SYBASE®

Developer Reference for Windows and Windows Mobile

Sybase Unwired Platform 1.5.2

Contents

Introduction to Developer Reference for Windows and Windows Mobile

This developer reference provides information about using advanced Sybase® Unwired Platform features to create applications for Microsoft Windows and Windows Mobile devices. The audience is advanced developers who are familiar working with APIs, but who may be new to Sybase Unwired Platform.

This guide describes requirements for developing a device application for the platform, how to generate application code, and how to customize the generated code using the client object API. Also included are task flows for the development options, procedures for setting up the development environment, and client object API documentation.

Companion guides include:

- *Sybase Unwired WorkSpace – Mobile Business Object*
- *Sybase Unwired WorkSpace – Device Application Development*
- *Tutorial: Windows Mobile Application Development (Device Application Designer)*
- *Tutorial: Windows Mobile Device Application Development (Custom Development)*
- *Troubleshooting for Sybase Unwired Platform*
- *C# documentation, which provides a complete reference to the APIs:*
    - Compiled help for the Device Framework API is installed to `<UnwiredPlatform_InstallDir>\Unwired_WorkSpace \VisualStudio\ComponentLibrary\help`.
    - You can integrate help for generated code from mobile business objects (MBOs) into your Visual Studio project. See *Integrating Help into a Project* on page 30.

See *Fundamentals* for high-level mobile computing concepts, and a description of how Sybase Unwired Platform implements the concepts in your enterprise.

Documentation Road Map for Unwired Platform

Learn more about Sybase® Unwired Platform documentation.

Table 1. Unwired Platform documentation

Document	Description
Sybase Unwired Platform Installation Guide	Describes how to install or upgrade Sybase Unwired Platform. Check the *Sybase Unwired Platform Release Bulletin* for additional information and corrections. Audience: IT installation team, training team, system administrators involved in planning, and any user installing the system. Use: during the planning and installation phase.
Sybase Unwired Platform Release Bulletin	Provides information about known issues, and updates. The document is updated periodically. Audience: IT installation team, training team, system administrators involved in planning, and any user who needs up-to-date information. Use: during the planning and installation phase, and throughout the product life cycle.
New Features	Describes new or updated features. Audience: all users. Use: any time to learn what is available.
Fundamentals	Describes basic mobility concepts and how Sybase Unwired Platform enables you design mobility solutions. Audience: all users. Use: during the planning and installation phase, or any time for reference.

Document	Description
System Administration	Describes how to plan, configure, manage, and monitor Sybase Unwired Platform. Use with the *Sybase Control Center for Sybase Unwired Platform* online documentation.
	Audience: installation team, test team, system administrators responsible for managing and monitoring Sybase Unwired Platform, and for provisioning device clients.
	Use: during the installation phase, implementation phase, and for ongoing operation, maintenance, and administration of Sybase Unwired Platform.
Sybase Control Center for Sybase Unwired Platform	Describes how to use the Sybase Control Center administration console to configure, manage and monitor Sybase Unwired Platform. The online documentation is available when you launch the console (**Start > Sybase > Sybase Control Center**, and select the question mark symbol in the top right quadrant of the screen).
	Audience: system administrators responsible for managing and monitoring Sybase Unwired Platform, and system administrators responsible for provisioning device clients.
	Use: for ongoing operation, administration, and maintenance of the system.
Troubleshooting	Provides information for troubleshooting, solving, or reporting problems.
	Audience: IT staff responsible for keeping Sybase Unwired Platform running, developers, and system administrators.
	Use: during installation and implementation, development and deployment, and ongoing maintenance.

Document	Description
Getting started tutorials	Tutorials for trying out basic development functionality. Audience: new developers, or any interested user. Use: after installation. - Learn mobile business object (MBO) basics, and create a mobile device application: *Tutorial: Mobile Business Object Development* *Tutorial: BlackBerry Application Development using Device Application Designer* *Tutorial: Windows Mobile Device Application Development using Device Application Designer* - Create native mobile device applications: *Tutorial: BlackBerry Application Development using Custom Development* *Tutorial: iPhone Application Development using Custom Development* *Tutorial: Windows Mobile Application Development using Custom Development* - Create a mobile workflow package: *Tutorial: Mobile Workflow Package Development*
Sybase Unwired WorkSpace – Mobile Business Object Development	Online help for developing MBOs. Audience: new and experienced developers. Use: after system installation.
Sybase Unwired WorkSpace – Device Application Development	Online help for developing device applications. Audience: new and experienced developers. Use: after system installation.

Document	Description
Developer references for device application customization	Information for client-side custom coding using the Client Object API.
	Audience: experienced developers.
	Use: to custom code client-side applications.
	• *Developer Reference for BlackBerry* • *Developer Reference for iPhone* • *Developer Reference for Mobile Workflow Packages* • *Developer Reference for Windows and Windows Mobile*
Developer reference for Unwired Server side customization – *Reference: Custom Development for Unwired Server*	Information for custom coding using the Server API.
	Audience: experienced developers.
	Use: to customize and automate server-side implementations for device applications, and administration, such as data handling.
	Dependencies: Use with *Fundamentals* and *Sybase Unwired WorkSpace – Mobile Business Object Development*.
Developer reference for system administration customization – *Reference: Administration APIs*	Information for custom coding using administration APIs.
	Audience: experienced developers.
	Use: to customize and automate administration at a coding level.
	Dependencies: Use with *Fundamentals* and *System Administration*.

Introduction to Developing Device Applications with Sybase Unwired Platform

A device application includes both business logic (the data itself and associated metadata that defines data flow and availability), and device-resident presentation and logic.

Within Sybase Unwired Platform, development tools enable both aspects of mobile application development:

- The data aspects of the mobile application are called mobile business objects (MBO), and "MBO development" refers to defining object data models with back-end enterprise information system (EIS) connections, attributes, operations, and relationships that allow

segmented data sets to be synchronized to the device. Applications can reference one or more MBOs and can include synchronization keys, load parameters, personalization, and error handling.

- Once you have developed MBOs and deployed them to Unwired Server, develop device-resident presentation and logic for your device application by using either:
  - The Device Application Designer – an Unwired Platform graphical design and development tool.
  - A native IDE – an API approach that uses your native IDE's Client Object API and Device Framework API. Unwired WorkSpace provides MBO code generation options targeted for specific development environments, for example, BlackBerry JDE for BlackBerry device applications, or Visual Studio for Windows Mobile device applications.
  - A combination of both development environments – begin device development with the Device Application Designer, and further customize with an IDE.

This guide provides reference material for the last two options.

The Client Object API uses the data persistence library to access and store object data in the database on the device. Code generation takes place in Unwired WorkSpace. You can generate code manually, or by using scripts. The code generation engine applies the correct templates based on options and the MBO model, and outputs client objects.

Note: See *Sybase Unwired WorkSpace – Mobile Business Object Development* for procedures and information about creating and deploying MBOs. See *Sybase Unwired WorkSpace - Device Application Development* for information about device application features and appearance.

Development Task Flows

This section describes the overall development task flows, and provides information and procedures for setting up the development environment, and developing device applications.

This diagram illustrates the two approaches by which you can develop a device application: directly from mobile business objects (MBOs), using the custom UI approach, or by extending Device Application Designer code by using the Designer Generated UI approach.
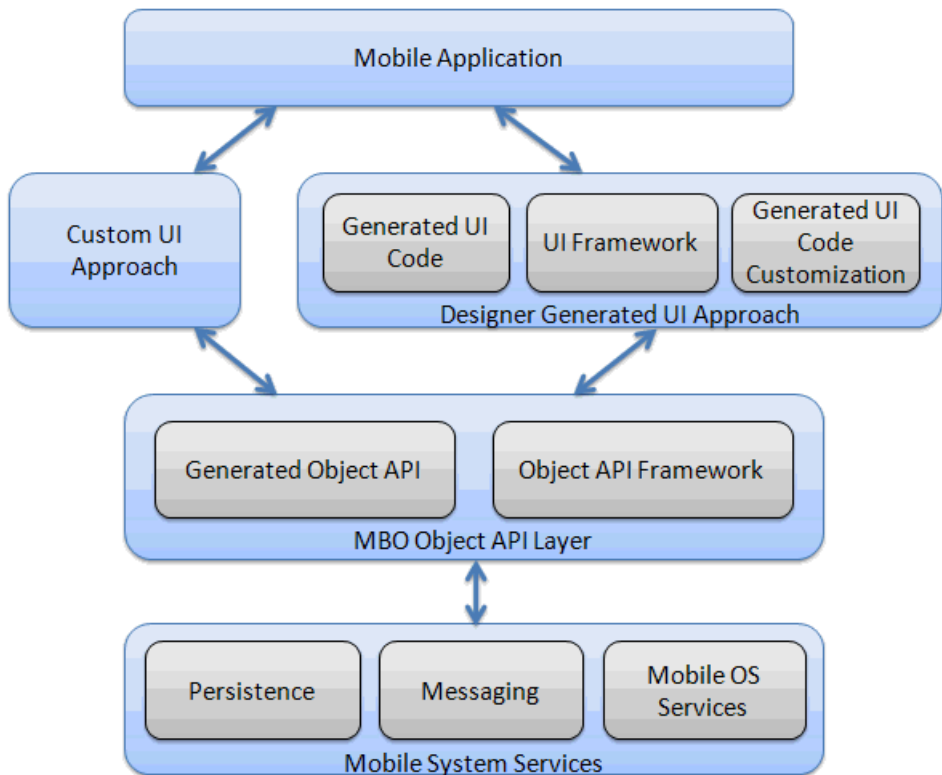
Task Flow for C# Development

This describes a typical task flow for creating a device application using Visual Studio and C#.

Highlevel steps:

1. *Configuring Your Windows Mobile Environment* on page 8.

  2. *Generating Windows or Windows Mobile Application Project Code* on page 12.
  3. *Developing a Windows Mobile Device Application Using Visual Studio* on page 25.
  4. Deploying applications:
   a. *Deploying replication-based applications* on page 34.
   b. *Deploying message-based applications* on page 34.

Task Flow for Device Application Designer and C# Development

This describes a typical task flow for creating a device application using the Device Application Designer with Visual Studio and C#.

Highlevel steps:

1. *Configuring Your Windows Mobile Environment* on page 8.
2. *Generating Windows Mobile Device Application Code from the Device Application Designer.* on page 16.
3. *Developing a Windows Mobile Device Application Using Visual Studio* on page 25.
4. Deploying applications:
 • *Deploying Replication-Based Applications* on page 34.
 • *Deploying Message-Based Applications to an Emulator or Device* on page 34.

Configuring Your Windows Mobile Environment

This section describes how to set up your Visual Studio development environment, and provides the location of required DLL files and client object APIs.

Configuring Connection Settings for the Synchronization Software

Install and configure Microsoft ActiveSync so you can deploy and run device applications on an emulator. If you install Visual Studio 2008, the Windows Mobile Device Emulators (Windows Mobile 5) and Device Emulator Manager are already installed.

Note: Microsoft ActiveSync is for Windows XP. If you are using Windows Vista or Windows 2008, you must install Virtual PC 2007 SP1 and Windows Mobile Device Center to manage synchronization settings. You can download the Windows Mobile Device Center from *http://www.microsoft.com/windowsmobile/en-us/downloads/microsoft/device-center-download.mspx*.

1. Install both the Windows Mobile 6 Standard SDK and the Windows Mobile 6 Professional SDK. You can download them from *http://www.microsoft.com/downloads/details.aspx?*

familyid=06111A3A-
A651-4745-88EF-3D48091A390B&displaylang=en#AdditionalInfo.

2. Download Microsoft ActiveSync from the *http://www.microsoft.com/windowsmobile/ en-us/help/synchronize/device-synch.mspx*. Save it to your local machine. Windows XP requires version 4.5.

3. In Windows Explorer, double-click **setup.msi** to run the ActiveSync installer.

4. Follow the steps in the ActiveSync installer to complete the installation.

5. When installation is complete, restart your machine.

   ActiveSync starts automatically, and its icon appears in the Windows toolbar.

6. Double-click the **ActiveSync** icon.

7. Select **File > Connection Settings**.

8. In the Connection Settings dialog, select all the check boxes.

9. Under "Allow connections to one of the following", select **DMA**.

10. Under "This computer is connected to", select **Work Network**.
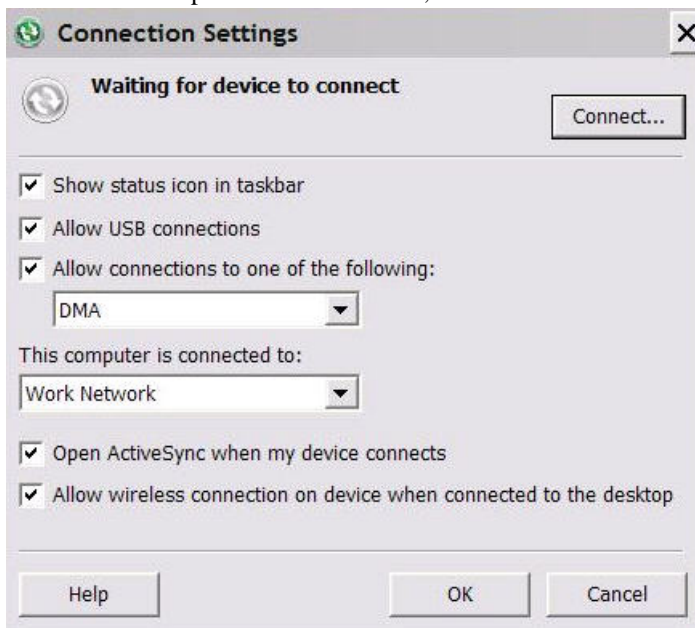
11. Click **OK**.

Configuring Windows Mobile Device Center

Before using the Windows Mobile Device Emulator, you need to change the settings of Windows Mobile Device Center.

1. Open Windows Mobile Device Center.

2. Click **Mobile Device Settings**.
3. Click **Connection Settings**.
4. Click on the **Allow connections to one of the following** checkbox.
5. Select **DMA** in the combobox.
6. On the **This computer is connected to** combobox, select **The Internet** if you want to allow the Windows Mobile device to access the Internet using Pocket IE.
7. Star the Windows Mobile Device Emulator.

Enabling Network Access from the Windows Mobile Device Emulator

When the Windows Mobile Device Emulator is started, you don't have network access by default on the device, so you must enable it.

You can start the Windows Mobile Device Emulator from Visual Studio or from the Device Emulator Manager.

1. To start the Emulator from Visual Studio 2008:
   a) Select **Tools > Device Emulator Manager**.
2. If a Device Emulator is not yet connected:
   a) Select a Device Emulator from the list and select **Connect**.
3. If you are using this Device Emulator for the first time:
   a) In the Emulator, select **File > Configure**.
   b) Click the **Network** tab.
   c) Check the **Enable NE2000 PCMCIA network adapter and bind to** checkbox.
   d) Select **Connected network card** from the list.
4. On the Emulator, configure the connection settings:
   a) In the Emulator, select **Start > Settings**.
   b) Select the **Connections** tab.
   c) Click **Connections**.
   d) Select the **Advanced** tab.
   e) Click on **Select Networks**.
   f) In the Settings window, select **My Work Network** in the first combobox.
   g) Select **File > Save State** and **Exit**.
   h) Restart the Emulator.
5. Right-click the current Emulator in Device Emulator Manager and select **Cradle**.

   ActiveSync starts. Once the connection is established, you should be able to access your PC and the Web from the Device Emulator.

Installing Required Components

During Sybase Unwired Platform installation, select **Windows Mobile .NET Components** to install the required files that allow you to customize the generated C# API object code.

Files include:

- Visual Studio Sybase Mobile Application project template.
- Online help for Windows Mobile Client Object API and Windows Mobile Framework.
- Toolbox registration for Windows Mobile controls.

Client API Dependencies

The client API assembly DLL dependencies are installed under the <UnwiredPlatform_InstallDir>\Servers\UnwiredServer\ClientAPI directory.

The contents of the Client API directory are:

- **Ultralite** – .NET Data Persistence Library and client database (UltraLite®) assemblies. This is used for replication-based synchronization client applications on Windows Mobile or Win32.
- **UltraliteJ** – Client assemblies for UltraliteJ.
- **dotnet** – Binaries of the framework classes for .NET.

  - ce: files for use on Windows CE based systems such as Windows Mobile 5+.
  - win32: files for use on full Windows based systems like Windows XP.
- **java** – The framework classes that are used by the generated classes (Java ME, J2se and RIM).
- **MoMessaging** – Files for installing client mobile messaging for message-based synchronization client applications.
- **SQLite** – Client assemblies for SQLite. These are used for message-based synchronization client applications.
- **ServerSync** – Used in replication-based synchronization applications for push notification synchronization support.
- **DeviceID** – Used for replication-based synchronization applications.

The .NET assemblies listed above support Compact Framework 2.0+ on Visual Studio 2008. These project types are supported:

- Full .NET Framework 2.0+ Application
- Windows CE .NET CF 2.0+ Application
- Pocket PC .NET CF 2.0+ Application
- Smartphone .NET CF 2.0+ Application

Mobile Business Object Code or Device Application Designer Code

Determine whether to develop a device application directly from mobile business object (MBO) generated code, or from Device Application Designer generated code, then generate the code according to your decision.

Note: Do not modify generated MBO API or Device Application Designer generated code directly. For Device Application Designer Code, use the customization pattern documented in this guide by either adding event handlers or customization classes. For MBO generated code, create a layer on top of the MBOs using patterns native to the mobile operating system development to extend and add functionality.

To avoid errors or inconsistent behavior, client applications must be regenerated whenever a mobile application package has been redeployed.

Generating Windows or Windows Mobile Application Project Code

After developing the mobile business objects (MBOs), generate the .cs files that implement the business logic and are required for Windows and Windows Mobile development.

Prerequisites
You must be connected to Unwired Server and the server-side (run-time) enterprise information system (EIS) data sources referenced by the MBOs in the deployed project before you generate object API code.

Task

1. From Unwired WorkSpace, right-click in the Mobile Application Diagram of the project for which you are generating code and select **Generate Code**.
2. (Optional) Enter the information for these options:

   Note: This page of the code generation wizard is seen only if you are using the Advanced developer profile.

Option	Description

Select code genera-tion configuration	Select either an existing configuration that contains code generation set-tings, or generate device client code without using a configuration:
	• Continue without a configuration – select this option to generate device code without using a configuration.
	• Select an existing configuration – select this option to either select an existing configuration from which you generate device client code, or create a new configuration. Selecting this option enables:
	    • Select code generation configuration – lists any existing configu-rations, from which you can select and use for this session. You can also delete any and all existing saved configurations.
	    • Create new configuration – enter the **Name** of the new configura-tion and click **Create** to save the configuration for future sessions. Select an existing configuration as a starting point for this session and click **Clone** to modify the configuration.

3. Click **Next**.

4. Enter the information for these configuration options, then click **Next**.

Option	Description
Language	Select **C#**.
Platform	Select the platform (target device) from the drop-down list for which the device client code is intended. • NET Framework for Windows • NET Compact Framework 2.0 for Win-dows Mobile • NET Compact Framework 3.5 for Win-dows Mobile
Unwired Server	Specify a default Unwired Server connection profile to which the generated code connects at runtime.
Server domain	Choose the domain to which the generated code will connect. If you specified an Unwired Serv-er to which you previously connected success-fully, the first domain in the list is chosen by default. You can enter a different domain man-ually.

Option	Description
Page size	Optionally, select the page size for the generated client code. If the page size is not set, the default page size is 16k at runtime. The page size should be larger than the sum of maximum length of all the attributes of the MBO. The page size must be valid for the database. **Note:** The page size option is not enabled for message-based applications.
Namespace	Enter a namespace for C#. **Note:** The namespace name should follow naming conventions for C#. Do not use ".com" in the namespace.
Destination	Specify the destination of the generated device client files. Enter (or **Browse**) to either a **Project path** (Mobile Application project) location or **File system path** location. Select **Clean up destination before code generation** to clean up the destination folder before generating the device client files.
Replication-based	Select to use replication-based synchronization.
Message-based	Select to use message-based synchronization.
Backward compatible	Select so the generated code is compatible with the SUP 1.2 release.

5. In Select Mobile Objects, select all the MBOs in the mobile application project or select MBOs under a specific domain, whose references, metadata, and dependencies (referenced MBOs) are included in the generated device code.
Dependent MBOs are automatically added (or removed) from the Dependencies section depending on your selections.

Note: .

6. (Optional) Select **Generate metadata classes** to generate metadata for the attributes and operations of each generated client object.

7. (Optional) Select **Generate metadata and object manager classes** to generate both the metadata for the attributes and operations of each generated client object and an object manager for the generated metadata.

The object manager allows you to retrieve the metadata of packages, MBOs, attributes, operations, and parameters during runtime using the name instead of the object instance.

8. Click **Finish** when done.

The class files include all methods required to create connections, synchronize deployed MBOs with the device, query objects, and so on, as defined in your MBOs.

By default, the MBO source code is generated in the project's Generated Code folder. The generated .cs files are located in the <MBO_project_name> folder under the src folder. In this example, code was generated for the Customer MBO:
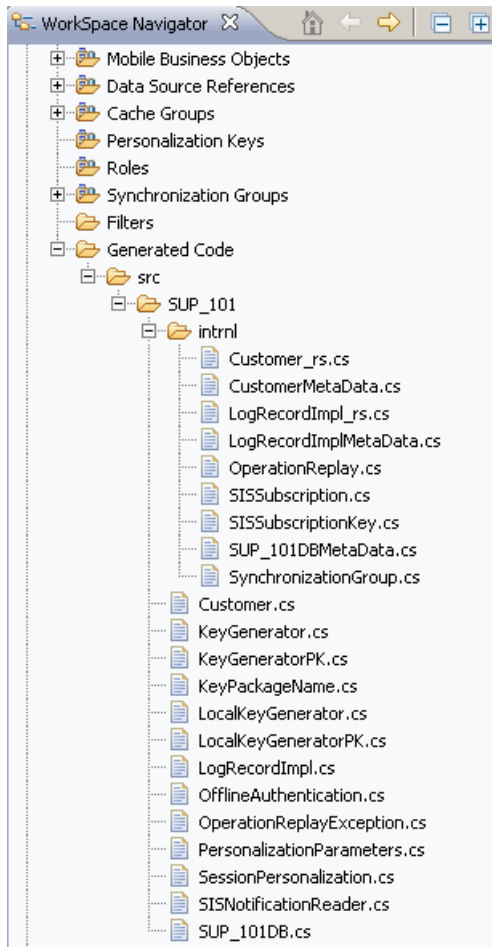
The frequently used files in this project, which you can view by double-clicking the file, include:

Table 2. Source Code File Descriptions

.cs File	Description
MBO class (for example, `Customer.cs`)	Includes all the attributes, operations, object queries, and so on, defined in the MBO.
Metadata class (for example, `Customer-MetaData.cs`)	Includes attribute and operation metadata.
Synchronization parameter class (for example, `CustomerSynchronization-Parameters.cs`)	Includes any synchronization parameters defined in this MBO.
Key generator classes (for example, `Key-Generator.cs`)	Includes surrogate key generator used to identify and track MBO instances and data.
Personalization parameter classes (for example, `PersonalizationParame-ters.cs`)	Includes any defined personalization keys.
Connection and synchronization classes (for example, `SUP_101DB.cs`)	Includes the Unwired Server connection information and synchronization methods

Generating Windows Mobile Device Application Code from the Device Application Designer

After developing the mobile business objects (MBOs), begin device application development using the Device Application Designer, then use the Generate Device Application wizard to generate the device application code required for further development in Visual Studio.

Use this procedure if you are developing Windows Mobile device applications using both the Device Application Designer and Visual Studio.

1. From Unwired WorkSpace, select **File > New > Device Application Designer**.
2. Follow the Device Application Designer wizard instructions to create a Device Application Designer project based on the developed mobile business objects (MBOs) appropriate for the type of Windows Mobile device application you are developing and click **Finish**:
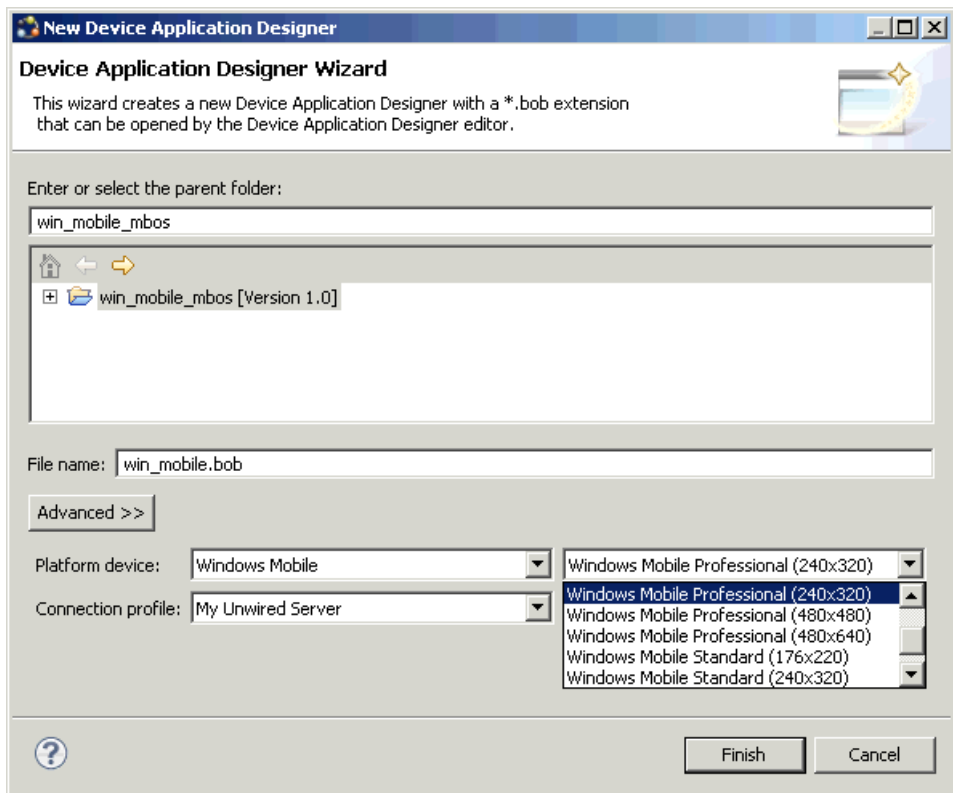
3. Develop as much of the device application as you want using the Device Application Designer.

   See the *Device Application Designer* documentation.

4. Generate the code for a Windows Mobile Device application, then extend and debug the code in Visual Studio.

Generating Code For a Windows Mobile Device Application
Use the Generate Device Application wizard to generate code for a Windows Mobile device application.

1. Click the Verify icon on the toolbar to verify the device application has no errors.
2. Click the code generation icon on the toolbar.
3. In the Generate Device Application wizard, select **Windows Mobile** and, optionally, select:

Option	Description
Locale	Expand this section to see a list of available locales from which you can select.
Advanced	Expand this section for advanced options: • Check mobile business object on Sybase Unwired Platform Server – select to verify that the mobile business objects that are used in the device application exist on the corresponding Unwired Server. • Mobile Business Object Group – the mobile business object group that contains the mobile business objects you want to verify. To generate code for the Mobile Application Project, click **Generate Code**.Use the wizard to generate the metadata classes for the selected mobile business objects.

4. Click **Next**.
5. Enter the information for the device application code generation options:

Option	Description
Favorite Configurations	Select a configuration.
Code Generation	• Target device – select the device. • Library version – choose the Microsoft .NET version of the library used to compile the generated code. **Note:** Microsoft .NET Compact Framework 2.0 requires Service Pack 2 (SP2).

Option	Description
Code Generation	• Visual Studio solutions folder – accept the default or click **Browse** to enter the location for the Visual Studio Solutions folder. **Note:** A .cab file is generated and placed in the Visual Studio solutions folder. • Solution name – enter the name of the Visual Studio solution. • Delete solution folder prior to generation – remove existing source folders before re-generating the Visual Studio solution. • Generate replication based application – select if the application has replication based synchronization. • Generate messaging based application – select if the application has message based synchronization.

Option	Description
Advanced	When the device application code is generated, two projects are created—the client project, which contains the user interface screens, and the mobile application project, which contains the mobile business objects that are used to access and update the data.
	• Client project name – enter the name of the project that contains the user interface.
	• Client project namespace – enter the namespace to use for the generated UI classes.
	• Client project assembly name – the name of the generated `.exe` file for the project. This is the name that appears on the mobile device.
	• Mobile application project name – the name of the Mobile Application Project that contains the mobile business objects used in the device application.
	• Mobile application project assembly name – accept the default or enter the name for the .dll of the mobile application project.
	• Client project icon – click **Browse** to select an icon with which to associate the generated `.exe` file. This is the icon that appears on the mobile device.
	• Generate source codes for Sybase Windows frameworks – select to generate sources for Sybase Unwired Platform Windows frameworks, including stock screens and actions.
	• Deploy to an ActiveSync connected device or emulator – select this option to deploy the generated code to a Windows Mobile device or emulator. ActiveSync enables the transferring and installation of the application on the mobile device.
	Note: ActiveSync is for Windows XP. Windows Vista, Windows 7, and Windows 2008 uses Windows Mobile Device Center.
	• Create a shortcut in Programs – select to create a shortcut in the Programs folder on the device.
	• Shortcut name – enter a name for the shortcut.

Option	Description
	• Deployment timeout (minutes) – the maximum time to wait for deployment to the device. • Perform silent install – use this option only when you are deploying to the emulator (not the device itself). This enables deployment to proceed with no user input.

6. (Optional) Click **Start Device Emulator Manager** if the Device Emulator Manager is not already running and you want to deploy the application to an emulator.

7. Click **Finish**.

Device Application Designer Generated Solution Files and Projects

The Device Application Designer generated code consists of a solution file and several projects, including the UI project, MBO project, Settings project, and settingProxy project.

Table 3. Solution file and projects

Name	Source Model	Description
Solution File	Solution name from the generation option.	Visual Studio solution file to manage all the generated projects. For example, Customer.sln.
Mobile Business Object (MBO) project	The MBO domain project is generated from the MBO domain model definition. The project name is defined in the project generation options.	This project contains the generated Client API code (for example, Customer_MBO.csproj).
Windows Mobile UI project	The Windows Mobile application project is generated from the Device Application Designer definition. The Project name and namespace are defined in the Windows Mobile application generation **Client project name** option.	This project contains all the UI-related application code. The name depends on the mobile business object project (for example, Customer_UI.csproj).
Sybase.UnwiredPlatform.Windows.StockScreen project	The Windows Mobile Device Framework, including UI framework, and a framework that approximates the Device Application Designer.	This framework can be generated as source or as a .dll, and is generated only if the **Generate source codes for Sybase windows frameworks** option is selected.

Name	Source Model	Description
Settings Project	The setting screen definitions in the Device Application Designer.	This project is generated when there are settings screens in the Device Application Designer, and if you use one of the Settings stock screens in DAD. The generated project provides a Settings application that allows users to configure various setting options of the application.
SettingProxy project	An addin to the setting screen definitions in the Device Application Designer.	This project is generated when there are settings screens in the Device Application Designer, and if you use one of the Settings stock screens in DAD. The addin allows users to invoke the Settings application directly from the Windows Mobile Settings folder.

Generation Gap Pattern Support
To allow developers to extend generated code and keep modifications, Sybase supports a generation gap pattern. This pattern makes use of a C# partial classes to provide customization. For certain logic-related classes, a "protected virtual" method lets you override the method with your own implementation.

An application consists of both generated code and customized code.
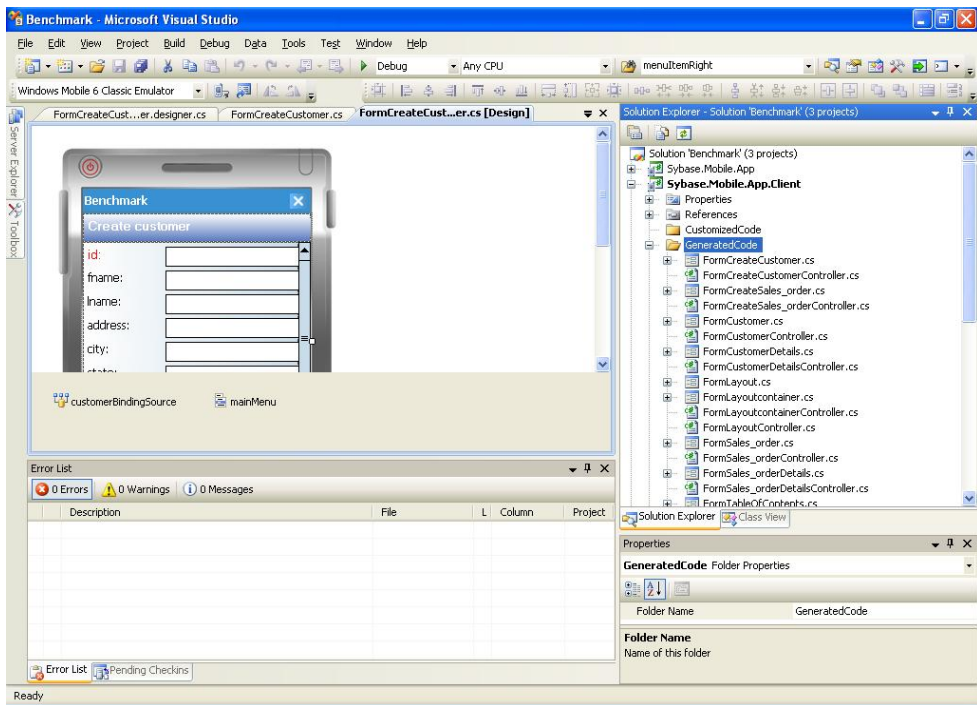
Following is an example of the `FormCreateCustomerController` class:

```
/// <summary>
/// The Base class of FormCreateCustomerController
/// </summary>
internal abstract class FormCreateCustomerControllerBase :
  ControllerBase
{
  public FormCreateCustomerControllerBase(IFormPart form)
    : base(form)
  {
  }
  // button (Submit) click event handler
  internal virtual void
    SubmitButton_Handler(FormsManagerDataObject dataObject)
  {
    ....
  }
}

/// <summary>
/// The Controller class of Form FormCreateCustomer
/// </summary>
internal partial class FormCreateCustomerController :
  FormCreateCustomerControllerBase
{
  public FormCreateCustomerController(IFormPart form)
```

```
  : base(form)
  {
  }
}
```

You may want to customize the business logic. For example, if you want to customize the **Submit** action, you could do the following:

- Create a `MyFormCreateCustomerController.cs` class in the CustomizedCode folder
- Change the default code to:

```
internal partial class FormCreateCustomerController
{
  internal override void
SubmitButton_Handler(Sybase.UnwiredPlatform.Windows.Forms.FormsMana
gerDataObject    dataObject)
  {
    // Add your custom actions here
    MessageBox.Show("Before Submit!");

    // Perform the default action
    base.SubmitButton_Handler(dataObject);
  }
}
```

Windows Mobile UI Project
The Windows Mobile UI project contains these classes and files:

Table 4. Classes and Files in the Windows Mobile UI Project

Class	Description
Program.cs	The main entry of the application.
ApplicationInit.cs	Code to initialize the application.
GlobalResource	All the localized text used by stock screen.
Generated Code\Form*.cs	The Generated Form class extends FormBase. It is generated from the Device Application Designer screen; the UI definition is generated in this screen.
Generated Code\Form*Controller.cs	The Controller class contains the data access and event handler code of the forms.
Generated Code\DataStore*DataStore.cs	The DataStore class wraps the access of the MBO object and keeps the context of the MBO.
Properties\DataSources*.datasource	The data sources for data binding.

Developing a Windows Mobile Device Application Using Visual Studio

After you import Unwired WorkSpace projects (mobile application or Device Application Designer) and associated libraries into the development environment, use the Client Object API, Windows Device Framework, and native APIs to create or customize your device applications.

Note: Do not modify generated MBO API or Device Application Designer generated code directly. For Device Application Designer Code, use the customization pattern documented in this guide by either adding event handlers or customization classes. For MBO generated code, create a layer on top of the MBOs using patterns native to the mobile operating system development to extend and add functionality.

Project Setup

You can create Visual Studio projects in three ways.

- Create the project from Visual Studio using the **SybaseMobileApplication** template. When creating message-based synchronization client projects **SybaseMobileApplication** template, there is some additional configuration to be performed.
- Create the project from Visual Studio (without using the **SybaseMobileApplication** template).
- Create the project from a Sybase Unwired Platform Device Application Designer generated project.

Creating a Mobile Application Project

This describes how to set up a project in Visual Studio without using the **SybaseMobileApplication** template. You must add the required libraries as references in the Visual Studio project. The libraries needed depend on client application platform and synchronization method (replication-based or message-based).

You can use this method to create replication-based and message-based sychronization client projects.

1. Add the following libraries as references in the Visual Studio project:

   Replication-based synchronization:
   - sup-client.dll – from <UnwiredPlatform_InstallDir>\Servers \UnwiredServer\ClientAPI\dotnet\<platform>.

- `iAnywhere.Data.UltraLite.dll` – from
  `<UnwiredPlatform_InstallDir>\Servers\UnwiredServer`
  `\ClientAPI\Ultralite\<platform>\Assembly\V2`.
- `iAnywhere.Data.UltraLite.resources.dll` (several languages are
  supported) – from `<UnwiredPlatform_InstallDir>\Servers`
  `\UnwiredServer\ClientAPI\Ultralite\<platform>\Assembly`
  `\V2\<language>`.

Message-based synchronization:
- `sup-client.dll` – from `<UnwiredPlatform_InstallDir>\Servers`
  `\UnwiredServer\ClientAPI\dotnet\<platform>`.
- For Windows 64-bit, `System.Data.SQLite.DLL` – from
  `<UnwiredPlatform_InstallDir>\Servers\UnwiredServer`
  `\ClientAPI\SQLite\x64`.
- For Windows 32-bit, `System.Data.SQLite.DLL` – from
  `<UnwiredPlatform_InstallDir>\Servers\UnwiredServer`
  `\ClientAPI\SQLite`

2. For replication-based synchronization, add the following libraries as items in the Visual
   Studio project. Set "Build Action" to "Content" and "Copy to Output Directory" to **Copy
   always**.
   - `ulnet11.dll` – from `<UnwiredPlatform_InstallDir>\Servers`
     `\UnwiredServer\ClientAPI\Ultralite\<platform>`.
   - `mlcrsa11.dll` (if HTTPS protocol is used) – from
     `<UnwiredPlatform_InstallDir>\Servers\UnwiredServer`
     `\ClientAPI\Ultralite\<platform>`.
   - `PUtilTRU.dll` - from `<UnwiredPlatform_InstallDir>\Servers`
     `\UnwiredServer\ClientAPI\DeviceID\<platform>`.
3. (Optional) This step is required only for Pocket PC and Smartphone clients. For message-
   based synchronization, add the following librariers as items in the Visual Studio project
   and set "Build Action" to "Content" and "Copy to Output Directory" to **Copy always**.
   - `SQLite.Interop.065.DLL` – from `<UnwiredPlatform_InstallDir>`
     `\UnwiredPlatform\Servers\UnwiredServer\ClientAPI\SQLite`
     `\CompactFramework`.

Creating a Replication-based Mobile Application Project Using the SybaseMobileApplication Template

Create a replication-based mobile application project using the SybaseMobileApplication
template which allows you to write C# device applications.

Prerequisites

To install the template, select **Windows Mobile .NET Components** during the Sybase
Unwired Platform installation process.

Note: This procedure is for replication-based synchronisation client projects.

For message-based applications, you need to remove the references to the Ultralite DLLs, and add references to the `System.Data.SQLite.dll` (for Windows and Windows Mobile) and `SQLite.Interop.065.DLL` (for Windows Mobile only).

Task

1. From the Visual Studio main menu, select **File > New > Project**.

   Select the project template, and include the **Name**, **Location**, and **Solution name**. To use the Sybase Mobile Application template, select **Visual C# > SybaseMobileApplication** as the project type. Click **OK**.

2. Enter:
   - Target platform – WM5, WM6, Win32, and so on.
   - Library version – .NET version 2.0, 3.5, and so on.
   - Language – the language used in the resource DLLs, to be included in the generated project.

   Based on your selections, different sets of DLLs are included in the project. The project contains all assemblies and runtime support DLLs required to access the Object API.

3. (Optional) Select **Include SUP Windows Mobile Controls** to reference the dependent Sybase Windows Mobile Component Library Assemblies in the generated project.

   This allows you to use Sybase Unwired Platform UI controls.

4. Click **OK** to create the project.

   Inspect the contents of the project from the Solution Explorer by expanding the various folders within the project.

5. Right-click on the project, and select select **Add > Existing Item**.

6. Browse to the folder `<UnwiredPlatform_InstallDir>\UnwiredPlatform \Servers\UnwiredServer\ClientAPI\DeviceID\WinCE\<platform>` in the created project, select `PUtilTRU.dll`, then select **Add > Add as Link**.

7. In its Properties view, set **Build Action** to **Content** and **Copy to Output Directory** to **Copy always**.

8. Follow the same steps to add the `ulnet11.dll`, which is located in `<UnwiredPlatform_InstallDir>\UnwiredPlatform\Servers \UnwiredServer\ClientAPI\UltraLite\<platform>` as a link.

Once you have generated and built a project, you can write custom mobile applications based on your selections.

Creating a Message-based Mobile Application Project Using the SybaseMobileApplication Template

Create a message-based mobile application project in Visual Studio using the SybaseMobileApplication template, which allows you to write C# device applications.

Prerequisites

To install the template, select **Windows Mobile .NET Components** during the Sybase Unwired Platform installation process.

Note: This procedure is for message-based synchronisation client projects.

For message-based applications, you need to remove the references to the Ultralite DLLs, and add references to the `System.Data.SQLite.dll` (for Windows and Windows Mobile) and `SQLite.Interop.065.DLL` (for Windows Mobile only).

Task

1. From the Visual Studio main menu select **File > New > Project**.
2. In the Visual C# project type, select **Visual C# > SybaseMobileApplication** and include the **Name**, **Location**, and **Solution name**, then click **OK**.
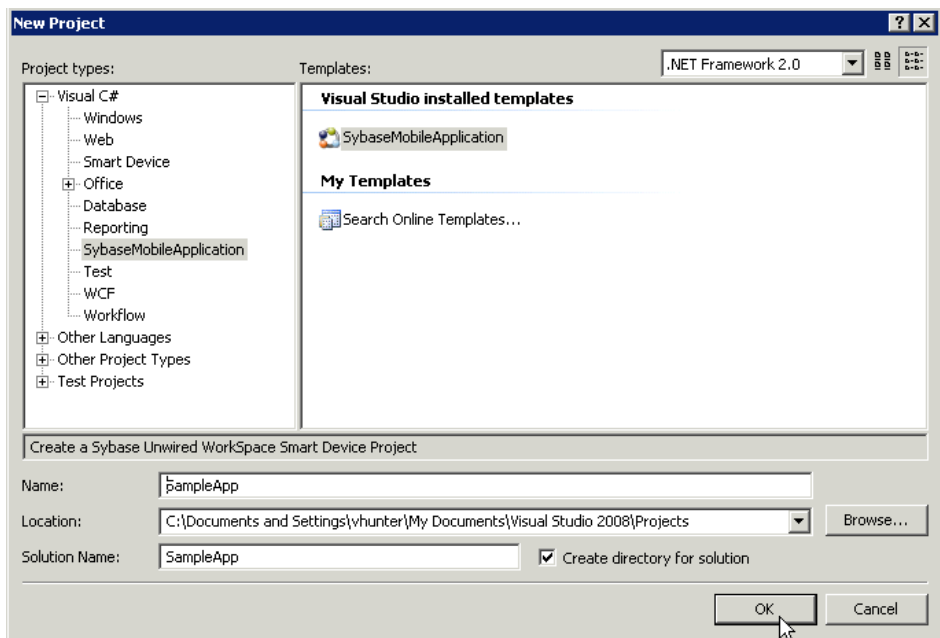
3. Enter:
   - Target platform – WM5, WM6, Win32, and so on.

- Library version – .NET version 2.0, 3.5, and so on.
- Language – the language used in the resource DLLs, to be included in the generated project.

Based on your selections, different sets of DLLs are included in the project. The project contains all assemblies and runtime support DLLs required to access the Object API.

4. (Optional) Select **Include SUP Windows Mobile Controls** to reference the dependent Sybase Windows Mobile Component Library Assemblies in the generated project.

This allows you to use Sybase Unwired Platform UI controls.

5. Click **OK** to generate the Visual Studio Project with the dependent Sybase Unwired Platform .NET assemblies.

6. Remove references to the `iAnywhere.Data.UltraLite.dll` and `iAnywhere.Data.UltraLite.resources.dll` files.

7. Add references to the `System.Data.SQLite.dll`.

   a) In the project, right-click the **References** folder and select **Add Reference**.

   b) Browse to the `<UnwiredPlatform_InstallDir>\UnwiredPlatform \Servers\UnwiredServer\ClientAPI\SQLite\CompactFramework \System.Data.SQLite.dll` file, select it, and click **OK**.

8. Add the following librariers as items in the Visual Studio project and set "Build Action" to "Content" and "Copy to Output Directory" to **Copy always**.

   - `SQLite.Interop.065.DLL` – from `<UnwiredPlatform_InstallDir> \UnwiredPlatform\Servers\UnwiredServer\ClientAPI\SQLite \CompactFramework`.
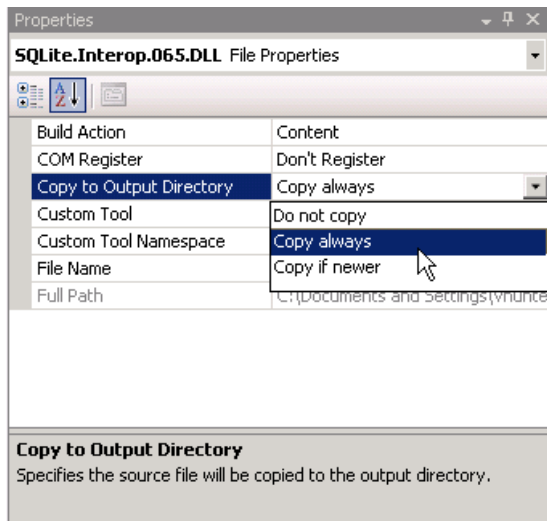
Windows Mobile Libraries

These Sybase Unwired Platform Windows Mobile Framework DLLs must be located in
`Unwired_WorkSpace\VisualStudio\ComponentLibrary`:

- `Sybase.UnwiredPlatform.Windows.dll`
- `Sybase.UnwiredPlatform.Windows.Forms.dll`
- `Sybase.UnwiredPlatform.Windows.StockScreens.dll`

Windows Mobile Development

Develop a Windows Mobile application by generating the Visual Studio 2008 projects in C#,
and running the application in the device or on a simulator to test.

1. Either generate an application using the Device Application Designer, or generate Mobile
   Business Objects (MBOs), then create a new Visual Studio project, the import generated
   MBOs, and create the user interface.
2. Add business logic to the generated code through the Windows Mobile Client Object API.
   See *Developer Reference for Windows and Windows Mobile > Reference > Client Object
   API*.
3. Change the default user interface through the Windows Mobile Device Framework. See
   *Developer Reference for Windows and Windows Mobile > Reference > Windows Mobile
   Device Framework API*.
4. Run the application in the device or on a simulator.

Integrating Help into a Project

When you generate MBOs or client applications for Windows Mobile from Unwired
WorkSpace, an XML file is generated for the MBOs. The generated Visual Studio project for
the forms can also generate a XML file. When you compile a project, an XML file is
generated. You can use these XML files to generate online help.

To generate online help for Visual Studio 2008, you can use Sandcastle and Sandcastle Help
File Builder. You can download and install Sandcastle and Sandcastle Help File Builder from
these locations:

- http://sandcastle.codeplex.com/Wikipage
- http://shfb.codeplex.com/releases/view/29710

To integrate help into your project build:

1. Add the /doc option in your project build, so that it can generate an XML file from the
   comments. You can also configure this option in the Visual Studio project properties. On
   the Build tab, select **XML documentation** and provide a file name.

2. Create a SandCastle Help File Builder project (.shfb file). Specify the assemblies and the XML file generated from the comments as input. You can also specify other help properties.

3. Use the `.shfb` project file in a script to build the document. For example:

```
<Target Name="Documentation">
    <Exec Command="$(SandCastleHelpBuilderPath) <shfb project
file>.shfb" />
</Target>
```

Debugging Windows and Windows Mobile Device Development

Device client and Unwired Server troubleshooting tools for diagnosing Microsoft® Windows and Windows Mobile development problems.

Client-side debugging

Problems on the device client side that may cause client application problems:

- Unwired Server connection failed.
- Data does not appear on the client device.
- Physical device problems, such as low battery or low memory.

To find out more information on the device client side:

- If you have implemented debugging in your generated or custom code (which Sybase recommends), turn on debugging and review the debugging information. See *Developer Reference for Windows and Windows Mobile* about using the MBOLogger class to add log levels to messages reported to the console.
- Check the log record on the device. Use the **DatabaseClass.GetLogRecord(Sybase.Persistence.Query)** or **Entity.GetLogRecords** methods.

  This is the log format

  ```
  level,code,eisCode,message,component,entityKey,operation,requestI
  d,timestamp
  ```

  Following is a log sample:

  ```
  level code eisCode message component entityKey operation requestId
  timestamp
   5,500,'','java.lang.SecurityException:Authorization failed:
  Domain = default Package = end2end.rdb:1.0 mboName =
  simpleCustomer action =
  delete','simpleCustomer','100001','delete','100014','2010-05-11
  14:45:59.710'
  ```

  - `level` – the log level currently set. Values include: 1 = TRACE, 2 = DEBUG, 3 = INFO, 4 = WARN, 5 = ERROR, 6 = FATAL, 7 = OFF.
  - `code` – Unwired Server administration codes.
    - Replication-based synchronization codes:

- 200 – success.
- 500 – failure.
- Message-based synchronization codes:
  - 401 – the client request had invalid credentials, or that authentication failed for some other reason.
  - 403 – the client request had valid credentials, but that the user does not have permission to access the requested resource [package, mobile business object (MBO) or operation].
  - 404 – the client attempted to access a non-existent package or mobile business object.
  - 405 – there is no valid license to check out for the client.
  - 500 – an unexpected (unspecified) server failure.
- `eisCode` – not currently used.
- `message` – the message content.
- `component` – Mobile Business Object (MBO) name.
- `entityKey` – MBO surrogate key, used to identify and track MBO instances and data.
- `operation` – operation name.
- `requestId` – operation replay request ID or messaging-based synchronization message request ID.
- `timestamp` – message logged time, or operation execution time.

Server-side debugging

Problems on the Unwired Server side that may cause device client problems:

- The domain or package does not exist. If you create a new domain, whose default status is disabled, it is unavailable until enabled.
- Authentication failed for the synchronizing user.
- The operation role check failed for the synchronizing user.
- Back-end authentication failed.
- An operation failed on the remote, replication database back end, for example, a table or foreign key does not exist.
- An operation failed on the Web Service, REST, or SAP® back end.

To find out more information on the Unwired Server side:

- Check the MMS server log files.
- For message-based synchronization mode, check the `SUPBridge` and `JMSBridge` logs. Modify the log level for the module `SUPBridge` and `JmsBridge` in `<server_install_folder>\UnwiredPlatform\Servers \MessagingServer\Data\TraceConfig.xml` to debug.

Setting Paging for Large Amounts of Data

If you receive a communication error while synchronizing the device, you may need to set the paging size in the Form${ScreenName}Controller.cs file.

For screens containing a paged table, add this code to the Form$ {ScreenName}Controller.cs file:

```
/// <summary>
      /// Gets the ${mbo name} in current context.
      /// </summary>
      /// <returns></returns>
      public BindingList<global::$(package name).${mbo name}> Get$
{mbo name}BindingSource()
      {
        //....
        //${mbo name}DataStore.Instance.AllObjects = ${mbo
name}DataStore.Instance.LoadData();
        // should be changed to
        ${mbo name}DataStore.Instance.AllObjects = ${mbo
name}DataStore.Instance.LoadData(this.table1_paging_size *
this.table1_paging_current_page, this.table1_paging_size);
        return new SortableBindingList<global::$(package name).$(mbo
name)>($(mbo name)DataStore.Instance.AllObjects);
      }
```

For screens that do not contain a paged table, add this code to the Form$ {ScreenName}Controller.cs file:

```
/// <summary>
      /// Gets the ${mbo name} in current context.
      /// </summary>
      /// <returns></returns>
      public BindingList<global::$(package name).${mbo name}> Get$
{mbo name}BindingSource()
      {
        //....
        //${mbo name}DataStore.Instance.AllObjects = ${mbo
name}DataStore.Instance.LoadData();  // this line should be
commented.

        return new SortableBindingList<global::$(package name).$(mbo
name)>($(mbo name)DataStore.Instance.AllObjects);
      }
```

Application Deployment to Devices

Deploy mobile applications to devices and register the devices with Unwired Server.

Deploying Replication-Based Applications

Deploy replication-based applications to a Device Emulator or connected device.

1. To deploy a replication-based application from Visual Studio, compile the project and deploy the application to the emulator or the real device.
2. If you are using a Device Emulator, define a shared folder and copy the file in that folder from your machine so the Emulator can access it.
3. Using Windows and a connected device, use the Virtual folder on your machine to copy the application's .cab file to the device or memory card.

Deploying Message-Based Applications to an Emulator or Device

The Sybase Messaging Client requires a .cab file. Pocket PCs require `SUPMessaging_Pro.cab`, and Smartphones require `SUPMessaging_Std.cab`.

Note: ActiveSync is for Windows XP. Windows Vista, Windows 7, and Windows 2008 uses Windows Mobile Device Center.

1. In Visual Studio, select **Tools > Emulator Manager** to deploy to an Emulator, or **Connect to Device** to deploy to a connected device.
2. From the list of devices, right-click the emulator to which to deploy the application and select **Connect**.
3. Right-click on the same emulator and select **Cradle**.
   Microsoft ActiveSync appears.
4. If the Microsoft ActiveSync configuration window appears, click **Cancel**.
5. In Microsoft ActiveSync, click **Explore**.
6. In the Mobile Device window, double-click **My Windows Mobile-based Device**.
   The device's file system root folder opens.
7. Navigate to `<UnwiredPlatform_InstallDir>\UnwiredPlatform \Servers\UnwiredServer\ClientAPI\MoMessaging\wm` and copy the `SUPMessaging_Pro.cab` file (for Pocket PC) or the `SUPMessaging_Std.cab` file (for Smartphone) to the device's root folder.
8. Use File Explorer on the device emulator to browse to where you placed the `.cab` file.
9. Click on `.cab` once to start the installation.

Message-based Synchronization Overview

The message-based synchronization model:

- Uses an asynchronous messaging protocol.
- Provides reliable messaging between the device and the server.

- Provides fine-grained synchronization (synchronization is provided at the data level—each process communicates only with the process it depends on).
- Best for always available mode.
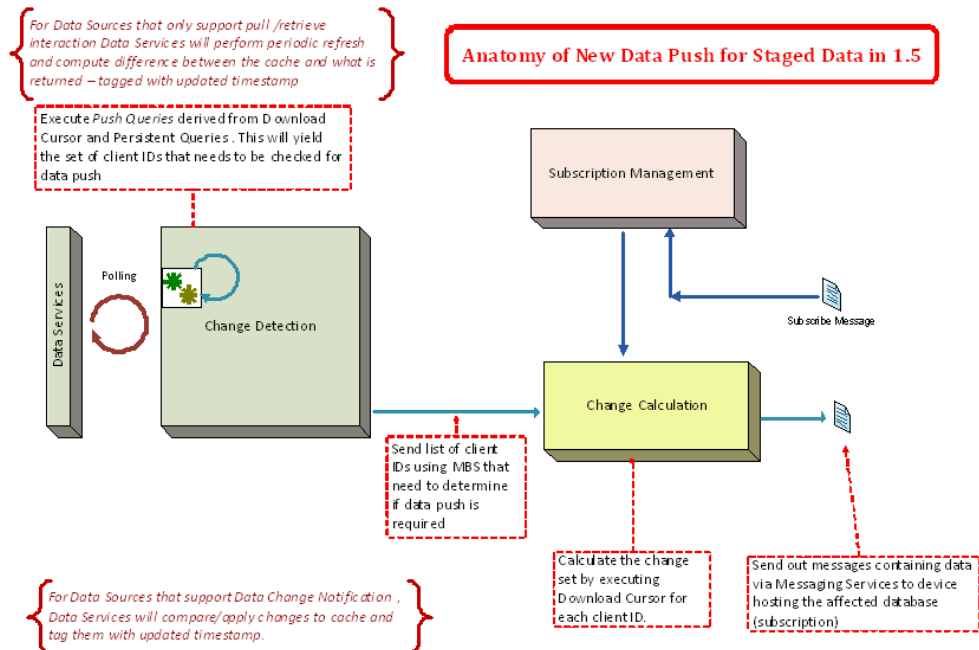
Device Registration

Messaging devices contain applications that send and receive data through messaging. An administrator must configure the device activation template properties for message-based synchronization (MBS) devices. Device activation requires user registration. Upon successful registration, the device is activated and set up with the template the administrator has selected.

Device registration pairs a user and a device once the user supplies the correct activation code. This information is stored in the messaging database, which contains extensive information about users and their corresponding mobile devices.

Users who are registered but who have not yet installed the software are listed in the window as registered, and their e-mail messages and PIM items are queued by Unwired Server for later delivery. Typically, device registration occurs when the user initially attempts to connect to Unwired Server. However, an administrator can force a user to reregister if there is data corruption on the device, or if the user is assigned a new device. This reestablishes the relationship between the user and the device, and refreshes the entire data set on the device

Note: For more information on device registration, see *Sybase Unwired Platform System Administration > Device and User Management > Messaging Devices*, *Sybase Unwired Platform System Administration > Device and User Management > Messaging Devices >*

Device Registration and Activation, and *Sybase Unwired Platform System Administration > Device and User Management > Device Provisioning*.

Device registration requires user registration from the physical device.

1. Locate the device registration program in `\Program Files\SybaseSettings`.
2. Double-click the program, then click the **Next Connection** icon.
   You see the Connection window.
3. Enter the information for your Sybase Messaging Server configuration and account:
   - **Server Name –** The Unwired Server servername.
   - **Server Port –** The port on which the Unwired Server is listening.
   - **Farm ID –** The company ID. In cases where a relay server is used, this parameter corresponds to the farm ID of the relay server.
   - **User Name –** The username registered with the Unwired Server.
   - **Activation Code –** The activation code.

Reference

This section describes the Client Object API and Device Framework API. Classes are defined and sample code is provided.

Generated API Help

Generated API help is included in the Sybase Unwired Platform installation directory.

C# documentation, which provides a complete reference to the APIs:
- Compiled help for the Device Framework API is installed to
  `<UnwiredPlatform_InstallDir>\Unwired_WorkSpace`
  `\VisualStudio\ComponentLibrary\help`.
- You can integrate help for generated code from mobile business objects (MBOs) into your Visual Studio project. See *Integrating Help into a Project*.

Windows Mobile Client Object API

Describes solutions and examples for tasks and uses of the Sybase Unwired Platform Windows Mobile Client Object API. The Client Object API enables you to customize mobile business object data flow and handling for the Windows Mobile device application.

Connection APIs

The Connection APIs contain methods for managing local database information, establishing a connection with the Unwired Server and authenticating.

ConnectionProfile
The `ConnectionProfile` class manages local database information. You can use it to set the encryption key, which you must do before creating a local database.

```
ConnectionProfile cp = SampleAppDB.GetConnectionProfile();
cp.SetEncryptionKey("Your key");
cp.Save();
```

SynchronizationProfile
Before synchronizing with Unwired Server, you must configure a client with information for establishing a connection with the Unwired Server where the mobile application has been deployed.

```
ConnectionProfile cp = SampleAppDB.GetSynchronizationProfile();
cp.DomainName = "default";
```

This example is to call relay server for replication-based synchronization:

```
ConnectionProfile cp = SampleAppDB.GetSynchronizationProfile();
    cp.ServerName = "Relay_Server";
    cp.PortNumber = 80;
  cp.NetworkStreamParams = "url_suffix=ias_relay_server/client/
rs_client.dll/Ryan.SUPFarm";
  cp.Save();
```

You can set certificate information in SynchronizationProfile.

```
ConnectionProfile profile = MyDatabase.GetSynchronizationProfile();
profile.DomainName = "default";
Reference
Developer Reference for Windows and Windows Mobile 47
profile.ServerName = "host-name";
profile.PortNumber = 2481;
profile.NetworkProtocol = "https";
profile.NetworkStreamParams =
"trusted_certificates=rsa_public_cert.crt";
```

Authentication

The generated package database class provides a default synchronization connection profile according to the Unwired Server connection profile and Server Domain selected during code generation. You can log in to the Unwired Server with your user name and credentials.

The package database class provides these methods for logging in to the Unwired Server:

`OnlineLogin` authenticates credentials against the Unwired Server.

`OfflineLogin` authenticates against the last successfully authenticated credentials. There is no communication with Unwired Server in this method.

`LoginToSync` synchronizes the KeyGenerator from the Unwired Server with the client. The KeyGenerator is an MBO for storing key values that are known to both the server and the client. On `LoginToSync` from the client, the server sends down a value that the client can use when creating new records (by using the method `KeyGenerator.generateId()` to create key values that the server will accept).

The KeyGenerator value increments each time the `GenerateId` method is called. A periodic call to `SubmitPending` by the KeyGenerator MBO sends the most recently used value to the Unwired Server, to let the Unwired Server know what keys have been used on the client side. Place this call in a try/catch block in the client application and ensure that the client application does not attempt to send more messages to the Unwired Server if `LoginToSync` throws an exception.

```
void LoginToSync(string user, string password);
```

`AsyncOnlineLogin` is available only for message-based synchronization, and it is the recommended login method for message-based synchronization. It functions similarly to `LoginToSync`, except that it sends the login request asynchronously (it returns without

waiting for a server response). Check for `OnLoginSuccess` or `OnLoginFailure` to be called in the callback handler.

```
void AsyncOnlineLogin(string user, string password);
```

Connect Using a Certificate
You can set certificate information in `ConnectionProfile`.

```
ConnectionProfile profile = MyDatabase.GetSynchronizationProfile();
profile.DomainName = "default";
profile.ServerName = "host-name";
profile.PortNumber = 2481;
profile.NetworkProtocol = "https";
profile.NetworkStreamParams =
"trusted_certificates=rsa_public_cert.crt";
```

Encrypt the Database
You can use `ConnectionProfile.EncryptionKey` to set the encryption key of a local database. Set the key during application initialization, and before creating or accessing the client database.

```
ConnectionProfile profile = MyDatabase.GetConnectionProfile();
profile.SetEncryptionKey("Your key");
```

Synchronization APIs

The client object API allows you to change synchronization parameters and perform mobile business object synchronization.

Changing Synchronization Parameters
Synchronization parameters determine the manner in which data is retrieved from the consolidated database during a synchronization session.

The primary purpose of synchronization parameters is to partition data. By changing the synchronization parameters, you affect the data you are working with, including searches, and synchronization.

```
CustomerSynchronizationParameters sp =
Customer.SynchronizationParameters;
            sp.State = "CA";
            sp.Save();
```

Performing Mobile Business Object Synchronization
To perform mobile business object (MBO) synchronization, you must save a Connection object. Additionally, you may want to set synchronization parameters.

For replication-based synchronization, this code synchronizes an MBO package using a specified connection:

```
SampleAppDB.Synchronize (string synchronizationGroup)
```

For message-based replication, before you can synchronize MBO changes with the server, you must subscribe the mobile application package deployed on server by calling `SampleAppDB.subscribe()`. This also downloads certain data to devices for those that have default values. You can use the `OnImportSuccess` method in the defined `CallbackHandler` to check if data download has been completed.

Then you can call the **SubmitPendingOperations(string synchronizationGroup)** operation through the publication as this example illustrates:

```
Product product_new = new Product();
product_new.Color="Yellow";
product_new.Description="";
product_new.Id=888;
product_new.Name = "ChildrenPants";
product_new.Prod_size = "M";
product_new.Quantity = 200;
product_new.Unit_price = (decimal)188.00;
product_new.Create();
SampleAppDB.SubmitPendingOperations("default");
while(SampleAppDB.HasPendingOperations())
{
    System.Console.Write(" . ");
    System.Threading.Thread.Sleep(1000);
}
```

You can use a publication mechanism, which allows as many as 32 simultaneous synchronizations. However, performing simultaneous synchronizations on several very large Unwired Server applications can impact server performance, and possibly affect other remote users. The following code samples demonstrate how to simultaneously synchronize multiple MBOs.

For message-based synchronization, synchronize multiple MBOs using:

```
SampleAppDB.SubmitPendingOperations();
```

Or you can use:

```
SampleAppDB.SubmitPendingOperations("my-pub");
```

where "my-pub" is the synchronization group defined.

For replication-based synchronization, synchronize multiple MBOs using:

```
SampleAppDB.Synchronize();
```

You can also use:

```
SampleAppDB.Synchronize("my-pub");
```

Query APIs

The Query APIs allow you to retrieve data from mobile business objects, to retrieve relationship data and paging data, and to retrieve and filter a query result set.

Retrieving Data from the local database

You can retrieve data from the local database through a variety of queries, including object queries, arbitrary find, and through filtering query result sets.

Object Queries

To retrieve data from a local database, use one of the static Object Query methods in the MBO class.

Object Query methods are generated based on the object queries defined by the modeler in Unwired WorkSpace. Object Query methods carry query name, parameters, and return type defined in Unwired WorkSpace. Object Query methods return either an object, or a collection of objects that match the specified search criteria.

The following examples demonstrate how to use the Object Query methods of the Customer MBO to retrieve data.

This method retrieves all customers:

```
public static Sybase.Collections.GenericList<Customer> FindAll()

Sybase.Collections.GenericList<Customer> customers =
Customer.FindAll();
```

This method retrieves all customers in a certain page:

```
public static Sybase.Collections.GenericList<Customer> FindAll(int
skip, int take)

Sybase.Collections.GenericList<Customer> customers =
Customer.FindAll(10, 5);
```

Suppose the modeler defined the following Object Query for the Customer MBO in Sybase Unwired Workspace:

- **name –** findByFirstName
- **parameter –** String firstName
- **query definition –** SELECT x.* FROM Customer x WHERE x.fname = :firstName
- **return type –** Sybase.Collections.GenericList

The preceding Object Query results in two generated methods in Customer.cs:

```
public static Sybase.Collections.GenericList<Customer>
FindByFirstName(string firstName)
```

Arbitrary Find

The arbitrary find method provides custom device applications the ability to dynamically build queries based on user input.

AttributeTest

In addition to allowing for arbitrary search criteria, the arbitrary find method lets the user specify a desired ordering of the results and object state criteria. A `Query` class is included in the client object API's core assembly sup-client.dll Sybase.Persistence namespace. The `Query` class is the single object passed to the arbitrary search methods and consists of search conditions, object/row state filter conditions, and data ordering information.

In MBO `Customer.cs`:

```
public static  Sybase.Collections.GenericList<sample.Customer>
FindWithQuery(Sybase.Persistence.Query query)
```

In Database class `SampleAppDB.cs`:

```
public static Sybase.Persistence.QueryResultSet
ExecuteQuery(Sybase.Persistence.Query query)
```

The following classes define arbitrary search methods and filter conditions, and provide methods for combining test criteria and dynamically querying result sets.

Table 5. Query and Related Classes

Class	Description
Query	Defines arbitrary search methods and can be composed of search conditions, object/row state filter conditions, and data ordering information.
AttributeTest	Defines filter conditions for MBO attributes.
CompositeTest	Contains a method to combine test criteria using the logical operators AND, OR, and NOT to create a compound filter.
QueryResultSet	Provides for querying a result set for the dynamic query API.

In addition queries support select, where, and join statements.

Define these conditions by setting properties in a query:

- **TestCriteria** – criteria used to filter returned data.
- **SortCriteria** – criteria used to order returned data.
- **Skip** – an integer specifying how many rows to skip. Used for paging.
- **Take** – an integer specifying the maximum number of rows to return. Used for paging.

TestCriteria can be an AttributeTest or a CompositeTest.

An AttributeTest defines a filter condition using an MBO attribute, and supports these conditions:

- IS_NULL
- NOT_NULL
- EQUAL
- NOT_EQUAL
- LIKE
- NOT_LIKE
- LESS_THAN
- LESS_EQUAL
- GREATER_THAN
- GREATER_EQUAL
- CONTAINS
- STARTS_WITH
- ENDS_WITH
- DOES_NOT_START_WITH
- DOES_NOT_END_WITH
- DOES_NOT_CONTAIN

Dynamic Query

User can use query to construct a query SQL statement as he wants to query data from local database. This query may across multiple tables (MBOs).

```
Query query2 = new Query();
query2.Select("c.fname,c.lname,s.order_date,s.region");
query2.From("Customer", "c");
//
// Convenience method for adding a join to the query
// Detailed construction of the join criteria
query2.Join("Sales_order", "s", "c.id", "s.cust_id");
AttributeTest ts = new AttributeTest();
ts.Attribute = ("fname");
ts.TestValue = "Beth";
query2.Where(ts);
QueryResultSet resultSet = SampleAppDB.ExecuteQuery(query2);
```

Paging Data

On low memory devices, retrieving up to 30,000 records from the database may cause the custom client to fail and throw an OutOfMemoryException.

Consider using the Query object to limit the result set:

```
Query props = new Query();
props.Skip =10;
props.Take = 5;
```

```
CustomerList customers = Customer.FindWithQuery(props);
```

CompositeTest

A `CompositeTest` combines multiple `TestCriteria` using the logical operators AND, OR and NOT to create a compound filter.

SortCriteria

`SortCriteria` defines a `SortOrder`, which contains an attribute name and an order type (ASCENDING or DESCENDING).

Complex Example

For example, to locate all customer objects based on this criteria:

- FirstName = John AND LastName = Doe AND (State = CA or State = NY)
- Customer is New or Updated
- Ordered by LastName ASC, FirstName ASC, Credit DESC
- Skip the first 10 and take 5

This code demonstrate the usage of `CompositeTest`, `SortCriteria` and `Query`:

```
Query props = new Query();
        //define the attribute based conditions
        //Users can pass in a string if they know the attribute
name. R1 column name = attribute name.
        CompositeTest innerCompTest = new CompositeTest();
        innerCompTest.Operator = CompositeTest.OR;
        innerCompTest.Add(new AttributeTest("state", "CA",
AttributeTest.EQUAL));
        innerCompTest.Add(new AttributeTest("state", "NY",
AttributeTest.EQUAL));
        CompositeTest outerCompTest = new CompositeTest();
        outerCompTest.Operator = CompositeTest.OR;
        outerCompTest.Add(new AttributeTest("fname", "Jane",
AttributeTest.EQUAL));
        outerCompTest.Add(new AttributeTest("lname", "Doe",
AttributeTest.EQUAL));
        outerCompTest.Add(innerCompTest);
        //define the ordering
        SortCriteria sort = new SortCriteria();

        sort.Add("fname", SortOrder.ASCENDING);
        sort.Add("lname", SortOrder.ASCENDING);
        //set the Query object
        props.TestCriteria = outerCompTest;
        props.SortCriteria = sort;
        props.Skip = 10;
        props.Take = 5;
        Sybase.Collections.GenericList<Customer> customers2 =
Customer.FindWithQuery(props);
```

QueryResultSet

The `QueryResultSet` class provides for querying a result set for the dynamic query API. `QueryResultSet` is returned as a result of executing a query.

Example

The following example shows how to filter a result set and get values by taking data from two mobile business objects, creating a Query, filling in the criteria for the query, and filtering the query results:

```
Sybase.Persistence.Query query = new Sybase.Persistence.Query();
          query.Select("c.fname,c.lname,s.order_date,s.region");
          query.From("Customer ", "c");
          query.Join("SalesOrder ", "s", " s.cust_id ", "c.id");
          AttributeTest at = new AttributeTest();
          at.Attribute = "lname";
          at.TestValue = "Devlin";
          query.TestCriteria = at;
          QueryResultSet qrs = SampleAppDB.ExecuteQuery(query);
          while(qrs.Next())
          {
              Console.Write(qrs.GetString(1));
              Console.Write(",");
              Console.WriteLine(qrs.GetStringByName("c.fname"));

              Console.Write(qrs.GetString(2));
              Console.Write(",");
              Console.WriteLine(qrs.GetStringByName("c.lname"));

              Console.Write(qrs.GetString(3));
              Console.Write(",");
Console.WriteLine(qrs.GetStringByName("s.order_date"));

              Console.Write(qrs.GetString(4));
              Console.Write(",");
              Console.WriteLine(qrs.GetStringByName("s.region"));
          }
```

Retrieving Relationship Data

A relationship between two MBOs allows the parent MBO to access the associated MBO. If the relationship is bi-directional, it also allows the child MBO to access the associated parent MBO.

Assume there are two MBOs defined in Unwired Server. One MBO is called Customer and contains a list of customer data records. The second MBO is called SalesOrder and contains order information. Additionally, assume there is an association between Customers and SalesOrder on the customer ID column. The Orders application is parameterized to return order information for the customer ID.

```
Customer customer = Customer.FindByPrimaryKey(101);
          Sybase.Collections.GenericList<SalesOrder> orders =
```

```
customer.Orders;
```

You can also use the Query class to filter the return MBO list data.

```
Query props = new Query();
… // set query parameters
Sybase.Collections.GenericList<SalesOrder> orders =
customer.GetOrdersFilterBy(props);
```

Operations APIs

Mobile business object operations are performed on an MBO instance. Operations in the model that are marked as create, update, or delete (CUD) operations create instances (non-static) of operations in the generated client-side objects.

Any parameters in the create, update, or delete operation that are mapped to the object's attributes are handled internally by the Client Object API, and are not exposed. Any parameters not mapped to the object's attributes are left as parameters in the Generated Object API.

Note: If the Sybase Unwired Platform object model defines one instance of a create operation and one instance of an update operation, and all operation parameters are mapped to the object's attributes, then a Save method can be automatically generated which, when called internally, determines whether to insert or update data to the local client-side database. In other situations, where there are multiple instances of create or update operations, it is not possible to automatically generate such a Save method.

Create Operation
To execute a create operation on an MBO, create a new MBO instance, set the MBO attributes, then call the Save() or Create() operation.

```
Customer cust = new Customer();
cust.Fname  =  "supAdmin" ;
cust.Company_name =    "Sybase";
cust.Phone = "777-8888";
cust.Create();// or cust.Save();
cust.SubmitPending();
```

Update Operation
To execute update operations on an MBO, get an instance of the MBO, set the MBO attributes, then call either the Save() or Update() operations.

```
Customer cust = Customer.FindByPrimaryKey(101);
cust.Fname = "supAdmin";
cust.Company_name = "Sybase";
cust.Phone = "777-8888";
cust.Update();// or cust.Save();
cust.SubmitPending();
```

Delete Operation

To execute delete operations on an MBO, get an instance of the MBO, set the MBO attributes, then call the `Delete()` operation.

```
Customer cust = Customer.FindByPrimaryKey(101);
cust.Delete();
```

Other Operation

Operations that are not create, update, or delete operations are called "Other" operations.

Suppose the Customer MBO has an Other operation "other", with parameters "p1" (string), "p2" (int) and "p3" (date). This results in a `CustomerOtherOperation` class being generated, with "p1", "p2" and "p3" as its attributes.

To invoke the Other operation, create an instance of `CustomerOtherOperation`, and set the correct operation parameters for its attributes. This code provides an example:

```
CustomerOtherOperation other = new CustomerOtherOperation();
other.P1 = "somevalue";
other.P2 = 2;
other.P3 = System.DateTime.Now;
other.Save(); // or other.Create()
other.SubmitPending();
```

Cascade Operations

Composite relationships are cascaded. Cascade operations allow a single synchronization to execute a chain of related CUD operations. Multi-level insert is a special case for cascade operations. It allows parent and children objects to be created in one round without having to synchronize multiple times.

Refer to Unwired WorkSpace documentation (Relationship Guidelines and Multi-level insert operations) for information about defining relationships that support cascading (composite) operations.

Consider creating a Customer and a new SalesOrder at the same time on the client side, where the SalesOrder has a reference to the new Customer identifier. The following example demonstrates a multilevel insert:

```
Customer customer = new Customer();
customer.Fname = "firstName";
customer.Lname = "lastName";
customer.Phone = "777-8888";
customer.Save();
SalesOrder order = new SalesOrder();
order.Customer = customer;
order.Order_date = DateTime.Now;
order.Region = "Eastern";
order.Sales_rep = 102;
customer.Orders.Add(order);
//Only the parent MBO needs to call Save()
customer.Save();
```

```
//Must submit parent
customer.SubmitPending();
```

To insert an order for an existing customer, first find the customer, then create a sales order with the customer ID retrieved:

```
Customer customer = Customer.FindByPrimaryKey(102);
SalesOrder order = new SalesOrder();
order.Customer = customer;
order.Order_date = DateTime.UtcNow;
order.Region = "Eastern";
order.Sales_rep = 102;
customer.Orders.Add(order);
order.Save();
customer.SubmitPending();
```

To update MBOs in composite relationships, perform updates on every MBO to change and call SubmitPending on the parent MBO:

```
Customer cust = Customer.FindByPrimaryKey(101);
Sybase.Collections.GenericList<SalesOrder> orders = cust.Orders;
SalesOrder order = orders[0];
order.Order_date = DateTime.Now;
order.Save();
cust.SubmitPending();
```

To delete a single child in a composite relationship, call the child's Delete method, and the parent MBO's SubmitPending.

```
Customer cust = Customer.FindByPrimaryKey(101);
Sybase.Collections.GenericList<SalesOrder> orders = cust.Orders;
SalesOrder order = orders[0];
order.Delete();
cust.SubmitPending();
```

To delete all MBOs in a composite relationship, call Delete and SubmitPending on the parent MBO:

```
Customer cust = Customer.FindByPrimaryKey(101);
cust.Delete();
cust.SubmitPending();
```

Note: For non-composite relationships, SubmitPending must be called on each and every MBO.

See the Sybase Unwired Platform online documentation for specific multilevel insert requirements.

Pending Operation
You can manage pending operations using these methods:

- **CancelPending** – cancels the previous create, update, or delete operations on the MBO. It cannot cancel submitted operations.
- **SubmitPending** – submits the operation so that it can be replayed on the Unwired Server. For message-based synchronization, a replay request is sent directly to the Unwired

Server. For replication-based synchronization, a request is sent to the Unwired Server during a synchronization.

- **SubmitPendingOperations** – submits all the pending records for the entity to the Unwired Server. This method internally invokes the `SubmitPending` method on each of the pending records.
- **CancelPendingOperations** – cancels all the pending records for the entity. This method internally invokes the `CancelPending` method on each of the pending records.

```
Customer customer = Customer.FindByPrimaryKey(101);
if(errorHappened)
{
  Customer.CancelPending();
}
else
{
  customer.SubmitPending();
}
```

Passing Structures to Operations

Structures hold complex datatypes (for example a string list, class or MBO object, or a list of objects) that enhance interactions with certain enterprise information systems (EIS) data sources, such as SAP and Web services, where the mobile business object (MBO) requires complex operation parameters.

An Unwired WorkSpace project includes an example MBO that is bound to a Remedy Web service data source that includes a create operation that takes a structure as an operation parameter. MBOs differ depending on the data source, configuration, and so on, but the principles are similar.

The SimpleCaseList MBO contains a create operation that has a number of parameters, including a parameter named _HEADER_ that is a structure datatype named AuthenticationInfo, defined as:

```
AuthenticationInfo
   userName: String
   password: String
   authentication: String
   locale: String
   timeZone: String
```

Structures are implemented as classes, so the parameter _HEADER_ is an instance of the AuthenticationInfo class. The generated Java code for the create operation is:

```
public  void Create(Authentication _HEADER_,string escalated,string
hotlist,
string orig_Submitter,string pending,string workLog);
```

This example demonstrates how to initialize the AuthenticationInfo class instance and pass them, along with the other operation parameters, to the create operation:

```
AuthenticationInfo authen = new AuthenticationInfo();
          authen.UserName = "Demo";
```

```
            SimpleCaseList newCase = new SimpleCaseList();
            newCase.Case_Type = "Incident";
            newCase.Category = "Networking";
            newCase.Department = "Marketing";
            newCase.Description = "A new help desk case.";
            newCase.Item = "Configuration";
            newCase.Office = "#3 Sybase Drive";
            newCase.Submitted_By = "Demo";
            newCase.Phone_Number = "#0861023242526";
            newCase.Priority = "High";
            newCase.Region = "USA";
            newCase.Request_Urgency = "High";
            newCase.Requester_Login_Name = "Demo";
            newCase.Requester_Name = "Demo";
            newCase.Site = "25 Bay St, Mountain View, CA";
            newCase.Source = "Requester";
            newCase.Status = "Assigned";
            newCase.Summary = "MarkHellous was here Fix it.";
            newCase.Type = "Access to Files/Drives";
            newCase.Create_Time = System.DateTime.Now;

          newCase.Create (authen, "Other", "Other", "false", "work
log");
            newCase.SubmitPending();
```

Local Business Object

A business object can be either local or mobile. A Local Business Object (LBO) is a client-only object. LBOs are useful to persist an application's local data without updating the backend. The difference between a LBO and an MBO is that an MBO's operations are sent to the backend. LBO's operations are updated only to the local state do not affect the backend. For example, an LBO would be well suited to store some bookkeeping information on an application device.

An example of a Local Business Object:

```
LoginStatus status= new LoginStatus ();
   status.Id = 123;
   status.Time = DateTime.Now;
   status.Success = true;
   status.Create();

   long savedId = 123;
   LoginStatus status = LoginStatus.Find(savedId);
   status.Success = false;
   status.Update();


   long savedId = 123;
   LoginStatus status = LoginStatus.Find(savedId);
   status.Delete();
```

Personalization APIs

Personalization keys allow the application to define certain input parameter values that differ (are personalized) from each mobile user. The Personalization APIs allow you to manage personalization keys, and get and set personalization key values.

Type of Personalization Keys

There are three types of personalization keys: client, server, and session. Client personalization keys are persisted in the local database. Server personalization keys are persisted on the Unwired Server. Session personalization keys are not persisted and are lost after the device application terminates.

A personalization parameter can be a primitive or complex type. This is shown in the code example.

Get or Set Personalization Key Values

The `PersonalizationParameters` class is generated automatically for managing personalization keys.

The following code provides an example on how to set a personalization key, and pass an array of values and array of objects:

```
PersonalizationParameters pp =
SampleAppDB.GetPersonalizationParameters();
pp.MyIntPK = 10002;
pp.Save();
Sybase.Collections.IntList il = new Sybase.Collections.IntList();
il.Add(10001);
il.Add(10002);
pp.MyIntListPK = il;
pp.Save();
Sybase.Collections.GenericList<MyData> dl = new
Sybase.Collections.GenericList<MyData>(); //MyData is a structure
type defined in tooling
MyData md = new MyData();
md.IntMember = 123;
md.StringMember = "abc";
dl.Add(md);
pp.MyDataListPK = dl;
pp.Save();
```

If a synchronization parameter is personalized, you can overwrite the value of that parameter with the personalization value.

Note: For detailed description on personalization key usage, see the *Sybase Unwired Platform online help*.

Object State APIs

The object state APIs provide methods for returning information about the state of an entity.

Entity State Management

The object state APIs provide methods for returning information about entities in the database. All entities that support pending state have the following attributes:

Name	C# Type	Description
IsNew	bool	Returns true if this entity is new (but has not been created in the client database).
IsCreated	bool	Returns true if this entity has been newly created in the client database, and one the following is true: • The entity has not yet been submitted to the server with a replay request. • The entity has been submitted to the server, but the server has not finished processing the request. • The server rejected the replay request (replayFailure message received).
IsDirty	bool	Returns true if this entity has been changed in memory, but the change has not yet been saved to the client database.
IsDeleted	bool	Returns true if this entity was loaded from the database and was subsequently deleted.
IsUpdated	bool	Returns true if this entity has been updated or changed in the database, and one of the following is true: • The entity has not yet been submitted to the server with a replay request. • The entity has been submitted to the server, but the server has not finished processing the request. • The server rejected the replay request (replayFailure message received).
Pending	bool	Returns true for any row that represents a pending create, update, or delete operation, or a row that has cascading children with a pending operation.

Name	C# Type	Description
PendingChange	char	If pending is true, then 'C' (create), 'U' (update), 'D' (delete), 'P' (to indicate that this MBO is a parent in a cascading relationship for one or more pending child objects, but this MBO itself has no pending create, update or delete operations). If pending is false, then 'N'.
ReplayCounter	long	Returns a long value which is updated each time a row is created or modified by the client. This value is derived from the time in seconds since an epoch, and increases each time a row is changed.
ReplayPending	long	Returns a long value. When a pending row is submitted to the server, the value of ReplayCounter is copied to ReplayPending. This allows the client code to detect if a row has been changed since it was submitted to the server (that is, if the value of ReplayCounter is greater than ReplayPending).
ReplayFailure	long	Returns a long value. When the server responds with a ReplayFailure message for a row that was submitted to the server, the value of ReplayCounter is copied to ReplayFailure, and ReplayPending is set to 0.

Entity State Example

This table shows how the values of the entities that support pending state change at different stages during the MBO update process. The values that change between different states appear in bold.

Note the following entity behaviors:

- The IsDirty flag is set if the entity changes in memory but is not yet written to the database. Once you save the MBO, this flag clears.
- The ReplayCounter value that gets sent to the Unwired Server is the value in the database before you call SubmitPending. After a successful replay, that value is imported from the Unwired Server.
- The last two entries in the table are two possible results from the operation; only one of these results can occur for a replay request.

Description	Flags/Values
After reading from the database, before any changes are made.	IsNew=false IsCreated=false IsDirty=false IsDeleted=false IsUpdated=false Pending=false PendingChange='N' ReplayCounter=33422977 ReplayPending=0 ReplayFailure=0
One or more attributes are changed, but changes not saved.	IsNew=false IsCreated=false IsDirty=**true** IsDeleted=false IsUpdated=false Pending=false PendingChange='N' ReplayCounter=33422977 ReplayPending=0 ReplayFailure=0

Description	Flags/Values
After `entity.Save()` or `entity.Update()` is called.	IsNew=false IsCreated=false IsDirty=**false** IsDeleted=false IsUpdated=**true** Pending=**true** PendingChange=**'U'** ReplayCounter=**33424979** ReplayPending=0 ReplayFailure=0
After `entity.SubmitPending()` is called to submit the MBO to the server	IsNew=false IsCreated=false IsDirty=false IsDeleted=false IsUpdated=true Pending=true PendingChange='U' ReplayCounter=33424981 ReplayPending=**33424981** ReplayFailure=0

Description	Flags/Values
Possible result: the Unwired Server accepts the update, sends an import and a `ReplayResult` for the entity, and the refreshes the entity from the database.	IsNew=false
	IsCreated=false
	IsDirty=false
	IsDeleted=false
	IsUpdated=**false**
	Pending=**false**
	PendingChange=**'N'**
	ReplayCounter=**33422977**
	replayPending=**0**
	ReplayFailure=0
Possible result: The Unwired Server rejects the update, sends a `ReplayFailure` for the entity, and refreshes the entity from the database	IsNew=false
	IsCreated=false
	IsDirty=false
	IsDeleted=false
	IsUpdated=true
	Pending=true
	PendingChange='U'
	ReplayCounter=33424981
	ReplayPending=**0**
	ReplayFailure=**33424981**

Pending State Pattern

When a create, update, delete, or save operation is called on an entity, the requested change becomes pending. To apply the pending change, call `SubmitPending` on the entity, or `SubmitPendingOperations` on the MBO class:

```
Customer e = new Customer();
e.Name = "Fred";
e.Address = "123 Four St.";
e.Create(); // create as pending
e.SubmitPending(); // submit to server

Customer.SubmitPendingOperations(); // submit all pending Customer
rows to server
```

`SubmitPendingOperations` submits all the pending records for the entity to the Unwired Server. This method internally invokes the `SubmitPending` method on each of the pending records.

For message-based sychronization, the call to `SubmitPending` causes a JSON message to be sent to the Unwired Server with the `Replay` method, containing the data for the rows to be created, updated, or deleted. The Unwired Server processes the message and responds with a JSON message with the `ReplayResult` method (the Unwired Server accepts the requested operation) or the `ReplayFailure` method (the server rejects the requested operation).

If the Unwired Server accepts the requested change, it also sends one or more `Import` messages to the client, containing data for any created, updated, or deleted row that has changed on the Unwired Server as a result of the `Replay` request. These changes are written to the client database and marked as rows that are not pending. When the `ReplayResult` message is received, the pending row is removed, and the row remaining in the client database now contains data that has been imported from and validated by the Unwired Server. The Unwired Server may optionally send a log record to the client indicating a successful operation.

If the Unwired Server rejects the requested change, the client receives a `ReplayFailed` message, and the entity remains in the pending state, with its `ReplayFailed` attribute set to indicate that the change was rejected.

For replication-based synchronization, the call to `SubmitPending` creates a replay record in local database. When the `DBClass.Synchronize()` method is called, the replay records are uploaded to Unwired Server. Unwired Server processes the replay records one by one and either accepts or rejects it.

At the end of the synchronization, the replay results are downloaded to client along with any created, updated or deleted rows that have changed on the Unwired Server as a result of the Replay requests. These changes are written to the client database and marked as rows that are not pending.

When the operation is successful, the pending row is removed, and the row remaining in the client database now contains data that has been imported from and validated by the Unwired Server. If the Unwired Server rejects the requested change, the entity remains in the pending state, with its ReplayFailed attribute set to indicate that the change was rejected. The Unwired Server may optionally send a log record to the client.

The `LogRecord` interface for both message-based synchronization and replication-based synchronization has the following getter methods to access information about the log record:

Method Name	C# Type	Description
Component	string	Name of the MBO for the row for which this log record was written.

Method Name	C# Type	Description
EntityKey	string	String representation of the primary key of the row for which this log record was written.
Code	int	One of several possible HTTP error codes: • 200 indicates success. • 401 indicates that the client request had invalid credentials, or that authentication failed for some other reason. • 403 indicates that the client request had valid credentials, but that the user does not have permission to access the requested resource (package, MBO, or operation). • 404 indicates that the client tried to access a nonexistent package or MBO. • 405 indicates that there is no valid license to check out for the client. • 500 to indicate an unexpected (unspecified) server failure.
Message	string	Descriptive message from the server with the reason for the log record.
Operation	string	The operation (create, update, or delete) that caused the log record to be written.
RequestId	string	The id of the replay message sent by the client that caused this log record to be written.
Timestamp	System.Date-Time?	Date and time of the log record.

If a rejection is received, the application can use the entity method `GetLogRecords` or the database class method `SampleDB.GetLogRecords(query)` to access the log records and get the reason:

```
Sybase.Collections.GenericList<Sybase.Persistence.ILogRecord> logs =
e.GetLogRecords();
for(int i=0; i<logs.Size(); i++)
{
Console.WriteLine("Entity has a log record:");
Console.WriteLine("Code = {0}",logs[i].Code);
Console.WriteLine("Component = {0}",logs[i].Component);
Console.WriteLine("EntityKey = {0}",logs[i].EntityKey);
Console.WriteLine("Level = {0}",logs[i].Level);
Console.WriteLine("Message = {0}",logs[i].Message);
Console.WriteLine("Operation = {0}",logs[i].Operation);
Console.WriteLine("RequestId = {0}",logs[i].RequestId);
```

```
Console.WriteLine("Timestamp = {0}",logs[i].Timestamp);
}
```

`CancelPendingOperations` cancels all the pending records for an entity. This method internally invokes the `CancelPending` method on each of the pending records.

Mobile Business Object States
A mobile business object can be in one of three states:

- Original state, the state before any create, update, or delete operation.
- Downloaded state, the state downloaded from the Unwired Server.
- Current state, the state after any create, update, or delete operation.

The mobile business object class provides properties or methods for querying the original state and the downloaded state:

```
public sample.Customer GetOriginalState()
public Customer DownloadState;
```

The original state is valid only before the application synchronizes with the Unwired Server. After synchronization has completed successfully, the original state is cleared and set to null.

```
Customer cust = Customer.FindByPrimaryKey(101); // state 1
cust.Fname = "supAdmin";
cust.Company_name = "Sybase";
cust.Phone = "777-8888";
cust.Save(); // state 2
Customer org = cust.GetOriginalState(); // state 1
//suppose there is new download for Customer 101 here
Customer download = cust.DownloadState; // state 3
cust.CancelPending(); // state 3
```

Using all three states, the application can resolve most conflicts that may occur.

Refresh Operation
The refresh operation of an MBO allows you to refresh the MBO state from the client database.

The following code provides an example:

```
Customer cust = Customer.FindByPrimaryKey(101);
cust.Fname = "newName";
cust.Refresh();// newName is discarded
```

Utility APIs
The Utility APIs allow you to customize aspects of logging, callback handling, and generated code.

Using the Logger and LogRecord APIs

LogRecord is used to store two types of logs.

- Operation logs on the Unwired Server. These logs can be downloaded to the device.
- Client logs. These logs can be uploaded to the Unwired Server.

DBClass.GetLogger – gets the log API. The client can write its own records using the log API. For example:

```
ILogger logger = SampleAppDB.GetLogger();
    logger.Debug("Write this string to the log records table");
    SampleAppDB. SubmitLogRecords();
```

DBClass.GetLogRecords – gets the log records received from the server. For example:

```
Query query = new Query();
query.TestCriteria =
Sybase.Persistence.AttributeTest.Equal("component", "Customer");
Sybase.Persistence.SortCriteria sortCriteria = new
Sybase.Persistence.SortCriteria();
sortCriteria.Add("requestId",
Sybase.Persistence.SortOrder.DESCENDING);
query.SortCriteria = sortCriteria;

GenericList<ILogRecord> loglist = SampleAppDB.GetLogRecords(query);
```

GenerateId

You can use the GenerateId method in the LocalKeyGenerator or KeyGenerator classes to generate an ID when creating a new object for which you require a primary key or surrogate key.

This method in the LocalKeyGenerator class generates a unique ID for the package on the local device:

```
public static long GenerateId()
```

This method in the KeyGenerator class generates a unique ID for the same package across all devices:

```
public static long GenerateId()
```

Callback Handlers

To receive callbacks, you must register a CallBackHandler with the generated database class, the entity class, or both. You can create a handler by extending the DefaultCallbackHandler class, or by creating a new handler by implementing the interface, ICallbackHandler.

In your handler, override the particular callback that you are interested in (for example, OnImport). The callback is executed in the thread that is performing the action (for example, import). When you receive the callback, the particular activity is already complete.

Note: Message-based synchronization and replication-based synchronization share the same CallbackHandler interface. Some of the callbacks are applicable to message-based

synchronization only or to replication-based synchronization only, while others are shared by both.

Callbacks in the CallbackHandler interface include:

```
namespace Sybase.Persistence
{
    /// <summary>
    /// Call back interface which would get called based on event.
    /// </summary>
    /// <remarks>
    /// MBS and RBS share the same CallbackHandler interface.
   /// Some of the callbacks are applicable to MBS only or RBS only,
others are common.
    /// Please see the method comments for the details.
    /// </remarks>
    public interface ICallbackHandler
    {
        /// <summary>
        /// Called when a import message successfully applies to
databases.
        /// <param name="mbo">The Mobile Business Object was just
imported.</param>
        /// </summary>
        /// <remarks>MBS only</remarks>
        void OnImport(object o);

        /// <summary>
        /// Called when login fails.
        /// </summary>
        /// <remarks>MBS only</remarks>
        void OnLoginFailure();

        /// <summary>
        /// Called when login succeeds.
        /// </summary>
        /// <remarks>MBS only</remarks>
        void OnLoginSuccess();

        /// <summary>
        /// Called when a replay request fails
        /// <param name="mbo">The Mobile Business Object to replay.</
param>
        /// </summary>
        void OnReplayFailure(object o);

        /// <summary>
        /// Called when a replay request succeeds
        /// <param name="mbo">The Mobile Business Object to replay.</
param>
        /// </summary>
        void OnReplaySuccess(object o);

        /// <summary>
        /// Called when a backend search fails
        /// <param name="mbo">The backend search object</param>
```

```
        /// </summary>
        void OnSearchFailure(object o);

        /// <summary>
        /// Called when a backend search succeeds
        /// <param name="mbo">The backend search object</param>
        /// </summary>
        void OnSearchSuccess(object o);

        /// <summary>
        /// Called when subscribe succeeds
        /// </summary>
        /// <remarks>MBS only</remarks>
        void OnSubscribeSuccess();

        /// <summary>
        /// Called when subscribe fails
        /// </summary>
        /// <remarks>MBS only</remarks>
        void OnSubscribeFailure();

        /// <summary>
       /// Called when last import message has already been processed
successfully regarding subscribe\resume\recover request.
        /// </summary>
        /// <remarks>MBS only</remarks>
        void OnImportSuccess();

        /// <summary>
        /// Called when unsubscribe succeeds.
        /// </summary>
        /// <remarks>MBS only</remarks>
        void OnUnsubscribeSuccess();

        /// <summary>
        /// Called when unsubscribe fails.
        /// </summary>
        /// <remarks>MBS only</remarks>
        void OnUnsubscribeFailure();

        /// <summary>
        /// Called when suspend subscription succeeds.
        /// </summary>
        /// <remarks>MBS only</remarks>
        void OnSuspendSubscriptionSuccess();

        /// <summary>
        /// Called when suspend subscription fails.
        /// </summary>
        /// <remarks>MBS only</remarks>
        void OnSuspendSubscriptionFailure();

        /// <summary>
        /// Called when resume subscription succeeds.
        /// </summary>
        /// <remarks>MBS only</remarks>
```

```
        void OnResumeSubscriptionSuccess();

        /// <summary>
        /// Called when resume subscription fails.
        /// </summary>
        /// <remarks>MBS only</remarks>
        void OnResumeSubscriptionFailure();

        /// <summary>
        /// Called when recover succeeds.
        /// </summary>
        /// <remarks>MBS only</remarks>
        void OnRecoverSuccess();

        /// <summary>
        /// Called when recover fails.
        /// </summary>
        /// <remarks>MBS only</remarks>
        void OnRecoverFailure();

        /// <summary>
        /// Called when reset succeeds.
        /// </summary>
        /// <remarks>MBS only</remarks>
        void OnResetSuccess();

        /// <summary>
        /// Called at the specified status of the synchronization.
        /// <param name="groups">a list of synchronization groups.</
param>
        /// <param name="context">synchronization context.</param>
        /// </summary>
        /// <remarks>RBS only</remarks>
        SynchronizationAction
OnSynchronize(Sybase.Collections.GenericList<Sybase.Persistence.ISy
nchronizationGroup> groups, SynchronizationContext context);

        /// <summary>
        /// Called if the synchronization failed.
        /// </summary>
        /// <param name="groups">a list of synchronization groups.</
param>
        /// <remarks>RBS only</remarks>
        void
OnSynchronizeFailure(Sybase.Collections.GenericList<Sybase.Persiste
nce.ISynchronizationGroup> groups);

        /// <summary>
        /// Called if synchronization succeed.
        /// </summary>
        /// <param name="groups">a list of synchronization groups.</
param>
        /// <remarks>RBS only</remarks>
        void
OnSynchronizeSuccess(Sybase.Collections.GenericList<Sybase.Persiste
nce.ISynchronizationGroup> groups);
```

```
        /// <summary>
        /// Called when subscription end.
        /// </summary>
        /// <remarks>MBS only</remarks>
        void OnSubscriptionEnd();


        /// <summary>
       /// Other callbacks in this interface (whose names begin with
"on") are invoked inside a database transaction. If the transaction
will be rolled back due to an unexpected exception, then this
operation will be called with the exception (before rollback occurs).
        /// </summary>
        /// <remarks>MBS only</remarks>
        void OnMessageException(System.Exception ex);

        /// <summary>
       /// Other callbacks in this interface (whose names begin with
"on") are invoked inside a database transaction. If the transaction
is successfully committed, then this operation will be invoked after
commit.
        /// </summary>
        /// <remarks>MBS only</remarks>
        void OnTransactionCommit();

        /// <summary>
       /// Other callbacks in this interface (whose names begin with
"on") are invoked inside a database transaction. If the transaction
is rolled back, then this operation will be invoked after rollback.
        /// </summary>
        /// <remarks>MBS only</remarks>
        void OnTransactionRollback();

        /// <summary>
        /// Called before applying an import message to database.
        /// <param name="mbo">The Mobile Business Object to be
imported.</param>
        /// </summary>
        /// <remarks>MBS only</remarks>
        void BeforeImport(object o);

        /// <summary>
        /// Called when device storage is critically low
        /// </summary>
        /// <remarks>MBS only</remarks>
        void OnStorageSpaceLow();

        /// <summary>
        /// Called when device storage becomes sufficient again.
        /// </summary>
        /// <remarks>MBS only</remarks>
        void OnStorageSpaceRecovered();
    }
}
```

This code example shows how to create and register a handler to receive callbacks:

```
public class MyCallbackHandler : DefaultCallbackHandler
{
    // implementation
}

CallbackHandler handler = new MyCallbackHandler();
MyDatabase.RegisterCallbackHandler(handler);
//or Customer.RegisterCallbackHandler(handler);
```

Client Database APIs
The generated package database class provides methods for managing the client database.

```
public static void CreateDatabase()
public static void DeleteDatabase()
```

Typically, `CreateDatabase` does not need to be called since it will be called internally when necessary. An application may use `DeleteDatabase` when the client database contains corrupted data and needs to be cleared.

Exceptions
Reviewing exceptions allows you to identify where an error has occurred during application execution.

Handling Exceptions
The Client Object API defines server-side and client-side exceptions.

Server-Side Exceptions
Exceptions thrown on the Unwired Server are logged in both the server log and in `LogRecord`. For `LogRecord`, the exception gets downloaded to the device automatically during synchronization (replication-based synchronization) or when importing a message (message-based synchronization).

Client-Side Exceptions
Device applications are responsible to catch and handle exceptions thrown by the client object API.

For message-based synchronization, you can catch exceptions for background thread message processing by creating a callback handler and implementing `OnMessageException` methods.

Note: Refer to *Callback Handlers* on page 62 for more information.

Exception Classes

The Client Object API supports exception classes for queries and for the messaging client.

- **SynchronizeException –** this exception is thrown when an error occurs during synchronization.
- **PersistenceException –** this exception is thrown when trying to load an MBO that is inside the local database.
- **SystemException –** this exception is thrown for uncategorized errors, and is typically unrecoverable.
- **ObjectNotFoundException –** this exception is thrown when trying to load an MBO that is not inside the local database.
- **NoSuchOperationException –** this exception is thrown when trying to call a method (using the Object Manager API) but the method is not defined for the MBO.
- **NoSuchAttributeException –** this exception is thrown when trying to access an attribute (using the Object Manager API) but the attribute is not defined for the MBO.

MetaData and Object Manager API

The MetaData and Object Manager API allows you to access metadata for database, classes, entities, attributes, operations, and parameters.

MetaData and Object Manager API

Some applications or frameworks may wish to operate against MBOs in a generic manner by invoking MBO operations without prior knowledge of MBO classes. This can be achieved by using the MetaData and Object Manager APIs.

These APIs allow retrieving the metadata of packages, MBOs, attributes, operations and parameters during runtime. The APIs are especially useful for a runtime environment without a reflection mechanism such as J2ME.

You can generate metadata classes using the -md code generation option. You can use the -rm option to generate the object manager class.

You can also generate metadata classes by selecting the option "Generate metadata classes" or "Generate metadata and object manager classes" option in the code generation wizard in the mobile application project.

ObjectManager

The ObjectManager class allows an application to call the Object API in a reflection style.

```
IObjectManager rm = new MyDatabase_RM();
ClassMetaData customer = MyDatabase.Metadata.GetClass("Customer");
AttributeMetaData lname = customer.GetAttribute("lname");
OperationMetaData save = customer.GetOperation("save");
object myMBO = rm.NewObject(customer);
rm.SetValue(myMBO, lname, "Steve");
rm.Invoke(myMBO, save, new ObjectList());
```

DatabaseMetaData

The `DatabaseMetaData` class holds package level metadata. You can use it to retrieve data such as synchronization groups, default database file, and MBO metadata .

```
DatabaseMetaData dmd = SampleAppDB.Metadata;
    foreach (String syncGroup in dmd.SynchronizationGroups)
    {
        Console.WriteLine(syncGroup);
    }
```

EntityMetaData

The `EntityMetaData` class holds metadata for the MBO, including attributes and operations.

```
EntityMetaData customerMetaData = Customer.GetMetaData();
    AttributeMetaData lname =
customerMetaData.GetAttribute("lname");
    OperationMetaData save = customerMetaData.GetOperation("save");
```

AttributeMetaData

The AttributeMetaData class holds metadata for an attribute such as attribute name, column name, type, and maxlength.

```
Console.WriteLine(lname.Name);
Console.WriteLine(lname.Column);
Console.WriteLine(lname.MaxLength);
```

Replication-Based Synchronization APIs

The following operations are available when performing replication-based synchronization.

IsSynchronized() and GetLastSynchronizationTime

For replication-based synchronization applications, the package database class provides the following two methods for querying the synchronized state and the last synchronization time of a certain synchronization group:

```
/// Returns if the synchronizationGroup was synchronized
public static bool IsSynchronized(string synchronizationGroup)

/// Returns the last synchronization time of the synchronizationGroup
public static System.DateTime GetLastSynchronizationTime(string
synchronizationGroup)
```

Push Configuration APIs

The push configuration APIs provide methods for configuring push through lightweight polling (LWP).

Note: To use the push notification API in the Object API, the Sybase Server Sync Tool must be installed on the device. You can get the installer from
`\<UnwiredPlatform_InstallDir>\Servers\UnwiredServer`
`\ClientAPI\ServerSync\ce\Installer\*.CAB`.

LWPPush

The following APIs support registering or unregistering for push notification in the generated database class:

```
MyDatabase.RegisterCallbackHandler(new PushListener());
MyDatabase.GetSynchronizationProfile().SISAppname = "TestSIS"
MyDatabase.GetSynchronizationProfile().SISIntervalMS = 10000
MyDatabase.GetSynchronizationProfile().SISNotificationFilePath =
ServerSyncRegistry.NewInstance().FilePath;
MyDatabase.StartBackgroundSynchronization();
MyDatabase.StopBackgroundSynchronization();
```

The client should set the SIS push configuration using `SynchronizationGroup`.

```
Sybase.Persistence.ISynchronizationGroup
GetSynchronizationGroup(string syncGroup)

ISynchronizationGroup sg = End2EndDB.GetSynchronizationGroup("ofs");
sg.EnableSIS = true;
sg.Interval = 0;
sg.Save();
Sybase.Persistence.SynchronizationManager.Instance.RegisterServerSy
ncConfiguration();
```

Creating a Replication-based Push Application

The device application must meet these requirements to utilize the replication-based Push Synchronization APIs described in this section.

You can develop the push application directly from generated mobile business object (MBO) code, or from the Device Application Designer.

1. Properly configure and deploy the mobile business objects (MBOs).
   a) Create a Cache Group and set the cache policy to **Scheduled** and set some value for the **Cache interval**, 30 seconds for example.
   b) Create a Synchronization Group and set some value for the **Change detection level**, one minute for example.
   c) Place all Mobile Application project MBOs in the same Cache Group and Synchronization Group.

---

d) Deploy the Mobile Application Project as **Replication-based** in the Deployment wizard.

2. Configure the Emulator or device:

a) Copy the server synchronization tool `SybaseServerSync.v2.CAB` or `SybaseServerSync.v35.CAB` from `ClientAPI\ServerSync\ce\installer` to the Emulator or device.

b) Click on the CAB file to install it on the Emulator or device.

3. Develop the application in Device Application Designer.

4. Add the Push Settings and Synchronization stock screens.

5. Generate the device application code.

6. Run the application in the Emulator or device.

7. In the Push Settings screen, define the polling interval and notification file path.
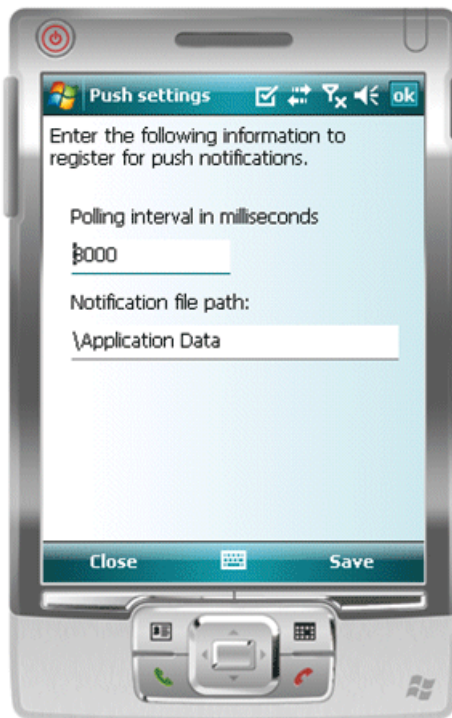
8. In the Synchronization screen, select the **Enable push** checkbox, choose a push notification mode, and click **Save**.
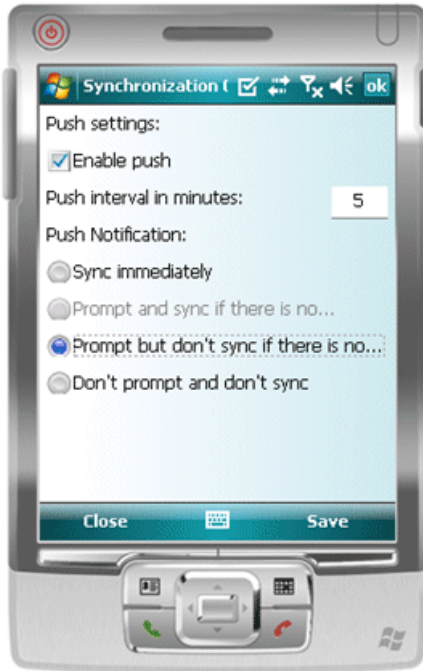
Setting Up Lightweight Polling for a Single Client

If you do not want to set the lightweight polling configuration on Unwired Server for multiple device clients, use the client-side program for a single client.

The **poll_every** unit should be set to seconds (not minutes, hours, or a combination of the two). The lightweight poller listener on the client can be turned on/off if you do not want to receive notifications during a specific period; do not just change the interval.

1. In your program, look for where you specify the polling option right after: `...;poll_notifier=UALIGHTWEIGHT;poll_key....`

2. Change the polling option, for example: `....;poll_notifier=UALIGHTWEIGHT;poll_every=180;poll_key=.....`

Note: Do not set the client **poll_every** value to a shorter time interval than the server value. This does not result in receiving push notifications any faster, and can cause the client to see the same notification multiple times, causing multiple useless synchronizations. Only set this value on the client if for some reason you do not want to see notifications as frequently as the server checks for pending notifications.

Notification Handling APIs

The notification handling APIs provide method for configuring notifications through lightweight polling (LWP).

LWPPush

To register to receive push messages through lightweight polling, the client should use these methods:

```
RegisterCallbackHandler(Sybase.Persistence.ICallbackHandler
_handler)
```

```
End2EndDB.RegisterCallbackHandler(new MyCallbackHandler(this))
```

Message-Based Synchronization APIs

Message-based synchronization uses the publish/subscribe model. During a subscription, the subscriber indicates the data to be received from the publisher. When a subscription is established, the publisher must send all relevant data to the subscriber (to maintain the data state on the client).

The following operations are available when performing message based synchronization.

Subscribe Data

The following example shows how to notify the server of your subscription to a specific package.

```
SampleAppDB.Subscribe();
```

Unsubscribe Data

If the client does not require the subscribed data, it can send an unsubscribe request to remove it. The following example shows how to notify the server to remove a subscription so it does not have to push to the application/device any longer.

```
SampleAppDB.Unsubscribe()
```

Calling to `CleanAllData()` also cleans up all data on the local database.

```
SampleAppDB.CleanAllData();
```

Suspend Operation

This operation is used if the device is going offline or user has no need to receive updates for a significant amount of time. The following example shows how to notify the server to stop delivering data change notifications for a specific package.

```
MyPackageDB.Suspend();
```

Resume Operation

The Resume operation notifies the server to resume sending the data change notifications from the last suspension. All modified data since suspension is pushed the application/device.

```
MyPackageDB.Resume();
```

Recover Operation

If data on the device is corrupted or a Resume request is rejected, you can use this operation to recover the data.

```
MyPackageDB.Recover();
```

Start Background Synchronization

This operation starts background synchronization for the database class.

```
MyPackageDB.StartBackgroundSynchronization();
```

Stop Background Synchronization

This operation stops background synchronization for the database class.

```
MyPackageDB.StopBackgroundSynchronization();
```

HasPendingOperations Operation

The HasPendingOperation operation returns false if there are some requests that have not yet been processed by server. The following code shows how to wait until the replay operation is processed by the server.

```
Customer c = new Customer();
c.Id = 900;
c.Address = "Beijing";
c.Create();
c.SubmitPending();

//use this code to wait for the replay operation result:

while (MyPackageDB.HasPendingOperations())
 {
     Thread.Sleep(1000);
  }
```

Windows Mobile Device Framework API

Describes solutions and examples for tasks and uses of the Sybase Unwired Platform Device Application Designer API, which allows you to customize the Windows Mobile device user interface.

Add Controls Manually to a Screen

You can add controls manually to a screen by using the Visual Studio Form Designer, or by editing the Form*.designer.cs file.

To add controls from the designer, open the form in Visual Studio Form Designer, and drag and drop controls from the toolbox to the form.

Another method for adding controls is to open and edit Form*.designer.cs.

```
//
// myEditBox
//
this.myEditBox.Anchor = ((System.Windows.Forms.AnchorStyles)
(System.Windows.Forms.AnchorStyles.None |
System.Windows.Forms.AnchorStyles.Top |
System.Windows.Forms.AnchorStyles.Left |
System.Windows.Forms.AnchorStyles.Right)) ;
this.myEditBox.BackColor = System.Drawing.Color.FromArgb(
    0xFF,0xFF,0xFF);
this.myEditBox.Font =
    new System.Drawing.Font("Tahoma", 9F ,
        ((System.Drawing.FontStyle)
         (System.Drawing.FontStyle.Regular)));
this.myEditBox.ForeColor =
    System.Drawing.Color.FromArgb(0x00,0x00,0x00);
this.myEditBox.LogicalType =
    Sybase.UnwiredPlatform.Windows.Forms.LogicalType.Phone;
this.myEditBox.Size = new System.Drawing.Size(150,20);
this.myEditBox.TabIndex = 16;
this.myEditBox.Tag = "16";
this.myEditBox.Text = "";
this.myEditBox.BorderStyle =
    Sybase.UnwiredPlatform.Windows.Forms.BorderStyle.BottomLine;
this.myEditBox.Location = new System.Drawing.Point(87,180);
this.myEditBox.Name = "myEditBox";
this.displayMainPanel.SetColumn(this.myEditBox, 1);
this.displayMainPanel.SetRow(this.myEditBox, 7);
this.displayMainPanel.SetRowSpan(this.myEditBox, 1);
this.displayMainPanel.SetColumnSpan(this.myEditBox, 1);
this.displayMainPanel.SetCellAnchorStyles(this.myEditBox,
    ((System.Windows.Forms.AnchorStyles)
     (System.Windows.Forms.AnchorStyles.None |
      System.Windows.Forms.AnchorStyles.Top |
      System.Windows.Forms.AnchorStyles.Left |
      System.Windows.Forms.AnchorStyles.Right)));
this.displayMainPanel.Controls.Add(this.myEditBox);

private Sybase.UnwiredPlatform.Windows.Forms.TextBox myEditBox;
```

Customize Controller

In the CustomizedCode folder, you can add new classes to customize a controller.

This is an example of the FormCreateCustomerController code

```
/// <summary>
/// The Base class of FormCreateCustomerController
/// </summary>
internal abstract class FormCreateCustomerControllerBase :
ControllerBase
{
   public FormCreateCustomerControllerBase(IFormPart form)
      : base(form)
   {
   }

   // button (Submit) click event handler
   internal virtual void SubmitButton_Handler(FormsManagerDataObject
dataObject)
   {
      // Generated code
   }
}

/// <summary>
/// The Controller class of Form FormCreateCustomer
/// </summary>
internal partial class FormCreateCustomerController :
FormCreateCustomerControllerBase
{
   public FormCreateCustomerController(IFormPart form)
      : base(form)
   {
   }
}
```

The following code example and illustration describe a partial class where you can override the virtual methods defined in FormCreateCustomerControllerBase and provide your own business logic.

```
internal partial class FormCreateCustomerController
   {
      internal override void
SubmitButton_Handler(Sybase.UnwiredPlatform.Windows.Forms.FormsMana
gerDataObject dataObject)
      {
         // Add your custom actions here
         MessageBox.Show("Before Submit!");

         // Perform the default action
         base.SubmitButton_Handler(dataObject);
      }
   }
```
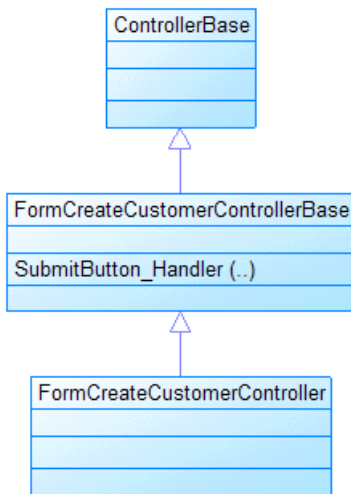
Customize Widget Event Code

You can customize widget event code in the Device Application Designer. Define the widget event in control's Actions|Coding tab.

After code generation, a widget event handler method is generated in the controller class. For example:

```
// button (Button Events) click event handler
      internal virtual void
ButtonEventsButton_ClickHandler(FormsManagerDataObject dataObject)
      {
          // actions
          try
          {
          }
          catch (Exception __ex__)
          {
              if (__ex__.InnerException != null)
              {
                  MessageBox.Show(__ex__.Message + " [" +
__ex__.InnerException.Message + "]", "Error",
System.Windows.Forms.MessageBoxButtons.OK,
System.Windows.Forms.MessageBoxIcon.Exclamation,
System.Windows.Forms.MessageBoxDefaultButton.Button1);
              }
              else
              {
                  MessageBox.Show(__ex__.Message, "Error",
System.Windows.Forms.MessageBoxButtons.OK,
System.Windows.Forms.MessageBoxIcon.Exclamation,
System.Windows.Forms.MessageBoxDefaultButton.Button1);
              }

Sybase.UnwiredPlatform.Windows.Util.Logger.Instance.Log(__ex__);
```

```
            }
        }
```

You can add your action code in this method or define the action code in a partial class so your
code will not be overridden during the next generation. For example:

```
/// <summary>
   /// The Controller class of Form FormCreateCustomerController
   /// </summary>
   internal partial class FormCreateCustomerController
   {
      internal override void
ButtonEventsButton_ClickHandler(FormsManagerDataObject dataObject)
      {
         // Add your event handler code here

         // Call base method
         base.ButtonEventsButton_ClickHandler(dataObject);
      }
   }
```

Add Validators

A validator defines a set of standard classes for performing common data validation checks,
for example, a phone number input field must be numbers. A component can have one or more
validators.

```
this.editbox.Validating += new
System.ComponentModel.CancelEventHandler
(editbox_Validating);
```

Perform UI Binding to an MBO

You can perform UI binding to an MBO through the DataBindings method.

```
this.editbox.DataBindings.Add(new
System.Windows.Forms.Binding("Text",
this.CustomerBindingSource, "Id", true,
System.Windows.Forms.DataSourceUpdateMode.Never));
```

Access Pending Operations and Operation Logs

You can access pending operations directly from the MBO.

```
public static
Sybase.Collections.GenericList<OrdersManagment.Customer>
GetPendingObjects()
public
Sybase.Collections.GenericList<Sybase.Persistence.ILogRecord>
GetLogRecords()
```

Connect to Unwired Server

You can call PackageDB.LoginToSync(username,password) to connect to the
Unwired Server.

Add or Modify Navigation

The Form Manager implements the navigation of forms. You can add or modify the navigation by using a connection action.

```
Sybase.UnwiredPlatform.Windows.Action.Action
connectionAction1_From_FormCustomer_To_FormCustomerDetails =
    Sybase.UnwiredPlatform.Windows.Action.ActionFactory
    .CreateScreenAction(typeof(FormCustomerDetails),
    (this.Form as FormBase),dataObject);
connectionAction1_From_FormCustomer_To_FormCustomerDetails
    .Execute();
```

You can also call the Form Manager directly.

```
FormBase.FormsManager.ShowForm(…)
```

Add or Modify Actions

To add or modify actions, you can create a partial class for the controller and override the event handler or you can modify the event handler code directly.

Create and Assign Variables

Variables are managed by the VariableManager.

To create a variable:

```
Sybase.UnwiredPlatform.Windows.Variable.VariableManager
   .Instance.UserVaraibles
   .AddVariable("test", null);
```

To read a variable:

```
Sybase.UnwiredPlatform.Windows.Variable.VariableManager
  .Instance.GetVariableValue(
    Sybase.UnwiredPlatform.Windows.Variable.VariableType.User,
    "test")
```

Assign PIM Actions to Controls

You can add a PIM action to a control to integrate the control with a Windows Mobile PIM application.

To add a PIM attribute to a control:

```
[Sybase.UnwiredPlatform.Windows.Forms.CustomAttributes.PIMLogicalTy
peAttribute
("Contact","FirstName")]
private Sybase.UnwiredPlatform.Windows.Forms.TextBox editbox4;
[Sybase.UnwiredPlatform.Windows.Forms.CustomAttributes.PIMLogicalTy
peAttribute
("Contact","LastName")]
private Sybase.UnwiredPlatform.Windows.Forms.TextBox editbox6;
```

To add a PIM action in the control event handler:

```
Sybase.UnwiredPlatform.Windows.Action.Action action0 =
new Sybase.UnwiredPlatform.Windows.Action.PimAction
((this.Form as FormBase),"Contact",true,"Display");
action0.Execute();
```

Change Default Layout

In Windows Mobile, the default layout in the generated application is TableLayout. To change the layout, you can open the form in the Visual Studio Form Designer, and manually change the layout.

Windows Mobile Device Framework Assemblies

The Windows Mobile Device Framework consists of the following three assemblies, which provide device component integration, custom controls, and Device Application Designer actions:

Table 6. Sybase Windows Device Framework Assemblies

Name	Function
Sybase.UnwiredPlatform.Windows	Provides drawing improvement and device component integration. For example, integration with the device's phone, e-mail, or camera functionalities.
Sybase.UnwiredPlatform.Windows.Forms	Provides customized control, MVP pattern, action framework, and validation framework.
Sybase.UnwiredPlatform.Windows.StockScreens	Provides Device Application Designer specific features. For example, stock screens, prepared stock actions, and variables.

Sybase.UnwiredPlatform.Windows

The Sybase.UnwiredPlatform.Windows assembly provides nonvisual components.

Table 7. Sybase.UnwiredPlatform.Windows Non-visual Components

Component	Description
PictureCamera	Allows developers to use the device camera to take pictures.
VideoCamera	Allows developers to use the device camera to shoot videos.
Email	Allows developers to send e-mail messages.
SMS	Allows developers to send SMS messages.

Component	Description
Phone	Allows developers to make phone calls.
GPS	Allows developers to interact with the GPS.

PictureCamera Component

Because of the many types of devices that exist, a developer must check whether a device has a picture camera, and must display a message if the device lacks a camera. The PictureCamera component simplifies developing an application that can work correctly on all devices, by allowing the use of the PictureCamera component without writing a lot of code.

The PictureCamera supports Windows Mobile Standard and Professional 5.x, 6.0, 6.1, and 6.5.

VideoCamera Component

Because of the many types of devices that exist, a developer must check whether a device has a video camera, and must display a message if the device lacks a camera. The VideoCamera component simplifies developing an application that can work correctly on all devices, by allowing the use of the VideoCamera component without writing a lot of code.

Using the VideoCamera Component

How to use the VideoCamera component.

To use the VideoCamera component:

* Add the Sybase.UnwiredPlatform.Windows assembly.
* Open a form.
* Drag and drop the VideoCamera component from toolbox to the form, or manually add VideoCamera in the form's designer.cs file.
* Set the VideoCamera object properties if required.
* Add an event handler in your form (for example, when clicking on button) to capture a video using the VideoCamera object.

Example of code generated by the form designer:

```
private Sybase.UnwiredPlatform.Windows.Device.VideoCamera
videoCamera1;
this.videoCamera1 = new
Sybase.UnwiredPlatform.Windows.Device.VideoCamera();
```

The following sample code for the event handler includes two buttons on the form: the first button captures a video and returns the video file name. The second button plays the video.

```
private void button1_Click(object sender, EventArgs e)
{
    This.FileName = this.videoCamera1.CaptureVideoFile(this);
    this.button2.Enabled = !String.IsNullOrEmpty(this.FileName);
}
```

```
private void button2_Click(object sender, EventArgs e)
{

Sybase.UnwiredPlatform.Windows.Device.Process.OpenFile(this.FileNam
e);
}
```

Email Component

The e-mail component allows a device application to set e-mail properties and to compose or send an e-mail message using an e-mail object.

The e-mail component supports Windows Mobile Standard and Professional 5.x, 6.0, 6.1, and 6.5.

Using the E-mail Component

To use the E-mail component:

• Add the Sybase.UnwiredPlatform.Windows assembly.
• Open a form.
• Drag and drop the e-mail component from toolbox to the form, or manually add e-mail in the form's `designer.cs` file.
• Set the e-mail object properties if required.
• Add an event handler in your form (for example, when clicking a button) to compose or send an e-mail message using the `Email` object.

Example of code generated by the form designer:

```
private Sybase.UnwiredPlatform.Windows.Device.Email email1;
this.email1 = new Sybase.UnwiredPlatform.Windows.Device.Email();
```

The following sample code for the event handler uses the `ComposeEmailForm()` method to send an e-mail message. The user can review the e-mail message before sending it. The addressBox1 TextBox control defines the e-mail address, the subjectBox2 TextBox control defines the e-mail subject, the emailBox3 TextBox control defines the e-mail text.

```
private void button1_Click(object sender, EventArgs e)
{
    // You can always try to open the send email form or send an email
    this.email1.ComposeEmailForm(this.addressBox1.Text,
this.subjectBox2.Text, this.emailBox3.Text, null);
}
```

SMS Component

The SMS component allows a device application to set SMS properties and to compose or send an SMS message.

The SMS component supports Windows Mobile Standard and Professional 5.x, 6.0, 6.1, and 6.5.

Using the SMS Component

To use the SMS component, you must:

- Add the Sybase.UnwiredPlatform.Windows assembly.
- Open a form.
- Drag and drop the SMS component from toolbox to the form, or manually add SMS in the form's `designer.cs` file.
- Set the SMS object properties if required.
- Add an event handler in your form (for example, when clicking a button) to compose or send an SMS message using the SMS object.

Example of code generated by the form designer:

```
private Sybase.UnwiredPlatform.Windows.Device.SMS sms1;
this.sms1 = new Sybase.UnwiredPlatform.Windows.Device.SMS();
```

The following sample code for the event handler uses the `ComposeSmsForm()` method to send a SMS. The user can review the message before sending it. The phoneBox1 TextBox control defines the SMS phone number, the smsBox2 TextBox control defines the SMS text.

```
private void button1_Click(object sender, EventArgs e)
{
    // You can always try to open the send SMS form or send a SMS
    this.sms1.ComposeSmsForm(this.phoneBox1.Text, this.smsBox2.Text);
}
```

Phone Component

The phone component .

The phone component supports Windows Mobile Standard and Professional 5.x, 6.0, 6.1, and 6.5.

Using the Phone Component

To use the phone component:

- Add the Sybase.UnwiredPlatform.Windows assembly.
- Open a form.
- Drag and drop the Phone component from toolbox to the form, or manually add the Phone component in the form's `designer.cs` file.
- Set the Phone object properties if required.
- Add an event handler in your form (for example, when clicking a button) to make a phone call using the Phone object.

Example of code generated by the form designer:

```
private Sybase.UnwiredPlatform.Windows.Device.Phone phone1;
this.phone1 = new Sybase.UnwiredPlatform.Windows.Device.Phone();
```

The following sample code for the event handler uses the `Call()` method to make a phone call. The phoneBox1 TextBox control defines the phone number.

```
private void button1_Click(object sender, EventArgs e)
{
    // You can always try to call
    this.phone1.Call(this.phoneBox1.Text, false);
}
```

Sybase.UnwiredPlatform.Windows.Forms

The Sybase.UnwiredPlatform.Forms assembly provides custom controls used in Windows Mobile and Windows code generation. These controls extend the capability and improve the look and feel of standard Windows Mobile 6.x controls.

The following controls provide features not available in the standard Windows Mobile 6.x controls:

- Transparent background
- Gradient-filled background
- Background pictures
- Background transparency level (alpha-channel)
- Border thickness and color
- Rounded corners
- Auto-ellipsis for labels
- Phone, email, and SMS integration for LinkLabel
- Button with gradient, image and text
- CheckBox and RadioButton controls with image and text
- Automatic handling of Undo, Copy, Paste, InputPanel, and zoom for TextBox control
- Watermarker for TextBox and PictureBox controls
- Camera integration in PictureBox control
- Edit and ReadOnly modes

The following provide controls that do not exist in Windows Mobile 6.x:

- ImageButton
- Masked TextBox
- Signature
- FlowLayoutPanel
- TableLayoutPanel
- PowerList
- FreeFormView
- Separator
- Touch screen optimized toolbar
- Title bar

- TabControl for Windows Mobile Standard
- RadioButton for Windows Mobile Standard

There are several versions of this assembly for different targets:

- Windows Mobile Professional/PocketPC
- Windows Mobile Standard/Smartphone
- Windows

Table 8. Sybase.UnwiredPlatform.Windows.Form Custom Controls

Component	Description
FormBase	Extends the standard Windows Mobile Form.
Button	Extends the standard Windows Mobile Button control.
CheckBox	Extends the standard Windows Mobile CheckBox control.
DateTimePicker	Extends the standard Windows Mobile DateTimePicker control.
FlowLayoutPanel	Similar to the FlowLayoutPanel control for Windows. Windows Mobile does not have such a control.
FreeFormView	Provides similar Form features as a control. It also supports paging feature if you have many controls and need to put controls on different pages.
ImageButton	A push button with image, text, or both.
Label	Extends the standard Windows Mobile Label control.
LinkLabel	Extends the standard Windows Mobile LinkLabel control.
Maps	Shows the map using Goolge Maps.
MaskedTextBox	Similar to the MaskedTextBox control for Windows. Windows Mobile does not have such a control.
Notification	Extends the standard Windows Mobile Notification control. It also supports Windows Mobile Standard and Windows.
Panel	Extends the standard Windows Mobile Panel control.
PictureBox	Extends the standard Windows Mobile PictureBox control.
PowerList	The PowerList control is a powerful list control that is optimized for touch screens.
ProgressBar	Shows the progress of a task.

Component	Description
RadioButton	Extends the standard Windows Mobile RadioButton control. It also supports Windows Mobile Standard.
Separator	Draws a horizontal or vertical separator.
Signature	Allows the user to add signature capture.
TabControl	Extends the standard Windows Mobile TabControl control.
TabPage	Extends the standard Windows Mobile TabPage control.
TableLayoutPanel	Similar to the TableLayoutPanel control for Windows. Windows Mobile does not have such a control.
TextBox	Extends the standard Windows Mobile TextBox control.
Toolbar	Provides a touch screen optimized Toolbar control.
TitleBar	Provides an easier to use TitleBar control for forms or panels.

FormBase

The FormBase is a base form class that provides common form management functions. It supports the following features:

- Gradient filled background
- Background picture
- Form navigation support

Button Control

The Button control is similar to the Windows Mobile Button control, but improves the look and feel of the standard control.

The Button control provides the following additional features:

- Gradient fill styles
- Rounded corner
- PushButton, CheckBox, ImageButton styles
- Image
- Image and text alignments
- Pushed font, code, image

CheckBox Control

The CheckBox control is similar to the Windows Mobile CheckBox control, but improves the look and feel of the standard control.

The CheckBox control provides the following additional features:

- Image
- Image and text alignments
- Pushed font, code, image
- Transparent background

DateTimePicker Control

The DateTimePicker control is similar to the Windows Mobile DateTimePicker control.

The DateTimePicker control provides the following additional features:

- Border style
- Text alignment
- Edit and ReadOnly modes
- Transparent background

FlowLayoutPanel Control

The FlowLayoutPanel control is similar to the Windows version, but provides additional features that are useful for developing Windows Mobile applications.

The FlowLayoutPanel control provides the following additional features:

- Stack layout, wrap layout, and uniform layout
- Border thickness and color
- Rounded corner
- Gradient fill
- Background image
- Transparency level

FreeFormView Control

The FreeFormView control provides similar Form features as a control. It also supports a paging feature, if you have many controls that must be placed on different pages.

ImageButton Control

The ImageButton control is identical to the Button control with the ImageButton style.

Label Control

The Label control is similar to the Windows Mobile Label control.

The Label control provides the following additional features:

- Auto ellipsis
- Text alignment
- Transparent background

LinkLabel Control

The Label control is similar to the Windows Mobile LinkLabel control.

The LinkLabel control provides the following additional features:

- Auto ellipsis
- Text alignment
- Transparent background
- Phone, e-mail, SMS, Web URL, and file path integration

Maps Control

The Maps control shows a map using the Google Maps service.

It provides the following features:

- Show map view
- Show satellite view
- Go to the current location of the GPS
- Show an address on the map
- Show a geolocation on the map
- Convert geolocation to an address
- Zoom In and Zoom Out
- Scroll

MaskedTextBox Control

The MaskedTextBox control is similar to the Windows version of MaskedTextBox. This control is a subclass of TextBox.

Note: Windows Mobile does not support the Windows version of MaskedTextBox.

The MaskedTextBox control provides the following additional features:

- Edit and ReadOnly modes
- Border style
- Watermark text
- Phone, email, SMS, Web URL, and file path integration

Notification Control

The Notification control is an extension of the Windows Mobile Notification control.

It supports the following additional features:

- Support for default icons
- Support for HTML text
- Return parsed action defined in HTML text

- Support for Smartphone and Windows

Notification Event

If HTML text is used and the user selects a button or hyperlink, a ResponseSubmitted event handler is called with the event argument Notification.ResponseSubmittedEventArgs. The notification event contains the following parsed information:

- **Response –** The original response.
- **Action –** Action defined in the HTML text. Allowable values are "notify", "hyperlink" or "unknown."
- **Arguments –** Notify arguments (lstbx=1&chkbx=on) or hyperlink name.
- **NameValues –** The name values map.

Panel Control

The Panel control extends the Windows Mobile standard Panel control.

It provides the following additional features.

- Border thickness and color
- Rounded corners
- Gradient fill
- Background image
- Transparent background
- Transparency level

PictureBox Control

The PictureBox control extends the Windows Mobile standard Panel control.

The PictureBox control provides the following features:

- Border
- Transparent background
- Watermark
- Camera integration
- Browse pictures
- Rotate pictures
- Load and save pictures
- Limit picture size

PowerList Control

The PowerList control is a powerful list control that is optimized for touch screens.

The PowerList control provides the following features:

- Allows the use of a different layout for selected and unselected items.

- Touch scrolling
- Grouping items by category
- Sorting
- Search support

RadioButton Control

The Panel control extends the Windows Mobile standard RadioButton control, and provides the following additional features.

- Enhanced radio button bitmap
- Rounded corners
- Auto ellipsis
- Support picture
- Support picture and text alignments
- Transparent background

Separator Control

The Separator control draws a line horizontally or vertically.

Signature Control

The Signature control is a subclass of PictureBox that allows users to sign their signature.

Because Signature is a subclass of PictureBox, all PictureBox properties can be used.

TabControl Control

The TabControl is similar to the Windows Mobile TabControl, but also supports smartphone.

TabPage Control

The TabPage control is similar to the Windows Mobile TabPage control, but also supports smartphone.

TableLayoutPanel Control

The TableLayoutPanel control is similar to the Windows Mobile TableLayoutPanel control, but also supports Pocket PC and Smartphone.

TextBox Control

The TextBox control is similar to the Windows Mobile TextBox control, but provides the following additional features:

- Edit and ReadOnly modes
- Border style
- Transparent background

- Phone, email, SMS, Web URL and file path support
- Watermark
- Built-in input panel support
- Built-in undo/cut/copy/paste support
- Zoom window for multiline text

Toolbar Control
The Toolbar control provides a Toolbar control optimized for touch screen or Smartphone.

TitleBar Control
The TitleBar control provides a more usable TitleBar control for forms or panels, and provides the following features:

- Icon
- Capture
- Left button
- Right button

Sybase.UnwiredPlatform.Windows.StockScreens

Provides Device Application Designer specific features. For example, variables, prepared stock actions, and stock screens.

Variables
Variables are key-values. There are three types of variables: user-defined variables, table context variables, and system variables.

A system variable is a predefined variable to retrieve system information such as device date and time, RAM, flash memory, and other parameters, and information about the Unwired Server.

The table context variable is related to the MBO used in the current context (for example, operation).

Table 9. Variables

Variable Name	Variable Constant	Description
Device Date	DEVICE_DATE	The date set on the device.
Device Time	DEVICE_TIME	The time set on the device.
OS version	DEVICE_OS	The OS version running on the device.
Package Name	PACKAGE_NAME	The package name used for the logged-in user.

Variable Name	Variable Constant	Description
Password	LOGIN_PASSWORD	The password used to log in to Sybase Unwired Platform.
User Name	LOGIN_USER_NAME	The user name used to log in to Sybase Unwired Platform.
Server Name	SERVER_NAME	The name of the Unwired Server where the user logged in.
Unique ID	GUID	A 32-bit unique ID.
Unwired Server URL	SERVER_URL	The URL used for connecting to the Sybase Unwired Server.

The following sample code shows how to access the system variable "Package Name."

```
String packageName =
Sybase.UnwiredPlatform.Windows.Variable.VariableManager.Instance.

GetVariableValue(Sybase.UnwiredPlatform.Windows.Variable.VariableTy
pe.System, "PACKAGE_NAME");
```

Actions
You can implement ActionFactory to create different types of actions that are predefined in the Device Application Designer.

Table 10. Creating Predefined Actions Using ActionFactory

Action Type	Description
Alert	Shows a message to alert the user.
Connection	An action for navigating to another screen.
Navigation Back	An action for navigating to the previous screen.
Exit	Exits the client application.
Logout	Logs out the user, and clears the user name and password. The user will be required to enter login credentials during the next login attempt.
Operation	An action for performing a mobile object operation.
Persist	Persists local variables assigned to this screen.
Refresh	Refreshes the current screen.
Synchronize	Synchronizes the MBO in the current screen.
Object Query	An action for performing an object query of a mobile business object

Alert

The following sample code for the Alert action causes an error message box to appear:

```
string description = "Hello world";
Sybase.UnwiredPlatform.Windows.Action.Action alertAction =
Sybase.UnwiredPlatform.Windows.Action.ActionFactory.CreateAlertActi
on(description,"ERROR");
alertAction.Execute();
```

Connection

The following sample code for the Connection action navigates from the current screen to the FormUpdateProduct screen:

```
Sybase.UnwiredPlatform.Windows.Action.Action connectionAction =

Sybase.UnwiredPlatform.Windows.Action.ActionFactory.CreateScreenAct
ion(typeof(FormUpdateProduct),
                                (this.Form as FormBase),  dataObject);
connectionAction.Execute();
```

Navigation Back

The following sample code for the Navigation Back action is an InlineAction that instructs the FormsManager to close the current screen and navigate to the previous screen:

```
Sybase.UnwiredPlatform.Windows.Action.Action action0 =
Sybase.UnwiredPlatform.Windows.Action.ActionFactory.CreateInlineAct
ion(
            delegate(Object[] args,out string message )
            {
                FormBase.FormsManager.CloseForm();
                message = "";
                return false;
            }
            );
            action0.Execute();
```

Exit

The following is sample code for the Exit action:

```
Sybase.UnwiredPlatform.Windows.Action.Action exitAction =
Sybase.UnwiredPlatform.Windows.Action.ActionFactory.CreateExitActio
n();
exitAction.Execute();
```

Logout

The following is sample code for the Logout action:

```
Sybase.UnwiredPlatform.Windows.Action.Action logoutAction =
Sybase.UnwiredPlatform.Windows.Action.ActionFactory.CreateLogoutAct
ion(this.Form as Form);
 logoutAction.Execute();
```

Operation

The following sample code for the Operation action calls the Create method of the mobile business object "newProduct," and creates a new instance of Product:

```
List<Object> createParameterList = new List<object>();
List<Type> createParameterTypeList = new List<Type>();
//create
Product newProduct = new Product();
newProduct.Id=100;
newProduct.Name = "SUP";


//Call Product.Create()
Sybase.UnwiredPlatform.Windows.Action.Action operationAction =

Sybase.UnwiredPlatform.Windows.Action.ActionFactory.CreateOperation
Action(
                    typeof(Product), newProduct,"Create",

createParameterList.ToArray(),createParameterTypeList.ToArray());
 operationAction.Execute();
```

Persist

The following sample code for the Persist action persists the user variable named "UserVar":

```
String variable1Value = "Hello";
Sybase.UnwiredPlatform.Windows.Action.Action persistAction =
Sybase.UnwiredPlatform.Windows.Action.ActionFactory.CreatePersistAc
tion(

Sybase.UnwiredPlatform.Windows.Variable.VariableType.User,"UserVar"
,variable1Value);
 persistAction.Execute();
```

Refresh

The following sample code for the Refresh action refreshes the current screen:

```
Sybase.UnwiredPlatform.Windows.Action.Action refreshAction =

Sybase.UnwiredPlatform.Windows.Action.ActionFactory.CreateRefreshAc
tion((this.Form as FormBase));
refreshAction.Execute();
```

Synchronize

The following sample code for the Synchronize action starts synchronization of the synchronization group where the "Product" mobile business object is a member:

```
Sybase.UnwiredPlatform.Windows.Action.Action syncAction =

Sybase.UnwiredPlatform.Windows.Action.ActionFactory.CreateSynchroni
zationAction(
                          Product.GetMetaData().GetPublication(),
```

```
"Product");
syncAction.Execute();
```

Object Query

An Object Query action can perform an object query of a mobile business object by using the `FreeMethodAction` method to delegate to the object query operations of the mobile business object, as shown in the following sample code for the Object Query action:

```
Sybase.UnwiredPlatform.Windows.Action.Action namedQueryAction =

Sybase.UnwiredPlatform.Windows.Action.ActionFactory.CreateFreeMetho
dAction(
            delegate(Object[] args,out string message )
            {
                System.Int32 p_Id = Convert.ToInt32(@"100");
                IList<Customer> findById_List =
CustomerDataStore.Instance.FindByPrimaryKey(p_Id);

                this.SetTable1Datasource(findById_List);
                message = "";
                return false;
            }
            );
 namedQueryAction.Execute();
```

Stock Screens

The following stock screens are available in the Windows Mobile UI framework.

Table 11. Stock Screens

Screen Type	Description
Login	The Login screen allows entering a user name and password.
Logs	The Logs screen shows the operation logs in the device.
Pending Operations	The Pending Operation screen shows the MBO's pending operations, and allows you to submit or delete them.
Personalization	The Personalization screen allows modifying the values of Personalization keys defined in the specific package.
Search	The Search screen allows a search on specific MBOs.
Synchronize	The Synchronize screen lets you manually synchronize the specific synchronization group and its included MBOs.

Login

The following sample code creates a Login screen:

```
FormsManagerDataObject FormLoginDataObject = new
FormsManagerDataObject();
Form loginScreen= FormBase.FormsManager.GetForm(typeof( FormLogin));
```

```
FormBase.FormsManager.FirstForm = loginScreen;
Application.Run(loginScreen);
```

Search
The following sample code creates a Search screen to search the Customer mobile business object:

```
FormsManagerDataObject dataObject = new FormsManagerDataObject();
dataObject ["ContextMBOType"] = typeof(Customer);
Sybase.UnwiredPlatform.Windows.Action.Action screenAction =
Sybase.UnwiredPlatform.Windows.Action.ActionFactory.CreateScreenAct
ion(typeof(FormSearch), (this.Form as FormBase), dataObject);
screenAction.Execute();
```

Synchronize
The following sample code creates a Synchronize screen.

```
FormsManagerDataObject dataObject = new FormsManagerDataObject();
Sybase.UnwiredPlatform.Windows.Action.Action screenAction =
Sybase.UnwiredPlatform.Windows.Action.ActionFactory.CreateScreenAct
ion(typeof(FormSynchronization), (this.Form as FormBase),
dataObject);
screenAction.Execute();
```

To enable the Synchronize screen, call this registration method:

```
MBOInformationRegistry.Instance.DatabaseClassType =
typeof(MyDatabase);
```

Index

A

actions 95
    alert 93
    connection 93
    exit 93
    logout 93
    navigation back 93
    operation 94
    persist 94
    synchronize 94
ActiveSync, installing and configuring 8
alert action 93
arbitrary find method 44
AttributeMetadata 69
AttributeTest 44

C

callback handler 62
certificates 41
ClassMetadata 69
client database 67
client object API 39
code, generating 12
common APIs 58
CompositeTest 44
connection action 93
ConnectionProfile 39, 41
ConnectionProfile.EncryptionKey 41
controls:adding to a screen 75
CreateDatabase 67
customization:of a controller 76

D

database:client 67
DatabaseMetadata 69
debugging 31
Delete operation 49
DeleteDatabase 67
dependencies 11
deploying
    configuring ActiveSync for 8

    message-based applications 34
Device Application Designer
    generated solution files and projects 21
DLL dependencies 11
documentation roadmap
    document descriptions 2

E

entity states 54, 55
exceptions
    server-side 67
exit action 93

G

generated API help 1, 39
GenerateId 62
generating code 12
generation gap pattern 22
GetLastSynchronizedTime() 69

H

HasPendingOperations operation 74

I

installing
    Microsoft ActiveSync 8
    synchronization software 8
IsSynchronized() 69

K

KeyGenerator 62

L

layout 80
libraries 25
local business object 52
LocalKeyGenerator 62
LoginToSync 40, 78

W
widget events 77

Windows Mobile Device Center 9

Sybase Unwired Platform