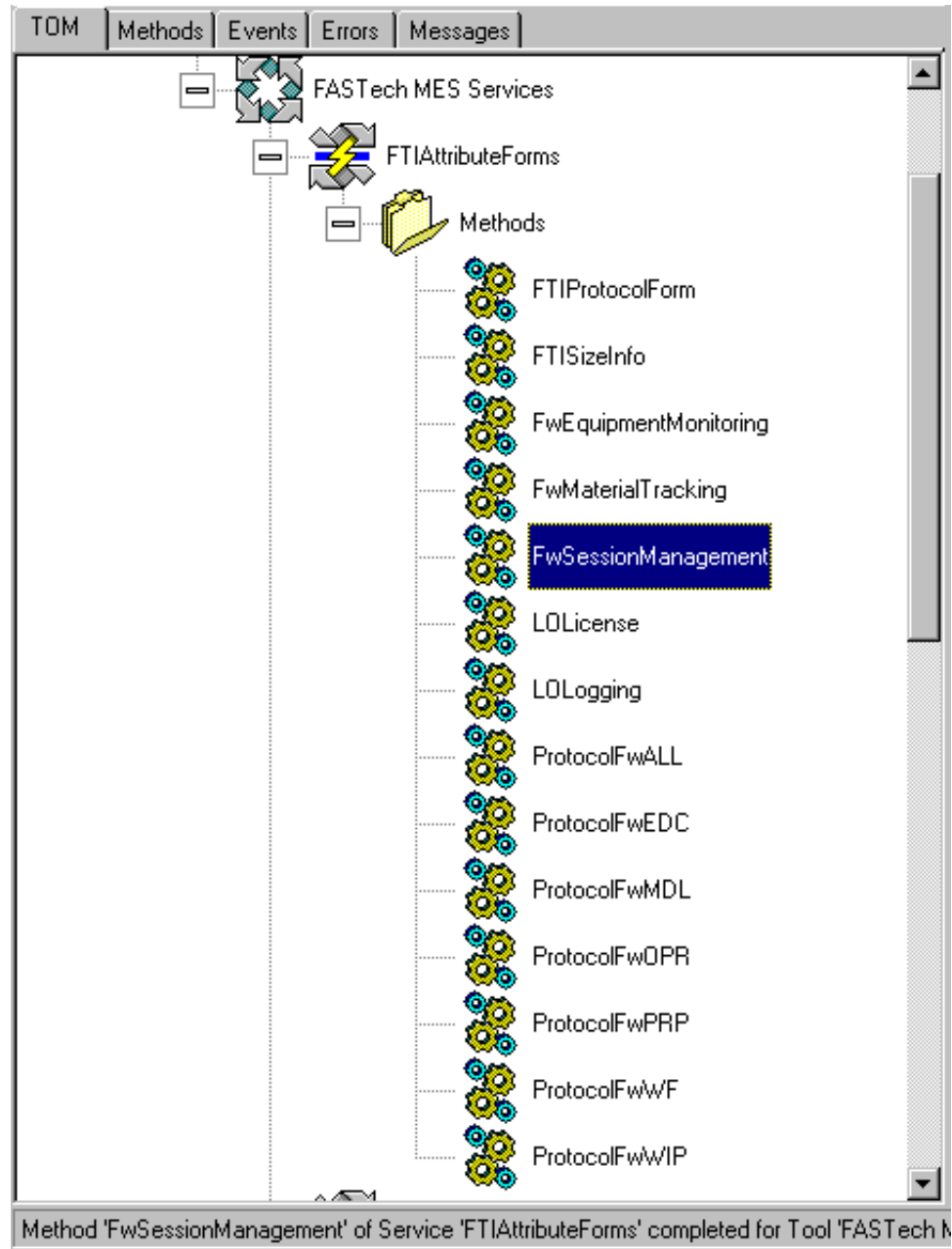
- ◆ `Refresh`—Retrieves the latest Attribute settings.
- ◆ `OK` (or `Enter`)—Sets the Attributes and closes the form.
- ◆ `Cancel` (or `ESC`)—Cancels settings of Attributes and closes form.
- ◆ `Apply`—Sets the Attributes and leaves the form open.
- ◆ `Help`—Launches TOM Help for the Service whose attributes you are viewing and/or setting.

Attributes

None

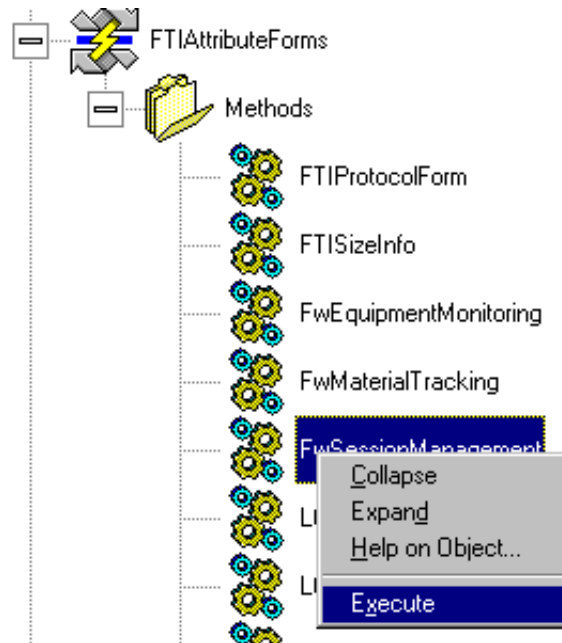
Methods

The *FTIAttributeForms* Service dynamically creates Methods whose names correspond to the Services of the Tool it is assigned to.



To see the Methods, expand the *FTIAttributeForms* Service's Methods under the Tool. Here you can see a list of Methods, each with the name of one of the Tool's Services (see preceding illustration).

To execute the Attributes form for one of the corresponding Services, you right click on the Method and select `Execute`.



You then see the Attributes form displaying that Service's Attributes.

Events

None

NOTE

Since the Methods are created in the `OnInitialize` routine, Methods are not created for cloned Services.

FTISizeInfo

This Service provides information on the TOM application size using API calls and indicates the number of messages per second being transmitted/received.

Other Services Required

For this Service to load, it requires two other Services be in the database:

- *ProtocolTimer*
- *LOLogging service*

Remarks

When you load *FTISizeInfo*, it clones the *LOLogging* Service and names the clone *FTISizeInfo_log* then turns off any other logging options on this clone. The `Log file base path` Attribute of this Service is available to set when the `Logging` Attribute has been set to `True`.

Each Poll interval executes a `Trace Method` containing these Attributes:

- `Application Memory Size`
- `Tool Message Rate`
- `CPU percentage`

The Method traces these attributes into a file, where they are separated by a TAB character, so you can easily import the log file values into a spreadsheet or data analysis program.

Attributes

`Application Memory Size`

`Long`. Size of TOM Application in KB

`Log file base path`

`String`. Points to the Log file base path of the clone of the *LOLogging* Service this Service creates.

`Logging`

`Boolean`. Turns logging on/off for the updates

`Poll Interval`

`Long`. Interval in seconds (1-360) that the `Application Memory Size` and `Tool Message Rate` should be updated.

`Tool Message Rate`

`Single`. messages/sec

CPU percentage

String. Defined by dividing the application's process time by system clock ticks.

Methods

None

Events

None

Code for Initialize Tool Service

F

Introduction

This appendix lists the code for the `init` (initialize tool) Service's class, from the `sample2.cls` file. This Service is the one that contains the `StartTool` Method illustrated in Chapter 8.

Complete Code for the Init.sample2 Service

```
' ---- FASTech Integration.  Copyright 1999-2000
' 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.

Option Explicit

' Object Names
Private SERVICE_NAME As String

Private Const SRV_PROTOCOLSECS = "ProtocolSECS"
Private Const SRV_GEMESTABCOMMS = "GemEstablishCommunications"
Private Const SRV_GEMIDENTIFICATION = "GemIdentification"
Private Const SRV_LOLOGGING = "LOLogging"
Private Const SRV_PROTTIMER = "ProtocolTimer"

'DataDef names
Private Const METH_START = "StartTool"

'Boolean that indicates whether or not Full Verification is on
Private m_oFullVerification As Boolean

' Constants for sequence of verification
Private Const CaseStep1 = 1
Private Const CaseEnd = 4

' References
Private m_oService As tom.Service 'Service that owns this class

' Global reference to a custom method
Private m_StartTool As tom.Method

Public Sub OnCreate(ByVal Service As tom.Service)

    Dim ServiceSpecificDataDef As tom.DataDef
    Dim StartTool As tom.Method

    ' Save Service reference
    Set m_oService = Service

    Debug.Print "Entering OnCreate"

    ' Here is the StartTool Method Object
    Set StartTool = srvDefineMethod(m_oService, METH_START, "StartTool Method")

    Debug.Print "Leaving OnCreate"
End Sub
```

```

Private Sub Class_Initialize()
    SERVICE_NAME = App.Title + TypeName(Me)
End Sub

Public Sub LetAttribute(ByVal AttributeName As String, NewValue As Variant)
    Debug.Print "Entering LetAttribute"
    'no attributes to set
    Debug.Print "Leaving LetAttribute"
End Sub

Public Function GetAttribute(ByVal AttributeName As String) As Variant
    Debug.Print "Entering GetAttribute"
    'no attributes to get
    Debug.Print "Attribute is ", AttributeName
    Debug.Print "Leaving GetAttribute"
End Function

Public Sub OnInitialize()
    Dim localAttribute As String

    Debug.Print "Entering OnInitialize"
    ' Perform initialization that must happen after Attributes are
    ' set and/or other services are started.

    ' Here is how to check to be sure a required service is present
    ' If the service is present, it is registered in the NT registry
    srvRequiredService m_oService, SRV_PROTOCOLSECS
    srvRequiredService m_oService, SRV_GEMESTABCOMMS
    srvRequiredService m_oService, SRV_GEMIDENTIFICATION
    srvRequiredService m_oService, SRV_LOLOGGING
    srvRequiredService m_oService, SRV_PROTTIMER

    ' Subscribe to events your service requires
    ' This event occurs when you execute the Open Method of ProtocolSECS
    srvSubscribeEvent m_oService, SRV_PROTOCOLSECS, "Connect"
    ' These events occur when you execute Connect Method of GemEstablishCommunications
    srvSubscribeEvent m_oService, SRV_GEMESTABCOMMS, "Established communications"
    srvSubscribeEvent m_oService, SRV_GEMESTABCOMMS, "Changed"

    ' Set whether or not other services require notification
    ' Pass this handler support routine tomNotifyAlways or tomNotifyNever
    srvSetEventNotification m_oService, SRV_PROTOCOLSECS, "Connect", tomNotifyAlways
    srvSetEventNotification m_oService, SRV_GEMESTABCOMMS, _
        "Established communications", tomNotifyAlways
    srvSetEventNotification m_oService, SRV_GEMESTABCOMMS, "Changed", tomNotifyAlways

    'Make use of an attribute in OnInitialize rather than in OnCreate

```

```

    Debug.Print "Leaving OnInitialize"
End Sub

Public Sub OnExecute(ByVal ExecuteMethod As tom.Method)
    Dim MethodToExec As tom.Method
    Dim InvokingMethod As tom.Method

    Debug.Print "Entering OnExecute"

    On Error GoTo ErrorTrap

    Select Case ExecuteMethod.Name

        Case METH_START

            Debug.Print "StartTool Method Executing"
            'Set MethodToExec = srvCloneMethod(m_oLoLogging, "Stop")
            Set MethodToExec = srvCloneMethod(m_oService, "Stop", SRV_LOLOGGING)
            'Set MethodToExec.Tag = srvCloneMethod(m_oLoLogging, "Start")
            Set MethodToExec.Tag = srvCloneMethod(m_oService, "Start", SRV_LOLOGGING)
            Set InvokingMethod = ExecuteMethod
            'the method executed from the TOM Explorer or the IDE Browser
            'becomes the invoking method
            Set m_StartTool = InvokingMethod 'for use by OnSubscribedEvent
            'Without this Method object, OnSubscribedEvent doesn't know invoking Method
            srvExecute MethodToExec, m_oService, InvokingMethod

        End Select

    Debug.Print "Leaving OnExecute"
    Exit Sub

ErrorTrap:
    Dim ErrorState As t_ErrorState
    srvSaveErrorState ErrorState
    Set ExecuteMethod = Nothing
    srvRestoreErrorState ErrorState
    srvExtendError "OnExecute"

End Sub

Public Sub OnMethodCompleted(ByVal CompletedMethod As tom.Method, ByVal InvokingMethod
As tom.Method)
    Debug.Print "Entering OnMethodCompleted", CompletedMethod.Name

    If InvokingMethod Is Nothing Then
        ' Do Verification
        lVerify CompletedMethod.Tag
    Else
        ' Take actions that should occur after method completes

```

```

        lCompleted CompletedMethod, InvokingMethod
    End If
    Debug.Print "Leaving OnMethodCompleted"
End Sub

Private Sub lVerify(Index As Variant)
    Dim VerifyingMethod As tom.Method
    Dim ExecuteMethod As tom.Method

    On Error GoTo ErrorTrap

    Select Case Index
        Case CaseStep1
            Set VerifyingMethod = srvCloneMethod(m_oService, METH_START)
            VerifyingMethod.Tag = CaseEnd

        Case CaseEnd
            srvVerified m_oService
            Exit Sub
    End Select

    srvExecute VerifyingMethod, m_oService, Nothing

ErrorTrap:
    Dim ErrorState As t_ErrorState
    srvSaveErrorState ErrorState
    Set ExecuteMethod = Nothing
    srvRestoreErrorState ErrorState
    srvExtendError "lVerify"

End Sub

Private Sub lCompleted(ByVal CompletedMethod As tom.Method, ByVal InvokingMethod As tom.Method)
    Dim FinishedSteps As Boolean
    Dim ExecuteMethod As tom.Method
    Dim MethodToExec As tom.Method
    Dim TheCount As Integer

    TheCount = 0

    If (CompletedMethod.Error.ErrorCode <> 0) Then
        srvCompleted InvokingMethod, FailedMethod:=CompletedMethod
        FinishedSteps = False
        Debug.Print "Method Failed: ", InvokingMethod.Name
    Else
        Select Case CompletedMethod.Name
            Case "Stop"

```

```

        'This is completion of the Lologging Stop Method to ensure no error
        'on initiating logging.
        Debug.Print "StartTool Executing Start Method of Lologging Service"
        Set MethodToExec = CompletedMethod.Tag
        'Gets Start method from tag of Stop method
        Set MethodToExec.Tag = srvCloneMethod(m_oService, "Close", _
            SRV_PROTOCOLSECS) 'Sets tag to the next method, Close
        srvExecute MethodToExec, m_oService, InvokingMethod

    Case "Start"
        'This is completion of the Lologging Start Method.
        Debug.Print "StartTool Executing Close Method of ProtocolSECS Service"
        Set MethodToExec = CompletedMethod.Tag 'Gets Close method from tag of
        Start method
        Set MethodToExec.Tag = srvCloneMethod(m_oService, "Open", _
            SRV_PROTOCOLSECS) 'Sets tag to the next method, Open
        srvExecute MethodToExec, m_oService, InvokingMethod

    Case "Close"
        'This is completion of the ProtocolSECS Close method to ensure no error
        'when executing the Open method.
        Debug.Print "StartTool Executing Open Method of ProtocolSECS Service"
        Set MethodToExec = CompletedMethod.Tag
        'Gets Open method from tag of Close method
        srvExecute MethodToExec, m_oService, InvokingMethod

    Case "Open"
        'After the Open Method, Service waits for Connect Event.

    Case "Connect"
        'After the Connect Method, Service waits for the Established
        'communication or Changed Event.

    End Select
End If
End Sub

Public Sub OnVerify(ByVal FullVerification As Boolean)

    Debug.Print "Entering OnVerify"
    m_oFullVerification = FullVerification
    lVerify CaseStep1
    Debug.Print "Leaving OnVerify"
End Sub

Public Function Version() As String
    Debug.Print "Entering Version"
    Version = srvVersion

```

```
    Debug.Print "Leaving Version"
End Function

Public Sub OnTerminate()
    Debug.Print "Entering OnTerminate"
    Set m_oService = Nothing
    Debug.Print "Leaving OnTerminate"
End Sub

Public Sub OnSubscribedEvent(ByVal tomEvent As tom.Event)
    Debug.Print "Entering OnSubscribedEvent"
    On Error GoTo ErrorTrap

    Dim ReceivedEvent As tom.Event 'Event subscribed to
    Dim ExecuteMethod As tom.Method
    Dim MethodToExec As tom.Method
    Dim TheCount As Integer

    Set ReceivedEvent = tomEvent

    Select Case ReceivedEvent.Name

        Case "Connect"
            Debug.Print "Received notification of Connect Event from ProtocolSECS"
            Debug.Print "Completing StartTool Method's Opening of Port"
            srvGetService(m_oService, SRV_GEMESTABCOMMS).Attributes.Item("Inter-
            val").Value = 5
            Set MethodToExec = srvCloneMethod(m_oService, "Connect", SRV_GEMESTABCOMMS)
            srvExecute MethodToExec, m_oService, m_StartTool

        Case "Established communications", "Changed"
            Debug.Print "Received notification that Communication with Tool has_
            been established"
            Debug.Print "StartTool Method has communicated with Tool"
            If Not m_StartTool Is Nothing Then
                srvCompleted m_StartTool
                Set m_StartTool = Nothing
            End If

    End Select

    Debug.Print "Leaving OnSubscribedEvent"
    Exit Sub

ErrorTrap:
    Dim ErrorState As t_ErrorState
    srvSaveErrorState ErrorState
    Set ReceivedEvent = Nothing
```

```
    srvRestoreErrorState ErrorState
    srvExtendError "OnSubscribedEvent"
End Sub

Private Sub Class_Terminate()
    Me.OnTerminate
End Sub
```


Index

A

actions on startup
 initiating
 handler methods required 3-2

Attributes

 creating in database 4-28
 handler methods required 3-2
 Service requirements 1-2, 6-3
 setting 5-3
 settings for verification process 3-35
 what Service can do with 4-28
 when available 3-10, 3-16

Attributes of ProtocolSECS

 setting 8-8

Auto Refresh

 toggling in TOM Explorer 5-6

C

Class 5-3

class methods

 required 3-4
 Terminate 3-41

cloning

 purpose and advantages of 3-23

communication

 establishing with Tool 8-2

container Services

 choosing standard Service to contain 6-2
 creating handler methods 6-3
 creating in database 6-3
 dictionaries 6-3
 when to use 6-1

D

data values

 initialize using Attributes 3-16
 where to initialize 3-6

database

 adding Service 5-2

 setting for TOM Explorer 5-4

DataDefs

 creating clones of 4-27
 creating references to children 3-8
 Service requirements 1-2, 6-3, 8-7
 Service specific area
 creating 3-7
 loading 3-7

DataItems

 creating for Events 3-11
 creating for Methods 3-9
 setting for Methods 3-9

debugging Services 5-1

description

 parameters for handling errors 7-4

Dictionaries

 Service requirements 1-2, 6-3

Dictionary

 Service specific area
 creating 3-7

E

error notifications

 setting up 3-18

error strings

 associating error with 7-4

errors

 extending 7-3
 handling in OnMethodCompleted 3-27
 notification to TOM application
 sending 3-18
 raising 7-6
 raising vs. extending vs. triggering 7-2
 triggering 7-8

Event notifications

 canceling 3-18
 setting up 3-18

Events

- creating 3-11
 - DataItems
 - creating 3-11
 - debugging 5-10
 - defining in your Service 3-11
 - other Service's
 - using in your Service 3-31
 - subscribing to 3-17
 - handler methods required 3-2
 - testing 5-10
 - triggering for higher level Service 3-31
 - verifying 3-36
- F**
- Full Verification
 - setting in TOM Explorer 5-11
 - toggling on in TOM Explorer 3-33
 - full verification 3-34
 - FullVerification 3-34
 - functions
 - GetAttribute 3-15
- G**
- GetAttribute 3-15
 - sample code 3-15
- H**
- handler methods
 - Attributes
 - handling 3-2
 - GetAttribute 3-15
 - LetAttribute 3-14
 - OnCreate 3-5
 - OnExecute 3-20
 - OnInitialize 3-16
 - OnMethodCompleted 3-4, 3-25
 - OnSubscribedEvent 3-18, 3-30
 - OnTerminate 3-41
 - OnVerify 3-33
 - order TOM calls 3-3
 - required 3-2
 - startup actions
 - handling 3-2
 - subscribed Events
 - handling 3-2
 - timers
 - handling 3-2
 - TOM's use of 3-2
 - order of calling 3-3
 - Version 3-40
 - handler support routines
 - see Routines
- I**
- incompatible Services
 - checking for 3-16
 - invoking Method 3-25
 - defined 3-22
 - IPAddressLocal 8-8
 - IPAddressRemote 8-8
 - IPPortLocal 8-8
 - IPPortRemote 8-8
- L**
- ICompleted
 - sample code 3-29
 - LetAttribute 3-14
 - sample code 3-14
 - logging
 - starting 8-2
 - IVerify
 - sample code 3-39
- M**
- Method notification 5-10
 - Method objects
 - using in OnSubscribedEvent 8-6
 - Methods
 - chaining 8-4, 8-5
 - cloning 3-22
 - creating 3-9
 - DataItems
 - creating 3-9
 - setting 3-9
 - debugging 5-8
 - defining in your Service 3-9
 - determining which to execute in OnExecute 3-20
 - executing 3-22
 - steps to 3-22
 - executing in TOM Explorer 5-8
 - existing Services

- using 3-23
 - other Service's
 - executing in your Service 3-22
 - Properties
 - Notify
 - settings for 3-18
 - responding to completion of 8-5
 - verifying 3-36
 - where to code action 3-21
- N**
- notification
 - completion 3-23
- notifications
 - error 3-18
- Notify Property of a Method
 - settings for 3-18
- O**
- objects
 - cleaning up 3-41
- OnCreate 3-5
 - actions to take in 3-6
 - restrictions 3-12
 - sample code 3-12
- OnExecute 3-20
 - determining Method to execute 3-20
 - handling errors 3-21
 - sample code 3-24
 - starting StartTool Method in 8-4
 - trapping errors in 3-20
- OnInitialize 3-16
 - restrictions 3-19
 - sample code 3-19
- OnMethodCompleted 3-4, 3-25, 8-5
 - handling errors 3-27
 - major branches in 3-25, 3-26
 - sample code 3-29
 - seeing in action 5-9
 - verification process 3-26, 3-38
- OnSubscribedEvent 3-30
 - sample code 3-32
 - when required 3-18
- OnunsubscribedEvent 8-6
- OnTerminate 3-41
 - debugging 5-14
 - testing 5-14
- OnVerify 3-33
- P**
- partial verification 3-34
- port
 - opening 8-2
- Properties
 - Notify
 - settings for 3-18
 - Verification Completed 3-36
 - Verified 3-36
 - verify status 3-36
- ProtocolSECS Attributes
 - setting 8-8
- Provider 5-3
- R**
- required Services
 - checking for 3-17
- routine
 - srvServiceDataDef 3-7
- routines
 - srvAddDataItem 3-9, 3-36
 - srvCloneDataDef 4-27
 - srvCloneEvent 7-4
 - srvCloneMethod 3-22, 3-36, 7-4
 - srvCompleted 3-21, 3-23, 3-25, 3-27
 - srvDefineEvent 3-11
 - srvDefineMethod 3-9
 - srvExecute 3-22, 3-25, 3-36
 - srvExtendError 3-17, 7-3
 - srvGetService 3-17
 - srvIncompatibleService 3-16
 - srvLoadDataDef 3-8
 - srvRaiseError 7-6
 - srvRequiredService 3-17
 - srvRestoreErrorState 7-5
 - srvSaveErrorState 7-4
 - srvServiceDictionaryRoot 4-26
 - srvSetEventNotification 3-18
 - srvTriggerError 7-8
 - srvTriggerEvent 3-31, 3-36
 - srvVerified 3-35, 3-37
 - srvVersion 3-40
 - SubscribeEvent 3-17

S

Service

- adding to database 5-2
- Events subscribed to
 - handler methods for 3-2

Service reference

- where to save 3-6

Service specific area

- creating 3-7
- loading DataDefs from 3-7

ServiceProvider 5-3

Services

- actions on startup
 - handler methods for initiating 3-2
- Attributes
 - handler methods for 3-2
- compiling 5-15
- container
 - choosing standard service to contain
 - 6-2
 - purpose of 6-1
- debugging 5-1
- Events
 - creating 3-11
 - display in TOM Explorer 3-12
- incompatible
 - checking for 3-16
- levels
 - order TOM calls 3-3
- Methods
 - creating 3-9
 - display in TOM Explorer 3-10
- other
 - generating references to 3-17
- relationship to Tools 1-2
- required
 - checking for 3-17
- setting Class 5-3
- setting Provider 5-3
- start Tool 8-2
- subscribing to events 3-17
- testing 5-15

timers

- handler methods for 3-2
- verifying 5-11
- srvAddDataItem 3-9, 3-36
- srvCloneDataDef 4-27
- srvCloneEvent 7-4
- srvCloneMethod 3-22, 3-36, 7-4
- srvCompleted 3-21, 3-23, 3-25, 3-27
- srvDefineEvent 3-11
- srvDefineMethod 3-9
- srvExecute 3-22, 3-25, 3-36
- srvExtendError 3-17, 7-3
 - parameters for description 7-4
- srvGetService 3-17
- srvIncompatibleService 3-16
- srvLoadDataDef 3-8
- srvRaiseError 7-6
 - parameters for description 7-4
- srvRequiredService 3-17
- srvRestoreErrorState 7-5
- srvSaveErrorState 7-4
- srvServiceDataDef 3-7
- srvServiceDictionaryRoot 4-26
- srvSetEventNotification 3-18
- srvSubscribeEvent 3-17
- srvTriggerError 7-8
 - parameters for description 7-4
- srvTriggerEvent 3-31, 3-36
- srvVerified 3-35, 3-37
- srvVersion 3-40
- StartTool Method
 - starting 8-4

T

- Terminate class method 3-41
- timers
 - handler methods required 3-2
- TOM 5-10
- TOM Core 1-10
- TOM defined 1-10
- TOM Explorer
 - Auto Refresh
 - toggling 5-6
 - debugging with 5-1

- exiting 5-14
- Method notification 5-10
- setting database 5-4
- setting Full Verification 5-11

Tools

- conceptual 1-2
- establishing communication with 8-2
- relationship to Services 1-2
- starting with Service 8-2

V

values

- initialize using Attributes 3-16
- where to initialize 3-6

- verification
 - debugging 5-11
- verification process 3-33, 3-34, 3-36
 - Attribute settings 3-35
 - flow of code 3-38
 - OnMethodCompleted 3-38
 - Properties to set 3-36
 - sample 3-37
- Version 3-40

