Modifying Infor ERP SyteLine
Copyright © 2010 Infor
All rights reserved. The word and design marks set forth herein are trademarks and/or registered trademarks of Infor and/or its affiliates and subsidiaries. All rights reserved. All other trademarks listed herein are the property of their respective owners.

Important Notices
The material contained in this publication (including any supplementary information) constitutes and contains confidential and proprietary information of Infor.
By gaining access to the attached, you acknowledge and agree that the material (including any modification, translation or adaptation of the material) and all copyright, trade secrets and all other right, title and interest therein, are the sole property of Infor and that you shall not gain right, title or interest in the material (including any modification, translation or adaptation of the material) by virtue of your review thereof other than the non-exclusive right to use the material solely in connection with and the furtherance of your license and use of software made available to your company from Infor pursuant to a separate agreement ("Purpose").
In addition, by accessing the enclosed material, you acknowledge and agree that you are required to maintain such material in strict confidence and that your use of such material is limited to the Purpose described above.
Although Infor has taken due care to ensure that the material included in this publication is accurate and complete, Infor cannot warrant that the information contained in this publication is complete, does not contain typographical or other errors, or will meet your specific requirements. As such, Infor does not assume and hereby disclaims all liability, consequential or otherwise, for any loss or damage to any person or entity which is caused by or relates to errors or omissions in this publication (including any supplementary information), whether such errors or omissions result from negligence, accident or any other cause.

Trademark Acknowledgements
All other company, product, trade or service names referenced may be registered trademarks or trademarks of their respective owners.

Publication Information
Release: Infor ERP SyteLine 8.02
Publication Date: December 9, 2010
# Table of Contents

**About This Guide** .......................... v
  For More Information ............................... v
  System Requirements and Prerequisite Knowledge ......................... vi
  Contacting Infor Support .......................... vi

**Toolset Overview** ............................. 1-1
  Summary of Tools ..................................... 1-2
    Opening Development Tools ............................ 1-3
    Accessing Development Tools Help .................... 1-3
  Setup and Configuration ................................ 1-4
    Configuration Manager ................................ 1-4
    Service Configuration Manager ....................... 1-6
  Database Tier ........................................ 1-7
    Application Schema Forms .............................. 1-7
    User-Extended Tables Forms ......................... 1-7
  Middle (Objects) Tier ................................ 1-8
    IDO Editing Forms ..................................... 1-8
    IDO Runtime Development Server ..................... 1-9
  Presentation Tier ..................................... 1-10
    WinStudio ............................................. 1-10
    FormControl ........................................... 1-10
  Application Event System ................................ 1-11
    Event System Forms .................................... 1-11
  Application Upgrade and Maintenance ................................ 1-12
    FormSync ................................................ 1-12
    App Metadata Sync .................................... 1-12
    App Metadata Transport ............................... 1-13
  Diagnostics .......................................... 1-14
    Replication Tool ....................................... 1-14
    Log Monitor .......................................... 1-14
    IDO Runtime Development Server ..................... 1-14
    WinStudio ............................................. 1-14

**Architectural Guidelines for Customers Modifying SyteLine** .............. 2-1
  Prerequisites and Assumptions ................................ 2-2
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminology</td>
<td>2-3</td>
</tr>
<tr>
<td>Customizations</td>
<td>2-4</td>
</tr>
<tr>
<td>Modifications</td>
<td>2-6</td>
</tr>
<tr>
<td>- Do Not Use SQL 2008 Merge Function</td>
<td>2-6</td>
</tr>
<tr>
<td>- Build Installations as Additive SQL Scripts</td>
<td>2-6</td>
</tr>
<tr>
<td>- General Guidelines</td>
<td>2-7</td>
</tr>
<tr>
<td>- UI – Forms</td>
<td>2-7</td>
</tr>
<tr>
<td>- Database – Transact SQL Code (Stored Procedures, Triggers, Functions, Constraints)</td>
<td>2-8</td>
</tr>
<tr>
<td>- Application Database Schema Changes (Table Structure)</td>
<td>2-8</td>
</tr>
<tr>
<td>- Reports</td>
<td>2-9</td>
</tr>
<tr>
<td>External Touch Points</td>
<td>3-1</td>
</tr>
<tr>
<td>- Overview</td>
<td>3-2</td>
</tr>
<tr>
<td>- External Touch Point Code</td>
<td>3-3</td>
</tr>
<tr>
<td>- Non-report Stored Procedures That Output an @Infobar Parameter</td>
<td>3-3</td>
</tr>
<tr>
<td>- Non-report Stored Procedures That Do Not Output an @Infobar Parameter</td>
<td>3-4</td>
</tr>
<tr>
<td>- Report Stored Procedures</td>
<td>3-5</td>
</tr>
<tr>
<td>- Using External Touch Points</td>
<td>3-7</td>
</tr>
<tr>
<td>- Version Upgrades and Service Packs</td>
<td>3-8</td>
</tr>
<tr>
<td>Dynamic IDOs</td>
<td>4-1</td>
</tr>
<tr>
<td>- Overview</td>
<td>4-2</td>
</tr>
<tr>
<td>- Stored Procedure Calls</td>
<td>4-3</td>
</tr>
<tr>
<td>- Collection Sources</td>
<td>4-5</td>
</tr>
<tr>
<td>Performance Issues</td>
<td>5-1</td>
</tr>
<tr>
<td>- Locking and Blocking</td>
<td>5-2</td>
</tr>
<tr>
<td>- Read Locks</td>
<td>5-2</td>
</tr>
<tr>
<td>- Transaction Size</td>
<td>5-4</td>
</tr>
<tr>
<td>- Deadlocks</td>
<td>5-7</td>
</tr>
<tr>
<td>- Cursors</td>
<td>5-8</td>
</tr>
<tr>
<td>- Avoid Cursors</td>
<td>5-8</td>
</tr>
<tr>
<td>- Examples</td>
<td>5-9</td>
</tr>
<tr>
<td>- Using Cursors</td>
<td>5-11</td>
</tr>
<tr>
<td>- Triggers</td>
<td>5-13</td>
</tr>
<tr>
<td>- Dynamic SQL</td>
<td>5-13</td>
</tr>
<tr>
<td>- Temporary Tables and Table Variables</td>
<td>5-14</td>
</tr>
<tr>
<td>- Miscellaneous Coding Tips</td>
<td>5-15</td>
</tr>
<tr>
<td>- Procedures for Recompiling</td>
<td>5-16</td>
</tr>
</tbody>
</table>
# Table of Contents

Data Types ................................................................. 5-17
Diagnosing Performance Problems .................................. 5-18
  Viewing SQL Commands Initiated by the Framework .............. 5-18
  Timing Tests in SQL Server Management Studio .................. 5-18

INSTEAD OF INSERT and AFTER UPDATE Triggers ................. 6-1
  Overview ................................................................. 6-2
  INSTEAD OF INSERT Triggers ........................................ 6-2
  AFTER UPDATE Triggers ............................................... 6-2
  Maintenance Form ..................................................... 6-3
  Maintenance Tasks .................................................... 6-4

Creating Custom Application Messages .............................. 7-1
  How the Toolset Generates Application Messages ................. 7-2
    Using GetErrorMessageSp to Build the Output Message ......... 7-2
  Using Stored Procedures to Set Up Custom Application Messages ... 7-4
    Message Numbers .................................................. 7-7
    Message Type ...................................................... 7-8
    Object Type ....................................................... 7-9

Licensing ................................................................. 8-1
  Overview ................................................................. 8-2
  Licensing New and Copied Forms .................................... 8-3
    Multi-Site .......................................................... 8-4
    Copy License Information ......................................... 8-4
  Licensing New and Copied IDOs ..................................... 8-4
    IsModuleLicensed Method .......................................... 8-5
    Sample Script ...................................................... 8-5
    Sample Event Handler ............................................. 8-6

APIs for Strong Data Types ............................................ 9-1

APIs for the Graph Component ........................................ 10-1

APIs for the Tree Component .......................................... 11-1

Sample Scripts with APIs for Strong Types ......................... A-1

Suggestions for Development Environment Configuration .......... B-1
  Maintain Separate Read-Only SyteLine Environments ............... B-2
  Use SourceSafe with IDO Metadata ................................ B-2
Use SourceSafe for Application Database Scripts ........................................... B-2

Numeric Constants Used in the Forms Database.............................................. C-1

Scripting UET Definitions ............................................................................... D-1
  Extracting UET Definitions from the Source Environment....................... D-2
  Applying the Scripts in the Target Environment...................................... D-3
  Testing the Changes .................................................................................... D-3
  The ExportUETClassSp Stored Procedure ................................................. D-4

Effects of Service Packs and Upgrades on Customizations and Modifications . E-1
About This Guide

This guide provides an overview of SyteLine development tools, guidelines for customizing SyteLine, and information on some common development tasks not covered in the online help for the framework.

For More Information

The most current version of all documentation is available on the Infor support Web pages (see “Contacting Infor Support” on page -vi). The following documents are related to SyteLine development:

*Guide to the Application Event System* This document provides information about how the application event system can be used to customize the way applications such as SyteLine perform. In most cases, custom events and event handlers can be used without having to write procedural code. This guide includes in-depth conceptual information, reference tables for implementing events and handlers, and detailed descriptions of how custom events and handlers are synchronized with existing ones.

*Integrating IDOs with External Applications* This document describes types of integration between SyteLine and external applications, as well as the prerequisites, setup, and protocols needed.

*Infor ERP SyteLine Reporting Guide* This document describes how SyteLine handles reports. It includes a description of the reporting system architecture, specifics of creating Crystal Reports for SyteLine, how to link custom reports into the SyteLine system, specifics of using the Infor ERP TaskMan and RunReport executables, how to distribute reports to internal and external users, and how to troubleshoot problems with reports.

*Infor ERP SyteLine Replication Reference Guide* This document describes SyteLine replication concepts, the system forms used to set up replication,
how replication works "behind the scenes," how to handle UETs with replication, and how to handle replication errors.

**Converting WinStudio Script Code from VBA to VB.NET** This document provides instructions for converting WinStudio script code from Visual Basic for Applications (VBA) to Visual Basic (VB.NET). SyteLine version 7.05 and earlier versions used VBA in global scripts, form scripts, and in-line scripts. Beginning with version 8.0, SyteLine requires VB.NET.

**Infor ERP SyteLine DataMap Guide** This guide describes the SyteLine DataMap, an analytical tool designed to help you learn and analyze relationships between SyteLine forms, collections, stored procedures, and database schema. It also allows you to track and record changes you make using user-defined fields or otherwise modifying the standard product. The DataMap consists of a set of SQL queries and Microsoft Excel® spreadsheets that list SyteLine forms and fields and their data relationships, default authorization levels, and so on.

**Infor ERP SyteLine Help Customization** You can customize the SyteLine online Help by creating your own Help topics in the application. This document explains the process. It is part of a ZIP file that includes the necessary help templates, style sheets, and so on.

**About Form Synchronization** This document describes FormSync, which is an aid to preserving customizations when you upgrade to a newer version of SyteLine or when you apply a service pack. FormSync also provides a set of development and administrative utilities with which you can gather information about customizations, copy customizations from one database to another, and generate SQL scripts to apply customizations to a remote database.

**Changed Programs List** This document lists all programs that changed from one service pack to the next.

---

**System Requirements and Prerequisite Knowledge**

For the most up-to-date list of software and hardware requirements for Infor products, see the *Guide to Technology*. This document also lists typical system administration tasks you should be familiar with before attempting to install and administer Infor products.

**Contacting Infor Support**

If you cannot find the answer to a question in this document or in the product's online help, contact Infor Global Support at our support Web site [http://www.infor365.com](http://www.infor365.com). Simply log in and select your product area. From this site, you can do the following:
Gain easy access to critical support resources like the Knowledge Center, software updates, and release notes.
Obtain documentation for your product.
Log and track incidents.
Access the tools you need to keep your software running efficiently.
Link to additional Infor resources.

Planning Your Communication
To make sure the correct analyst is assigned to your case and to expedite the resolution of your questions, please have the following information available when you call us:
- Company name and phone number
- SyteLine version release and point release
- Database software version and release, if applicable
- Platform or environment (example: Windows® XP)
- If you received an error message, the full message text and error number
- If you are calling back on an existing case, the Case Number

Signing Up for Support
If you are not currently using support and would like more information on support options, please call your customer account representative. If you are not sure who your account representative is, contact Infor Customer Service.
Chapter 1: Toolset Overview

The development framework includes tools for creating and editing forms, IDOs, and application events and event handlers. The framework also provides administrative and development utilities for managing a development project and deploying an application.

This chapter gives an overview of the functionality of the toolset. Topics include:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary of Tools</td>
<td>1-2</td>
</tr>
<tr>
<td>Setup and Configuration</td>
<td>1-4</td>
</tr>
<tr>
<td>Database Tier</td>
<td>1-7</td>
</tr>
<tr>
<td>Middle (Objects) Tier</td>
<td>1-8</td>
</tr>
<tr>
<td>Presentation Tier</td>
<td>1-10</td>
</tr>
<tr>
<td>Application Event System</td>
<td>1-11</td>
</tr>
<tr>
<td>Application Upgrade and Maintenance</td>
<td>1-12</td>
</tr>
<tr>
<td>Diagnostics</td>
<td>1-14</td>
</tr>
</tbody>
</table>
Summary of Tools

The table below lists the tools available for customizing your SyteLine system and summarizes the basic functionality of each tool.

<table>
<thead>
<tr>
<th>Development Area</th>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup and configuration</td>
<td>Configuration Manager</td>
<td>Set up a development or production environment that includes databases, servers, and application-specific settings required by WinStudio, IDOs, FormControl, FormSync, and other framework components.</td>
</tr>
<tr>
<td></td>
<td>Service Configuration Manager</td>
<td>Configure framework services:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Event Service</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Replication</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• TaskMan</td>
</tr>
<tr>
<td>Database tier</td>
<td>Application Schema Forms</td>
<td>Define application schema metadata and extended schema metadata needed for generating custom table triggers.</td>
</tr>
<tr>
<td></td>
<td>User-Extended Tables Forms</td>
<td>Extend existing application database tables and add custom user fields to forms.</td>
</tr>
<tr>
<td>Middle (objects) tier</td>
<td>IDO Editing Forms</td>
<td>Create and edit IDOs and IDO property classes for the middle tier. Import and export IDOs, IDO property classes, and related custom assemblies.</td>
</tr>
<tr>
<td></td>
<td>IDO Runtime Development Server</td>
<td>Access the IDO Runtime as a free-standing executable on a local machine rather than as a Windows service on a utility server. View diagnostic information.</td>
</tr>
<tr>
<td>Presentation (WinStudio) tier</td>
<td>WinStudio</td>
<td>Create and edit forms and global form objects. Diagnose problems in form operations.</td>
</tr>
<tr>
<td></td>
<td>FormControl</td>
<td>Manage version control for WinStudio forms and global form objects in a vendor-developer or site-developer environment.</td>
</tr>
<tr>
<td>Application event system</td>
<td>Event System Forms</td>
<td>Create and edit application events and event handlers. Monitor activities in the application event system.</td>
</tr>
<tr>
<td>Application upgrade and maintenance</td>
<td>FormSync</td>
<td>Preserve customizations to WinStudio forms and global form objects in upgrades to new versions of an application. Manage form customizations with a set of utilities.</td>
</tr>
<tr>
<td></td>
<td>App Metadata Sync</td>
<td>Synchronize metadata in your system with metadata from SyteLine upgrades or from add-on (third-party) products. Metadata defines IDOs, IDO-related objects, and objects in the application event system.</td>
</tr>
<tr>
<td></td>
<td>App Metadata Transport</td>
<td>Import or export metadata that defines IDOs, IDO-related objects, and objects in the application event system.</td>
</tr>
</tbody>
</table>
### Development Area

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Diagnostics</strong></td>
<td></td>
</tr>
<tr>
<td>Replication Tool</td>
<td>View, correct, and resubmit inbound and outbound replication errors. View the status of sites linked to a site for replication.</td>
</tr>
<tr>
<td>Log Monitor</td>
<td>View all activity logged on a utility server, including messages related to WinStudio, TaskMan, IDOs, replication, and the application event system.</td>
</tr>
<tr>
<td>IDO Runtime Development Server</td>
<td>View IDO XML request and response documents, all application activity, and information about user sessions logged on to configurations.</td>
</tr>
<tr>
<td>WinStudio</td>
<td>View messages related to WinStudio activity.</td>
</tr>
</tbody>
</table>

### Opening Development Tools

Except as noted, development tools are installed on Administrative clients and can be opened from the Tools menu (Start > All Programs > Infor > Tools). The tools are available on all utility servers.

### Accessing Development Tools Help

All these development tools have their own Help files, accessible from within each tool. In addition, you can access the Help from within Infor ERP SyteLine, by selecting Customizing Forms from the Help menu.

In rare cases, usually when you are working in a local development environment, it is possible for the file that controls where the system looks for the tools Help files to be overwritten. If this happens, you will get a file-not-found error message, and Internet Explorer will display in the Address field a URL like the following:


If this happens, you must update the MG_Help.xml file. For the procedure to do this, see the Infor ERP SyteLine Installation Guide.
Setup and Configuration

Configuration Manager

A configuration encapsulates settings that:

- Provide access to framework databases
- Point to a middle-tier Web server
- Define a particular application, including the location of an IDO Request Service and specifications for how forms look and behave

Settings can be specified for a deployed application or for a development environment. A configuration is required to use WinStudio, FormControl, FormSync, and other framework components.

The specifications that comprise configurations are stored in an .xml file that resides on a system Web server.

Database Configuration

Types of framework databases are described in the following table.

<table>
<thead>
<tr>
<th>Database</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application database</td>
<td>Stores the primary data maintained by end users of an application. In an inventory application, for example, the database might contain product numbers and quantities. An application database also contains:</td>
</tr>
<tr>
<td></td>
<td>Metadata defining objects in the application event system. Developers specify metadata in the event system forms.</td>
</tr>
<tr>
<td></td>
<td>Stored procedures, user-defined data types, and other SQL Server objects that support application functionality.</td>
</tr>
<tr>
<td></td>
<td>Tables for storage of administrative information, such as user names and sign-in information.</td>
</tr>
<tr>
<td>Forms database</td>
<td>Stores specifications that define forms and form-related objects. Developers define forms in WinStudio design mode. In run mode, WinStudio retrieves the specifications to create the functional forms.</td>
</tr>
<tr>
<td>Objects database</td>
<td>Stores metadata that defines IDOs, IDO property classes, and IDO-related custom assemblies. Developers specify metadata using the IDO editing forms.</td>
</tr>
<tr>
<td>Templates database</td>
<td>Stores specifications for form templates defined in WinStudio design mode. Developers creating forms in a deployed application can access a vendor-supplied templates database.</td>
</tr>
</tbody>
</table>
Three databases are required in a deployed application: an application database, a forms database, and an objects database.

To support development environments in which FormControl is used, you can configure run-time and master versions of forms databases and templates databases, as well as archive databases for deleted forms and templates. (For more information, see “FormControl” on page 1-10).

Visual SourceSafe can be configured for form development with FormControl and for middle-tier development with the IDO editing forms. (For more information, see “FormControl” on page 1-10 and “IDO Editing Forms” on page 1-8.)

Application Configuration

Each configuration of databases must include a specification for an application. The ability to create bundles of application-specific settings allows you to deploy the information contained in a configuration of databases in different ways, customizable in Configuration Manager.

Configurable settings include:

- The relative path to the IDO Request Service. At run time, this path is appended to the Web server root URL to form a complete Web server URL
- The application name, version, copyright notice, and company name
- Graphics files for the splash screen, application icons, and wallpaper for the main WinStudio window
- Colors and fonts to be used when displaying WinStudio forms
- User-interface text for the Help menu, deletion prompts, and the displayed name of the Actions menu
- The URL and startup file for the application's online Help
- A display of validation error indicators

The application-specific specifications are stored in the same .xml file that stores other configuration settings. This file resides on a Web server.

Web Server Configuration

The middle tier of the framework resides on a Web server and is the control center for all requests to the system. Each configuration must include a specification for a Web server in the form of a root URL. At run time, the root URL is joined with a relative path specified in the application settings. WinStudio and other clients access the IDO Request Service by means of the complete path.

For more information, see the online Help for Configuration Manager.
Service Configuration Manager

To set up framework services, use the Service Configuration Manager. Windows services support the application event system, non-transactional replication, background task execution, and other framework operations.

<table>
<thead>
<tr>
<th>Service</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Infor Framework Event Service</td>
<td>Application event defaults for each available configuration</td>
</tr>
<tr>
<td>Infor Framework Replicator and Infor Framework Replication Queue Listener</td>
<td>Bootstrap configuration name, inbound bus configuration name, and other replication defaults and options</td>
</tr>
<tr>
<td>Infor Framework TaskMan</td>
<td>Configurations monitored by TaskMan and, if necessary, DSN settings that a configuration uses for reporting</td>
</tr>
</tbody>
</table>

For more information, see the online Help for Service Configuration Manager.
Database Tier

Application Schema Forms

If you create a custom table or a custom column on a standard table, you can use the application schema forms to define the application schema metadata and extended schema metadata that is needed for generating table triggers. The forms are located in the WinStudio Explorer under Master Explorer > System. These form names begin with Application Schema.

The system uses the metadata, which is stored in tables in the application database, when generating table triggers. The forms can also generate SQL scripts used to distribute the metadata to other databases.

For more information, see Help for the application schema forms and "Maintaining Application Schema Metadata" in the Help.

User-Extended Tables Forms

The User-Extended Tables (UET) feature gives you the ability to extend existing application database tables and add custom user fields to forms in the application. Use the feature to store information that is not currently in the application database schema.

These forms are located in the WinStudio Explorer under Master Explorer > System > UET.

For more information, see Help for the UET forms and "User Extended Tables Overview" in the Help.
Middle (Objects) Tier

IDO Editing Forms

NOTE: When editing IDOs with Visual SourceSafe (VSS) enabled, we recommend that you use the IDO Runtime Development Server, rather than the IDO Runtime Service on the utility server (see “IDO Runtime Development Server” on page 1-9). This allows you to check IDOs in and out without generating errors.

Middle-tier IDOs (Intelligent Data Objects) provide client software with access to the application database. The IDO editing forms allow you to create, edit, import, and export IDOs and related objects. The forms are located in the WinStudio Explorer under Master Explorer > System > IDO Editing.

All WinStudio interactions with an application database use the properties and methods of IDOs. In general, an IDO is designed to serve a specific WinStudio form. An IDO can also be used by clients other than WinStudio. Clients and IDOs interact by means of HTTP or HTTPS, exchanging documents in an XML request-response protocol.

A property of an IDO represents a datum. The value of a property may be retrieved from a table column in the application database, computed by a SQL expression, or passed in as a temporary datum typically used as an input parameter for specialized processing in the middle tier. A subcollection property specifies a child IDO collection that is filtered from a parent IDO.

IDOs support property classes, which encapsulate property attributes for reuse.

IDO methods consist of built-in procedures for all routine data maintenance tasks and procedures defined by users as custom methods. Custom methods are typically stored procedures implemented in Transact-SQL (T-SQL). Custom methods can also be implemented in .NET assemblies as IDO extension classes.

In the IDO editing forms, you specify metadata defining IDO properties, methods, property classes, project membership, and other attributes of IDOs. Metadata is stored in the objects database, which is deployed with the application. At run time, IDOs are built from the metadata and executed by the IDO Runtime service on a utility server.

The IDO editing forms support a multi-developer environment and integration with Microsoft Visual SourceSafe.

Two features of the IDO editing forms aid in upgrading customized IDOs:

- The Access As mechanism identifies the creator of IDO metadata objects and restricts the ability to edit IDO metadata to the creator. Customizations and add-on modules are kept separate from the base version, allowing automatic upgrades (see “App Metadata Sync” on page 1-12).
The **Extend** option allows you to create a new IDO that is based on, and inherits from, an existing IDO.

The **Extend and Replace** option allows you to replace a base IDO with an extended IDO. In this case, all client requests that specify the base IDO are redirected through the extended IDO.

For more information, see the online Help for the IDO editing forms.

**IDO Runtime Development Server**

**NOTE:** When editing IDOs with Visual SourceSafe (VSS) enabled, we recommend that you use the IDO Runtime Development Server, rather than the IDO Runtime Service on the utility server. This allows you to check IDOs in and out without generating errors.

The IDO Runtime Development Server can be run in place of the Infor Framework IDO Runtime Service. The Infor Framework IDO Runtime Service, normally a Windows service on a utility server, is the gateway to the application database. It accepts IDO XML requests, executes the requests, and returns a response. The WinStudio client and most other framework components connect to an instance of the Infor Framework IDO Runtime Service to access data in the application database.

IDO Runtime Development Server, a stand-alone executable, performs the same functions as the Infor Framework IDO Runtime Service but in a local development environment. It allows you to:

- View IDO XML request and response documents.
- View application activity in the integrated Log Monitor (see “Log Monitor” on page 1-14).
- View information about user sessions logged on to configurations.
- Remove user sessions logged on to a configuration.
- Clear the IDO metadata cache to allow changes to metadata to take effect.

For more information, see the online Help for IDO Runtime Development Server.
Presentation Tier

WinStudio

WinStudio is deployed on client workstations to present forms to end users of an application. Developers create and edit forms using WinStudio’s Design mode. All elements of the presentation tier are specified in Design mode, including form components, such as grids and combo boxes, event handlers, validation procedures, variables, and filters. WinStudio stores specifications for these objects in a forms database, which is deployed with the application.

In creating a form, the developer can select an IDO that returns the required primary collection (a result set) for the form from the application database. The developer selects properties that the IDO publishes and provides other attributes for each property (label, validation, drop-down list, and so forth).

For more information, see the online Help for WinStudio (from the Help menu, select Customizing Forms).

FormControl

FormControl is a version-control tool for developers using WinStudio to create and edit vendor versions of forms and form-related objects. It is not intended for use with customizations made at non-developer levels of editing permission (Basic or Full User), but—starting with SyteLine 8.02—developers with Site Developer editing permissions can use it for form development work.

FormControl tracks vendor versions of forms, global form objects, and form templates. It supports access to a repository of versions and an archive of deleted objects. You can specify integration with Visual SourceSafe in Configuration Manager. Although use of FormControl is optional, we recommend using this tool when several developers are working on an application.

The tool’s file name is FormControl.exe. The file is located on Administrative clients in the main program folder (the same folder that contains WinStudio.exe).

For more information, see the online Help for FormControl.
Application Event System

Event System Forms

Event system forms allow you to create custom events and event handlers in the multi-tier, application event system. The forms are located in the WinStudio Explorer under Master Explorer > System > Event System.

An application event can be generated when:

- An end user performs a particular action on a form and/or when a particular business process is activated
- A database calculation is performed
- Another event generates an event
- A certain amount of time has passed

An event can be generated in any tier:

- In the client tier, an event can be generated by a form through an event handler of the response type Generate Application Event.
- In the middle tier, an event can be generated by invoking an IDO method.
- In the database tier, an event can be generated by a stored procedure.
- In any tier, an event can generate another event, using the Generate Event action type.

There are more than twenty types of actions that an event handler can carry out, such as adding an entry to an audit log, calling a stored procedure, calling an IDO method, sending an e-mail message, and loading an IDO collection.

The Access As mechanism identifies the creator of application event system metadata and restricts the ability to edit metadata to the creator. Customizations and add-on modules are kept separate from the base version, allowing automatic upgrades (see “App Metadata Sync” on page 1-12).

For more information, see the online Help for the event system forms and Guide to the Application Event System on the support Web site.
Application Upgrade and Maintenance

FormSync

FormSync is an aid to preserving customizations to forms and global form objects when you upgrade an application to a newer version or install a service pack. The process of using FormSync is referred to as *form synchronization*. This process applies only to customizations made in WinStudio design mode.

FormSync also provides a set of utilities for administering customized versions of forms and global objects. The utilities support the following tasks:

- **Copy customized versions.** You can copy customized versions from a development forms database to a production database.
- **Delete customizations.** You can delete customizations from a forms database.
- **Script customizations.** You can generate SQL scripts that represent customized versions in a forms database. You can then run the scripts on a remote forms database to apply the customizations to the database.
- **List customizations.** You can create informational lists of customized versions in a forms database.
- **Compare objects.** You can compare one object with another.
- **Synchronize with a Site or Group version.** You can synchronize a Site or Group version of an object with other customized versions. In this type of synchronization, the Site or Group version, rather than a vendor version, serves as the base-level version.

For more information, see the online Help for FormSync, the *Infor ERP SyteLine Installation Guide*, and *About Form Synchronization* on the support Web site.

App Metadata Sync

Metadata processed by the App Metadata Sync utility is of two types:

- Metadata that defines IDOs, IDO property classes, and IDO custom assemblies
- Metadata that defines events, event handlers, and event actions in the application event system

App Metadata Sync synchronizes metadata from new sources with metadata that exists in a configuration. New metadata usually comes from two sources:

- Application upgrades and service packs
- Add-on (third-party) products

Only metadata associated with user-selected Access As identifiers is processed during synchronization. (For more information, see “IDO Editing Forms” on page 1-8.)
New metadata is stored in a source configuration; metadata to be updated is stored in a target configuration. App Metadata Sync deletes all metadata with a specified identifier in the target and copies new metadata with the specified identifier from the source to the target. The product of synchronization is the upgraded target database.

The utility’s file name is AppMetadataSync.exe. The file is located on Administrative clients in the main program folder (the same folder that contains WinStudio.exe).

For more information, see the online Help for the App Metadata Sync utility.

App Metadata Transport

The App Metadata Transport utility processes the same types of metadata as the App Metadata Sync utility. App Metadata Transport allows you to:

- Import application event system or IDO metadata from an .xml file to a target database.
- Export application event system or IDO metadata from a source database to an .xml file.

The utility’s file name is AppMetadataTransport.exe. The file is located on Administrative clients in the main program folder (the same folder that contains WinStudio.exe).

For more information, see the online Help for the App Metadata Transport utility.
Diagnostics

Replication Tool

Use the Replication Tool to:

- View, correct, and resubmit inbound and outbound replication messages that contain errors.
- View the status of sites linked to a site for replication.

For more information, see the online Help for Replication Tool and Replication Reference Guide on the support Web site.

Log Monitor

Log Monitor provides a single, consolidated view of all activity logged on a utility server. It shows information in real time from all parts of the framework, including WinStudio, TaskMan, IDOs, replication, and the application event system.

To open Log Monitor, do one of the following:

- Click the Log Monitor button on the WinStudio Diagnostics toolbar.
- On a utility server, in the application's main program folder (the same folder that contains WinStudio.exe), open the file LogMonitor.exe.

For more information, see the online Help for Log Monitor.

IDO Runtime Development Server

For types of diagnostic information provided by the utility, see “IDO Runtime Development Server” on page 1-9.

WinStudio

WinStudio supports a comprehensive, configurable diagnostics feature that allows you to view messages related to WinStudio activity. For information about configuring message logging and viewing messages, see the online Help for WinStudio.
This chapter provides recommendations and guidelines from the SyteLine Technology and Architecture team to developers making changes to SyteLine at a customer site. The chapter focuses on issues related to maintaining customer changes across service packs and version upgrades. The chapter augments other sources of information such as developer training and online Help. Topics include:

<table>
<thead>
<tr>
<th>Topics</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites and Assumptions</td>
<td>2-2</td>
</tr>
<tr>
<td>Terminology</td>
<td>2-3</td>
</tr>
<tr>
<td>Customizations</td>
<td>2-4</td>
</tr>
<tr>
<td>Modifications</td>
<td>2-6</td>
</tr>
</tbody>
</table>
Prerequisites and Assumptions

Readers should be familiar with the SyteLine toolset, Microsoft SQL Server, and Crystal Reports. Readers should have completed the SyteLine developer training course and have some experience working with the toolset. Terms such as IDO, form scope type, UET, and stored procedure are not explained here.
Terminology

The terms *customization* and *modification* will be used as follows:

**Customization**

Customizations are changes made at a customer site using the following mechanisms:

- WinStudio changes to forms (at a site default, group, and user level)
- User-Extended Tables (UETs)
- User-defined Fields (UDFs)
- IDOs (by means of the IDO editor forms)

These changes can automatically be migrated forward in conjunction with service packs and upgrades to the base product.

Customizations do *not* include changes to:

- SQL database schema
- .asp pages in the Data Collection module
- T-SQL stored procedures, triggers, or functions

Some groups use the term “personalization” for this class of changes.

**Modification**

Modifications are changes to:

- Vendor version (scope type 0) forms
- .asp pages in the Data Collection module
- SQL database schema apart from the UET mechanism
- T-SQL stored procedures, triggers, or functions

The SyteLine toolset does not automatically carry forward such changes in the service-pack process or the version upgrade process.

**Modifications and customizations as applied to reports**

Changes to reports represent a special case, combining customizations and modifications, as defined above. One component of a report is its front-end form, which can be customized like other forms; changes to the other components (the Crystal Reports .rpt file, data source stored procedure, and background task definition data) must follow the processes recommended for modifications. For more information, see “Reports” on page 2-9.
Customizations

We recommend that, to the extent possible, developers making changes to SyteLine at a customer site restrict their changes to those classified as customizations. Such changes will be carried forward in subsequent SyteLine upgrades and service packs without the need for additional, site-specific installation scripts or procedures. Note that SyteLine’s customization features allow developers to implement solutions for most site-specific needs.

In the service-pack process and the version upgrade process, the SyteLine toolset does not automatically carry forward changes classified as modifications. To implement and maintain modifications, we advise you to create T-SQL scripts to be applied with service packs and upgrades, as described in the section “Modifications” on page 2-6.

In the figure below, customizations made at the top three scope levels are supported in service packs and upgrades.

Customizations at the site level (scope type 1), group level (scope type 2), and user level (scope type 3) can be made to:

- Forms
- Variables
- Menus
- Strings
- Validators
- Scripts
- Property Class Extensions
- Component Classes
- UETs
- UDFs
Customizations at the three levels are made through the WinStudio user interface. Customizations include making copies of forms and global objects, creating new forms and global objects, and changing forms and global objects.

FormSync provides reports on such customizations, copies customizations between forms databases, generates SQL export scripts for customizations, and offers options for handling customizations during the upgrade process. The toolset does not, however, attempt to anticipate changes made to the base SyteLine product at the vendor level (scope type 0).

IDO changes are also upgraded automatically, as the IDO definitions you create are marked as separate from the base application and other add-on modules.

For changes to SyteLine that require special treatment in the service-pack and upgrade processes, see the following "Modifications" section.
Modifications

Modifications to SyteLine—that is, changes to .asp pages, SQL Server database schema, forms at scope type 0, and T-SQL stored procedures, triggers, and functions—can be overwritten by a service pack or an upgrade. (For details, see Appendix E, “Effects of Service Packs and Upgrades on Customizations and Modifications”). Modifications must be reapplied to the updated databases manually or with SQL scripts.

Several basic application objects can be modified. Each specific object carries its own set of rules and caveats. The different objects are defined below and the rules and caveats are also provided.

- **User Interface – Forms**
- **Database – Transact SQL Code**
- **Database – Application Database Schema**
- **Reports – Crystal Reports .rpt files, background task definitions data, and report forms**

Do Not Use SQL 2008 Merge Function

SyteLine does not support the SQL 2008 merge function.

Build Installations as Additive SQL Scripts

If you are building add-on modules for redistribution, use SQL scripts, when possible, to intelligently apply modifications to the databases.

Two tables in SyteLine provide an aid to tracking modifications by means of scripts.

- Both the forms and the application databases have a table named ProductVersion containing a column also named ProductVersion. The modification installation should check the ProductVersion before applying changes.
- The application database also has a table named ProductPatches, which contains a history of service packs applied to the database. If the version is known to be incompatible, the installation should prevent the user from updating the database. If the version is unknown (newer), you may want the install to generate warnings to the user, or you may want to document in an associated readme file the targeted version of the base application.

The following sections include more suggestions, specific to different tiers, about how the installation scripts can be made additive or safe to re-apply after upgrades.
General Guidelines

When preparing to update or upgrade modifications, keep in mind the following general guidelines:

1. To make installation as easy as possible, avoid requiring client updates unless absolutely necessary. Things that would require client updates include forms that include user controls and client-side images.

2. Write a complete list of all new and modified objects, including database objects and reports, so that you can identify potential upgrade and service-pack issues.

3. Use a consistent naming convention for all objects that are not replacements of base application objects. Names should clearly identify and group objects by type. Such a naming convention usually consists of a prefix, or system for a set of prefixes, affixed to object names.

4. These guidelines provide a set of techniques and practices that will minimize, but not eliminate, the amount of rework required by a modification when a new service pack or version is released. These are not intended to be a strict set of rules that must never be violated. The best solution to some problems might conflict with these guidelines.

5. As detailed in Appendix B, “Suggestions for Development Environment Configuration,” we recommend that you keep a separate environment containing the original SyteLine version you are modifying, so you can construct your own reports and queries to identify differences between your modifications at the various tiers and the original version.

UI – Forms

For modifications to the user interface and/or forms:

1. Do not modify base versions of a form. Form modifications receive scope type 0 when you are logged on to WinStudio as a Vendor Developer (edit level 4). Site, group, and user scopes are reserved for customizations only. Vendor Developer edit level should only be used for debugging purposes. Changes made at this level cannot be automatically migrated forward.
2. If major customizations to a form are needed, copy the form to a new name and customize the copy. When you need to alter a form, you have two options. You can copy it to a new name, and make the customization under the new name, or you can customize the original form. There are trade-offs to consider with each approach.

Generally, you should simply customize the form. That way, all references to the form from other forms and from the Explorer will point to your customized version of it.

The exception is when you want to alter the form radically, changing the business logic in substantial fashion so that the purpose of the form has changed. Such changes are difficult to migrate automatically into the next release. It is better to copy the form to a new name and maintain it yourself in future releases. Note that when you apply a service pack or version upgrade and then synchronize the forms databases, FormSync does not process renamed customized forms.

Database – Transact SQL Code (Stored Procedures, Triggers, Functions, Constraints)

If you need to modify a T-SQL database object, you have two options. You can modify the base object or copy the object to a new name and make your modification there. Modifying the base version introduces upgrade issues; the drawback of renaming the object is that you must then modify any other objects that call it to point to your renamed version. Use judgment as to which approach is best, and follow these guidelines:

1. Whenever possible and practical, when it is necessary to add trigger logic to a table, add a new trigger rather than modify the base trigger. This eliminates issues with upgrades and service packs, allowing your changes to be retained intact. Base triggers update the base table, which can lead to multiple firings of an update trigger. SyteLine code protects our base triggers against this, as described in “AFTER UPDATE Triggers” on page 6-2.

2. Avoid modifying code in base stored procedures, functions, and triggers. Note that, in SyteLine, each stored procedure contains code that allows you to redirect calls to the procedure to a custom procedure. For more information, see Chapter 3, "External Touch Points."

3. When modifying base application code, bracket your changes in comment lines. Use an easily-searchable keywords in all such comments.

Application Database Schema Changes (Table Structure)

When modifying application database schema:

1. Never remove base tables or columns.

2. Avoid schema changes to base tables. If you must extend a base product table, employ a UET. An alternative technique that might be preferable is to create a new table with the same key column(s) as the base product table plus any new columns required by the modification.
Service packs and upgrades do not remove a column you added to a base table, but it is possible that your new columns plus new columns added by means of a service pack or upgrade might cause the table to exceed the SQL maximum row size.

3. **Explicit collation should not be specified for columns in SQL scripts.** Explicit collation uses the server’s default collation, which is what the SyteLine tables also use.

## Reports

Reports are composed of three elements:

- A Crystal Reports report definition file (.rpt)
- A SQL Server stored procedure that the report is based on
- A background task definition stored in the application database (BGTaskDefinitions) table.

When a base report needs to be modified, you should copy and rename each of these elements, and then apply the modifications to the copy.

**NOTE:** In SyteLine, each stored procedure contains code that allows you to redirect calls to the procedure to a custom procedure. See Chapter 3, “External Touch Points.”

Also, there is typically a form that allows users to launch the report. You can customize these forms using standard WinStudio techniques.

If you need to produce multiple variations of a base report, consider building a version of the report that presents the user with all the various options they might need as components on the report form, corresponding to parameters passed to the .rpt file and the stored procedure.
In SyteLine, each stored procedure contains a block of code that calls a custom stored procedure if such a procedure exists. This code block is referred to as an external touch point. The custom stored procedure is referred to as an external stored procedure.

External touch points provide a way to modify stored procedures without altering standard system code. A modified stored procedure can coexist with the standard stored procedure but be invoked by calls to the standard stored procedure.

A benefit of using external touch points is that they simplify the process of applying version upgrades and service packs. With external touch points, you do not have to reapply custom code to standard stored procedures. (See “Version Upgrades and Service Packs” on page 3-8.)

This chapter covers concepts and practices associated with the use of these external touch points. Topics include:

<table>
<thead>
<tr>
<th>Topics</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>3-2</td>
</tr>
<tr>
<td>External Touch Point Code</td>
<td>3-3</td>
</tr>
<tr>
<td>Using External Touch Points</td>
<td>3-7</td>
</tr>
<tr>
<td>Version Upgrades and Service Packs</td>
<td>3-8</td>
</tr>
</tbody>
</table>
Overview

External touch point code checks for the existence of an external stored procedure. If one exists, it runs the procedure, passing (in and out) all the parameters of the standard stored procedure. It then performs some cleanup processes.

The name of an external stored procedure must consist of the prefix EXTGEN_ and the name of the standard stored procedure. The form is:

```
EXTGEN_standard_stored_procedure_name
```

An external touch point can conditionally run the standard stored procedure after executing the external stored procedure, depending on the type of stored procedure and the return value from the external procedure (see “External Touch Point Code” on page 3-3).

External touch points are not implemented in functions, triggers, or views. All stored procedures contain external touch points.
External Touch Point Code

There are three types of code blocks for external touch points. The types are for:

- Non-report stored procedures that output an @Infobar parameter
- Non-report stored procedures that do not output an @Infobar parameter
- Report stored procedures

Most stored procedures have an @Infobar output parameter to return error messages. Code blocks in the sections below are from SyteLine stored procedures. Procedure names are in bold.

Non-report Stored Procedures That Output an @Infobar Parameter

The following is an example of code that calls a non-report stored procedure that outputs an @Infobar parameter.

```
-- Check for existence of Generic External Touch Point routine
-- (this section was generated by SpETPCodeSp and inserted by
-- CallETPs.exe):
IF OBJECT_ID(N'dbo.EXTGEN_AllocSp') IS NOT NULL
BEGIN
    DECLARE @EXTGEN_SpName sysname
    SET @EXTGEN_SpName = N'dbo.EXTGEN_AllocSp'
    EXEC @EXTGEN_Severity int
    DECLARE @EXTGEN_Severity = @EXTGEN_SpName
    @CurPoNum, @ProcessLevel OUTPUT
    , @PromptMsg OUTPUT
    , @Buttons OUTPUT
    , @Infobar OUTPUT
    IF @EXTGEN_Severity <> 1
    RETURN @EXTGEN_Severity
END
-- End of Generic External Touch Point code.
```

Remarks

- The code block is at the top of the stored procedure, immediately after the AS keyword or the following BEGIN keyword.
Note that even if the external stored procedure exists, execution of the body of the standard stored procedure proceeds if the external routine returns a special value of 1.

Non-report Stored Procedures That Do Not Output an @Infobar Parameter

The following is an example of code that calls a non-report stored procedure that does not output an @Infobar parameter.

```sql
-- Check for existence of Generic External Touch Point routine
-- (this section was generated by SpETPCodeSp and inserted by
-- CallETPs.exe):
IF OBJECT_ID(N'dbo.EXTGEN_CoPackingSlipLoadSp') IS NOT NULL
BEGIN
    DECLARE @EXTGEN_SpName sysname
    SET @EXTGEN_SpName = N'dbo.EXTGEN_CoPackingSlipLoadSp'
    -- Invoke the ETP routine, passing in (and out)
    -- this routine's parameters:
    EXEC @EXTGEN_SpName
        @TPckCall
        , @CoNum
        , @CustNum
        , @CoitemCustNum
        , @CoitemCustSeq
        , @Whse
        , @FromCoLine
        , @ToCoLine
        , @FromCoRelease
        , @ToCoRelease
        , @FromDate
        , @ToDate
        , @Stat
        , @BatchId

    -- ETP routine must take over all desired functionality
    -- of this standard routine:
    RETURN 0
END
-- End of Generic External Touch Point code.
```

Remarks

The code block is at the top of the stored procedure, immediately after the AS keyword or the following BEGIN keyword.

Note that if the external routine exists, execution of the body of the standard stored procedure never proceeds because of the return value of 0.
Report Stored Procedures

The following is an example of code that calls a report stored procedure.

```sql
-- Check for existence of Generic External Touch Point routine
-- (this section was generated by SpETPCodeSp and inserted
-- by CallETPs.exe):
IF OBJECT_ID(N'dbo.EXTGEN_Rpt_JournalControlNumberSp') IS NOT NULL
BEGIN
    DECLARE @EXTGEN_SpName sysname
    SET @EXTGEN_SpName = N'dbo.EXTGEN_Rpt_JournalControlNumberSp'
    -- Invoke the ETP routine, passing in (and out)
    -- this routine's parameters:
    EXEC @EXTGEN_SpName
        @CurId,
        @Type,
        @ShowForeignAmounts,
        @PrefixStarting,
        @PrefixEnding,
        @SiteStarting,
        @SiteEnding,
        @YearStarting,
        @YearEnding,
        @PeriodStarting,
        @PeriodEnding,
        @ControlNumberStarting,
        @ControlNumberEnding,
        @DisplayHeader

    IF @@TRANCOUNT > 0
        COMMIT TRANSACTION
    EXEC dbo.CloseSessionContextSp @SessionID = @RptSessionID
    -- ETP routine must take over all desired functionality
    -- of this standard routine:
    RETURN
END
-- End of Generic External Touch Point code.

Remarks

This code block comes after the call to InitSessionContextSp and after the last statement that sets a parameter for the first time.

Reports contain special code to create a transaction, set the isolation level, initialize a session context, and then commit the transaction and close the session context at the end. If the external routine exists, it should run after
these structures have been set up. Since the commit and close statements are not reached, they are duplicated before returning.

Also, most report parameters are adjusted to convert NULL values; the external routine should run after these adjustments are made, so it is passed the adjusted values. Thus, you do not have to include similar adjustments in external routines.

Note that if the external routine exists, execution of the body of the standard stored procedure never proceeds.
Using External Touch Points

When you develop a stored procedure that is to be called by external touch point code, be sure to:

- Name the procedure `EXTGEN_standard_stored_procedure_name`.
- Ensure that parameters in the procedure are the same (in name, number, and order) as the parameters in the standard procedure.
- For a non-report procedure that outputs an @Infobar parameter, set the return value of the new procedure to 1 if you want to execute the body of the standard stored procedure after the new procedure completes. Return a value other than 1 to skip execution of the body of the standard stored procedure. (With other types of stored procedures, you cannot execute the body of the standard stored procedure after the new procedure completes.)
Version Upgrades and Service Packs

In a major version upgrade, such as from SyteLine 7.05 to 8.0, external stored procedures are dropped. You must re-add your external stored procedure to the application database.

When you apply a service pack, external stored procedures are not dropped.

In both version upgrades and service packs, if the signature of a standard stored procedure in the new version is different from the signature in the old version, you must modify the external stored procedure accordingly. A signature change is a change in the names, number, or order of parameters.
Chapter 4: Dynamic IDOs

This chapter provides information about working with dynamic IDOs. Topics include:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>4-2</td>
</tr>
<tr>
<td>Stored Procedure Calls</td>
<td>4-3</td>
</tr>
<tr>
<td>Collection Sources</td>
<td>4-5</td>
</tr>
</tbody>
</table>
Overview

Dynamic IDOs invoke stored procedures directly from the application database or work with collections of records directly from tables in the application database. The operations are not based on IDO metadata in the objects database.

NOTE: Dynamic IDOs provide an alternative to changing IDOs. Prior to SyteLine 8, this feature provided a convenient way to eliminate upgrade issues related to IDO modifications. However, in SyteLine 8, changes to IDOs upgrade automatically, and so utilizing dynamic IDOs may not be the best option for invoking stored procedures or working with collections of records.

To invoke a stored procedure, you normally specify an IDO name and a method on the IDO, which in turn invokes the stored procedure. With the dynamic IDO feature, you specify the keyword SP! instead of an IDO name and a stored procedure name instead of an IDO method name.

A collection source in SyteLine is normally a dedicated IDO. With the dynamic IDO feature, you can specify the keyword TABLE! and a table name to retrieve a collection. In a form, components can be bound to columns in the table.

Standard and dynamic IDOs behave the same at run time. However, standard IDOs, defined in the IDO editing forms, support additional features as a source for a collection:

- An IDO can publish data from multiple database tables.
- An IDO property name may be different from the name of the column to which it is bound.
- IDO properties can be calculated values that are not represented by a column in the database table.
- An IDO can publish nonpersistent properties, which are often used to pass user input as parameters to a process.
- An IDO can publish subcollection properties, which are required to use WinStudio's subcollection features.
Stored Procedure Calls

For the basic steps to create an event handler, see the WinStudio Help (from the Help menu, select Customizing Forms).

In the following example, a push button on a WinStudio form inserts a value in a component. An event handler on the button invokes a stored procedure that retrieves the value. The procedure takes three input parameters and one output parameter. (The example does not show development of the button, the custom event, or the stored procedure.)

To use a dynamic IDO in an event handler:

1. In the Event Handler Properties dialog box, specify Method Call as the response type.

2. Click theParmsrow, and then click the ellipsis (...) button.

3. To open theEdit Method Calldialog box, click Type Specific Parameters.

4. In theEdit Method Calldialog box, specify SP! in theIDO field.

5. In theMethod box, specify the name of the stored procedure.

   In the case of this example, the name isGetContextID.
6. Click **Parameters** and create (or specify) the parameters.

Parameters for a dynamic IDO are specified in the same way as parameters for an IDO method.

For this example, in the **Edit Method Parameters** dialog box, create the following parameters:

- **VAR C(FileTypeEdit)**
- **VAR C(SubdirectoryEdit)**
- **VAR C(ContextStringEdit)**
- **RVAR P(ContextId)**

![Edit Method Parameters dialog box]

Complete specification for the example:

```
SP!.GetContextID(PARMS(VAR C(FileTypeEdit), VAR C(SubdirectoryEdit), VAR C(ContextStringEdit), RVAR P(ContextId)))
```

Note that in the complete specification, a period (.) follows the exclamation point (!). The system adds the period.
Collection Sources

To use a dynamic IDO as a collection source for a form: In the Collection row of the Collections property sheet, specify TABLE! and a table name (with the syntax TABLE!table_name).

To bind a component to a column in the table: in the Binding row of the Component property sheet, specify object and a column name (with the syntax object.column_name).

**NOTE:** Like standard IDOs, the TABLE! feature of dynamic IDOs operates with standard SyteLine tables. You can create a compatible table by reverse engineering a SyteLine table, including required columns, triggers, constraints, and keys, and create a new table.

In this example, the collection source is the table AuditLog.

The table column InWorkflow is bound to the component InWorkflowGridCol.

The primary collection consists of rows from the table AuditLog. Values from the column InWorkflow appear in the grid column component InWorkflowGridCol.
Chapter 5: Performance Issues

This chapter examines some common performance issues that may arise in customizing SyteLine and offers suggestions on performance improvements.

The *Infor ERP SyteLine System Administration Guide* contains additional performance-related information on SQL Server settings and maintenance, diagnostics, hardware, and SyteLine data maintenance.

Topics in this chapter include:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locking and Blocking</td>
<td>5-2</td>
</tr>
<tr>
<td>Deadlocks</td>
<td>5-7</td>
</tr>
<tr>
<td>Cursors</td>
<td>5-8</td>
</tr>
<tr>
<td>Triggers</td>
<td>5-13</td>
</tr>
<tr>
<td>Dynamic SQL</td>
<td>5-13</td>
</tr>
<tr>
<td>Temporary Tables and Table Variables</td>
<td>5-14</td>
</tr>
<tr>
<td>Miscellaneous Coding Tips</td>
<td>5-15</td>
</tr>
<tr>
<td>Procedures for Recompiling</td>
<td>5-16</td>
</tr>
<tr>
<td>Data Types</td>
<td>5-17</td>
</tr>
<tr>
<td>Diagnosing Performance Problems</td>
<td>5-18</td>
</tr>
</tbody>
</table>
Locking and Blocking

Locking prevents users from reading data being changed by other users, and prevents multiple users from changing the same data at the same time. SQL Server enforces locking automatically. Locking can occur at record, page, or table level.

Blocking occurs when one user holds a lock and a second user requires a conflicting lock type. This forces the second user to wait, blocked by the first. Typically, the second user sees an hourglass while trying to process or save records. Most blocking problems happen because a single process holds locks for an extended period of time, causing a chain of blocked processes.

There are three ways to decrease record blocking:

- Take the minimum number of locks necessary.
- Hold locks for the shortest possible time.
- Improve general performance of T-SQL code to reduce the amount of time that a user holds a lock.

Techniques to reduce blocking are designed to optimize read operations, reduce the size and duration of transactions, and enhance code efficiency.

CAUTION: Changes to default locking behavior in SyteLine and SQL Server should be made with caution. Always consider possible adverse effects on the integrity of data in transactions before implementing changes.

Read Locks

Since most locks are taken during read operations, efforts to minimize read locks can reduce blocking and improve performance.

READ UNCOMMITTED Isolation Level

The SQL Server transaction isolation level controls locking behavior in transactions, determining the degree to which one transaction is isolated from other transactions. With the transaction isolation level READ UNCOMMITTED, SQL Server does not place locks on data being read. Also, the read process does not block on data that is locked but retrieves the data even if the transaction that created it is not yet committed. Users do not have to wait for long-running transactions that access the same set of data to complete before their queries can complete.

To set the base isolation level for loading IDO collections, use the Collection Read Mode field of the Process Defaults form. This setting applies to queries that load primary collections, secondary collections, and lists, and to in-collection validations. It also applies to background-task queries that
generate reports and to background-task stored procedures unless the setting is overridden on the **Background Task Definitions** form.

**NOTE:** The setting does not affect SELECT operations coded in stored procedures (IDO method calls).

To set the isolation level for individual reports and stored procedures, use the **Background Task Definitions** form. This setting overrides the **Process Defaults** setting. See the topic "Setting Transaction Isolation Levels" in the SyteLine online help. To access this setting, a stored procedure must include code like the following:

```sql
BEGIN TRANSACTION
SET XACT_ABORT ON
-- Set the isolation level specified for the background task
-- or use the system default.
IF dbo.GetIsolationLevel(N'PastDueJobOperationsReport') = N'COMMITTED'
    SET TRANSACTION ISOLATION LEVEL READ COMMITTED
ELSE
    SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
```

You can override the base setting at the form level in WinStudio design mode. For more information, see the topic "Set the Read Mode for a Collection" in the Help for WinStudio Design mode (from the **Help** menu, select **Customizing Forms**).

**READUNCOMMITTED Locking Hint**

The table locking hint READUNCOMMITTED reduces locking and can reduce blocking problems in stored procedures, functions, and triggers.

READUNCOMMITTED applies only to SELECT statements and overrides the current transaction isolation level in the statement for which it is specified. The locking hint is similar to the READ UNCOMMITTED isolation level. SQL Server does not place locks on the data being read, and the read process does not block on data that is locked but retrieves the data even if the transaction that created it is not yet committed. Dirty reads are possible.

Consider the use of READUNCOMMITTED in selects on table columns that contain relatively static data and data that does not affect business logic, such as description strings. Data is likely to be relatively static if it is not maintained by code in the database but is maintained manually through a form or is updated only infrequently. If all columns in a SELECT statement or all columns contained in the **WHERE** clause of a SELECT statement contain such data, then READUNCOMMITTED can generally be used safely.
Example:

```sql
SELECT @SequenceBy = periods_seq.sequence_by
FROM periods_seq WITH (READUNCOMMITTED)
WHERE periods_seq.fiscal_year = @ControlYear
AND periods_seq.jour_control_prefix = @ControlPrefix
```

NOTE: By default, SQL Server operates at the READ COMMITTED isolation level, which does not allow dirty reads and does apply locks on records.

**Transaction Size**

A long-running transaction may hold locks on records for an unacceptably long time, blocking other processes. You can reduce the duration of locks by decreasing the number of records in the transaction. This is accomplished by dividing a record set into discrete subsets and processing them sequentially in separate transactions.

In general, transactions should be started as late as possible and committed as soon as possible while still preserving data integrity. Statements that modify data—INSERT, UPDATE and DELETE—should never appear outside a transaction block.

**Save Each Modified Row in a Separate Transaction**

You can specify that WinStudio save each modified row in a separate transaction. By default, without this setting, all modified rows are sent to the middle tier to be processed within a single transaction.

Saving one row per transaction can alleviate blocking problems in some forms. Whether the setting enhances performance depends on the complexity of a form's save operation and the speed of the network connection. Forms that require highly complex saves might benefit from the setting, especially if connection speed is adequate. Forms that require simple save operations are less likely to benefit, especially if the additional network traffic required in saving one row at a time slows the system.

In default WinStudio behavior, if one row fails, the entire transaction is rolled back. When you save each row in a separate transaction and a row fails, all previous rows remain committed.

NOTE: If your form design requires that all modified rows be committed as a unit, with processing on the unit before and/or after a save operation, saving rows in separate transactions might not be appropriate. If an error occurs on a row, some rows might be committed and others might not be committed.

For instructions on setting this feature, see the topic "Save One Row Per Transaction" in the Help for WinStudio Design mode.
For a list of SyteLine forms that use this feature, see "Improving Performance" in the Infor ERP SyteLine System Administration Guide.

Non-Transactional IDO Methods

Many IDO methods run within the context of a transaction. A method call starts a transaction before the stored procedure itself is called. To reduce transactional overhead and blocking, you can set a method to run without an outer transaction.

Methods that do not update data in the database can likely be run without an outer transaction.

Methods that handle transactions internally can also be run without an external transaction. In this case, you must manually code smaller transactions in a stored procedure and commit them or roll them back individually. The general procedure is:

```
BEGIN TRANSACTION
<transaction block>
IF <no errors> COMMIT TRANSACTION
ELSE ROLLBACK TRANSACTION
```

To set an IDO method to run without a transaction, clear the Transactional check box in the New Method form or the IDO Methods form. The method then does not start a transaction before the stored procedure is called. To set a method to run within a transaction, select the Transactional check box.

Custom IDO Methods

The following code sample is a hand-coded IDO method that controls transaction scope.

The function EntryMethod is an extension-class method that is not transactional. It queries records based on the itemNo parameter and calls another method named WorkerMethod. WorkerMethod represents a function (either a hand-coded, extension-class method, or an IDO method based on a stored procedure) that updates the database. WorkerMethod is transactional.

If WorkerMethod succeeds, any updates that it performed are committed in the database. If it fails (raises an exception or returns an error result), any updates that it performed are rolled back, but only the updates for the call that failed. Updates performed by previous calls are still committed; that is, the scope of the transaction is a single call to WorkerMethod.
Public Function EntryMethod(ByVal itemNo As String, ByRef Infobar As String) As Integer

    Dim loadResp As LoadCollectionResponseData
    Dim invResp As InvokeResponseData
    Dim result As Integer = 0

    loadResp = Me.LoadCollection(
        "prop1, prop2", _
        String.Format("item = {0}", SqlLiteral.Format(itemNo)), _, 
        ",", -1)

    For row As Integer = 0 To loadResp.Items.Count - 1
        invResp = Me.Invoke("WorkerMethod", loadResp(row, 
            "prop1").Value, loadResp.Items(row).ItemID, _
            String.Empty)

        ' check result if necessary
        If invResp.IsReturnValueStdError() Then
            Infobar = invResp.Parameters(2).Value ' get Infobar
            ' Add Exit Function to quit processing after failure;
            ' otherwise we continue to process batch.
            End If
    Next

    Return result
End Function

Forms

Custom transaction handling should not be attempted in WinStudio forms or other clients. Transaction handling should be used only at the IDO level or the Transact-SQL level.
Deadlocks

Deadlocking occurs when two user processes have locks on separate objects and each process tries to acquire a lock on the other process's locked object. When this happens, SQL Server ends the deadlock by automatically aborting one of the processes, allowing the other to continue. The aborted transaction is rolled back, and an error message is sent to the user of the aborted process.

To help avoid deadlocks, use the following techniques:

- Arrange for multiple transactions to place locks on data resources in the same order. For example, two transactions lock the operations of a job, one-by-one, but each uses a different order. The possibility of a deadlock can be avoided by altering the transactions so they lock operations in the same order. In designing a new transaction, you should review existing transactions that operate on the same data.

- With parent-child data structures, lock the parent record before accessing child records. This has the effect of serializing access to the entire structure as a unit, avoiding deadlocks among child records. The issue of lock ordering must be considered only at the parent-record level.

- During transactions, do not allow user input. Collect input before the transaction begins.

- Keep transactions short and within a single batch (see “Transaction Size” on page 5-4).

- If appropriate, use as low an isolation level as possible for the user connection running the transaction (see “Read Locks” on page 5-2).

- Ensure that processes that do not go through IDOs initialize their sessions using the stored procedure `InitSessionContextSp` and close their sessions using `CloseSessionContextSp`. If a process does not perform this initialization, then it might try to access the session variables for another user’s session. This can lead to blocking or application errors.
Cursors

SQL Server is a batch-oriented system, designed to process large data sets. Processing records one at a time in a cursor increases overhead and decreases performance.

If a procedure that processes a large number of records is identified as slow and it has a cursor, the cursor is likely the problem. Rewriting the procedure to eliminate the cursor often results in a significant performance gain. The gain is proportional to the number of records in the cursor.

Avoid Cursors

Cursor loops that simply perform accumulations or assignments should normally be removed. Use of bulk INSERT and UPDATE statements (single statements that process more than one record) can improve performance over cursor loops. Bulk statements support the same JOIN and WHERE clauses that the SELECT statement supports.

Some cursor loops require simplification before it becomes apparent that they can be removed. Look for inefficient code in the cursor:

- Redundant SELECT statements are executed inside the loop when the record does not depend on the surrounding cursor variables. The statements can be moved elsewhere in the system.
- Some statements can be moved into the trigger or are already in the trigger. Moving code into a trigger should be done with caution because you may lower the performance of a different procedure.
- Some SELECT statements can become part of the DECLARE CURSOR statement because there is a one-to-one relationship between the records. Moving an individual SELECT statement to the cursor allows bulk reading of records and gives SQL Server the opportunity to minimize database accesses.
- Use INNER joins instead of LEFT OUTER joins if possible. Note that with LEFT OUTER joins, a filter in the FROM clause executes before the join, whereas the filter in the WHERE clause executes after the join.

In many cases, it is safe to use an INNER join only if there is an explicit foreign key relationship on the joined columns and the column you are joining on is non-nullable. Otherwise, you could drop some rows from the result set.
Examples

Example A shows a cursor loop in a trigger. Example A1 shows a rewritten version without the cursor.

Example A

```sql
DECLARE
    @RowPointer RowPointerType,
    @Site SiteType,
    @ReportsToSite SiteType

DECLARE site_hierarchyIupCrs CURSOR LOCAL STATIC
FOR SELECT
    ii.RowPointer,
    ii.site,
    ii.reports_to_site
FROM inserted ii
LEFT OUTER JOIN deleted AS dd ON
    dd.RowPointer = ii.RowPointer

OPEN site_hierarchyIupCrs
WHILE @Severity = 0
BEGIN -- cursor loop
    FETCH site_hierarchyIupCrs INTO
        @RowPointer,
        @Site,
        @ReportsToSite
    IF @@FETCH_STATUS = -1
        BREAK
    IF @Site = @ReportsToSite
        BEGIN
            EXEC @Severity = GetErrorMessageSp
                'E=ReportsToSiteSameAsSite', 5, @Infobar OUTPUT
        END
END

CLOSE site_hierarchyIupCrs
DEALLOCATE site_hierarchyIupCrs
```

Example A1

```sql
IF EXISTS(
    SELECT 1 FROM inserted ii
    WHERE ii.site=ii.reports_to_site
) BEGIN
    EXEC @Severity = dbo.MsgAppSp @Infobar OUTPUT,
```
The cursor loop in Example B is removed in Example B1.

**Example B**

```sql
DECLARE del_termDelCrs CURSOR LOCAL STATIC
FOR SELECT
d.delterm,
d.RowPointer
FROM deleted d

OPEN del_termDelCrs
WHILE @Severity = 0
BEGIN -- cursor loop
  FETCH del_termDelCrs INTO
    @Delterm,
    @RowPointer
  IF @@FETCH_STATUS = -1
    BREAK
  -- Make sure this delterm is not being used
  -- in the coitem table.
  BEGIN
    IF EXISTS ( SELECT 1 FROM coitem WHERE delterm = @Delterm
    and coitem.ship_site = @site)
      EXEC @Severity = GetErrorMessageSp 'DeltermDel1', 18, _
      @Infobar OUTPUT
  END
  -- Make sure this delterm is not being used
  -- in the citemh table.
  BEGIN
    IF EXISTS ( SELECT 1 FROM citemh WHERE delterm = @Delterm
    and citemh.ship_site = @site)
      EXEC @Severity = GetErrorMessageSp 'DeltermDel2', 18, _
      @Infobar OUTPUT
  END
END

CLOSE del_termDelCrs
DEALLOCATE del_termDelCrs
```
Example B1

-- Make sure this delterm is not being used in the coitem table.
IF EXISTS ( SELECT 1 FROM
    coitem INNER JOIN deleted AS dd
    ON coitem.delterm = dd.delterm
    and coitem.ship_site = @Site
)
    EXEC @Severity = dbo.GetErrorMessageSp 'DeltermDel1', 18, _
        @Infobar OUTPUT

-- Make sure this delterm is not being used in the citemh table.
IF EXISTS ( SELECT 1 FROM
    citemh INNER JOIN deleted AS dd
    ON citemh.delterm = dd.delterm
    and citemh.ship_site =@Site
)
    EXEC @Severity = dbo.GetErrorMessageSp 'DeltermDel2', 18, _
        @Infobar OUTPUT

Using Cursors

If a cursor loop cannot be replaced with bulk statements, you can take steps to reduce inefficiencies in the code. The first is to minimize data updates. It is better to perform one UPDATE that processes 100 records than it is to perform 100 separate UPDATEs. This approach reduces trigger processing. Each time you execute a trigger, SQL Server can evaluate the execution plan and recompile it—a time-consuming process.

A trade-off is to store intermediate results in a table variable (or if necessary, a temporary table) within the loop. After the loop, you can then perform a bulk statement to move the changes into the database. This course ensures that the SyteLine tables are locked for a minimum period and that trigger calls are reduced. A table variable generally contains columns to establish a unique join to the real table and columns for each changed database column.

It is important to make sure that enough indexes have been defined for the table variable to produce a good execution plan.

Other tips for using cursors:

- Avoid selecting unused columns.
- Minimize the number of OPEN statements. These statements can consume significant time.
- Move IF statements that restrict processing from inside the loop to the WHERE clause of the DECLARE CURSOR statement. This reduces the amount of data written to SQL Server’s tempdb.
Use dynamic cursors instead of static cursors when shared or exclusive locks are to be acquired on the cursor select. Static cursors retrieve the entire result set at the time the cursor is opened, locking each row of the result set at open time. Dynamic cursors do not retrieve rows until they are fetched; locks are not acquired on the rows until they have been fetched.
Triggers

The following are some tips for using triggers:

- Keep trigger code to a minimum.
- Avoid the use of cursors in triggers.
- If it is necessary to use a cursor in a trigger, perform validations before the cursor loop using NOLOCKs (when feasible). If errors are detected, roll the transaction back.
- Move blocks of code that are not in the common code path to stored procedures and call them conditionally from the trigger.
- INSTEAD OF INSERT and AFTER UPDATE triggers are used with almost all tables in SyteLine. If you modify a base table or its _all table, you should regenerate the INSTEAD OF and AFTER triggers with the Trigger Management form. For more information, see Chapter 6, “INSTEAD OF INSERT and AFTER UPDATE Triggers.”

Dynamic SQL

Reports and utilities that have many input parameters should use dynamic SQL if possible. Users can leave many of the input options blank; however, the report's stored procedure contains these options as part of the WHERE clause. If many of the parameters are blank, you can save time with a statement that does not use them in the WHERE clause, because SQL Server can then generate a better execution plan.

Use the `dbo.AndSqlWhere` routines to prevent SQL injection attacks.
Temporary Tables and Table Variables

When you design temporary tables for global or per-connection use, consider the possibility of creating indexes for the tables. Most temporary tables do not need, nor can they use, an index, but some larger temporary tables can benefit from indexes. A properly designed index on a temporary table can be as great a benefit as a properly designed index on a standard database table.

For best performance, if you need a temporary table in your T-SQL code, try to use a table variable instead of a temporary table.

- Table variables are created and manipulated in memory rather than in SQL Server's tempdb, making them faster.
- Table variables in stored procedures result in fewer compilations than do temporary tables, and transactions using table variables last only as long as the duration of an update on the table variable, requiring less locking and logging.

Some considerations might dictate that a temporary table be used instead of a table variable:

- Table variables can only be referenced within a procedure. Called subprocedures cannot reference them directly.
- Nonclustered indexes cannot be created on table variables other than system indexes created for primary or unique constraints.
Miscellaneous Coding Tips

The following are miscellaneous tips to help you write code that is efficient and performs well.

- To defer and reinitiate APS processing when inserting records into, or deleting records from, tables whose INSERT or DELETE triggers update APS-related tables, use the following at the start of the process:

  ```
  EXEC @Severity = dbo.ApsSyncDeferSp @Infobar = @Infobar output
  ```

  At the end of the process, use:

  ```
  EXEC @Severity = dbo.ApsSyncImmediateSp @Infobar = @Infobar output
  ```

  Examples of such tables are job, jobmatl, CO, and coitem.

- Use IF EXISTS() instead of the following construct:

  ```
  SELECT @rowpointer = xxx.rowpointer FROM xxx
  IF @rowpointer IS NULL THEN
  ```

- Take advantage of the built-in SQL Server query index optimizer by specifying the columns of the index in your SELECT statements. The following statement:

  ```
  SELECT co_num, co_line, co_release FROM coitem
  ```

  is better than the statement:

  ```
  SELECT * FROM coitem
  ```

- Use the `dbo` prefix in EXEC statements:

  ```
  EXEC @Severity = dbo.MsgAppSp @Infobar OUTPUT, 'E=ReportsToSiteSameAsSite'
  ```

- Where a join or a subquery can perform the same task in a query, generally the join is faster. This is not always the case, however; test the query using both methods to determine which is faster.

- For updates, only commit "real" changes; that is, only perform an update if the data actually changed. For example:

  ```
  WHERE coitem.qty_ordered <> @buffer.qty_ordered
  ```

- When too many tables are joined, SQL Server sometimes creates a suboptimal execution plan. SQL Server tries to optimize database access and the resulting plan can be inefficient. Splitting the join and storing the results in a table variable can increase performance.
Procedures for Recompiling

SQL Server does not use precompiled code. It compiles Transact-SQL source code into an execution plan that can be executed quickly. It stores the compiled code in memory for use by all users. It continues to use a compiled version until algorithms determine that a newer version is needed.

Reasons to recompile include the following:

- The maximum number of procedures that can be stored in memory has been reached and the oldest accessed procedure is removed from memory.
  
  In this case, the only way to diminish the frequency of recompilation is to add more memory.

- Tables referenced in a procedure have changed enough that the execution plan might not be reliable.
  
  In this case, SQL Server keeps statistics about every table/index. It uses these statistics to create the execution plan, determining what tables are accessed and what indexes are used to join the tables. Once enough data changes have been made to a table, SQL Server recompiles the procedure to make sure that it has the most efficient execution plan.

- Code includes a temporary table that has gone out of scope, and SQL Server creates a new one.
  
  In this case, dynamic activities cause SQL Server to reconsider the execution plan. A statement like

  ```sql
  SELECT * INTO #item FROM item WHERE 1 = 2
  ```

  is quick, but SQL Server recompiles this procedure every time. If you use such a statement, place it at the top of the procedure. Table variables do not cause this behavior and are at least as efficient as temporary tables.
Data Types

Choosing the appropriate data types can affect how quickly SQL Server can SELECT, INSERT, UPDATE, and DELETE data.

- Always choose the smallest data type you need to store data in a column.
  For example, if you are going to store only the numbers 1 through 10, then the `tinyint` data type is more appropriate than the `int` data type.
  You should similarly evaluate `char` and `nvarchar` data types, and specify the minimum number of characters for character columns. This allows SQL Server to store more rows in its data and index pages, reducing the amount of input-output needed to read them. Also, it reduces the amount of data moved from the server to the client, reducing network traffic and latency.

- If the text data in a column varies greatly in length, use the `nvarchar` data type instead of the `char` data type.
  Although the `nvarchar` data type has slightly more overhead than the `char` data type, the amount of space saved by using `nvarchar` over `char` on variable length columns can reduce input and output, improving overall SQL Server performance.

- If you need to store large strings less than 4,000 characters, use the `nvarchar` data type. For strings over 4,000 characters, use `nvarchar(MAX)`.

- If you have a column that is designed to hold only numbers, use a numeric data type, such as `integer`, instead of the `nvarchar` or `char` data type. Numeric data types generally require less space to hold a numeric value than does a character data type. This helps reduce the size of the columns, and can boost performance when the columns are searched (WHERE clause) or joined to another column.
Diagnosing Performance Problems

Viewing SQL Commands Initiated by the Framework

To see SQL commands run by the middle tier, open Log Monitor on a utility server or, in a development environment, open IDO Runtime Development Server.

- SQL commands that are executed by the IDO Runtime can be viewed in the Log Monitor or IDO Runtime Development Server with **Type = SQL**.
- The log shows each SQL command that is executed and its start time.
- The log shows stored procedure names. You can enable WinStudio diagnostics to show the values of parameters in calls initiated by WinStudio.
- Running a SQL command in SQL Server Management Studio should show the same execution time. This test helps eliminate WinStudio or IDOs as the problem. If a problem persists when a command is called directly, then the problem is not in the framework but in the back-end code.

Timing Tests in SQL Server Management Studio

For timings, use PRINT statements and wrap the SQL statement with BEGIN TRAN and ROLLBACK TRAN. TRAN logic prevents having to recreate or reset data between each timing. PRINT is preferable to SELECT because it does not show in a grid and is easier to read. A typical PRINT statement looks like this:

```
Print 'a: ' + convert(nvarchar, getdate(), 114)
```

Use subtraction to get an approximation of how long sections of the code take to execute, or use DATEDIFF and a PreviousTime variable.

These statements can be placed throughout a procedure to isolate which section of the code uses the most time.

Two other useful values to print out are @@rowcount, which indicates how many records were processed by the bulk statement, and @@cursor_rows, which indicates how many records are in a cursor.
In SyteLine, almost all tables are accompanied by INSTEAD OF INSERT triggers and AFTER UPDATE triggers. The triggers contain standard functionality common to most tables and are generated from metadata. This chapter provides information about these triggers. Topics include:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>6-2</td>
</tr>
<tr>
<td>Maintenance Form</td>
<td>6-3</td>
</tr>
</tbody>
</table>

**NOTE:** This chapter gives only an overview of trigger design and generation in SyteLine. For detailed information, see the document *Insert and Update Trigger Generation*, available from our support Web site.
Overview

Trigger names follow these conventions:

- **INSTEAD OF INSERT triggers**: `table_nameInsert`
- **AFTER UPDATE triggers**: `table_nameUpdatePenultimate`
- **Existing insert and update triggers**: `table_nameIup`

If the generated triggers handle all the functionality required by a table during insertion and update, then they stand alone (although assisted by replication and audit-trail triggers when dictated by system settings). Otherwise, the `lup` trigger also exists, contains the special functionality, and executes immediately after (or during, in the case of insert) the generated trigger for the current operation. In some cases, the `lup` trigger handles only insert or only update, depending on the remaining code after removal of generated sections. However, the names have not been changed to remove the now-defunct operation designator `I` or `up`. `lup` triggers are maintained manually.

Generation of INSTEAD OF INSERT triggers and AFTER UPDATE triggers occurs automatically for most IDO-compatible tables:

- During every schema upgrade using the SyteLine Upgrade Wizard
- After migration using the SyteLine Database Migration utility
- After impacting the schema for User Extended Tables (UET) with the **UET Impact Schema** form

Generation can also be accomplished by using the **Trigger Management** form in special development situations.

**INSTEAD OF INSERT Triggers**

INSTEAD OF INSERT triggers fire first upon execution of an INSERT statement and take the place of the requested insert. Essentially, the row(s) requested to be inserted are moved to the “inserted” pseudo-table, and the trigger is executed. When the INSTEAD OF INSERT trigger executes an INSERT statement for the same table, the actual insert then occurs (the trigger itself does not recur), at which time any AFTER INSERT triggers are fired immediately. Then any remaining statements in the INSTEAD OF INSERT trigger execute.

**AFTER UPDATE Triggers**

AFTER UPDATE triggers fire immediately after an UPDATE statement. The row(s) requested to be updated are moved to the “inserted” pseudo-table, the original row(s) are moved to the “deleted” pseudo-table, and the trigger(s) are executed. First the `lup` trigger executes, if it exists, because it is marked **First** in the trigger order. Then the generated **UpdatePenultimate** trigger fires, if it exists. Next the audit-trail trigger fires, along with any custom triggers. Finally the replication trigger fires, because it is marked **Last** in the trigger order.
When any AFTER UPDATE trigger executes an UPDATE statement for the same table, the requested row(s) are updated and the same AFTER UPDATE trigger(s) are fired immediately (although the trigger itself does not recur). Then any remaining statements in the triggering trigger execute. However, to avoid redundant work, all standard (non-custom) triggers are protected so that they exit immediately if triggered from either an **Iup** or an **UpdatePenultimate** trigger (replication and audit-trail triggers never touch the triggering table). The following table explains these protections:

<table>
<thead>
<tr>
<th>Trigger</th>
<th>Updates Triggering Table</th>
<th>Protects Against</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_nameIup</td>
<td>Maybe</td>
<td>table_nameUpdatePenultimate</td>
</tr>
<tr>
<td>table_nameUpdatePenultimate</td>
<td>Yes</td>
<td>table_nameIup</td>
</tr>
<tr>
<td>table_nameUpdAudit</td>
<td>No</td>
<td>table_nameIup table_nameUpdatePenultimate</td>
</tr>
<tr>
<td>custom</td>
<td>Maybe</td>
<td>Depends on the design</td>
</tr>
<tr>
<td>table_nameIupReplicate</td>
<td>No</td>
<td>table_nameIup table_nameUpdatePenultimate</td>
</tr>
</tbody>
</table>

**Maintenance Form**

The **Trigger Management** form generates INSERT and UPDATE triggers for tables in a specified range of table names in the current application database.

Use this form if you have added custom columns to a standard table or if you have added a custom table. The form generates or regenerates triggers with standard functionality.

You can regenerate triggers multiple times, as the result is identical for identical schemas and metadata.
Maintenance Tasks

After adding or dropping columns, creating a new table, or modifying trigger functionality, you must regenerate the Insert and UpdatePenultimate triggers with the Trigger Maintenance form.

Adding or dropping column(s) from an existing table

When columns are added to or dropped from an existing table, the generated triggers must be regenerated to reflect the change. If audit columns (CreateDate, CreatedBy, RecordDate, UpdatedBy), shadow columns, or shadowed columns are added or dropped, the appropriate setting code becomes present or absent from the regenerated triggers.

Adding a new table

When a new table is added to the schema, the generated triggers usually must be created, in addition to or in place of the standard Iup trigger that is created manually. The following considerations are required:

- Decide which standard behavior (if any) of generated triggers is applicable and should be directed into generated triggers.
- Decide if any additional behavior is required that would need to be placed into an Iup trigger.

Changing trigger behavior of an existing table

For information about performing the tasks in this section, see the document Insert and Update Trigger Generation, available from our support Web site.
In SyteLine's application architecture, messages generated at the database layer are called *application messages*. This document explains how the system generates application messages, and the process you should follow to set up your own application messages. Topics include:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>How the Toolset Generates Application Messages</td>
<td>7-2</td>
</tr>
<tr>
<td>Using Stored Procedures to Set Up Custom Application Messages</td>
<td>7-4</td>
</tr>
</tbody>
</table>
How the Toolset Generates Application Messages

When primary key columns are defined in a table, and a process attempts to add a new record with key values identical to an existing record, then users should receive an associated table constraint error message. During its database layer operations (insert/update/delete), the toolset identifies such database errors as table constraints, and then generates predefined application messages that are displayed to users.

For example, on the Users form in SyteLine, an administrator might attempt to add a new user record with the same UserID value as an existing record. When trying to save the new record, the administrator receives the following message:

**The User Name entered already exists.**

This application message is displayed due to a unique key constraint of UserNamesIx1 defined in the UserNames table. When the system tries to process the requested INSERT operation, the toolset receives an error from the database due to this constraint violation. The toolset then builds an application message based on the error returned from database (in this case, a table constraint error), using message text strings predefined in the following tables:

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplicationMessages</td>
<td>Maintains actual message text strings. These messages can be literal text or consist of substitution keywords (&amp;1, &amp;2, and so forth).</td>
</tr>
<tr>
<td>ObjectMainMessages</td>
<td>Maintains mappings between a database object and its associated text strings, found in the ApplicationMessages table. Database objects include table constraint name, table column name, and so forth.</td>
</tr>
<tr>
<td>ObjectBuildMessages</td>
<td>Maintains substitution field information: how to replace the substitution keywords found in message text strings, if any, and the text strings with which to replace them.</td>
</tr>
</tbody>
</table>

Using GetErrorMessageSp to Build the Output Message

The actual output message is built using the stored procedure `GetErrorMessageSp`. For the error message in our example, the code would be as follows:
DECLARE @P1 nvarchar(2802)
EXEC GetErrorMessageSp N'UserNamesIx1', 17, @P1 output
SELECT @P1

This stored procedure requires three parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Direction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Name</td>
<td>IN</td>
<td>Database object name such as the name of a table's unique constraint (in our example, UserNamesIx1).</td>
</tr>
<tr>
<td>Message Type</td>
<td>IN</td>
<td>Predefined message type (see &quot;Message Type&quot; on page 7-8).</td>
</tr>
<tr>
<td>Message Result</td>
<td>OUT</td>
<td>Application message output.</td>
</tr>
</tbody>
</table>

Based on the given parameter values, the stored procedure follows these steps to generate the application message output:

1. Looks up the message text, which is defined in the ApplicationMessages table, based on the given database object name and message type (in this example, UserNamesIx1 and 17).
2. Looks up substitution substring objects for the specified database object (in this example, UserNamesIx1), as defined in the ObjectBuildMessages table. If any substitution substrings are found, search for the actual text strings for those substrings (again from the ApplicationMessages table).
3. Replaces any substitution keywords (&1, &2, and so forth) found in the message text string (from Step 1) with those substrings (from Step 2).
Using Stored Procedures to Set Up Custom Application Messages

The following stored procedures are used to add custom application message data to the application message database tables:

<table>
<thead>
<tr>
<th>Stored Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddApplicationMessageSp</td>
<td>Adds message and/or substitution substrings to the ApplicationMessages table.</td>
</tr>
<tr>
<td>AddObjectMainMessageSp</td>
<td>Adds a record to the ObjectMainMessages table. This record contains the database object name, message type, and associated message text strings predefined in the ApplicationMessages table.</td>
</tr>
<tr>
<td>AddObjectBuildMessageSp</td>
<td>Adds a record into the ObjectBuildMessages table. This record contains the database object name, message type, substitution sequence number, and corresponding substring text predefined in the ApplicationMessages table.</td>
</tr>
</tbody>
</table>

To add custom application messages, run the stored procedures in the following order:

1. To add message template text and substitution substrings to the ApplicationMessages table, run AddApplicationMessageSp.
2. To add a record to the ObjectMainMessages table (database object name and message type, associated with a message number for the message template text), run AddObjectMainMessageSp.
3. To add records to the ObjectBuildMessages table (database object name, message type, substitution sequence number, associated with a message number for the substring text) run AddObjectBuildMessageSp. Repeat this step for as many substitution substrings as required in the message template text.

Each of these stored procedures is described in more detail in the following sections.

AddApplicationMessageSp

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>@MessageNo</td>
<td>Unique message number (see “Message Numbers” on page 7-7)</td>
</tr>
<tr>
<td>Parameter</td>
<td>Notes</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>@MessageText</td>
<td>Message text or template with substitution keywords</td>
</tr>
<tr>
<td>@MessageLanguage</td>
<td>Language ID value (optional)</td>
</tr>
<tr>
<td></td>
<td>The default is NULL, which uses the system default language ID value of en-US (English [United States]).</td>
</tr>
</tbody>
</table>

**AddObjectMainMessageSp**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>@ObjectName</td>
<td>Database object name</td>
</tr>
<tr>
<td>@MessageType</td>
<td>Message Type, predefined in MessageTypes table (see “Message Type” on page 7-8)</td>
</tr>
<tr>
<td>@MessageNo</td>
<td>Message number defined with AddApplicationMessageSp</td>
</tr>
<tr>
<td></td>
<td>This requires you to execute this stored procedure after AddApplicationMessageSp.</td>
</tr>
<tr>
<td>@MessageSeverity</td>
<td>Severity value returned:</td>
</tr>
<tr>
<td></td>
<td>0 – Success</td>
</tr>
<tr>
<td></td>
<td>1- 5 – Warning</td>
</tr>
<tr>
<td></td>
<td>6-16 – Failure</td>
</tr>
<tr>
<td></td>
<td>Currently, the framework identifies the severity value of 0 or 16. If failed (16), the toolset displays the application message to users.</td>
</tr>
<tr>
<td>@ObjectMessageDesc</td>
<td>Brief description of this application message</td>
</tr>
<tr>
<td>@ObjectType</td>
<td>Object type, predefined in ObjectTypes table (see “Object Type” on page 7-9)</td>
</tr>
</tbody>
</table>

**AddObjectBuildMessageSp**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>@ObjectName</td>
<td>Database object name</td>
</tr>
<tr>
<td>@MessageType</td>
<td>Message Type, predefined in MessageTypes table (see “Message Type” on page 7-8)</td>
</tr>
<tr>
<td>@SequenceNo</td>
<td>Message substitution sequence number</td>
</tr>
<tr>
<td>@MessageNo</td>
<td>Message number defined with AddApplicationMessageSp</td>
</tr>
<tr>
<td></td>
<td>This requires you to execute this stored procedure after AddApplicationMessageSp.</td>
</tr>
</tbody>
</table>
Continuing the earlier example, the following material shows how to set up the following message based on the UserNamesIx1 unique key constraint on the UserNames table:

**The User Name entered already exists.**

The application message text string and its substrings are defined as follows:

<table>
<thead>
<tr>
<th>Constraint</th>
<th>UserNamesIx1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message</td>
<td>100093 (The &amp;1 entered already exists.)</td>
</tr>
<tr>
<td>Substitution</td>
<td>250039 (User Name)</td>
</tr>
</tbody>
</table>
You can add an application message like this by executing the following SQL script:

-- STEP 1:
-- Adding a message template text and its substitution substring to
-- "ApplicationMessages" table:
EXEC AddApplicationMessageSp 100093,
  'The &1 entered already exists.'
  Go

EXEC AddApplicationMessageSp 250039,
  'User Name'
  Go

-- STEP 2:
-- Adding a record, with the information on database object,
-- message type, and which message to associate with, into
-- "ObjectMainMessages" table:
EXEC AddObjectMainMessageSp 'UserNamesIx1',
  17, -- Constraint Message Type
  100093, 16,
  'Note: a message displayed when PRIMARY KEY ''UserNamesIx1''
  constraint fails',
  0 -- Table Object Type
  GO

-- STEP 3:
-- Adding a substitution information, for the database object, into
-- "ObjectBuildMessages" table:
EXEC AddObjectBuildMessageSp 'UserNamesIx1',
  17, -- Constraint Message Type
  1, 250039
  GO

Message Numbers

Infor has defined ranges of application message numbers that are reserved for specific features and users.

Numbers from 1 to 999,999 are reserved for Infor developers and customer developers.

- Infor toolset: 1 – 100,000
- SyteLine core application: 100,000 – 799,999
- Customer use: 800,000 – 999,999

Numbers from 1,000,000 to 7,999,999 are reserved for business partners:
- Single Source: 1,000,000 – 1,099,999
- Lake: 2,000,000 – 2,099,999
- RSVP: 3,000,000 – 3,099,999
- DRI: 4,000,000 – 4,099,999
- BTA: 5,000,000 – 5,099,999

Numbers from 8,000,000 to 8,999,999 are reserved for localization partners.

- Country Pack for Mexico: 8,500,000 – 8,510,000
- Country Pack for France: 8,510,001 – 8,520,000
- Country Pack for China: 8,520,001 – 8,530,000
- Country Pack for Thailand: 8,530,001 – 8,540,000

**NOTE:** The current product upgrade process drops all application messages and rebuilds with the updated tables. Therefore, you must reapply your SQL scripts to re-add custom messages after an upgrade.

### Message Type

The following message types apply to SyteLine systems:

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Message Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DELETE</td>
<td>Delete Message Type</td>
</tr>
<tr>
<td>1</td>
<td>EXISTS</td>
<td>Exists Message Type</td>
</tr>
</tbody>
</table>
| 2            | KEY          | Key Field Message Type  
Disallow Updates to Clustering key. 
Associated message might read: 
The &1 field is a key column and can not be updated. |
| 3            | NOUPD        | No Update Message Type  
Disallow any updates to rows in this table. Associated message might read: 
Rows in the &1 table are read-only and can not be updated. |
| 4            | CHILD        | Child Exists Message Type |
| 5            | SPST1        | Sp Step 1 Message Type  
Used in messages generated through stored procedures |
| 6            | SPST2        | Sp Step 2 Message Type  
Used in messages generated through stored procedures |
| 7            | SYSDEL       | Delete of System Type |
| 8            | VALID        | Data Entered is not Valid |
SyteLine's database tables have a number of constraints defined, so you can see many messages with message type 17 in its application message tables.

Object Type

The following object types identify the type of database object associated with each message:

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Object Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TABLE</td>
<td>Table object</td>
</tr>
<tr>
<td>1</td>
<td>COLUMN</td>
<td>Column object</td>
</tr>
<tr>
<td>2</td>
<td>SP</td>
<td>Stored procedure object</td>
</tr>
<tr>
<td>3</td>
<td>TRIGGER</td>
<td>Trigger object</td>
</tr>
</tbody>
</table>

The Object Type is informational only. During development and testing of a new trigger, you might receive a system error similar to the following:

**Error Message does not exist. Object:test_table, Type:3**

If so, use the text following "Object" to determine the Object Type to use when creating the appropriate application error message. In the example above, the object type would be TABLE.
Chapter 8: Licensing

SyteLine requires licensing for each module you want to use. This chapter contains information about licensing of new and copied forms and IDOs. Topics include:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>8-2</td>
</tr>
<tr>
<td>Licensing New and Copied Forms</td>
<td>8-3</td>
</tr>
<tr>
<td>Licensing New and Copied IDOs</td>
<td>8-4</td>
</tr>
<tr>
<td>IsModuleLicensed Method</td>
<td>8-5</td>
</tr>
</tbody>
</table>
Overview

After installing SyteLine, you can see records in the `ModuleMembers` table with a predefined set of forms and IDOs according to the modules you purchased. For example, the records for the form ABCAnalysis might look like the following table:

<table>
<thead>
<tr>
<th>ObjectType</th>
<th>ObjectName</th>
<th>ModuleName</th>
<th>OriginalModuleName</th>
<th>ModuleMemberSpec</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ABCAnalysis</td>
<td>SyteLineTrans</td>
<td>SyteLineTrans</td>
<td>&lt;...encrypted text..&gt;</td>
</tr>
<tr>
<td>0</td>
<td>ABCAnalysis</td>
<td>SyteLineTransMultiSite</td>
<td>SyteLineTrans-MultiSite</td>
<td>&lt;...encrypted text..&gt;</td>
</tr>
</tbody>
</table>

After applying a valid license document on the `License Management` form, the `ModuleMemberSpec` column values are updated with encrypted text. The encrypted text is based on the SQL Server machine name, SyteLine application database name, and the other table column values (`ObjectType`, `ObjectName`, `ModuleName`, and `OriginalModuleName`).

When requesting a form through WinStudio, the system decrypts the string text to verify that the form requested is valid for the environment (SQL Server machine name and application database name) and that the requesting user belongs to one of the modules (such as SyteLineTrans) assigned for the form. User-specific module records are kept in the `UserModules` table. When you add new records using the `User Modules` form, the framework again produces the encrypted text value for its `UserModuleSpec` column. The encrypted text consists of values from `UserId`, `ModuleName`, SQL Server name, application database name, and so forth, which must match when checking the validity of the requested form for the requesting user.
Licensing New and Copied Forms

If you create your own form, you must apply the license to it. For a newly created form (created with the New Form Wizard), the ModuleMembers table is populated as in the following table:

<table>
<thead>
<tr>
<th>ObjectType</th>
<th>ObjectName</th>
<th>ModuleName</th>
<th>OriginalModuleName</th>
<th>ModuleMemberSpec</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;FormName&gt;</td>
<td>CustomerForms</td>
<td>SyteLineTrans</td>
<td>&lt;..encrypted text..&gt;</td>
</tr>
</tbody>
</table>

The ModuleName value is CustomerForms, and the OriginalModuleName value is SyteLineTrans. If you copy an existing form, the ModuleName value is CopiedForms, and the OriginalModuleName value is set to the module to which the source form was assigned.

Beginning with SyteLine 7.03, the framework creates ModuleMembers table records for both SyteLineTrans and SyteLineTransMultiSite when creating a new form through the New Form Wizard or when copying existing ones. For SyteLine 7.02 and earlier only SyteLineTrans was created.

With SyteLine 7.02 and earlier, if you need to license a new form to SyteLineTrans and SyteLineTransMultiSite, then in the target database you must perform the following steps:

1. In the desired target database, use WinStudio to copy a form belonging to both the SyteLineTrans and SyteLineTransMultiSite modules with the new desired name.
2. Use FormSync (without the Copy License option enabled) to copy the real form from some other source database into the target forms database. In this scenario, the WinStudio-copied form has licensing records for both SyteLineTrans and SyteLineTransMultiSite, and the FormSync-copied form (since licensing is not specified) simply uses that and does not overlay it.

Beginning with SyteLine 7.03, generated licenses allow for more than one UserForm module, and both SyteLineTrans and SyteLineTransMultiSite are the UserModule specifications included in SyteLine 7.03 license document. The framework for SyteLine 7.03 also supports multiple UserForm module specifications, so creating a new form from scratch creates license records for modules SyteLineTrans and SyteLineTransMultiSite. To copy these forms to other databases, use the FormSync copy feature with the Copy License option enabled.

Custom forms for entity databases must be copied from existing forms in the entity database, so that they can inherit the entity licensing. Because they will run in an entity, they cannot use the SyteLineTrans or SyteLineTransMultiSite license.
Multi-Site

If you want a newly created form to be available in additional sites and you are not using the intranet licensing option with a master site, you must copy the form and license information, using FormSync’s copy feature, to the forms and application databases in the sites.

For information about intranet licensing, see the Help for the Sites/Entities form and Multi-Site Planning Guide on the support Web site.

Copy License Information

You can create forms in a test forms database and then copy them to a production forms database. Enable the FormSync Copy License option to copy license information to the production forms database.

Licensing New and Copied IDOs

With SyteLine 8, the licensing of new and copied IDOs is parallel to the licensing of new and copied forms.

IDOs created with the New IDO Wizard are assigned to the CustomerIDOs internal module, which in turn inherits all licensing through the SyteLineAutomation module. The SyteLineAutomation module is the IDO equivalent of the UserForms module. The CustomerIDOs license module is similar to the CustomerForms license module.

IDOs developed with the Extends / Extend and Replace options are equivalent to copied and modified forms. Extended IDOs are assigned to the CopiedExtendedIDOs module, which allows the new IDO to inherit the same licensing modules as the original source IDO from which it was extended. The CopiedExtendedIDOs license module is similar to the CopiedForms license module. If a base IDO belongs to more than one license module, then a CopiedExtendedIDOs ModuleMember record is created for each license module to which the base IDO belongs.

Licensing for IDOs imported by means of the IDO Import Wizard form or the App Metadata Transport utility (AppMetadataTransport.exe) is similar to licensing for new or extended IDOs. Licensing is executed on import.
IsModuleLicensed Method

The **IsModuleLicensed** method indicates whether a module is licensed. **IsModuleLicensed** is implemented as a method on the **Licenses** IDO. It can be called in a script or in a WinStudio event handler of the type Method Call.

The result of the call is passed back in an output parameter. The return value integer 1 indicates that a module is licensed; 0 indicates that the module is not licensed.

Sample Script

The following script determines whether the module named in ModuleName is licensed.

```vbp
Function TestModule(ByVal ModuleName As String) As Integer

    Dim Response As InvokeResponseData
    Dim ModuleLicensedFlag As Integer
    TestModule = 0
    Response = IDOClients.Invoke("MGCore.Licenses", ", "IsModuleLicensed", ModuleName, ModuleLicensedFlag)

    ModuleLicensedFlag = Response.Parameters (1).GetValue(Of Integer)()

    If Convert.ToBoolean(ModuleLicensedFlag) Then
        Application.ShowMessage("Module Is Licensed")
    Else
        Application.ShowMessage("Module Is Not Licensed")
    End If

End Function
```

Remarks

The integer return value is not synonymous with the IsLicensedFlag return value. It simply indicates whether the call itself succeeded. A non-zero value indicates that the call failed.

The parameters passed in must match the type of the parameters required by the method or be of type Object.

As with all **IDOClients.Invoke** calls, the values of the output parameters are retrieved using a **Response.Parameters(index).Value** [or **GetValue**]. Parameters are zero-based. In this case, ModuleName is the first parameter. The return output parameter value can be placed in, for example, a Visual Basic variable, a WinStudio component, a property, or a variable.

To use the output parameter value immediately in a Boolean check without assigning it to an intermediate container, you can do the following:
If Response.Parameters(1).GetValue(Of Boolean)() Then
    Application.ShowMessage("Module is licensed.")
Else
    Application.ShowMessage("Module is not licensed.")
End If

Sample Event Handler

The following is the text in theParmsfield of a Method Call event handler. The event handler return value does not represent the IsModuleLicensed value. It simply indicates (as do most IDO method calls) whether the call itself succeeded.

Licenses.IsModuleLicensed(PARMS(BSTR C(ModuleNameEdit), RBSTR _
V(ModuleLicensed)))

Remarks

Parameter 1 is the module name. In this case, the parameter is a component value.

Parameter 2 (output) is the 1 or 0 result of the license check. In this case, output is assigned to a variable named ModuleLicensed.
The material that was formerly contained in this chapter has been updated and incorporated into the online help.
The material that was formerly contained in this chapter has been updated and incorporated into the online help.
Chapter 11: APIs for the Tree Component

The material that was formerly contained in this chapter has been updated and incorporated into the online help.
Appendix A: Sample Scripts with APIs for Strong Types

The material that was formerly contained in this chapter has been updated and incorporated into the online help.
Appendix B: Suggestions for Development Environment Configuration

This appendix provides guidelines for how to configure your development environment in such a way that you can most easily reintegrate it with new versions of the SyteLine application. Topics include:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintain Separate Read-Only SyteLine Environments</td>
<td>B-2</td>
</tr>
<tr>
<td>Use SourceSafe with IDO Metadata</td>
<td>B-2</td>
</tr>
<tr>
<td>Use SourceSafe for Application Database Scripts</td>
<td>B-2</td>
</tr>
</tbody>
</table>
Maintain Separate Read-Only SyteLine Environments

For the purposes of being able to generate difference analyses at each tier of the application, maintain a complete set of read-only databases (application, objects, and forms) for each major release of SyteLine, where those databases remain separate from your actual development environment’s databases.

Use SourceSafe with IDO Metadata

For each major version of your modifications, maintain an objects database and an associated SourceSafe project tied into the configuration the developers use. Developers will then create project histories in SourceSafe as they check IDOs in and out in IDO editing forms.

Use SourceSafe for Application Database Scripts

While the toolset does not address application database changes, and so a wide variety of approaches are possible in managing source code for this tier, we recommend that you also set up a SourceSafe project storing the SQL scripts that make up the application database, and have an instance of this SourceSafe project for each major version of your modifications.
Appendix C: Numeric Constants Used in the Forms Database

This appendix contains the contents of one of the toolset C# enum files. It might be useful in constructing SQL against the forms database.

```csharp
namespace Mongoose.WinStudio.Enums
{
    #region Form

    [FlagsAttribute]
    public enum FormFlags
    {
        NONE = 0x00000000,
        HORIZONTAL_SPLITTER = 0x00000001,
        VERTICAL_SPLITTER = 0x00000002,
        MODAL = 0x00000004,
        COLLECTION_COMMIT = 0x00000008,
        PANE_ZERO_HIDDEN = 0x00000010,
        PANE_ONE_HIDDEN = 0x00000020,
        NO_SPLITTER_VSCROLL_PANE_0 = 0x00000040,
        NO_SPLITTER_HSCROLL_PANE_0 = 0x00000080,
        NO_SPLITTER_VSCROLL_PANE_1 = 0x00000100,
        NO_SPLITTER_HSCROLL_PANE_1 = 0x00000200,
        CUSTOM_FORM = 0x00000400,
        NO_CLOSEBOX = 0x00000800,
        NO_MINMAX = 0x00001000,
        NO_WALLPAPER_PANE_0 = 0x00002000,
        NO_WALLPAPER_PANE_1 = 0x00004000
    }

    public enum FormTypes : short
    {
    }
```
detail = 0,
editcollection = 1,
query = 2,
editcolsmall = 3,
editcolright = 4
}

[FlagsAttribute]
public enum CollectionOperations
{
    None = 0x00000000,
    Refresh = 0x00000001,
    New = 0x00000002,
    Add = 0x00000004,
    Save = 0x00000008,
    Delete = 0x00000010,
    Navigate = 0x00000020,
    Copy = 0x00000040,
    Filter = 0x00000080,
    Notes = 0x00000100,
    Graph = 0x00000200,
    Validate = 0x00000400, // this goes with Save in all but
                           // programmatic generation
    SpreadsheetDisabled = 0x00000800
}

#region FormComponent

Executing a function to set a flag for the form component.

/// <summary>
/// ComponentTypes: formerly, MD_COMP_TYPE_
/// </summary>

public enum ComponentTypes
{
    Pointer = -1,
    None = -1,
    Static = 0,
    Edit = 1,
    Combobox = 2,
    DropList = 3,
    List = 4,
    Checkbox = 5,
Groupbox = 6,
RadioButton = 7,
Button = 8,
DefButton = 9,
MenuItem = 10,
ToolbarButton = 11,
Notebook = 12,
NotebookTab = 13,
Grid = 14,
GridColumn = 15,
Graph = 16,
GraphSeries = 17, // not implemented
MultilineEdit = 18,
Tree = 19,
TreeBranch = 20, // not implemented
TreeLeaf = 21, // not implemented
Slider = 22, // not implemented
SpinButton = 23, // not implemented
FormPage = 24,
ActivexControl = 25,
DateCombo = 26,
EnhancedCombo = 27,
HyperlinkButton = 38,
Browser = 39,
CancelButton = 40,
Diagram = 41,

[FlagsAttribute]
public enum ComponentFlags
{
    NONE = 0x00000000,
    SPLITTER_PANE_1 = 0x00000001, // by default, pane 0
    STATIC_RIGHT_JUST = 0x00000002, // applies to pushbuttons/checkboxes too
    VALUE_IS_LIST_INDEX = 0x00000002, // applies to listboxes, combos, and combos in grid cols
    ACTIVEX_WANT_RIGHT_CLICK = 0x00000002, // only activex components
    NO_SCROLL_NOTEBOOK = 0x00000002, // for Notebooks
    STATIC_CENTER_JUST = 0x00000004, // for pushbuttons, means Left Justify!
    EDIT_PASSWORD = 0x00000004, // for edits
    LIST_SORT = 0x00000004, // for combos and listboxes
    GRID_COL_CHECKBOX = 0x00000008,
    GROUP = 0x00000008,
    DEFAULT_ON_COPY = 0x00000010,
    STATIC_NO_COLON = 0x00000010,
    LIST_MULTISELECT = 0x00000010,
VALIDATE_IMMEDIATELY = 0x00000020,
GRID_COL_FROZEN = 0x00000040,
STATIC_FRAME = 0x00000040,
GLUE_RIGHT = 0x00000080,
GLUE_BOTTOM = 0x00000100,
REQUIRED = 0x00000200,
NO_CLEAR_ON_NEW = 0x00000400,
UPPERCASE = 0x00000800,
NO_FILTER_IN_PLACE = 0x00001000,
READ_ONLY_FOR_EXISTING = 0x00002000,
READ_ONLY_FOR_NEW = 0x00004000,
ENABLE_ON_NON_MOD_OBJ = 0x00008000,
ENABLE_ON_REQ_DATA_ENTERED = 0x00010000,
TBD = 0x00020000,
EDITCOMP_RIGHT_JUSTIFY = 0x00040000, // grid columns too
STATIC_RECT = 0x00040000,
HTML_BLEND = 0x00040000,
NO_TAB_STOP = 0x00080000,
COMBO_AUTO_COMPLETE = 0x00100000, // grid columns too
VAL_ERR_STATUS_LINE = 0x00200000,
GRID_DATE_COMBO = 0x00400000,
READ_ONLY_FOR_DELETED = 0x00800000,
MOVE_LOCKED = 0x01000000,
SIZE_LOCKED = 0x02000000,
NO_INTERPRET_CONTENTS = 0x04000000, // for any text-showing/editing types
INTERPRET_BOUND_CONTENTS = 0x08000000 // for any text-showing/editing types

public enum ComponentBindings
{
    NONE = 0,
    STANDARD_OBJECT = 1,
    OTHER_OBJECT = 2,
    FORM_COLLECTION = 3,
    SUB_COLLECTION = 4,
    OBJECT_ERROR = 5,
    VARIABLE = 6,
    FILTER = 7,
    SECONDARY_COLLECTION = 8,
    COLLECTION_ID = 9,
    GRAPH = 10,
    GRAPH_XLIMIT = 11,
    GRAPH_NEXT = 12,
    GRAPH_PREVIOUS = 13,
    GRAPH_REDRAW = 14,
    GRAPH_PROPERTIES = 15,
    TREE = 16,
#endregion

#region FormEventHandler

public enum FormEventHandlerResponseType
{
    NONE = -1,
    // The response string has the format "server.method( PARMS(type <val1>,<type2 <val2>>etc <ERRORMESSAGE()>)"
    OLE = 0,
    // The response string has the format "dllname.functionname" <parms>; not currently implemented
    DLL = 1,
    // The response string has the format "executable.ext" <parms>
    EXE = 2,
    // The response string has the format "NameOfFormToCall"
    CALL_FORM = 3,
    // The response string has the format "NameOfFormToGoto"
    GOTO_FORM = 4,
    // The response string has the format "NumberOfFormPageToGoTo" or nothing, which is next
    GOTO_FORMPAGE = 5,
    // The response string has the format "NameOfFormToRunAsLinkedChild"
    RUN_FORM_LINKED_CHILD = 6,
    // The response string has the format "NameOfFormToRun"
    RUN_FORM = 7,
    // The response string has the format "scriptname <parms>"
    RUN_INTERNAL_SCRIPT = 8,
    // Note: The subcollection stuff, though still supported, displays as obsolete...use
    // the COLLECTION responses to achieve the same things in general fashion.
    // The response string has the format "componentname
    next/prev/first/last"
    SUBCOLLECTION_NAVIGATE = 9,
    // The response string has the format "componentname"
    SUBCOLLECTION_NEW = 10,
    // The response string has the format "componentname"
    SUBCOLLECTION_SAVE = 11,
    // The response string has the format "componentname"
    SUBCOLLECTION_DELETE = 12,
    // The response string has the format "componentname"
    SUBCOLLECTION_REFRESH = 13,
    // The response string has the format "componentname"
    SUBCOLLECTION_COPY = 14,
    // The response string has the format "componentname"
SUBCOLLECTION_REFRESH_CURRENT = 15,
   // The response string has the format "NameOfFormToCall"
CALL_FORM_FOR_FILTER = 16,
   // The response string has the format "componentname"
SUBCOLLECTION_NOTES = 17,
   // The response string has the format "componentname"
SUBCOLLECTION_CUR_NOTES = 18,
   // The response string is JOBNAME(), JOBPARMS1(), and
   // JOBPARMS2()
RUN_BACKGROUND_TASK = 19,
   // ??
unused = 20,
   // The response string is the name of the event to generate
GENERATE_EVENT = 21,
   // The response string contains SETVARVALUES, SETPROPVALUES,
   // and/or SETCOMPVALUES
SET_VALUES = 22,
   // The response string is the cache id, "object", "object2",
   // "object.prop", followed by NEXT, etc.
COLLECTION_NAVIGATE = 23,
   // The response string is the cache id, "object", "object2",
   // "object.prop"
COLLECTION_NEW = 24,
COLLECTION_COPY = 25,
COLLECTION_SAVE = 26,
COLLECTION_DELETE = 27,
COLLECTION_REFRESH = 28,
COLLECTION_REFRESH_CURRENT = 29,
COLLECTION_NOTES = 30,
COLLECTION_CUR_NOTES = 31,
COLLECTION_INITAUTOINSERT = 32,
   // The response string is the script, including any "ReturnValue =" required
INLINE_SCRIPT = 33,
   // The response string is the cache id, "object", "object2",
   // "object.prop"
COLLECTION_MAKECURRENT = 34,
   // The response string is the method name, and possibly parms
ACTIVEX_SCRIPT_METHOD = 35,
   // the response string is the set of parms in the
   // DoConditionalAction language, in ACTION() keyword
CONDITIONAL_ACTION = 36,
   // the response string is ACTION(Import/Export/View)
COMPONENT(compname)
   // the response string contains
BINARY_DATA_ACTION = 37,
   // The response string contains
ADD_AUDITLOG_ENTRY = 38,
   // the response is a url...executed in another instance of IE
GOTO_URL = 39,
PRINT_PREVIEW = 40,
// the response is a forward-slash delimited explorer folder
// specification, with MASTER, PUBLIC, USERS, MYFOLDERS as hard-
// wired roots
GOTO_EXPLORER_FOLDER = 41,
// response format tbd
INVOKE_WEB_SERVICE = 42, // postponed until post 6.00
// response format EVENT(eventtogen) <SYNCHRONOUS()> PARMS(
// name1=value1, name2=value2, etc. )
GENERATE_APP_EVENT = 43,
// unused--was going to be POST_APP_EVENT
UNUSED2 = 44,
// response format COMPONENT(userControlComponentName)
PARMS(comma-delimited parms string)
INVOKE_USER_CONTROL = 45

// from the events table, reserved (standard) events
// These have default implementations that are callable by
// generating the events,
// and which are hooked to the object menu items. If you specify
// a handler, you may
// override or supplement the default implementation: for those
types of
// handlers that can return an rc (non .exe), a non-zero return
code means
// "don't perform the default implementation".
public class StandardEvents
{
    StandardEvents()
    {
    }
    
    public const String FORM_REFRESH = "StdFormRefresh";
    public const String FORM_NEW = "StdFormNew";
    public const String FORM_COPY = "StdFormCopy";
    public const String FORM_SAVE = "StdFormSave";
    public const String FORM_SAVE_CURRENT = "StdFormSaveCurrent";
    public const String FORM_FILTER = "StdFormFilter";
    public const String FORM_DELETE = "StdFormDelete";
    public const String FORM_FIRST = "StdFormFirst";
    public const String FORM_LAST = "StdFormLast";
    public const String FORM_NEXT = "StdFormNext";
    public const String FORM_PREVIOUS = "StdFormPrevious";
    public const String FORM_REFRESH_CURRENT = "StdFormRefreshCurrent";
    public const String FORM_NOTES = "StdFormNotes";
    public const String FORM_NOTES_COMPLETED = "StdFormNotesCompleted";
public const String FORM_CURRENT_NOTES = "StdFormCurrentNotes";
public const String FORM_CURRENT_NOTES_COMPLETED = "StdFormCurrentNotesCompleted";
public const String FORM_LINK_UPDATE = "StdFormLinkUpdate";
public const String FORM_COL_FIND = "StdFormColFind";
public const String FORM_COL_REPLACE = "StdFormColReplace";
public const String FORM_COL_SORT = "StdFormColSort";
public const String FORM_GRAPH = "StdFormGraph";
public const String FORM_SPREADSHEET_COPY = "StdFormSpreadsheetCopy";
public const String FORM_VIEW_WORKFLOW = "StdFormViewWorkflow";
// These are notification-only events, and are only fired if the event
// they precede completed successfully
public const String FORM_REFRESH_COMPLETED = "StdFormRefreshCompleted";
public const String FORM_NEW_COMPLETED = "StdFormNewCompleted";
public const String FORM_COPY_COMPLETED = "StdFormCopyCompleted";
public const String FORM_SAVE_COMPLETED = "StdFormSaveCompleted";
public const String FORM_SAVE_CURRENT_COMPLETED = "StdFormSaveCurrentCompleted";
public const String FORM_FILTER_COMPLETED = "StdFormFilterCompleted";
public const String FORM_DELETE_COMPLETED = "StdFormDeleteCompleted";
public const String FORM_FIRST_COMPLETED = "StdFormFirstCompleted";
public const String FORM_LAST_COMPLETED = "StdFormLastCompleted";
public const String FORM_NEXT_COMPLETED = "StdFormNextCompleted";
public const String FORM_PREVIOUS_COMPLETED = "StdFormPreviousCompleted";
public const String FORM_REFRESH_CURRENT_COMPLETED = "StdFormRefreshCurrentCompleted";
public const String FORM_LINK_UPDATE_COMPLETED = "StdFormLinkUpdateCompleted";
public const String FORM_COL_FIND_COMPLETED = "StdFormColFindCompleted";
public const String FORM_COL_REPLACE_COMPLETED = "StdFormColReplaceCompleted";
public const String FORM_COL_SORT_COMPLETED = "StdFormColSortCompleted";
public const String FORM_GRAPH_COMPLETED = "StdFormGraphCompleted";
public const String FORM_SPREADSHEET_COPY_COMPLETED = "StdFormSpreadsheetCopyCompleted";
public const String FORM_VIEW_WORKFLOW_COMPLETED = "StdFormViewWorkflowCompleted";
// These are cache-level events...they have default implementations that you can
// cancel with a handler, but generating them won't do anything-you have to
// programmatically invoke the low-level cache functionality they represent by
// scripting, or event handlers of various response types.

public const String OBJ_REFRESH = "StdObjectRefresh";
public const String OBJ_NEW = "StdObjectNew";
public const String OBJ_COPY = "StdObjectCopy";
public const String OBJ_SAVE = "StdObjectSave";
public const String OBJ_SAVE_CURRENT = "StdObjectSaveCurrent";
public const String OBJ_DELETE = "StdObjectDelete";
public const String OBJ_DELETE_NEW = "StdObjectDeleteNew";
public const String OBJ_DELETE_EXISTING = "StdObjectDeleteExisting";
public const String OBJ_FIRST = "StdObjectFirst";
public const String OBJ_LAST = "StdObjectLast";
public const String OBJ_NEXT = "StdObjectNext";
public const String OBJ_PREVIOUS = "StdObjectPrevious";
public const String OBJ_REFRESH_CURRENT = "StdObjectRefreshCurrent";
public const String OBJ_SELECT_CURRENT = "StdObjectSelectCurrent";

// These are notification-only events, and are only fired if the event
// they precede completed successfully

public const String OBJ_REFRESH_COMPLETED = "StdObjectRefreshCompleted";
public const String OBJ_NEW_COMPLETED = "StdObjectNewCompleted";
public const String OBJ_COPY_COMPLETED = "StdObjectCopyCompleted";
public const String OBJ_SAVE_COMPLETED = "StdObjectSaveCompleted";
public const String OBJ_SAVE_CURRENT_COMPLETED = "StdObjectSaveCurrentCompleted";
public const String OBJ_DELETE_COMPLETED = "StdObjectDeleteCompleted";
public const String OBJ_DELETE_NEW_COMPLETED = "StdObjectDeleteNewCompleted";
public const String OBJ_DELETE_EXISTING_COMPLETED = "StdObjectDeleteExistingCompleted";
public const String OBJ_FIRST_COMPLETED = "StdObjectFirstCompleted";
public const String OBJ_LAST_COMPLETED = "StdObjectLastCompleted";
public const String OBJ_NEXT_COMPLETED = "StdObjectNextCompleted";
public const String OBJ_PREVIOUS_COMPLETED = "StdObjectPreviousCompleted";
public const String OBJ_REFRESH_CURRENT_COMPLETED = "StdObjectRefreshCurrentCompleted";
public const String OBJ_SELECT_CURRENT_COMPLETED = "StdObjectSelectCurrentCompleted";

// This one can be generated by a "Prev" button on a wizard-type form
public const String FORMPAGE_POP = "StdFormPagePop";

// These are events which are automatically generated during various phases of the
// form's operations, but which have no default implementations - they are merely
// opportunities for handlers to get control.

public const String FORM_PREDISPLAY = "StdFormPredisplay";
public const String FORM_CALLED_FORM_RETURNED = "StdFormCalledFormReturned";
    public const String FORM_CLOSE = "StdFormClose";
public const String FORM_CURRENT_CACHE_CHANGING = "StdFormCurCacheChanging";
    public const String FORM_CURRENT_CACHE_CHANGED = "StdFormCurCacheChanged";
public const String FORM_INIT_COMPLETED = "StdFormInitCompleted";

// This event is raised by the form timer
// A form timer is started by response type Timer
public const String FORM_TIMER_TICK = "StdFormTimerTick";

// These are events which are automatically generated during various phases of the
// form's operations, and which have default implementations.
For handlers that
// can return a return code (non exe), non-zero rc means "don't perform the default
// implementation".
public const String FORM_LOAD_BOUND_COMPONENT_VALUES = "StdFormLoadBoundValues";
    public const String FORM_LOAD_DERIVED_COMPONENT_VALUES = "StdFormLoadDerivedValues";
public const String FORM_GET_BOUND_COMPONENT_VALUES = "StdFormGetBoundValues";
    public const String FORM_PUT_BOUND_COMPONENT_VALUES = "StdFormPutBoundValues";
public const String FORM_PERFORM_VALIDATIONS = "StdFormPerformValidations";
    public const String FORM_EXIT_OK = "StdFormExitOk";
public const String FORM_EXIT_CANCEL = "StdFormExitCancel";
    public const String FORM_HELP = "StdFormHelp";
public const String FORM_COMPONENT_HELP = "StdFormComponentHelp";

// These are notification only, and are generated only after the corresponding
public const String FORM_VALIDATIONS_COMPLETED = "StdFormValidationsCompleted";

// These events are usually generated by double-click or right-click menu
// on a component...they are the "drill-down" and "drill-across" actions
public const String CUR_COMPONENT_FIND = "StdCurCompFind";
public const String CUR_COMPONENT_DETAILS = "StdCurCompDetails";
public const String CUR_COMPONENT_ADD = "StdCurCompAdd";
public const String CUR_COMPONENT_GRAPH_PROPERTIES = "StdCurCompGraphProperties";
public const String CUR_COMPONENT_GRIDCOL_MAINT = "StdCurCompEditGridColumns";

// These events expose the usual clipboard operations
public const String CUR_COMPONENT_CLIPBOARD_COPY = "StdCurCompClipboardCopy";
public const String CUR_COMPONENT_CLIPBOARD_PASTE = "StdCurCompClipboardPaste";
public const String CUR_COMPONENT_CLIPBOARD_CUT = "StdCurCompClipboardCut";
public const String CUR_COMPONENT_CLIPBOARD_PASTE_APPEND = "StdCurCompClipboardPasteAppend";
public const String CUR_COMPONENT_CLIPBOARD_PASTE_OVERWRITE = "StdCurCompClipboardPasteOverwrite";

// These are events which are fired only for query forms, and which have
// default implementations that are overridable by event handlers
public const String QUERY_FORM_INITIALIZE = "StdQueryFormInitialize";
public const String QUERY_FORM_ADD_ADDITIONAL = "StdQueryFormAddAdditional";
public const String QUERY_FORM_REMOVE_ADDITIONAL = "StdQueryFormRemoveAdditional";
public const String QUERY_FORM_CLEAR = "StdQueryFormClear";
public const String QUERY_FORM_SELECT_ADDITIONAL_PROPERTY = "StdQueryFormSelectAdditionalProperty";

// These are events associated with filter in place mode, and have
// default implementations that are overridable by event handlers
public const String FILTER_IN_PLACE_BEGIN = "StdFormFilterInPlaceBegin";
public const String FILTER_IN_PLACE_OK = "StdFormFilterInPlaceExecute";
public const String FILTER_IN_PLACE_CANCEL = "StdFormFilterInPlaceCancel";
public const String FILTER_IN_PLACE_TOGGLE = "StdFormFilterInPlaceToggle";

// These are events associated with filter in place mode, and are notification only
public const String FILTER_IN_PLACE_BEGIN_COMPLETED = "StdFormFilterInPlaceBeginCompleted";
public const String FILTER_IN_PLACE_OK_COMPLETED = "StdFormFilterInPlaceExecuteCompleted";
public const String FILTER_IN_PLACE_CANCEL_COMPLETED = "StdFormFilterInPlaceCancelCompleted";

// These are generated by a "hot" click on a graph component's data element; the second is notification only
public const String GRAPH_ELEMENT_CLICKED = "StdGraphElementClicked";
public const String GRAPH_ELEMENT_CLICKED_COMPLETED = "StdGraphElementClickedCompleted";

// A notification only event generated when the user presses ENTER from the IME editor
public const String LAST_IME_VALUE_UPDATED = "StdLastIMEValueUpdated";

// This event will be generated in an inbox custom response form after the payload variables have been initialized.
public const String INBOX_PAYLOAD_INIT_COMPLETED = "StdInboxPayloadInitCompleted";

// These events have application-wide, as opposed to only form-level, built-in responses
public const String APP_LAUNCH_INBOX = "StdAppLaunchInbox";
public const String APP_LAUNCH_TASKS = "StdAppLaunchTasks";

#endregion

#region ComponentBase (previously PropertyDefault)

// formerly MD_PROPDEF_FLAGS_ [FlagsAttribute]
public enum ComponentBaseFlags
{
    NONE = 0x00000000,
    READ_ONLY = 0x00000001
}

#endregion

#region Validator

public enum ValTypes

#endregion
{  
  NONE = -1,  
  ALPHANUM = 0,  
  ALPHA = 1,  
  NUMERIC = 2,  
  DATE = 3,  
  TIME = 4,  
  DATETIME = 5,  
  CURRENCY = 6,  
  OLE = 7,  
  DLL = 8,  
  IN_COLLECTION = 9,  
  IN_LISTCACHE = 10,  
  SCRIPT = 11,  
  IN_INLINELIST = 12,  
  LISTCACHE_ONLY = 13,  
  INLINE_SCRIPT = 14  
}  

#endregion  

#region Explorer  

public enum ExplorerObjectType  
{  
  NormalForm = 'R',  
  QueryForm = 'Q',  
  AllForms = 'A',  
  Folder = 'F',  
  Workspace = ‘W’  
}  

public enum ExplorerTopLevelFolderIds  
{  
  Master = -1,  
  Public = -2,  
  Users = -3,  
  User = -4,  
  AllForms = -5  
}  

public class ExplorerFolderStringNames  
{  
  public ExplorerFolderStringNames()  
  {  
  }  
}  

public const string EXP_FOLDER_ID_MASTER = "MASTER";
public const string EXP_FOLDER_ID_PUBLIC = "PUBLIC";
public const string EXP_FOLDER_ID_USERS = "USERS";
public const string EXP_FOLDER_ID_MYFOLDERS = "MYFOLDERS";

public const String EXP_MASTER_STRING = "stdMasterExplorer";
public const String EXP_PUBLIC_FOLDERS_STRING = "stdPublicFolders";
public const String EXP_USER_FOLDERS_STRING = "stdUserFolders";
public const String EXP_MY_FOLDERS_STRING = "stdMyFolders";
public const String EXP_EXPLORER_CAPTION = "stdExplorerCaption";
public const String EXP_AUTORUN_STRING = "stdAutoRun";
public const String EXP_PRELOAD_STRING = "stdPreLoad";
}

#endregion

#region Templates

public enum TemplateFormTypes
{
    MultiView = 0,
    GridOnly = 1,
    Query = 2,
    BuildFromScratch = 3
}

// For form template property component classification
// I think these actually match the ComponentTypes...could be done
// more elegantly at some point. Formerly FWIZ_COMP_CLASS_

public enum FormWizardCompClass
{
    NONE = 0,
    EDIT = 1,
    CHECKBOX = 2,
    GRID = 7
}

// for form template property container types...formerly
FWIZ_CONT_TYPE

public enum FormWizardContainerType
{
    DEFAULT = 0,
    NOTEBOOK = 1,
    FORM_PAGE = 2
}
// for interpretation of form template property flags
bits...formerly FWIZ_FLAGS_

[FlagsAttribute]
public enum FormWizardPropertyFlags
{
    NONE = 0x00000000,
    QUERY_CRITERIA_PRIMARY = 0x00000001,
    QUERY_CRITERIA_ADDITIONAL = 0x00000002,
    QUERY_RESULTS = 0x00000004,
    DETAIL_GRID = 0x00000001
}

#region
This appendix describes how to create scripts that recreate UET field definitions when executed against the application database. The issue of form changes—adding components bound to UET properties—is not addressed here and is not different from other form changes with regard to scope type and maintenance of customizations.

A manual technique to apply the scripts to a target environment is described. An installation program can perform the same actions programmatically.

Topics in this appendix include:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extracting UET Definitions from the Source Environment</td>
<td>D-2</td>
</tr>
<tr>
<td>Applying the Scripts in the Target Environment</td>
<td>D-3</td>
</tr>
<tr>
<td>Testing the Changes</td>
<td>D-3</td>
</tr>
<tr>
<td>The ExportUETClassSp Stored Procedure</td>
<td>D-4</td>
</tr>
</tbody>
</table>
Extracting UET Definitions from the Source Environment

UET definitions can be extracted from the source environment using the stored procedure `ExportUETClassSp`, which is supplied with this document and is not part of the SyteLine application. `ExportUETClassSp` creates a SQL script containing INSERT statements to put the necessary data into the following tables in the target environment: `user_class`, `user_fld`, `user_class_fld`, `user_index`, `user_index_fld`, and `table_class`. These tables constitute the UET schema.

The following are the steps to generate the script:

1. Open SQL Server Management Studio.
2. Load the file `ExportUETClassSp.sql` in a new query window (this file contains the `ExportUETClassSp` stored procedure and Quote function).
3. Execute the query to create the stored procedure and the function.
4. Create a new query window and execute `ExportUETClassSp`, providing the class name `ExportUETClassSp uet_class_name`, where `uet_class_name` is the name of the class you have defined and want to script. The script will be created in the results pane.
5. Save the script to an operating system file.

Repeat the previous step for each class you want to export to the target environment.
Applying the Scripts in the Target Environment

To apply scripts in the target environment:

1. Open SQL Server Management Studio.
2. Load the scripts generated with ExportUETClassSp and run each one.
3. Open SyteLine.
5. Apply the schema changes based on the information loaded into UET Schema through the script in the previous step:

6. Close the SyteLine session as indicated in the UET Impact Schema process.

Testing the Changes

Start a new SyteLine session and open a form that contains UET customizations. A field bound to the UET property should be in place and functional.
The ExportUETClassSp Stored Procedure

The following is the text of the stored procedure ExportUETClassSp:

```sql
SET QUOTED_IDENTIFIER OFF
GO
SET ANSI_NULLS ON
GO
IF OBJECT_ID('dbo.ExportUETClassSP') IS NOT NULL
    DROP PROCEDURE ExportUETClassSP
GO
CREATE PROCEDURE ExportUETClassSP (  
    @PClassName Infobar
)  
AS
DECLARE  
    @ClassName ClassNameType  
    , @ClassLabel LabelType  
    , @ClassDesc DescriptionType  
    , @SysHasFields ListYesNoType  
    , @SysHasTables ListYesNoType  
    , @SysApply UetSysApplyType  
    , @SysDelete ListYesNoType  
    , @FldName FldNameType  
    , @FldDataType UetDataTypeType  
    , @FldInitial UetDefaultType  
    , @FldDecimals UetScaleType  
    , @FldDesc ToolTipType  
    , @FldUDT sysname  
    , @FldPrec tinyint  
    , @IndexName IndexNameType  
    , @IndexDesc DescriptionType  
    , @IndexUnique ListYesNoType  
    , @IndexWord ListYesNoType  
    , @IndexSeq UetIndexSeqType  
    , @IndexAsc ListYesNoType  
    , @TableName TableNameType  
    , @TableRule QueryExpressionType  
    , @ExtendAllRecs ListYesNoType  
    , @AllowRecordAssoc ListYesNoType  
    , @Active ListYesNoType  
    , @SQLCmd InfobarType  
    , @SQLCmdWait InfobarType  
    , @Severity INT  
    , @Quote NCHAR(1)  
    , @RecordDate CurrentDateType  
    , @RowPointer RowPointerType  
    , @CommittedRowPointer RowPointerType
```
SET @Severity = 0
SET @Quote = '''
DECLARE UserClassCrs CURSOR LOCAL STATIC
FOR SELECT
    uc.class_name
    , uc.class_label
    , uc.class_desc
    , uc.sys_has_fields
    , uc.sys_has_tables
    , uc.sys_apply
    , uc.sys_delete
FROM user_class uc
WHERE class_name = @PClassName
OPEN UserClassCrs
WHILE @Severity = 0
BEGIN -- cursor loop
    FETCH UserClassCrs INTO
        @ClassName
        , @ClassLabel
        , @ClassDesc
        , @SysHasFields
        , @SysHasTables
        , @SysApply
        , @SysDelete
    IF @@FETCH_STATUS = -1
        BREAK
    SET @SQLCmd = 'INSERT INTO user_class ( class_name, class_label,'
    SET @SQLCmd = @SQLCmd + 'class_desc, sys_has_fields, ' +
    SET @SQLCmd = @SQLCmd + 'sys_has_tables, sys_apply, sys_delete ) VALUES ( ' +
    SET @SQLCmd = @SQLCmd + dbo.Quote(@ClassName) + ', ' +
    SET @SQLCmd = @SQLCmd + dbo.Quote(@ClassLabel) + ', ' +
    SET @SQLCmd = @SQLCmd + dbo.Quote(@ClassDesc) + ', ' +
    SET @SQLCmd = @SQLCmd + STR(ISNULL(@SysHasFields, 0)) + ', ' +
    SET @SQLCmd = @SQLCmd + STR(ISNULL(@SysHasTables, 0)) + ', ' +
    SET @SQLCmd = @SQLCmd + dbo.Quote(@SysApply) + ', ' +
    SET @SQLCmd = @SQLCmd + STR(ISNULL(@SysDelete, 0)) + ' )'
    print @SQLCmd
    DECLARE UserClassFldCrs CURSOR LOCAL STATIC
    FOR SELECT
        ucf.class_name
        , ucf.fld_name
        , ucf.sys_apply
        , ucf.sys_delete
        , ucf.sys_has_fields
        , ucf.sys_has_tables
        , ucf.sys_apply
        , ucf.sys_delete
FROM user_class_field ucf
WHERE class_name = @PClassName
FROM user_class_fld ucf
WHERE class_name = @PClassName
OPEN UserClassFldCrs
WHILE @Severity = 0
BEGIN -- cursor loop
FETCH UserClassFldCrs INTO
@ClassName
, @FldName
, @SysApply
, @SysDelete
IF @@FETCH_STATUS = -1
BREAK
SET @SQLCmd = 'INSERT INTO user_class_fld ( class_name, fld_name, sys_apply, sys_delete ) VALUES ( '
SET @SQLCmd = @SQLCmd + dbo.Quote(@ClassName) + ', ' +
dbo.Quote(@FldName) + ', ' + dbo.Quote(@SysApply) + ', ' +
STR(ISNULL(@SysDelete, 0)) + ' )'
SET @SQLCmdWait = @SQLCmd
SELECT @FldName = fld_name
, @FldDataType = fld_data_type
, @FldInitial = fld_initial
, @FldDecimals = fld_decimals
, @FldDesc = fld_desc
, @SysApply = sys_apply
, @SysDelete = sys_delete
, @FldUDT = fld_UDT
, @FldPrec = fld_prec
FROM user_fld uf
WHERE uf.fld_name = @FldName
SET @SQLCmd = 'INSERT INTO user_fld ( fld_name, fld_data_type, fld_initial, fld_decimals, fld_desc, sys_apply, sys_delete, fld_UDT, fld_prec ) VALUES ( '
SET @SQLCmd = @SQLCmd + dbo.Quote(@FldName) + ', ' +
dbo.Quote(@FldDataType) + ', ' + dbo.Quote(@FldInitial) +
', ' + dbo.Quote(@FldDecimals) + ', ' + dbo.Quote(@SysApply) +
', ' + STR(ISNULL(@SysDelete, 0)) + ', ' + dbo.Quote(@FldDesc) +
', ' + dbo.Quote(@FldUDT) + ', ' + STR(ISNULL(@FldPrec, 0)) + ' )'
PRINT @SQLCmd
PRINT @SQLCmdWait
END -- Cursor Loop UserClassFld
CLOSE UserClassFldCrs
DEALLOCATE UserClassFldCrs
DECLARE UserIndexCrs CURSOR LOCAL STATIC
FOR SELECT
ui.class_name
, ui.index_name
, ui.index_desc
, ui.index_unique
, ui.index_word
FROM user_index ui
WHERE class_name = @PClassName
OPEN UserIndexCrs
WHILE @Severity = 0
BEGIN -- cursor loop
FETCH UserIndexCrs INTO
@ClassName
, @IndexName
, @IndexDesc
, @IndexUnique
, @IndexWord
, @SysApply
, @SysDelete
IF @@FETCH_STATUS = -1
BREAK
SET @SQLCmd = 'INSERT INTO user_index ( class_name, index_name, index_desc, index_unique, index_word, sys_apply, sys_delete )
VALUES ( ' + dbo.Quote(@ClassName) + ', ' + dbo.Quote(@IndexName) + ', ' + dbo.Quote(@IndexDesc) + ', ' + STR(ISNULL(@IndexUnique, 0)) + ', ' + STR(ISNULL(@IndexWord, 0)) + ', ' + dbo.Quote(@SysApply) + ', ' + STR(ISNULL(@SysDelete, 0)) + ' )'
PRINT @SQLCmd
DECLARE UserIndexFldCrs CURSOR LOCAL STATIC
FOR SELECT
uif.class_name
, uif.index_name
, uif.index_seq
, uif.fld_name
, uif.index_asc
FROM user_index_fld uif
WHERE class_name = @PClassName
AND index_name = @IndexName
OPEN UserIndexFldCrs
WHILE @Severity = 0
BEGIN -- cursor loop
FETCH UserIndexFldCrs INTO
@ClassName
, @IndexName
, @IndexSeq
, @FldName
, @IndexAsc
IF @@FETCH_STATUS = -1
BREAK
SET @SQLCmd = 'INSERT INTO user_index_fld ( class_name,
index_name, index_seq, fld_name, index_asc ) VALUES ( ' 
SET @SQLCmd = @SQLCmd + dbo.Quote(@ClassName) + ', ' + 
dbo.Quote(@IndexName) + ', ' + STR(ISNULL(@IndexSeq, 0)) 
SET @SQLCmd = @SQLCmd + ', ' + dbo.Quote(@FldName) + ', ' + 
STR(ISNULL(@IndexAsc, 0)) + ' )' 
PRINT @SQLCmd 
END -- Cursor Loop UserIndexFld 
CLOSE UserIndexFldCrs 
DEALLOCATE UserIndexFldCrs 
END -- Cursor Loop UserIndex 
CLOSE UserIndexCrs 
DEALLOCATE UserIndexCrs 
DECLARE TableClassCrs CURSOR LOCAL STATIC 
FOR SELECT 
tc.table_name 
, tc.class_name 
, tc.table_rule 
, tc.extend_all_recs 
, tc.sys_apply 
, tc.sys_delete 
, tc.allow_record_assoc 
, tc.active 
FROM table_class tc 
WHERE class_name = @PClassName 
OPEN TableClassCrs 
WHILE @Severity = 0 
BEGIN -- cursor loop 
FETCH TableClassCrs INTO 
@TableName 
, @ClassName 
, @TableRule 
, @ExtendAllRecs 
, @SysApply 
, @SysDelete 
, @AllowRecordAssoc 
, @Active 
IF @@FETCH_STATUS = -1 
BREAK 
SET @SQLCmd = 'INSERT INTO table_class ( table_name, class_name, 
table_rule, extend_all_recs, sys_apply, sys_delete, 
allow_record_assoc, active ) VALUES ( ' 
SET @SQLCmd = @SQLCmd + dbo.Quote(@TableName) + ', ' + 
dbo.Quote(@ClassName) + ', ' + dbo.Quote(@TableRule) 
SET @SQLCmd = @SQLCmd + ', ' + STR(ISNULL(@ExtendAllRecs, 0)) + 
', ' + dbo.Quote(@SysApply) 
SET @SQLCmd = @SQLCmd + ', ' + STR(ISNULL(@SysDelete, 0)) + ', ' + 
STR(ISNULL(@AllowRecordAssoc, 0)) 
SET @SQLCmd = @SQLCmd + ', ' + STR(ISNULL(@Active, 0)) + ')' 
PRINT @SQLCmd 
END -- Cursor Loop TableClass
CLOSE TableClassCrs
DEALLOCATE TableClassCrs
END -- Cursor Loop UserClass
CLOSE UserClassCrs
DEALLOCATE UserClassCrs
RETURN @Severity
GO
SET QUOTED_IDENTIFIER OFF
GO
SET ANSI_NULLS ON
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
IF OBJECT_ID('dbo.Quote') IS NOT NULL
    DROP Function dbo.Quote
GO
CREATE FUNCTION dbo.Quote (    @ColumnValue LongListType)
    RETURNS LongListType
AS
BEGIN
DECLARE
    @Quote NCHAR(1)
SET @Quote = '''
SET @ColumnValue = ISNULL(@ColumnValue, 'NULL')
IF @ColumnValue != 'NULL'
    SET @ColumnValue = 'N' + @Quote + @ColumnValue + @Quote
RETURN @ColumnValue
END
GO
Appendix E: Effects of Service Packs and Upgrades on Customizations and Modifications

The following tables summarize the effects of service packs and upgrades on customizations and modifications.

<table>
<thead>
<tr>
<th>Object</th>
<th>Database</th>
<th>Type of Customization or Modification</th>
<th>Effect of Service Pack on Customization or Modification</th>
<th>Effect of Upgrade on Customization or Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form *</td>
<td>Forms</td>
<td>New object</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Scope: User, Group, or Site</td>
<td></td>
<td>Update to object</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Deletion of object</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Form *</td>
<td>Forms</td>
<td>New object</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Scope: Vendor</td>
<td></td>
<td>Update to base object</td>
<td>Update may be overwritten</td>
<td>Update is overwritten</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Deletion of base object</td>
<td>Object may be reinserted</td>
<td>Object is reinserted</td>
</tr>
<tr>
<td>IDO metadata</td>
<td>Objects</td>
<td>New object</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Extension of base object</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Object</td>
<td>Database</td>
<td>Type of Customization or Modification</td>
<td>Effect of Service Pack on Customization or Modification</td>
<td>Effect of Upgrade on Customization or Modification</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>----------</td>
<td>---------------------------------------</td>
<td>--------------------------------------------------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>.asp page: Data Collection module</td>
<td>N/A</td>
<td>New object</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Update to base object</td>
<td>Page may be overwritten</td>
<td>Page is overwritten</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Deletion of base object</td>
<td>Page may be added</td>
<td>Page is added</td>
</tr>
<tr>
<td>SQL Server table column</td>
<td>Application</td>
<td>New object</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Update to base object</td>
<td>Update may be overwritten</td>
<td>Update may be overwritten</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Deletion of base object</td>
<td>Column may be reinserted</td>
<td>Column may be reinserted</td>
</tr>
<tr>
<td>SQL Server trigger</td>
<td>Application</td>
<td>New object</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Update to base object</td>
<td>Trigger may be overwritten</td>
<td>Trigger is overwritten</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Deletion of base object</td>
<td>Trigger may be reinserted</td>
<td>Trigger is reinserted</td>
</tr>
<tr>
<td>SQL Server stored procedure or function</td>
<td>Application</td>
<td>New object</td>
<td>None</td>
<td>Dropped</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Update to base object</td>
<td>Update may be overwritten</td>
<td>Update is overwritten</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Deletion of base object</td>
<td>Object may be reinserted</td>
<td>Object may be reinserted</td>
</tr>
<tr>
<td>Report .rpt file</td>
<td>N/A</td>
<td>New object</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Update to base object</td>
<td>File may be overwritten</td>
<td>File may be overwritten</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Deletion of base object</td>
<td>File may be added</td>
<td>File may be added</td>
</tr>
<tr>
<td>Report BGTask definition or stored procedure</td>
<td>Application</td>
<td>New object</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Update to base object</td>
<td>Update may be overwritten</td>
<td>Update may be overwritten</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Deletion of base object</td>
<td>Object may be reinserted</td>
<td>Object may be reinserted</td>
</tr>
</tbody>
</table>

* Form objects include form components, variables, menus, strings, validators, scripts, property class extensions, component classes, UETs, and UDFs.
Index

Symbols
@Infobar Parameter 3-3

A
AFTER UPDATE Trigger 6-2
Application Messages 7-2
   Custom 7-4

C
Configuration Manager 1-4
Copy License Information 8-4
Cursors 5-8
   Using 5-11
Custom Application Messages 7-4

D
Data Types 5-17
Database
   Schema Changes 2-8
Deadlocks 5-7
Dynamic IDO 4-2
   Collection source 4-5
   Event Handlers 4-3
   Stored Procedure Calls 4-3
Dynamic SQL 5-13

E
Environment Configuration B-1
External Touch Points
   Code 3-3
      using 3-7

F
Form Creation
   Licensing 8-3
FormControl 1-10
Forms Database
   Numeric Constants C-1
FormSync 1-12

I
IDO
   Dynamic 4-2
      Non-transactional methods 5-5
INSTEAD OF INSERT Trigger 6-2

L
Licensing
   Copy License Information 8-4
   Form Creation 8-3
   License Management 8-2
   ModuleMembers table 8-2
   Multi-Site 8-4
Locking
   READ UNCOMMITTED 5-2
   Locking and Blocking 5-2
      Read Locks 5-2
      Transaction Size 5-4

M
Messages
   Application 7-2
metadata.h C-1
ModuleMembers table 8-2
Multi-Site
   Licensing 8-4

N
Numeric Constants C-1

P
Performance
   Diagnosing 5-18
   SQL Commands 5-18
prerequisite knowledge -vi

R
Read Locks 5-2
   READ UNCOMMITTED 5-2
   READ UNCOMMITTED 5-2
   recompiling, procedures 5-16
Reports 2-9
requirements
   system -vi

S
Schema Changes 2-8
Service Packs 3-8
   Impact on Customizations and Modifications E-1
SourceSafe B-2
Stored Procedures
   Custom Messages 7-4
   support
      signing up for -vii
      system requirements -vi

T
Table Variables 5-14
Temporary Tables 5-14
Transaction Size 5-4
Triggers 5-13
   AFTER UPDATE 6-2
   INSTEAD OF INSERT 6-2
   Maintenance 6-4
   Trigger Management form 6-3

U
UET
   Applying Scripts in Target Environment D-3
ExportUETClassSp  D-4
Extracting Definitions from Source Environment  D-2

V
Version Upgrades  3-8

W
WinStudio  1-10

VSS  B-2