Disclaimer
This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it. Some examples are for illustration only and are fictitious. No real association is intended or inferred. This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes only.

Sample Code Warranty disclaimer
Microsoft Corporation disclaims any warranty regarding the sample code contained in this documentation, including the warranties of merchantability and fitness for a particular purpose.

License agreement
Use of this software is covered by a license agreement provided with it. If you have any questions, please call the Customer Assistance Department at 800-456-0025 (in the United States or Canada) or +1-701-281-6500.

Copyright
© 2014 Microsoft Corporation. All rights reserved.

Publication Date
September 2014
# Contents

## Introduction
- About This Manual ................................................................. 1
- Module Overview ........................................................................ 1
- Benefits of Using the Microsoft Dynamics SL Software Development Kit ................................................................. 1
- Setting Up a Development Environment ........................................... 3
- Developer Guidelines for the Microsoft Dynamics SL SDK .................. 4
- The .NET Framework Development Environment ......................... 4
- Screen Design ............................................................................. 5
- Database Considerations .......................................................... 9

## Special Application Requirements
- Passing Parameters Between Applications ...................................... 15
  - Using Parameter-Passing Functions ............................................ 15
  - Building and Passing Parameters .............................................. 16
  - Retrieving Parameters .......................................................... 16
  - Returning Parameters .......................................................... 17
- Creating Applications Compatible with Customization Manager ....... 18
  - Using Required Components ................................................... 18
  - Using Required Coding Practices ............................................ 18
  - Using Supported Controls ....................................................... 19
  - Ensuring Customization Manager Usability ................................... 19
- Displaying Text in the Status Bar .................................................. 20
- Providing Database Configuration and Import Definition Files with Applications ................................................................. 22
  - Overview of the Configuration File .............................................. 22
  - [Scenarios] Section ................................................................ 22
  - [Scenario#] Sections ............................................................... 23
  - [MSSQL System Scripts] and [MSSQL Applications Scripts] ......... 24
  - [Update Scenarios] Section ...................................................... 27
  - [Update ScenarioX] Section ...................................................... 28
  - Import File Format .................................................................. 30
- Adding Custom HTML Help ....................................................... 31
  - Setting Up Help ..................................................................... 31
  - Understanding Help Requirements .......................................... 31
  - Help File Design Logic ......................................................... 31
  - Building Help Files .................................................................. 31
- Utilities and Add-Ins ................................................................. 33
  - Adding Custom Controls .......................................................... 33
  - Detail Levels .......................................................................... 33
  - Pasting a Control onto a Form from the Visual Basic Toolbox .......... 34
  - Defining New Tables ............................................................. 35
  - Defining New Fields ................................................................ 36
- Creating Tables ........................................................................... 37
  - Microsoft Dynamics SL Standard File Name Extensions .................. 37
  - Defining the Table Name and Fields ........................................... 37
  - Generate an SQL CREATE TABLE Script, SDO, and DH File .......... 38
  - Creating the Table in the Database ......................................... 38
  - Creating a Unique Index on the New Table ......................... 38
- Checking Code ........................................................................... 40
Using the VB Code Inspector ........................................................................................................ 40
Validating Data-Binding Values on Manually Bound Controls .................................................. 40
Checking for Required Controls, References, and Forms ...................................................... 41
Changing the Default Processing Configuration ..................................................................... 41
The VB Code Inspector Log File .............................................................................................. 44
Required Fields .......................................................................................................................... 46
Adding Applications to the Menu System ............................................................................... 47
The Parent Application ............................................................................................................... 47
System Database Tables .......................................................................................................... 47
Menu Cache Files ..................................................................................................................... 47
Maintenance Screens ................................................................................................................ 47

Reference .................................................................................................................................. 49
Controls ..................................................................................................................................... 49
DSLDate Control ....................................................................................................................... 49
DSLCheck Control ..................................................................................................................... 51
DSLCCombo Control ................................................................................................................... 52
DSLFloat Control ...................................................................................................................... 53
DSLGrid Control ....................................................................................................................... 54
DSLInteger Control .................................................................................................................... 56
DSLMaskedTextBox Control ..................................................................................................... 57
DSLOption Control ..................................................................................................................... 59
DSLUpdate Control ................................................................................................................... 61
Properties ................................................................................................................................... 62
Alignment Property ................................................................................................................... 62
BackColor Property .................................................................................................................... 62
BlankErr Property ..................................................................................................................... 63
Caption Property ....................................................................................................................... 63
ColsFrozen Property ................................................................................................................. 64
Custom Property Page ............................................................................................................... 64
Customizable Property .............................................................................................................. 64
DBNav Property ......................................................................................................................... 65
DecimalPlaces Property ............................................................................................................ 67
Default Property ....................................................................................................................... 67
DragIcon Property ..................................................................................................................... 68
DragMode Property ..................................................................................................................... 68
Enabled Property ....................................................................................................................... 68
FalseText Property .................................................................................................................... 69
FieldClass Property ................................................................................................................... 69
Flex Key Fieldclasses ................................................................................................................. 70
Non-Flex Key Fieldclasses ......................................................................................................... 72
FieldName Property ................................................................................................................... 76
Font Property ............................................................................................................................. 77
ForeColor Property ..................................................................................................................... 77
Heading Property ....................................................................................................................... 78
Height Property ......................................................................................................................... 78
HelpContextID Property ........................................................................................................... 79
InGrid Property .......................................................................................................................... 79
Left Property .............................................................................................................................. 80
Level Property ............................................................................................................................ 80
Levels Property .......................................................................................................................... 81
List Property .............................................................................................................................. 82
Mask Property ............................................................................................................................ 83
Max Property .............................................................................................................................. 84
Min Property ............................................................................................................................... 84
MouseIcon Property ................................................................................................................. 85
MousePointer Property ............................................................................................................. 85
Name Property ........................................................................................................................... 86
NoteButton Property ................................................................. 86
NoteColumn Property ................................................................. 87
PV Property ............................................................................. 87
Separator Property ................................................................. 89
Spin Property ........................................................................ 89
SpinIncrement Property ......................................................... 90
TabIndex Property ................................................................. 90
TabStop Property .................................................................... 91
Tag Property ........................................................................... 91
ToolTipText Property ............................................................. 91
Top Property ........................................................................... 92
Trigger Property ..................................................................... 92
TrueText Property .................................................................. 94
Visible Property ..................................................................... 94
Width Property ....................................................................... 95

Events ......................................................................................... 97
Cancel Event ............................................................................ 97
Chk Event ............................................................................... 98
Default Event .......................................................................... 101
Delete Event ........................................................................... 102
DragDrop Event ........................................................................ 104
DragOver Event ........................................................................ 104
Finish Event ............................................................................ 104
GotFocus Event ......................................................................... 105
LineChk Event .......................................................................... 106
LineGotFocus Event .................................................................. 107
LostFocus Event ....................................................................... 108
MemoryLoad Event ................................................................... 109
MouseDown Event ...................................................................... 109
MouseMove Event ...................................................................... 110
MouseUp Event ......................................................................... 111
NewLevel Event ....................................................................... 112
QuickPrint Event ...................................................................... 114
Update Event ............................................................................ 115

Methods .................................................................................... 117
AboutBox Method ...................................................................... 117
Drag Method ............................................................................ 117
Move Method ............................................................................ 117
SetFocus Method ...................................................................... 117
ShowWhatsThis Method .......................................................... 118
ZOrder Method ......................................................................... 118

API Function Calls .................................................................... 119
ApplGetParms Function .......................................................... 119
ApplGetParmValue Function .................................................. 120
ApplGetReturnParms Function ................................................ 122
ApplInit Statement .................................................................. 122
ApplSetFocus Statement ......................................................... 123
ApplSetParmValue Statement ................................................... 124
AutoNbrDefault Function ....................................................... 126
Button_Form_Change Statement ............................................. 128
Button_Level_Change Statement ............................................ 128
CallApplic Statement ............................................................. 129
CallApplicWait Statement ....................................................... 130
CallChks Statement .................................................................. 131
ChkSqlException Function ..................................................... 132
ChkCuryRateType Function ..................................................... 133
CurrencyField Statement ....................................................... 135
CurrencyInfo Statement .......................................................... 138
<table>
<thead>
<tr>
<th>Function/Statement</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CuryEffDateSet Statement</td>
<td>140</td>
</tr>
<tr>
<td>CuryFieldCalcSet Statement</td>
<td>141</td>
</tr>
<tr>
<td>CuryInfoEnable Statement</td>
<td>143</td>
</tr>
<tr>
<td>CuryInfoGet Statement</td>
<td>145</td>
</tr>
<tr>
<td>CuryInfoInit Statement</td>
<td>148</td>
</tr>
<tr>
<td>CuryIdSet Statement</td>
<td>151</td>
</tr>
<tr>
<td>CuryInfoSet Statement</td>
<td>152</td>
</tr>
<tr>
<td>CuryRateTypeSet Statement</td>
<td>155</td>
</tr>
<tr>
<td>CurySelFieldEnable Statement</td>
<td>157</td>
</tr>
<tr>
<td>DateCmp Function</td>
<td>160</td>
</tr>
<tr>
<td>DateMinusDate Function</td>
<td>161</td>
</tr>
<tr>
<td>DatePlusDays Statement</td>
<td>161</td>
</tr>
<tr>
<td>DatePlusMonthSetDay Statement</td>
<td>162</td>
</tr>
<tr>
<td>DateToIntStr Function</td>
<td>163</td>
</tr>
<tr>
<td>DateToStr Function</td>
<td>163</td>
</tr>
<tr>
<td>DateToStrSep Function</td>
<td>163</td>
</tr>
<tr>
<td>DBNavFetch Functions</td>
<td>164</td>
</tr>
<tr>
<td>DecimalPlaces Statement</td>
<td>165</td>
</tr>
<tr>
<td>DetailLoad Statement</td>
<td>167</td>
</tr>
<tr>
<td>DetailSave Statement</td>
<td>169</td>
</tr>
<tr>
<td>DetailSetup Functions</td>
<td>170</td>
</tr>
<tr>
<td>DetailSetupExtend Function</td>
<td>172</td>
</tr>
<tr>
<td>DispField Statements</td>
<td>173</td>
</tr>
<tr>
<td>DispForm Statement</td>
<td>174</td>
</tr>
<tr>
<td>DisplayMode Statement</td>
<td>175</td>
</tr>
<tr>
<td>DisplayModeSetProps Statement</td>
<td>175</td>
</tr>
<tr>
<td>DParm Function</td>
<td>176</td>
</tr>
<tr>
<td>ExportCustom Function</td>
<td>177</td>
</tr>
<tr>
<td>FPAdd Function</td>
<td>179</td>
</tr>
<tr>
<td>FParm Function</td>
<td>180</td>
</tr>
<tr>
<td>FPDiv Function</td>
<td>180</td>
</tr>
<tr>
<td>FPMult Function</td>
<td>181</td>
</tr>
<tr>
<td>FPPlus Function</td>
<td>182</td>
</tr>
<tr>
<td>FPSub Function</td>
<td>183</td>
</tr>
<tr>
<td>FtoA Function</td>
<td>184</td>
</tr>
<tr>
<td>GetCuryRate Statement</td>
<td>184</td>
</tr>
<tr>
<td>GetModulePeriod Function</td>
<td>186</td>
</tr>
<tr>
<td>GetPrecCury Function</td>
<td>187</td>
</tr>
<tr>
<td>GetPrecision Function</td>
<td>188</td>
</tr>
<tr>
<td>GetSWIMDefaultPrintInfo Function</td>
<td>189</td>
</tr>
<tr>
<td>GetSWIMPrtInfo Function</td>
<td>191</td>
</tr>
<tr>
<td>GetSysDate Statement</td>
<td>191</td>
</tr>
<tr>
<td>GetSysTime Statement</td>
<td>191</td>
</tr>
<tr>
<td>Grid_Sortable Statement</td>
<td>192</td>
</tr>
<tr>
<td>HideForm Statement</td>
<td>193</td>
</tr>
<tr>
<td>HideNoteButtons Statement</td>
<td>193</td>
</tr>
<tr>
<td>ImportCustom Function</td>
<td>194</td>
</tr>
<tr>
<td>IncrStrg Statement</td>
<td>197</td>
</tr>
<tr>
<td>InitLocalizationSubsystem Function</td>
<td>197</td>
</tr>
<tr>
<td>IntToStrToDate Statement</td>
<td>198</td>
</tr>
<tr>
<td>IParm Function</td>
<td>199</td>
</tr>
<tr>
<td>IsAdministrator Function</td>
<td>200</td>
</tr>
<tr>
<td>ISAppAutomating Function</td>
<td>200</td>
</tr>
<tr>
<td>IS_AppServer Function</td>
<td>200</td>
</tr>
<tr>
<td>IS_TI Function</td>
<td>201</td>
</tr>
<tr>
<td>IsMultiCompany Function</td>
<td>201</td>
</tr>
<tr>
<td>Level_SetDefaults Statement</td>
<td>201</td>
</tr>
<tr>
<td>Function/Statement</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>LoadForm Statement</td>
<td>203</td>
</tr>
<tr>
<td>LoadStr Function</td>
<td>205</td>
</tr>
<tr>
<td>Localize Statement</td>
<td>206</td>
</tr>
<tr>
<td>MArrayCnt Function</td>
<td>207</td>
</tr>
<tr>
<td>MCallChks Statement</td>
<td>208</td>
</tr>
<tr>
<td>MClear Statement</td>
<td>209</td>
</tr>
<tr>
<td>MClose Statement</td>
<td>209</td>
</tr>
<tr>
<td>MDelete Function</td>
<td>210</td>
</tr>
<tr>
<td>MDisplay Statement</td>
<td>211</td>
</tr>
<tr>
<td>Mess Statement</td>
<td>213</td>
</tr>
<tr>
<td>MessBox Statement</td>
<td>215</td>
</tr>
<tr>
<td>Messf Statement</td>
<td>217</td>
</tr>
<tr>
<td>MessGetText Function</td>
<td>219</td>
</tr>
<tr>
<td>MessResponse Function</td>
<td>220</td>
</tr>
<tr>
<td>MExtend Function</td>
<td>221</td>
</tr>
<tr>
<td>MFirst Function</td>
<td>222</td>
</tr>
<tr>
<td>MGetDelHandle Function</td>
<td>222</td>
</tr>
<tr>
<td>MGetLineStatus Function</td>
<td>223</td>
</tr>
<tr>
<td>MGetRowNum Function</td>
<td>223</td>
</tr>
<tr>
<td>MInsert Statement</td>
<td>224</td>
</tr>
<tr>
<td>MKey Statement</td>
<td>225</td>
</tr>
<tr>
<td>MKeyFind Function</td>
<td>226</td>
</tr>
<tr>
<td>MKeyFld Statement</td>
<td>228</td>
</tr>
<tr>
<td>MKeyHcti Statement</td>
<td>229</td>
</tr>
<tr>
<td>MKeyOffset Statement</td>
<td>230</td>
</tr>
<tr>
<td>MLast Function</td>
<td>230</td>
</tr>
<tr>
<td>MLoad Statement</td>
<td>233</td>
</tr>
<tr>
<td>MNext Function</td>
<td>233</td>
</tr>
<tr>
<td>MOpen Functions</td>
<td>234</td>
</tr>
<tr>
<td>MPrev Function</td>
<td>235</td>
</tr>
<tr>
<td>MSet Statement</td>
<td>237</td>
</tr>
<tr>
<td>MSetLineStatus Function</td>
<td>237</td>
</tr>
<tr>
<td>MSetProp Statement</td>
<td>239</td>
</tr>
<tr>
<td>MSetRow Statement</td>
<td>240</td>
</tr>
<tr>
<td>MSort Statement</td>
<td>240</td>
</tr>
<tr>
<td>MUpdate Statement</td>
<td>241</td>
</tr>
<tr>
<td>NoteCopy Function</td>
<td>241</td>
</tr>
<tr>
<td>PasteTemplate Function</td>
<td>242</td>
</tr>
<tr>
<td>PeriodCheck Function</td>
<td>242</td>
</tr>
<tr>
<td>PeriodMinusPeriod Function</td>
<td>243</td>
</tr>
<tr>
<td>PeriodPlusPerNum Function</td>
<td>243</td>
</tr>
<tr>
<td>PVChk Function</td>
<td>244</td>
</tr>
<tr>
<td>PVChkFetch Functions</td>
<td>244</td>
</tr>
<tr>
<td>SaveTemplate Statement</td>
<td>245</td>
</tr>
<tr>
<td>ScreenExit Statement</td>
<td>246</td>
</tr>
<tr>
<td>ScreenInit Statement</td>
<td>247</td>
</tr>
<tr>
<td>SDelete Statement</td>
<td>248</td>
</tr>
<tr>
<td>SDeleteAll Function</td>
<td>248</td>
</tr>
<tr>
<td>SetAddr Statement</td>
<td>249</td>
</tr>
<tr>
<td>SetAutoNbrFlag Statement</td>
<td>253</td>
</tr>
<tr>
<td>SetButton Statement</td>
<td>253</td>
</tr>
<tr>
<td>SetDefaults Statement</td>
<td>254</td>
</tr>
<tr>
<td>SetKeysEnabledOnly Statement</td>
<td>255</td>
</tr>
<tr>
<td>SetLevelChg Statement</td>
<td>256</td>
</tr>
<tr>
<td>SetProps Statement</td>
<td>257</td>
</tr>
<tr>
<td>SetRestart Statement</td>
<td>259</td>
</tr>
<tr>
<td>SetStatusBarText</td>
<td>262</td>
</tr>
<tr>
<td>SetSWIMPrintInfo Function</td>
<td>263</td>
</tr>
</tbody>
</table>
Appendix A: Toolset Limitations
Microsoft Dynamics SL SDK Limitations

Appendix B: Requirements for System Table Views Stored Procedures

Appendix C: Extending Doc Share's Capabilities
Introduction ............................................................................................................................... 303
Common Doc Share Terms...................................................................................................... 303
Database Tables for Doc Share ............................................................................................... 303
Adding an Entity Type .............................................................................................................. 304
Adding a Document Type to a New Instance Type ................................................................. 306
Adding a Document Type to an Existing Instance Type.......................................................... 308
Modifying a SharePoint Site.................................................................................................... 310
Microsoft Dynamics SL SharePoint Client Methods............................................................... 310
Implementation Examples ....................................................................................................... 316
Creating a Site .......................................................................................................................... 316
Uploading a Document .......................................................................................................... 317

Appendix D: Microsoft Dynamics SL SDK Application Upgrade Utility
Available Upgrade Options......................................................................................................... 320
Steps to Upgrade Applications ................................................................................................. 322
Post Upgrade Issues and How to Fix Them ............................................................................. 323

Appendix E: Running the .NET Framework-connected Applications from a Network

Appendix F: Customizing Role Center Data
Adding Activities entries......................................................................................................... 327
Adding an Activity Group...............................................................................................328
Adding an Activity Cue ..........................................................................................329
Adding Quick List Entries ..........................................................................................331
Adding a Quick List ..............................................................................................332
Adjusting the Display Behavior for Role Centers ....................................................333
Data Definitions ........................................................................................................334
  RCAactivity .........................................................................................................334
  RCCue ................................................................................................................334
  RCPart ..............................................................................................................335

Index .......................................................................................................................337
Introduction

About This Manual
This manual is organized based on the assumption that you have already attended the Microsoft Dynamics® SL Software Development Kit (Microsoft Dynamics SL SDK) hands-on training course. The course will help you quickly become productive with the Microsoft Dynamics SL SDK and achieve maximum return on your investment. It is advisable that you periodically see the training manual for hands-on reinforcement on the topics in this manual and the online help.

Module Overview
The Microsoft Dynamics SL SDK is the same development platform used by Microsoft developers to create Microsoft Dynamics SL. These tools allow you to leverage your existing knowledge of Microsoft® SQL Server® and Microsoft® Visual Basic® to create a tightly-integrated, complete line of business applications.

Note: Microsoft Dynamics SL SDK is compatible with Visual Studio 2010 and Visual Studio 2012. The SDK requires that you install the Microsoft .NET Framework 4.5.

Projects that you create must target either the .NET Framework 4 or the .NET Framework 4.5.

Benefits of Using the Microsoft Dynamics SL Software Development Kit
There are many reasons for using the Microsoft Dynamics SL SDK. Following is a list of some of the main benefits you will gain from using these tools:

- Common look and feel with the rest of Microsoft Dynamics SL
  If you want, your programs can be almost indistinguishable from the rest of the product. This includes full integration with the toolbar, F3 lookup, and standardized messaging.

- Support for the Customization Manager module
  Users can customize your applications exactly the way they can customize existing Microsoft Dynamics SL applications. Create source code that meets most of the needs of many different clients or customers, then let them or their reseller apply the finishing touches using Customization Manager!

- Integration with Microsoft Dynamics SL’s robust security functionality
  By simply naming your program files correctly, making sure the caption (Form.Text) on your main form is correct, and inserting a single record into one of the system tables, your application automatically supports standard security functionality. Without a single line of code, users can be given levels of access to your programs ranging from no access at all to full access.

- Built-in multi-user contention checking
  Probably one of the best features of the Microsoft Dynamics SL SDK is that you are developing full featured, client/server applications for a system that can support from one to hundreds of simultaneous users. However, when you are writing your code, you can write as though you were developing a single-user, single-access system. This is because the Microsoft Dynamics SL kernel has a built-in optimistic updating algorithm, which automatically handles cases in which more than one user or process attempts to update the same record, or logically related records at the same time. The system will automatically cancel changes from the second user, and display appropriate messages.

- Speed of development
Finally, after you have mastered Microsoft Dynamics SL SDK, you will find that much of the standard, routine programming work required for all database applications is handled automatically for you. This will make your development time significantly shorter than with traditional PC-based development tools.

Note that the referential integrity, data flow, and business rules are not incorporated in either the kernel or the database itself. Maintaining the logical integrity of the data in the database is the responsibility of the programmer, although features of the Microsoft Dynamics SL SDK make this easier.
Setting Up a Development Environment

There is no right or wrong way to configure your development environment. This section is intended to suggest an approach that may work for you.

In setting up your environment, there are several issues you may want to consider:

- There are many standard files that must be included in any project. Most of these files should never be modified.
- Almost every data entry, maintenance, or process window you might need will probably be a completely separate program file.
- You may be developing for multiple versions of Microsoft Dynamics SL.
- You may be moving your development environment from one computer to another (for example, from a desktop to a laptop).
- You may be using version control software.
Developer Guidelines for the Microsoft Dynamics SL SDK

The Microsoft Dynamics SL SDK provides the development environment to create Microsoft Dynamics SL applications. It provides a collection of extensions and components that provide many of the common UI characteristics and global features that are available throughout Microsoft Dynamics SL.

To take full advantage of many of these features, the application developer must implement development standards to ensure that these features can be used to their fullest extent and to ensure that the user experience is common throughout all Microsoft Dynamics SL applications.

This section provides developers with a set of guidelines to ensure that applications created with Microsoft Dynamics SL SDK compile, exhibit little or no unexpected behavior, and take optimal advantage of these Microsoft Dynamics SL SDK global facilities:

- Application server
- Copy and paste
- Customization manager
- Detail grids
- International localization
- Mouse and keyboard conventions
- Notes
- Attachments
- Object Model
- Possible value windows
- Process status window
- Security model
- Status bar
- Templates
- Toolbar navigation
- Transaction Import
- Required Fields notations

The following topics describe the .NET development environment, screen design, and database considerations.

The .NET Framework Development Environment

The following files are already included in the Microsoft Dynamics SL application template project (VBTPProject.vbp):

- Dynamics.SL.SDK.vb
- ApplicationEvents.vb
- Form1.frm

The following references are required in every Microsoft Dynamics SL SDK project file and are included in the template project:

- Microsoft.Dynamics.SL.Controls (Microsoft.Dynamics.SL.Controls.dll)
- Solomon.Kernel (Solomon.Kernel.dll)
Screen Design

Controls

Update control — Only one Update control is allowed for a project. It is placed on Form1. This is checked by VB Code Inspector.

CBegProcessing — For a process application to be available to the application server, a button control must exist on the form with the name cBegProcessing. This button is used to begin the process.

Form1

The Form1.Text property must contain the .exe file name in parentheses in the form of MyAppTitle (MM.SSS.SS) where the application name is MMSSSSS.exe. Failure to follow this standard results in a runtime error saying that the application does not match the caption. This is checked by VB Code Inspector.

Events

The Form1 Form_Load events process in the following order:

- LoadForm() calls
- Applinit
- SetAddr calls
- SqlCursorEX calls
- CurrencyInfo calls
- ScreenInit
- DetailSetup calls

Form_Unload — ScreenExit(“”, “”) should be called here. The use of “End” instead of this call does not “clean up” all of the objects created by Swim.

Update1_Update — TranEnd should never be called from within the Update event. A transaction begins just after the “UpdateStart” level in the Update1_Update event is run. Levels are then processed from the header level to the most detailed level. If a Tranend is called in an intermediate level when prior levels have already been processed, the earlier level data will be committed to the database without its corresponding detail level data.

This is checked by VB Code Inspector.

Tab_Click — There should be no code in the Tab_Click events. In Customize mode, these events are not suppressed and result in unexpected behavior.

This is optionally checked by VB Code Inspector if the “Customization Compatibility” Inspection Option is selected.

Button_Click — Logical code in the Button_Click event that is used to populate a subform will not be processed in Transaction Import mode.

Chk — Code should not be placed in a Chk event that loads a subscreen, and enables fields. This causes problems with Transaction Import.

Because Transaction Import loads all subscreens at application load time, it does not know that the fields will be enabled and will result in partial data being saved to the database.

Transaction Import — Special consideration needs to be made for Transaction Import when code exists in the Form_Load event of a subform that acts upon data previously entered in the main form.

Because Transaction Import loads all subscreens at application load time, the data expected is not available.
Property Considerations

**DSLMaskedText** — a Mask property is required for this control type. This is checked by VB Code Inspector. The Mask property should never be larger than the actual field size. That would result in the value for the control not displaying at all. This is checked by VB Code Inspector.

The Mask property should in most cases, match the length of the field defined for that control. If one application populates the full length and another application shows only a portion of that value, information is truncated, and if updated to other records the truncated value is passed along. This is optionally checked by VB Code Inspector if the Warning Option for “Controls with a Mask that is smaller than the field’s defined size” is selected.

**Setting property values at runtime** — Avoid using `control.property = somevalue`. Customization Manager has no knowledge of properties set using this method. The correct method is to use the Setprops and/or MsetProp Microsoft Dynamics SL APIs. This is optionally checked by VB Code Inspector if the Warning Option for “Explicit Property Assignments vs SetProps” is selected.

**Triggers** — The maximum number of controls with a trigger property that points to a single control is 8.

**PV or DBNav property** — If the stored procedure being used contains a LIKE keyword, the “wildcard” check box should be selected. The property should be populated for the last key field of every Navigation and Lookup level. This is optionally selected by VB Code Inspector if the Warning Option “No PV or DBNAV property on last key field of Navigation level” is selected.

The maximum number of parameters allowed for a PV or DBNav is 8. When passing an integer parameter to DBNAV to retrieve multiple rows, use the BETWEEN keyword in the WHERE clause of the Select statement.

**Control Size** — The size of all controls that contain text information (that is, DSLMaskedText, Labels, DSLCombo) should be 35% larger size than the English text requires (for international language support).

**Min and Max** — If a field class of 124, 125, or 126 is specified for a DSLFloat field, the Min and Max property values should be filled with all 9s.

If the Min and Max properties contain some other value, the correct precision still displays, but when precision flexes to fewer than the number of decimal places in the Decimalplaces property, the field size does not expand into the extra digits.

This is optionally checked by VB Code Inspector if the Warning Option “Controls with Improper Min and Max (values International)” is selected.

**Tab order**

The tabindex of a label that corresponds to a control should immediately precede the tabindex of that control. When using Hot Key mnemonics (that is, &Detail) for control positioning, the Microsoft Dynamics SL kernel positions the cursor at the control whose tabindex value follows the tabindex value for the label where the hot key is identified. If this is not done and a user customizes the label to add a Hot Key, the cursor is positioned on the next control in tab order—not necessarily the next control visually.

When in a grid screen, it is important to make the grid container (that is, Frame) the first tabindex value, then all controls and labels within the frame should have contiguous tabindex values. This is optionally checked by VB Code Inspector if the Warning Option “Tabindex values that jump out and back into a container” is selected.

The tabindex should never be set to allow the cursor to move from inside the grid to outside of the grid, then back into the grid. This is optionally checked by VB Code Inspector if the Warning Option “Tabindex values that jump out and back into a container” is selected.

**Levels**

The levels identified in the Level property of the Update control must match the levels specified in the SQLCursorEX calls.

The data for this property is formatted so that the level identified as LEVEL0 should be first. LEVEL1 second, and so forth. The choices for Level Type are:
• **N** — Navigation Level.
• **L** — Lookup Level. This level must be preceded by an N Level.
• **D** — Detail Level. This level is associated with a grid.
• **DA** — Detail (application loaded). If the application uses DetailLoad to load the grid, the Microsoft Dynamics SL kernel automatically handles the update. If the application does not use DetailLoad, the application must handle all updates for this level manually.
• **C** — Constant Level. Used when no database navigation is necessary for this level. The same record is always displayed. This is checked by VB Code Inspector.

Although you can have multiple D and DA levels, they should appear last in the Level property unless the levels beyond the D or DA levels are intended to be non-navigable. This is optionally checked by VB Code Inspector, by selecting the Warning Option “Order of Levels specified in Update Levels property”.

The level property is formatted as:

```
LEVEL0 alias;LevelType,LEVEL1 alias;LevelType
```

The level alias has a maximum of 19 characters. This is checked by VB Code Inspector.

Controls are templated in tab order within each level. Controls with a blank in the level property are processed as LEVEL0. The maximum number of levels allowed is 10 (LEVEL0 – LEVEL9). This is checked by VB Code Inspector.

The maximum number of N(navigation) type levels is 2. This is checked by VB Code Inspector.

The Level number in Setaddr must match the Level Number in SQLCursorEX. The Cursor defined in SQLCursorEX must match the cursor defined in the DetailSetup call.

**SetAddr calls**

A SetAddr call must be made for every record referenced by a control on the screen. This is checked by VB Code Inspector.

Additional records that are accessed by the application do not require a SetAddr call. However, making a SetAddr call for the record, makes that record easily available for customization. Also, the use of the SetAddr call properly initializes the buffer to blanks for strings (instead of nulls). There is, however, a possible performance tradeoff in making too many SetAddr calls.

It is important for the second parameter of the SetAddr call (buffername) to be a “b” followed by the full table name. However, it is often necessary to retrieve to separate instances of the same table. When it is necessary to create a second buffer for the additional instance, a number can be placed at the end of the btablename (that is, bcustomer1). When this guideline is not followed, errors may occur in the application.

**Grids**

For grid functionality, it is important for all controls contained within the grid as well as the grid control itself, to be “contained” within the frame (or whatever container is selected). This can be tested by attempting to move a control outside the frame that surrounds the grid data (in design mode).

If the control cannot be moved outside the frame, it is contained within the frame. If it can be moved outside the frame, these controls must be selected, then Cut, then focus on the frame and Paste. Keep in mind that this operation will probably alter the tabindex properties and they will need to be reset.

The maximum number of grids per application is 10. This is checked by VB Code Inspector. The maximum number of tables associated with a single grid is 16.

No visual controls within a grid container should be placed outside the visual boundaries of that container. This approach makes controls visible in Grid view, but not in Form view. This is optionally checked by VB Code Inspector, if the Warning Option “Visible Controls outside the Grid container” is selected.

It is now possible to have an inquiry grid on the applications that can be sorted by the user by clicking on the header of a column (see “Grid_Sortable Statement” on page 192).
Memory arrays

When looping through memory arrays in code, it is important that you preserve your original position in the array and reset back to that position when you are finished with the loop. You do this with the MgetRowNum, and MsetRow functions. Failure to do this causes the application to display several errors about the memory array not matching the grid. (that is, bulletproof 10061, 11008, 20007). It can also cause the Cut-Copy-Paste function to fail.

An application may have an unlimited number of memory arrays. The maximum number of keys allowed for a single memory array is 5.

Messages

The Microsoft Dynamics SL Message APIs should be used in all cases. Transaction Import is unable to recognize and respond to Visual Basic message calls. The use of Visual Basic message calls will cause an Application Server to stop responding when running the application.

Messages should never be hard-coded in an application. Messages.csv or an equivalent method should be used, so that the messages may be translated if necessary.

Avoid concatenating multiple messages. The syntax of many languages is different, so translation of the individual pieces may be difficult.

For process status messages, labels are often placed on the form with their visible property set to False. This way, the application can reference the control.caption or control.text property to display the message.

Never display a messages during an open transaction. A transaction is open after a TranBeg has been called and inside the Update1_Update or Update1_Delete events.

Never call TranEnd from within the Update1_Update event to allow a message to be displayed.

Status()

Make status calls only in process applications. In these cases, the Microsoft Dynamics SL kernel does not set TranBeg automatically and will not interfere with these calls.

Never make a status call in the Update1_Update Event. In this case, the kernel has already started a transaction. Placing a status call here would cause record or page locking errors.

Unique control names

All controls within an application (including subforms) MUST have unique names. This is checked by VB Code Inspector.

Key fields

All Navigation and Lookup levels should have at least one key field. This is optionally checked by VB Code Inspector if the Warning Option “No Key field defined for Navigation or Lookup level” is selected.

All key fields (.k) should have their BlankErr properties set to True for “D” or “DA” levels. This is checked by VB Code Inspector.

The last key field for every level, must have its PV or DBNav property populated. This is optionally checked by VB Code Inspector if the Warning Option “No PV or DBNAV property on last key field of Navigation level” is selected.

The maximum number of key fields per level is 5. This is checked by VB Code Inspector.

Special coding considerations

CallApplicWait will have trouble finding the .exe file if you do not specify the full path relative to the Microsoft Dynamics SL folder.

Unbound controls

To define unbound controls place values in the Field Offset, Field Type, and Field Length properties (on the Fieldname Property page).
• If a value exists in any of these properties, VB Code Inspector identifies the control as unbound and attempts to calculate its offset. So, if you do not want to do this yourself, you can place a 99 in any of the properties, then run VB Code Inspector and it will calculate the values for you.

• Even if the field is **not** unbound (that is, the table exists in the database), but values exist in the Field Offset, Field Type, or Field Length property, VB Code Inspector attempts to make the calculation. However, the Microsoft Dynamics SL kernel knows that the field is not unbound and these values are not used. This is checked by VB Code Inspector.

**Third-party controls**

Third-party controls have limited properties that may be customized. They are not, however, accessed by templating, which raises issues with application server as well, because templates are the way that application server fills the screen that is to be processed.

In this case, data keyed into third party controls is not captured in the template and therefore, is unavailable to the application server process. This is optionally checked by VB Code Inspector if the Warning Option “Third Party controls used in the Application” has been selected.

**Microsoft Dynamics SL SDK Size Limitations**

The Microsoft Dynamics SL SDK size limitations are:

• Maximum number of characters in a table name is 17.
• Maximum number of characters in a column name is 20.
• Maximum number of columns in a table is 500.
• Maximum length of a Microsoft Dynamics SL SDK control name is 30.
• Maximum number of characters in a Microsoft Dynamics SL SDK combo box list property is 32,000.
• Maximum size of one description line for Microsoft Dynamics SL SDK combo box is 81.
• Maximum number of bytes that can be written in a single insert or fetched in a single fetch is 30,500.
• Maximum number of database tables is 1000.
• Maximum number of forms per project is 30.
• Maximum number of fields returned in an SQL Select statement is 255.

**Application Menu and Toolbar**

The screens in all of the Microsoft Dynamics SL applications and the sample application include the menu and toolbar. They are automatically added when the applications load. No special coding is required.

**Database Considerations**

**Cursors**

The maximum number of cursors available to an application is 50 per database (50 for accessing the system database and 50 for accessing the application database). The maximum number of cursors available to customizations is 30 per database.

If an SQLCursor or SQLCursorEX call is not made for a cursor, it is automatically declared as SQLReadOnly and is not freed automatically.

SQLCursorEX is the preferred API call. It allows the application to specify additional parameters that optimize data access for SQL Server. This is optionally checked by VB Code Inspector if the Warning option “Use of SQLCursor vs SQLCursorEX in the application” is selected.

Call SQLFree when you are finished using the cursor.
Indexes

All Microsoft Dynamics SL database tables require a unique index, which is the full table name with a zero appended. (for example, Customer0).
The maximum number of fields in an index is 8.

Solomon Data Object Files

Solomon Data Object (.sdo) files contain classes that represent the database tables. Each class has properties representing the fields in a table. Each property has attributes that submit size and offset information to the Microsoft Dynamics SL SDK.

The example below shows a declaration of class ScreenMenu, and illustrates the following important concepts related to using Solomon Data Objects:

- Inheritance from base class SolomonDataObject.
- Declaration of properties containing getters and setters that call the base class SolomonDataObject to get or set the data requested.
- Code attributes attached to each property describing to the Microsoft Dynamics SL kernel characteristics like offset, size, and type of the property. This information is used by the Microsoft Dynamics SL kernel to construct a memory layout compatibility with existing data binding.
- The PropertyIndex attribute must be sequential and starts at zero.
- Declaration of variables of the type ScreenMenu, and assigning a new instance of that type to the variable. Note the comment embedded by the VBTools Conversion Utility, which warns the developer not to use direct assignment of one variable to another. With .NET Framework, the assignment of one class variable to another changes the variable’s reference to be same as the other variable. In other words, both variables reference the same location in memory. This is important if the variable in question is passed to SetAddr. In this case, the Microsoft Dynamics SL kernel tracks the memory location of the variable specified in the SetAddr call. If the contents of the variable are accidentally changed by assigning the variable to another variable of the same type, the application will cease to function properly. In order to copy the contents of one SolomonDataObject variable to another, the method CopyClass should be used. For example, bScreenMenu.CopyClass(nScreenMenu) assigns the contents of nScreenMenu to bScreenMenu.

Example:

```vbnet
Module ScreenMenuDH

Public Class ScreenMenu
    Inherits SolomonDataObject
    <DataBinding(PropertyIndex:=0, StringSize:=10)> Public Property CategoryID() As String
        Get
            Return Me.GetPropertyValue("CategoryID")
        End Get

        Set(ByVal Value As String)
            Me.SetPropertyValue("CategoryID", Value)
        End Set
    End Property

    ...<DataBinding(PropertyIndex:=26, StringSize:=1)> Public Property UserType() As String
        Get
            Return Me.GetPropertyValue("UserType")
        End Get

        Set(ByVal Value As String)
```
Me.SetPropertyValue("UserType", Value)
End Set

End Property
End Class

'DO NOT REASSIGN THESE VARIABLES, i.e. bSomeType = nSomeType. Use API CopyClass(bSomeType,nSomeType)
Public bScreenMenu As ScreenMenu = New ScreenMenu,
nScreenMenu As ScreenMenu = New ScreenMenu

SolomonDataObject Members
SolomonDataObject is the class used to provide data binding between the application and the SQL
database. An application class inherits from SolomonDataObject, which adds the desired properties
corresponding to the SQL table, declares a variable of the class type, and sets the variable to a new
instance of the class type. The following tables list the methods exposed by the SolomonDataObject
type:

**AddressOf**

Public Function [AddressOf](ByVal propName As String) As System.IntPtr

**Summary**
AddressOf is used to obtain the memory address of a specific property named in propName.

**Parameters**
propName: The property name for which an address will be obtained.

**Return values**
IntPtr value representing a memory address.

**Remarks**
AddressOf is useful in order to pass the address of an individual property to the following API calls:
- Detailsetup
- Detailsetup8
- CurrencyField
- CurrencyInfo
- CurrencyInfo2
- DecimalPlaces

**Example**
MemHandle = DetailSetup(c2, Spread1, 
bGLTran.AddressOf("LineNbr"), 
bGLTran, PNULL, PNULL, PNULL) -

**Compare** (overloaded)

Public Function Compare(ByVal classInstance As Solomon.Kernel.SolomonDataObject) As Boolean

**Summary**
Compare will compare the current instance with the instance specified in parameter classInstance
to determine equality.

**Parameters**
classInstance: The class instance that will have its contents compared.

**Return values**
True if classInstance is equal to the current instance.

**Remarks**
Compare will examine the underlying memory belonging to the two class instances and compare the allocated memory buffers.

**Example**

```vbnet
If bAddress1.Compare(bAddress2) = True Then
    Console.Write( "The bAddress1 equals bAddress2")
End If
```

**Public Shared Function Compare** (ByVal classInstance1 As _
Solomon.Kernel.SolomonDataObject, ByVal classInstance2 As _
Solomon.Kernel.SolomonDataObject) As Boolean

**Summary**

Compare will compare two class instances for equality.

**Parameters**

classInstance1: First class instance to have its contents compared. classInstance2: Second class instance to have its contents compared.

**Return values**

True if classInstance1 is equal to classInstance2.

**Remarks**

Compare will examine the underlying memory belonging to the two class instances and compare the allocated memory buffers.

**Example**

```vbnet
If bAny.Compare(bAddress1, bAddress2) = True Then
    Console.Write("The bAddress1 equals bAddress2")
End If
```

**CopyClass**

**Public Sub** CopyClass (ByVal sourceClassInstance As Solomon.Kernel.SolomonDataObject)

**Summary**

CopyClass is used to copy the contents of one SolomonDataObject instance to another.

**Parameters**

sourceClassInstance: The class instance that will have its contents copied.

**Remarks**

This method **must** be used if the caller wants to set the contents of one SolomonDataObject to another. In Visual Basic 6.0, this was done by direct assignment of one structure to another. An important distinction to be aware of is that SolomonDataObject is a class, **not** a structure. If a class instance is used in a direct assignment statement in the current version of Visual Basic, it will change what the instance points to, not its contents. For SolomonDataObject-derived classes, direct assignment of one class instance to another will likely result in errors and cause the application to function incorrectly. CopyClass should be used when the caller wants to set the contents of one class to another.

**Example**

bAccount.CopyClass(nAccount)

**GetPropertyLength** (overloaded)

**Public Function** GetPropertyLength (ByVal propName As String) As Short

**Summary**

GetPropertyLength returns the length of the property. For arrays, the length of the entire array is returned. For individual array elements, the length of a single element is returned.

**Parameters**

propName: The name of the property being set.
Return values
Short integer length of property.

Example
Dim propertyLength As Short = _
bAccount.GetPropertyLength("AcctType")

Summary
GetPropertyLength returns the length of the property. For arrays, the length of the entire array is returned. For individual array elements, the length of a single element is returned.

Parameters
propName: The name of the property being set. index0: Index representing the element within the array.

Return values
Short integer length of property.

Example
Dim propertyLength As Short = _
btaxcalc.GetPropertyLength("CuryDetTaxAmt", 0)
Special Application Requirements

Passing Parameters Between Applications

The Microsoft Dynamics SL kernel provides the ability to pass parameters between Microsoft Dynamics SL screens and reports, customizations created with the Basic Script Language (BSL), and custom applications developed with the Microsoft Dynamics SL SDK.

Each Microsoft Dynamics SL or Microsoft Dynamics SL SDK application is a separate program file (EXE), enabling you to create a single solution comprised of two separate applications. For example, you can implement custom “Drill Down” or “Quick Print” functionality by calling the appropriate program file or the Report Option Interpreter (ROI) with the specified parameters.

The architecture to support this functionality is a “layered” design so that parameters passed using different methods remain isolated. This prevents parameters that are in use by two applications from being compromised if a BSL customization uses different parameters on the same applications. Transaction Import starts any target application screen by using command line parameters and therefore must keep its own parameters isolated, so that any parameters used by BSL or the underlying Microsoft Dynamics SL SDK application are also not compromised.

Using Parameter-Passing Functions

Microsoft Dynamics SL provides the following functions to pass parameters using BSL or the Microsoft Dynamics SL SDK:

<table>
<thead>
<tr>
<th>Function</th>
<th>Where Available</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplSetParmValue</td>
<td>Microsoft Dynamics SL SDK, BSL</td>
<td>Adds parameters to pass to an application.</td>
</tr>
<tr>
<td>CallApplic</td>
<td>Microsoft Dynamics SL SDK</td>
<td>Launches a second, modeless application.</td>
</tr>
<tr>
<td>CallApplicWait</td>
<td>Microsoft Dynamics SL SDK</td>
<td>Launches a second, modal application.</td>
</tr>
<tr>
<td>Launch</td>
<td>BSL</td>
<td>Launches a second application.</td>
</tr>
<tr>
<td>ApplGetParmValue</td>
<td>Microsoft Dynamics SL SDK, BSL</td>
<td>Retrieves parameters from the parameter file.</td>
</tr>
<tr>
<td>ApplGetParms</td>
<td>Microsoft Dynamics SL SDK, BSL</td>
<td>Retrieves parameters from the command line.</td>
</tr>
<tr>
<td>ApplGetReturnParms</td>
<td>Microsoft Dynamics SL SDK</td>
<td>Retrieves parameters from a terminated secondary Microsoft Dynamics SL SDK application.</td>
</tr>
<tr>
<td>ScreenExit</td>
<td>Microsoft Dynamics SL SDK</td>
<td>Sends parameters back to the calling application.</td>
</tr>
</tbody>
</table>

These functions allow parameters to be processed separately for Microsoft Dynamics SL SDK, BSL, and Transaction Import (TI) applications, and allow an application to access any parameter type from any other Microsoft Dynamics SL SDK, BSL, or TI application.

After an application adds parameters, a temporary, uniquely named parameter file (with a .PRM extension) is automatically created by the Microsoft Dynamics SL kernel. This file contains the parameter values that are to be passed to the receiving program.

The parameter file is similar in format to a Windows INI file and contains sections, entries, and values. It is created in the Microsoft Dynamics SL program directory or the directory specified by the TempDirectory entry in the [Miscellaneous] section of the Solomon.ini file. After the called application has completed, the Microsoft Dynamics SL kernel deletes the parameter file.

Note that an application can pass parameter values using a single command line string consisting of the name of the program file and the parameters to pass. However, this method is limited by the operating system maximum of 128 characters for a command line.
The **CallApplic**, **CallApplicWait**, and **Launch** functions all support the ability to pass parameters using the command line, and the **ApplGetParms** function can retrieve any parameter passed via the command line.

To pass parameter information using a string of any length, use the **ApplSetParmValue** and **ApplGetParmValue** functions with the temporary parameter file.

### Building and Passing Parameters

Use **ApplSetParmValue** to set the parameter values to be passed to another application. **ApplSetParmValue** allows you to specify the exact section, entry, and value in the parameter file.

The first run of this statement creates the temporary parameter file and sets the first parameters; subsequent runs of this statement within the same program add additional parameters to the file.

When the parameter file is complete, use **CallApplicWait**, **CallApplic**, or **Launch** to launch the other application. The file name of the parameter file is passed on the command line that calls the second application.

The following examples show how to add a new section, entry, and parameter value to a parameter file, and then call the second application:

**BSL**

```text
Call ApplSetParmValue("[MyScreen]", "CustomerId", "C299")
serr1 = Launch("0826000 ", True, True, 0)
```

**Microsoft Dynamics SL SDK**

```text
Call ApplSetParmValue("[MyScreen]", "CustomerId", "C299")
Call CallApplicWait(0826000)
```

Either call to **ApplSetParmValue** above creates a parameter file with the following contents:

```
[MyScreen]
CustomerId=C299
```

### Retrieving Parameters

Use **ApplGetParmValue** in the Load event of an application to retrieve values from the parameter file passed from another application. **ApplGetParmValue** retrieves the section, entry, and value that was placed in the parameter file.

The following examples show how to retrieve parameter values from a parameter file:

**BSL**

```vbnet
'Code for Form1_Display event of 08.260 Customer screen
Dim PassedParm As String
PassedParm = ApplGetParmValue("[MyScreen]", "CustomerId")
If Trim$(PassedParm) <> "" Then
  'Set field value only if parameter is not blank
  serr1 = SetObjectValue("ccustid", PassedParm)
End If
```

**Microsoft Dynamics SL SDK**

```vbnet
'Code for Form1_Load event of 08.260 Customer screen
Dim PassedParm As String
PassedParm = ApplGetParmValue("[MyScreen]", "CustomerId")
If Trim$(PassedParm) <> "" Then
  'Set field value only if parameter is not blank
  bCustomer.CustId = PassedParm
```
Call DispField(ccustid)
End If

Returning Parameters

Use Screenexit to pass values back to the calling application. In the calling application, use ApplGetReturnParms to retrieve values from a parameter file passed from another application.

The following examples show how to retrieve parameter values:

**BSL**

' Code in the Called App
ScreenExit(APPLICRETURNPARMS, Parms )

' Code in the Calling App
Dim PassedParm As String
PassedParm = ApplGetReturnParms()

If Trim$(PassedParm) <> "" Then
'Set field value only if parameter is not blank
End If

**Microsoft Dynamics SL SDK**

' Code in the Called App
ScreenExit(APPLICRETURNPARMS, Parms )

' Code in the Calling App
Dim PassedParm As String
PassedParm = ApplGetReturnParms()

If Trim$(PassedParm) <> "" Then
'Set field value only if parameter is not blank
    Call DispField(ccustid)
End If
Creating Applications Compatible with Customization Manager

Any application developed with Microsoft Dynamics SL SDK can be customized with Customization Manager. To support all features of Customization Manager, an application must conform to the following requirements.

Using Required Components

For an application to be customizable with Customization Manager, a Microsoft Dynamics SL SDK project must include the following components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form1</td>
<td>Form</td>
<td>Contains the DSLUpdate control that determines whether an application can be customized.</td>
</tr>
<tr>
<td>DSLUpdate</td>
<td>Control</td>
<td>On Form1, determines if the application can be customized; to allow customization, set the property Update.Customizable to True; otherwise, set to False.</td>
</tr>
</tbody>
</table>

All of the required components are contained in the Microsoft Dynamics SL SDK project template file, VBTProject.vbp.

Using Required Coding Practices

To ensure that an application is customizable with Customization Manager, use the following coding practices:

- Do not modify the settings created by the VB Code Inspector for the required components previously listed.
- Do not create code for the Click event for a TabControl.
- Ensure that control names are unique across the project (including subforms).
- Ensure that all controls within a grid frame are within the coordinate boundaries of the frame by following these property rules:
  - control.Top > 0
  - control.Left > 0
  - control.Top + control.Height < frame.Height
  - control.Left + control.Width < frame.Width
- Ensure that required property settings are not violated by evaluating conditions before using SetProps in code to modify property settings.
- Ensure that all controls within a grid frame have contiguous values for their Tabindex properties.

To make sure that an application is customizable with VBA, you must add a post build event to set the /NXCOMPAT flag. For more information about the NXCOMPAT flag, see “Applications Using Older ATL Components May Experience Conflicts With DEP” at the following location: [http://support.microsoft.com/kb/948468](http://support.microsoft.com/kb/948468)

Example: The following is a Visual Studio 2008 Post Build event on the Compile tab

```vbscript
call "$(DevEnvDir)\tools\vsvars32.bat"
editbin.exe /NXCOMPAT:NO "$(TargetPath)"
```

Post build may need a strong name called again if you need a signed build. The following code includes a strong name in the post build:

```vbscript
call "$(DevEnvDir)\tools\vsvars32.bat"
editbin.exe /NXCOMPAT:NO "$(TargetPath)"
SN -R "$(TargetPath)" <yourCodeSignKey>
```
Using Supported Controls

Customization Manager provides extended support for many controls, in the form of control-specific properties or functionality. These controls include:

**Microsoft Visual Basic Controls**
- System.Windows.Forms.Form
- System.Windows.Forms.Label
- System.Windows.Forms.GroupBox
- System.Windows.Forms.Panel
- System.Windows.Forms.TabControl (and TabPages through this control)

**Microsoft Dynamics SL SDK Controls**
- Microsoft.Dynamics.SL.Controls.DSLMaskText
- Microsoft.Dynamics.SL.Controls.DSLUpdate
- Microsoft.Dynamics.SL.Controls.DSLInteger
- Microsoft.Dynamics.SL.Controls.DSLDate
- Microsoft.Dynamics.SL.Controls.DSLFloat
- Microsoft.Dynamics.SL.Controls.DSLCombo
- Microsoft.Dynamics.SL.Controls.DSLCheck
- Microsoft.Dynamics.SL.Controls.DSOption
- Microsoft.Dynamics.SL.Controls.DSLGrid

In addition, Customization Manager provides customization support for some common properties associated with any control, including third-party controls. These properties include:
- Left
- Top
- Height
- Width
- Enabled
- Visible

Ensuring Customization Manager Usability

Use the following guidelines to promote a successful user experience for those using Customization Manager to customize your application.

- Controls bound to Table.Field entries from the database are unavailable for insertion by the Insert Object Wizard in Customization Manager. For example, if a table contains 10 fields, and controls on your application are bound to 8 of these fields, only the 2 unbound fields are available to the Insert Object Wizard. In the wizard, the user will see only tables that are bound programmatically through code with a SetAddr() statement.

- All controls that are secure at the application level cannot be unsecured through Customization Manager. Customization Manager can only be more restrictive, not less restrictive. Modifications to the properties of secured controls are allowed in Customize mode, but with the following restrictions:
  - A control whose Enabled property is set to False cannot be set to True.
A control whose **Visible** property is set to False cannot be set to True.

- The **BlankErr** property of a required control cannot be changed.
- Changes to the **Minimum** and **Maximum** values for a range can only narrow the range.
- List items cannot be added to combo box or text box controls if data already exists in the **List** or **PV** property for these controls.

- A hidden control that should not be customized by the user should be positioned to the far left of the form so it does not appear in Customize mode. For example, set the control’s **Left** property to a large negative number such as -2000. (You cannot use this method to position a control within a grid frame.)

- Avoid using labels with empty **Text** properties.

- Run VB Code Inspector on every build of your application and correct all errors found. Most of the errors that Code Inspector is designed to detect are those that cause subsequent errors when an end user attempts to customize the application. Keeping VB Code Inspector visible in the Visual Basic Add-ins menu ensures that you are automatically prompted to run Code Inspector when creating a program file.

**Testing Applications**

Test your applications for Customization Manager compatibility. Load each form in your application, and then select Customize mode from the Microsoft Dynamics SL application menu. Grab handles should appear around the form, and no error message should appear.

All third-party controls should be tested in Customize mode to determine if they can be customized without error. Select each control in Customize mode and press F4 to display the property window. The property window should display the limited property entries as described earlier (**Left**, **Top**, **Height**, **Width**, **Enabled**, and **Visible**).
Displaying Text in the Status Bar

The SetStatusBarText function is available for displaying text in the status bar of the application window. To facilitate language localization, avoid hardcoded string literals for text displayed in the status bar, and avoid hardcoding direct references to code points on code pages. Rather, use a message from Microsoft Dynamics SL’s message database or the text of a hidden label.

The maximum allowable length for a status message is 48 characters. If the string value to be displayed is greater than this length, it is automatically truncated before display, with no errors or warnings. Each new status message completely replaces the existing message displayed in the status bar.

Specifying an empty string for the tooltip displays the entire status bar text for the tooltip. Optionally, a replacement tooltip can be specified.

Here is a sample view of how the status bar would appear if the developer sets the text of the text pane of the status bar to “The big brown bear jumped over the little white hare”:

*Figure 1: Sample status bar*
Providing Database Configuration and Import Definition Files with Applications

A configuration file is used by Database Maintenance (98.290.000) to drive the creation and updating of databases. The configuration file (DBBUILD.INI) is a Windows .ini formatted file. The Windows and Visual Basic Profile APIs are used to retrieve data from the configuration file to provide the characteristics of the databases to be created.

Overview of the Configuration File

The configuration file contains a [Scenarios] section for describing the available database scenarios and individual [Scenario#] sections for specifying information about the database for each scenario.

The configuration file contains two additional sections that create the database schema for the system database and application database(s): [MSSQL System Scripts] and [MSSQL Applications Scripts]. These sections contain keys that indicate file names of scripts that create the database schema. Every database created from the scenarios in the [Scenarios] section uses the SQL statements and the data contained in the files.

The [Update Scenarios] and the [Update ScenarioX] sections enable the developer to perform schema updates and imports of data.

[Scenarios] Section

The [Scenarios] section lists all possible scenarios that can be selected from Database Maintenance (98.290.000). Several scenarios may exist in this section to describe the available database scenarios. Each item must be of the format

Scenario#="Descriptive Text"

where # is a number and “Descriptive Text” is a short description of what the scenario is. The maximum length of the “Descriptive Text” is 4,096 characters.

The maximum number of scenarios that can be listed within this section is limited only by the 64K data limit of the Windows ListBox control, which gets populated with items from this [Scenarios] section.

The Scenario# items must start with Scenario1 and increment by 1 for each scenario. For example, the following is a valid [Scenarios] section:

[Scenarios]
Scenario1="Empty Databases"
Scenario2="Demo - Business Ready Editions"
Scenario3="Demo - Professional Edition"
Scenario4="Demo - Standard Edition"
Scenario5="Additional Empty Application Database"

The example below is not valid:

[Scenarios]
Scenario3="Empty Databases"
Scenario4="Demo - Professional Edition"
Scenario6="Demo - Standard Edition"

The second example is not valid for two reasons. First, the scenario items must start with Scenario1; therefore, Scenario3 is not valid. Second, the scenario items must have numeric values at the end of the key name that increment by 1 for each scenario, thus having Scenario6 after Scenario4 is not valid. Scenario5 is the only valid key name for the next scenario after Scenario4.
[Scenario#] Sections

Each scenario listed in the [Scenarios] section must have a corresponding section within the same configuration file. Each individual [Scenario#] section contains data specific to each system and application database to be created. See the following example when reading the explanation of how this section works:

```
[Scenarios]
Scenario1 = "Empty Databases"
Scenario2 = "Demo – Professional Edition"

[Scenario1]
SysName = SLSystem
SysSize = 10
SysImport = empty \system\EmptySysImport.lst
AppNames = SLApplication
AppSizes=40
AppImports=empty\app\EmptyAppImport.lst

[Scenario2]
SysName = SLDemoSystem
SysSize = 10
SysImport = demo\professional\Sys\import.lst
AppNames = SLDemoApp10;SLDemoApp60
AppSizes=40;50
AppImports=demo\professional\app10\import.lst;demo\professional\app60\import .lst
```

Six key names that may be specified within each Scenario# section. Three of the keys contain information about the system database, and three of them contain information about the application database. These keys specify default values and may be overridden by using Advanced on the Create Databases tab of Database Maintenance (98.290.00).

SysName

The SysName key indicates the default database name of the system database. This name must conform to Microsoft SQL Server database naming conventions.

If a database with this name already exists on the destination server, and no application databases are specified for this scenario when the Create button is clicked, an error message appears, indicating that the database exists. If a database with this name already exists on the destination server, and at least one application database is specified to be created for this scenario, then only the application databases will be created, and they will be associated with the system database specified here. If a database this same name does not exist on the destination server, then it will be created.

SysSize

The SysSize key indicates the initial size in MB of the system database when it is created.

SysImport

The SysImport key specifies a relative path and filename of an import file to use when importing data into the system database created with this scenario. The import file specifies the name of the table to import data into, the name of the file that contains the data, and the format of the data. The path specified is relative to the path of the configuration file in which this setting is specified. The import file must exist in the same directory as the import.lst file specified in this key.
For example, assume a configuration file is located in C:\Program Files\Microsoft Dynamics SL\DB.
Then the full path and file name of the SysImport key value under Scenario2 in the example would be
C:\Program Files\Microsoft Dynamics SL\DB\demo\professional\Sys\import.lst.

Below is an example of an import file that could be used here:

```
Pvrec, pvrec.bcp, /n
Acctxref, AcctXref.bcp, /nRegistitem, registit.csv
SubXref, SubXref.bcp, /n
```
For more information on the import file format and BCP options, see “Import File Format.”

**AppNames**

The **AppNames** key indicates the default database names of the application databases to be created.
These names must conform to Microsoft SQL Server database naming conventions and must be
separated by semicolons if more than one application database name is specified. There must be a
corresponding size specified in the **AppSizes** key for each database name specified.

If a database with this name already exists on the destination server when the **Create** button is
clicked, an error message will be written to the dbbuild.log file that indicates the database exists.

**AppSizes**

The **AppSizes** key indicates the initial size in MB of the application databases when they are created.
Each size corresponds to the database names specified in the **AppNames** key, and the sizes must be
separated by a semicolon.

**AppImports**

The **AppImports** key specifies a relative path and filename of an import file to use when importing data
into the application databases created with this scenario. The path specified is relative to the path of
the configuration file in which this setting is specified. If multiple import files are specified, they must
be separated by a semicolon and they correspond to the database names specified in the **AppNames**
key. The entire length of values specified in the **AppImports** key is limited by 4,096 characters.

**[MSSQL System Scripts] and [MSSQL Applications Scripts]**

The [MSSQL System Scripts] and [MSSQL Applications Scripts] sections contain keys that indicate file
names of scripts that create the database schema. Every database created from the scenarios in the
[Scenarios] section uses the SQL statements and the data contained in the files specified in the
configuration file.

There are six key names that may or may not be specified in these sections to specify the files to be
used. The primary reason for six different key names is so **Database Maintenance** (98.290.00) follows
a consistent order when creating schema objects. The system database is created first, followed by
the application databases. These databases are created with the compatibility level set to 90 by
default so that statements that use deprecated SQL 2005 syntax will still work. Each database is
created in the following order:

1. Run Create Table statements (tables key)
2. Importing of data (import key)
3. Run Create Index statements (indexes key)
4. Run Create Procedure statements (procedures key).
5. Run Create View statements (views key).
6. Association of the application database to the system database through creation of a domain
   record in the system database for the application database being created.

See the following example in reading the explanation of each key:

[MSSQL System Scripts]

```
tables=common\system\0MSCRT.SQL; common\system\APPCRT.SQL
```
Special Application Requirements

indexes= common\system\1MSNDX.SQL; common\system\APPMNSNDX.SQL
procedures= common\system\SYPROCS.SQL;common\system\APPPRCS.SQL
views=common\system\SYVIEWS.SQL; common\system\APPVIEWS.SQL
triggers=common\system\DSD_Triggers.sql
defaults=common\system\DSD_Defaults.sql
constraints=common\system\DSD_Constraints.sql
rules=common\system\DSD_Rules.sql
import=common\system\Import.lst
scriptpath=c:\scripts\mssql\sysdb

[MSSQL Application Scripts]
tables= common\application\MSCRT300.SQL; common\application\DSDCRT300.SQL
indexes= common\application\MSNDX300.SQL; common\application\DSDNDX300.SQL
procedures= common\application\MSPRC300.SQL;
common\application\DSDPRC300.SQL
views= common\application\MSVIEWS300.SQL; common\application\DSDVIEWS300.SQL
triggers=common\application\FMG_Triggers.sql
defaults=common\application\FMG_Defaults.sql
constraints=common\application\FMG_Constraints.sql
rules=common\application\FMG_Rules.sql
import= common\application\Import.lst
scriptpath=c:\scripts\mssql\appdb

Tables
The **tables** key specifies the location and name of files that contain “Create Table ...” statements for the databases. Multiple files can be specified and must be separated by a semicolon. The length of the key value cannot exceed 4,096 characters, including the semicolons.

A relative path may be specified for each file listed. The path will be relative to the location of the configuration file or the path specified in the **scriptpath** section if it exists. Absolute pathnames may not be specified.

Indexes
The **indexes** key specifies the location and name of files that contain “Create Index ...” statements for the databases. Multiple files can be specified and must be separated by semicolons. The length of the key value cannot exceed 4,096 characters, including the semicolons.

A relative path may be specified for each file listed. The path will be relative to the location of the configuration file or the path specified in the **scriptpath** setting if it exists. Absolute pathnames may not be specified.

Procedures
The **procedures** key specifies the location and name of files that contain “Create Procedure...” statements for the application databases. Multiple files can be specified and must be separated by semicolons. The length of the key value cannot exceed 4,096 characters, including the semicolons.

A relative path may be specified for each file listed. The path will be relative to the location of the configuration file or the path specified in the **scriptpath** setting if it exists. Absolute pathnames may not be specified.

Views
The **views** key specifies the location and name of files that contain “Create View ...” statements for the application databases. Multiple files can be specified and must be separated by a semicolon. The length of the key value cannot exceed 4,096 characters, including the semicolons.
A relative path may be specified for each file listed. The path will be relative to the location of the configuration file or the path specified in the scriptpath setting if it exists. Absolute pathnames may not be specified.

Views of every system database table will be created in an application database automatically.

**Triggers**

The triggers key specifies the location and name of files that contain Create Trigger statements for the databases. Multiple files can be specified and must be separated with a semicolon. The length of the key value cannot exceed 4,096 characters, including the semicolons. A relative path may be specified for each file listed. The path will be relative to the location of the configuration file or the path specified in the scriptpath section if it exists. Absolute pathnames do not work; paths must be relative.

**Constraints**

The constraints key specifies the location and name of files that contain Alter Table...Add Constraint statements for the databases. Multiple files can be specified and must be separated with a semicolon. The length of the key value cannot exceed 4,096 characters, including the semicolons. A relative path may be specified for each file listed. The path will be relative to the location of the configuration file or the path specified in the scriptpath section if it exists. Absolute pathnames did not work. paths must be relative.

**Defaults**

The defaults key specifies the location and name of files that contain Create Default statements for the databases. Multiple files can be specified and must be separated with a semicolon. The length of the key value cannot exceed 4,096 characters, including the semicolons. A relative path may be specified for each file listed. The path will be relative to the location of the configuration file or the path specified in the scriptpath section if it exists. Absolute pathnames do not work; paths must be relative.

**Rules**

The rules key specifies the location and name of files that contain Create Rule statements for the databases. Multiple files can be specified and must be separated with a semicolon. The length of the key value cannot exceed 4,096 characters, including the semicolons. A relative path may be specified for each file listed. The path will be relative to the location of the configuration file or the path specified in the scriptpath section if it exists. Absolute pathnames do not work; paths must be relative.

**Import**

The import key specifies the location and name of the import file or files that hold names of tables and corresponding data files. The import files need to be located in the directory specified in the scriptpath setting in the configuration file or, if scriptpath is not specified, in the same directory as the file specified in the import key.

The .csv file specified needs to specify data for ALL fields in the table.

In the example under “[MSSQL System Scripts] and [MSSQL Applications Scripts]” on page 24, import.lst is the file that contains the import information. Here is an example of import.lst:

```
UserRec, userrec.csv
UserGrp, usergrp.csv, /n
RegistItem, registit.csv, /c /t# /r
APDoc, APDoc.csv, /c /t; /r	 /E
```

The first line indicates that userrec.csv is a comma-delimited ASCII file with carriage return/linefeed characters as the row delimiter and a comma as the column delimiter. This file contains the data that will be imported into the UserRec table.

The second line indicates that the usergrp.csv file is in Microsoft SQL Server’s native BCP format and contains the data to import into the UserGrp Table.
The third line indicates that the registit.csv file is in ASCII format, and uses the # character as the column delimiter and carriage return/linefeed characters as the row delimiter. This file contains the data to import into the RegistItem table.

The fourth line indicates that the apdoc.csv is in ASCII format, the columns are delimited by a semi colon and the rows are delimited by a tab character, and we want to explicitly set any identity fields to a specific value. This file contains the data to import into the APDoc table.

For more information on the import file format and BCP options, see “Import File Format” on page 30.

Scriptpath
The scriptpath key indicates an absolute path location to the directory that holds all the script files. The path specified here will also become the location to which all paths specified in the tables, indexes, views, and procedures keys are relative.

The script files specified in the configuration file must be Microsoft SQL Server Transact-SQL Create statements. The script file specified for the tables key should contain T-SQL formatted “Create Table” statements. The script file specified for the indexes key should contain T-SQL formatted “Create Index” statements. The script file specified for the procedures key should contain T-SQL formatted “Create Procedure” statements. The script file specified for the views key should contain T-SQL formatted “Create View” statements.

[Update Scenarios] Section
The [Update Scenarios] section of DBBUILD.ini lists all possible update scenarios that can be selected on the Update tab in Database Maintenance (98.290.000). Several update scenarios may exist in this section to describe the action to take when it is selected.

Each item must be of the format:

    Update ScenarioX="Descriptive Update Text".

X is the scenario number and “Descriptive Update Text” is a short description of what the update scenario does.

The maximum length of the “Descriptive Update Text” is 4,096 characters. The maximum number of scenarios that can be listed within this section is limited only by the 64 KB data limit of the Windows ListBox control that is populated with items from this [Update Scenarios] section.

The Update ScenarioX items must start with “Update Scenario1” and increment by one for each update scenario. The last numeric digits of the keyname for the item must be sequential; that is, do not skip numbers.

For the update scenario to be recognized, there must be a space between “Update” and “ScenarioX.”

For example the following is a valid [Update Scenarios] section:

[Update Scenarios]
Update Scenario1="6.5x, 7.x, 2011x, to 2015"
Update Scenario2="Execute Master Indexes, Views and Stored Procedures"
Update Scenario3="Execute Master Views and Stored Procedures"
Update Scenario4="Field and Record Maintenance Update"
Update Scenario5="Synchronize All Database Ownership & Security"
Update Scenario6="Synchronize Selected Application Database Ownership & Security"
[Update Scenario1]
AppTables=Scripts\App\appTableUpdate.sql;Scripts\App\pjtext.sql;Scripts\App\appTableDataUpdate.sql
AppIndexes=Scripts\App\appIndexes.sql
AppProcedures=Scripts\App\appProcs.sql
AppViews=Scripts\App\appViews.sql;Scripts\App\quickqueryViews.sql
AppTriggers=Scripts\App\appTriggers.sql
AppDefaults=
AppConstraints=
AppRules=
AppImport=Update\App\AppUpdateImport.lst
SysTables=Scripts\Sys\SystemTableUpdates.sql;Scripts\Sys\SystemTableDataUpdates.sql
SysIndexes=Scripts\Sys\SystemIndexes.sql
SysProcedures=Scripts\Sys\SystemProcUpdates.sql
SysViews=Scripts\Sys\SystemViewUpdates.sql
SysTriggers=Scripts\Sys\SystemTriggers.sql
SysDefaults=
SysConstraints=
SysRules=
SysImport=Common\Sys\UpdateImport.lst

[Update ScenarioX] Section

Each key name listed in the [Update Scenarios] section of DBBUILD.ini must have a corresponding section within the same configuration file. Each individual Scenario section contains data specific to each System and Application database to be created.

There are 18 key names that may be specified within each [Update ScenarioX] section:

- Nine of the keys contain information about the system database.
- Nine of them contain information about the application database.

These keys specify the location and name of files that contain SQL statements used to perform schema updates and imports of data. The length of the key value for each of these keys cannot exceed 4,096 characters, including the semicolons.

When a specific update scenario is selected on the Update tab of Database Maintenance (98.290.000), all the scripts specified in the following sections are run for that specific update scenario. These scripts are applied to the system database and all selected application databases. Multiple application databases may be selected from the Update tab.

**Apptables**

The **apptables** key specifies the location and name of files that contain Create Table statements for the application databases. See the description of the Tables key in the [MSSQL Applications Scripts] and [MSSQL System Scripts] section for a description of this key. The same rules and limits apply here.

**Appindexes**

The **appindexes** key specifies the location and name of files that contain Create Index statements for the application databases. See the description of the Indexes key in the [MSSQL Applications Scripts] and [MSSQL System Scripts] section for a description of this key. The same rules and limits apply here.

**Appprocedures**

The **appprocedures** key specifies the location and name of files that contain Create Procedure statements for the application databases. See the description of the Procedures key in the [MSSQL Applications Scripts] and [MSSQL System Scripts] section for a description of this key. The same rules and limits apply here.
Appviews
The **appviews** key specifies the location and name of files that contain Create View statements for the application databases. See the description of the Views key in the [MSSQL Applications Scripts] and [MSSQL System Scripts] section for a description of this key. The same rules and limits apply here.

Appimport
The **appimport** key specifies the location and name of the file that holds names of tables and data files to import into. See the description of the Import key in the [MSSQL Applications Scripts] and [MSSQL System Scripts] section for a description of this key. The same rules and limits apply here.

Apptriggers
The **apptriggers** key specifies the location and name of files that contain Create Trigger statements for the application databases. See the description of the Triggers key in the [MSSQL Applications Scripts] and [MSSQL System Scripts] section for a description of this key. The same rules and limits apply here.

Appconstraints
The **appconstraints** key specifies the location and name of files that contain Alter Table...Add Constraint statements for the application databases. See the description of the Constraints key in the [MSSQL Applications Scripts] and [MSSQL System Scripts] section for a description of this key. The same rules and limits apply here.

Appdefaults
The **appdefaults** key specifies the location and name of files that contain Create Default statements for the application databases. See the description of the Defaults key in the [MSSQL Applications Scripts] and [MSSQL System Scripts] section for a description of this key. The same rules and limits apply here.

Apprules
The **apprules** key specifies the location and name of files that contain Create Rule statements for the application databases. See the description of the Rules key in the [MSSQL Applications Scripts] and [MSSQL System Scripts] section for a description of this key. The same rules and limits apply here.

Systables
The **systables** key specifies the location and name of files that contain Create Table statements for the system databases. See the description of the Tables key in the [MSSQL Applications Scripts] and [MSSQL System Scripts] section for a description of this key. The same rules and limits apply here.

Sysindexes
The **sysindexes** key specifies the location and name of files that contain Create Index statements for the system databases. See the description of the Indexes key in the [MSSQL Applications Scripts] and [MSSQL System Scripts] section for a description of this key. The same rules and limits apply here.

Sysprocedures
The **sysprocedures** key specifies the location and name of files that contain Create Procedure statements for the system databases. See the description of the Procedures key in the [MSSQL Applications Scripts] and [MSSQL System Scripts] section for a description of this key. The same rules and limits apply here.

Sysviews
The **sysviews** key specifies the location and name of files that contain Create View statements for the system databases. See the description of the Views key in the [MSSQL Applications Scripts] and [MSSQL System Scripts] section for a description of this key. The same rules and limits apply here.
Sysimport
The sysimport key specifies the location and name of the file that holds names of the tables and data files into which to import. Please see the description of the Import key in the [MSSQL Applications Scripts] and [MSSQL System Scripts] section for a description of this key. The same rules and limits apply here.

Systriggers
The systriggers key specifies the location and name of files that contain Create Trigger statements for the system databases. See the description of the Triggers key in the [MSSQL Applications Scripts] and [MSSQL System Scripts] section for a description of this key. The same rules and limits apply here.

Sysconstraints
The sysconstraints key specifies the location and name of files that contain Alter Table...Add Constraint statements for the system databases. See the description of the Constraints key in the [MSSQL Applications Scripts] and [MSSQL System Scripts] section for a description of this key. The same rules and limits apply here.

Sysdefaults
The sysdefaults key specifies the location and name of files that contain Create Default statements for the system databases. See the description of the Defaults key in the [MSSQL Applications Scripts] and [MSSQL System Scripts] section for a description of this key. The same rules and limits apply here.

Sysrules
The sysrules key is used to specify the location and name of files that contain Create Rule statements for the system databases. Please see the description of the Rules key in the [MSSQL Applications Scripts] and [MSSQL System Scripts] section for a description of this key. The same rules and limits apply here.

Import File Format
The import file holds names of tables and corresponding data files. Each line of the import file consists of two required fields and one optional field, all separated by commas. The first field specifies the name of the table into which to import the data. The second field specifies the name of the file that contains the data. The third field specifies the Microsoft SQL Server utility Bulk Copy Program (BCP) options to use when importing the data. The BCP options are:

- `/f <format file>` — Specifies a file that indicates to SQL Server the format of the data file you are using. This file must reside in the directory specified in the scriptpath setting in the configuration file or, if scriptpath is not specified, in the same directory as the configuration file.
- `/n` — Tells SQL Server to use the native BCP format.
- `/c` — Tells SQL Server to use ASCII format. If /c is used, you must also specify the following flags:
  - `/t` — Indicates the column delimiter characters to use
  - `/r` — Indicates the row delimiter characters to use
  If neither of these options is specified, the default column delimiter is the comma and the default row delimiter is the carriage return/linefeed.
- `/E` — Indicates to SQL Server that you want to set identity fields. If you do not specify this option and your table has an identity field, the value of the field will be incremented from the last identity value used in that table.

**Note:** The E must be an uppercase E.
Adding Custom HTML Help

Use the guidelines in this section if you want to add your own help files into the Microsoft Dynamics SL help system.

Setting Up Help

For proper operation, all help books should be installed in the Help folder under the Microsoft Dynamics SL root directory. Help books are stored in files with a .chm file extension (for example, the General Ledger help file is named SL_GL.chm).

The SL_TOC.chm is the primary Microsoft Dynamics SL help file. This file contains references to all installed help system files. In addition, it allows cross-searching and indexing of all Microsoft Dynamics SL help files that are installed.

Understanding Help Requirements

Use the following guidelines for implementing online help for a Microsoft Dynamics SL SDK application:

- “What’s This” help is currently not supported.
- Each independent software vendor (ISV) help system can have one or more help files.
- You cannot add books or topics to any book provided by Microsoft.
- Microsoft does not support the modification of a book name without changing the name of its corresponding help file.
- The compiled HTML help that is standard with Microsoft Dynamics SL does not support adding books to the help table of contents. However, a book can be added to a user’s menu or to favorites that appear in their home pane on the Microsoft Dynamics SL window. For more information, see “Home Pane” or “Using Menu Maintenance” in the System Manager online help or user guide.

Help File Design Logic

The screen-level help is based on module and screen names. The help file name is derived from the module associated with the screen, and the help topic is derived from the screen title. The details behind the screen-level help design logic are as follows:

- Each form has a Visual Basic help provider component named SAFHelpProvider.
- If the SAFHelpProvider.HelpNamespace property is empty, the help file is located in C:\Program Files\Microsoft Dynamics SL\Help. The help file name contains the two-letter acronym associated with the screen program name.
- If the Forms HelpNavigator property is set to AssociatIndex and the Forms HelpKeyword is empty, the help topic for the entire form is based on the form title. For example, the help topic for the Registration screen takes the title “Registration (95.250.00)” and converts it to match the help generation tool topic “SL_SM_rtf/Registration_95_250_00.htm”.

You can customize screen-level help by adding your own HelpProvider component that has a name other than SAFHelpProvider to each form. See the Visual Studio documentation for information about using the HelpProvider component.

Screen-level help is invoked by pressing the F1 key or the Help button on the application toolbar. Access to custom help can be designed to invoke when the user presses F1, when they select help on their menu, or when they select a help favorite on their home pane. For more information, see “Home Pane” or “Using Menu Maintenance” in the System Manager online help or user guide.

Building Help Files

You can use industry-standard help authoring tools to create online help files that are compatible with the Windows rich client Help system and with the Microsoft Dynamics SL Help system.
These tools create input to an HTML help compiler. Usually, the help tool also produces “map files” containing context ID numbers for each topic page in the resulting help file. The output of the HTML help compiler is a compiled help file. Each of these context IDs is specified in your Microsoft Dynamics SL SDK project for the HelpContextID property.

Use a help authoring tool to create topic help as well as field-sensitive help.

**Enabling Help**

To place your help information into the Microsoft Dynamics SL Help system, perform the following steps:

1. In the Project Properties dialog for your application, add the Help File Name of the help file on the General tab. Do not include path information with the name.
2. ApplInit() now sets the Visual Basic object property App.HelpFile by looking up the help file path and appending the help file name you entered on the Project Properties screen. ApplInit() determines this path by appending \Help to the Parent Directory value stored in the system registry at HKEY_LOCAL_MACHINE\SOFTWARE\Solomon\Solomon IV Tools For Visual Basic.
3. Select a control for which you want to provide field-sensitive help.
4. In the Properties window, select HelpContextID.
5. Enter a context ID number for the help topic associated with the control.
6. Repeat steps 3 through 5 for any additional controls.
Utilities and Add-Ins

Adding Custom Controls
You can add custom controls to Microsoft Dynamics SL Visual Basic projects. A full set of custom controls are defined within the Microsoft Dynamics SL kernel. These controls appear in the Toolbox Window of Visual Basic when the kernel program is referenced in a project. The following table shows what these controls look like in the Toolbox and a brief description of each control:

<table>
<thead>
<tr>
<th>Icon</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>![icon]</td>
<td>DSLUpdate</td>
<td>Required for all Microsoft Dynamics SL SDK projects. Communicates information about the program to the Microsoft Dynamics SL kernel, and provides events associated with database updates and deletes.</td>
</tr>
<tr>
<td>![icon]</td>
<td>DSLMaskedText</td>
<td>Provides an area to display and input text data.</td>
</tr>
<tr>
<td>![icon]</td>
<td>DSLInteger</td>
<td>Provides an area to display and input whole number values.</td>
</tr>
<tr>
<td>![icon]</td>
<td>DSDate</td>
<td>Provides an area to display and input calendar date data.</td>
</tr>
<tr>
<td>![icon]</td>
<td>DSLFloat</td>
<td>Provides an area to display and input non-whole numbers. Used for all monetary fields in Microsoft Dynamics SL.</td>
</tr>
<tr>
<td>![icon]</td>
<td>DSLCombo</td>
<td>Allows the user to select from a fixed list of possible values. (Behaves like a drop-down list box.)</td>
</tr>
<tr>
<td>![icon]</td>
<td>DSLCheck</td>
<td>Provides an area to display and input true or false information.</td>
</tr>
<tr>
<td>![icon]</td>
<td>DSOption</td>
<td>Allows the user to select from a fixed list of possible values.</td>
</tr>
<tr>
<td>![icon]</td>
<td>DSLGrid</td>
<td>Allows for the display and input of multiple records of information.</td>
</tr>
</tbody>
</table>

The same field from the database will frequently appear in several different programs throughout the Microsoft Dynamics SL product. This means that the same control will be needed, and many of the properties for those controls should be set exactly the same way in each program where the same database field is being accessed.

Detail Levels
Detail levels in a Microsoft Dynamics SL SDK program automatically support two different views of the data. The spreadsheet view is commonly referred to as “grid view.” If the user double left clicks on the grid, or presses the F4 key, the kernel hides the spreadsheet control so the data controls underneath it can be seen. This view is called “form view” and allows the user to see all the fields for a single record in the grid at one time.

The kernel uses the controls in the container object to determine what columns should be included in the spreadsheet (grid view), how the columns should be formatted, and what buffer variables the columns should be bound to.

Because of the “form view” functionality, it is important to place the data controls for the detail level in an esthetically pleasing way.

The level property of controls attached to a NOLEVEL buffer is often a point of confusion for developers. However, if you consider the purpose of the level property of a control, versus the level
“property” (what level the buffer is associated with in the SetAddr call) of a buffer variable, there is not any mystery.

- Level Property of a buffer variable — Tells the Microsoft Dynamics SL kernel that it should use that buffer variable to store records retrieved from the primary table for the level.

- Level Property of a control:
  - If the kernel refreshes the data associated with a level, then redisplay the information in all controls with the same level.
  - If a CHK event fires on a control, then set a “dirty flag” for the level associated with the control to indicate that the data in that level has been updated.

**Pasting a Control onto a Form from the Visual Basic Toolbox**

If you are adding a field, manually add the control from the Visual Basic toolbox. For a list of the Microsoft Dynamics SL SDK controls that appear in your Visual Basic toolbox, see “Adding Custom Controls” on page 33.

Microsoft Dynamics SL text controls are designed to display strings, integers, dates, or floating point values. When you manually paste a text control, it is important to match the data type, buffer variable, and text control.

**To paste a control onto a form from the Visual Basic Toolbox:**

1. In a Visual Basic project, choose **View | Project Explorer**.
2. In the *Project Explorer* window *Forms* folder, double left-click on a Form object to display its design window.
3. From the Visual Basic toolbar choose **View | Toolbox**.
4. In the *Toolbox* window, double-click on the control you want to add.
5. After the control appears on the Form design view window, reposition and resize the control as needed.
Defining New Tables

When Microsoft Dynamics SL developers add new tables to a database, they follow the naming convention of not starting table names with an “X”. Therefore, if third-party developers always create their table names starting with an “X”, they will know for sure that future updates of the Microsoft Dynamics SL product will not cause a naming conflict. (Be Aware That there is no formal convention to prevent multiple third-party developers from creating tables with the same names.)

**Note:** Although the “X” is the only explicitly stated naming convention for tables, there are some implicit restrictions. These restrictions result from the fact that Microsoft Dynamics SL uses the name assigned to tables as the names for variables in Microsoft Dynamics SL SDK programs. This means that any naming restrictions that apply to Visual Basic variables must also be followed for Microsoft Dynamics SL tables. Some of these restrictions are as follows:

- Table names should not contain spaces.
- Table names should not begin with numbers.
- Other than underscores, no special characters should be used in a table name.
- Table names should not end in a number (because of the kernel’s implementation of its SetAddr function).
Defining New Fields

After a new table is defined, the fields for that table must be defined. As with the table names, when you install the Microsoft Dynamics SL SDK, the definitions for most of the existing Microsoft Dynamics SL fields are installed.

Also like table names, field names are used within Microsoft Dynamics SL SDK programs as variable names. Therefore, exactly like table names, field names must follow standard variable naming restrictions.

**Note:** Although the standard fields can be accessed in this program, the physical structure of the standard Microsoft Dynamics SL tables (that is, field data types, field sizes, number and order of fields, and so on) should never be altered! For a Microsoft Dynamics SL SDK program to access a record from the database, a variable that has the identical structure as the record being selected must be defined and created at the time that the program is written. This is true for all the standard Microsoft Dynamics SL programs in addition to any programs written by third parties. Because Microsoft Dynamics SL usually retrieves records by performing “SELECT * FROM Table”, any changes to the structure of a table would cause records from that table to no longer correctly fit in the variables that were created to hold those records.
Creating Tables

The steps in creating a table are as follows:

1. Defining the Table Name and Fields.
2. Generating an SQL CREATE TABLE Script File and DH File.
3. Creating the Table in the Database.
4. Creating a Unique Index on the New Table.
5. Creating and Running a Stored Procedure.

Standard Microsoft Dynamics SL file name extensions are all recognized by a Microsoft Dynamics SL utility program called Database Update which can read these files and update the current database by using the information found in them. See “Microsoft Dynamics SL Standard File Name Extensions” for more information.

The table schema in each Microsoft Dynamics SL Application database that is attached to a System database should be consistent with other Application databases that are attached to the same System database. Therefore, you have to create custom tables, views, or stored procedures in all Application databases that are attached to the same System database.

Microsoft Dynamics SL Standard File Name Extensions

Microsoft Dynamics SL has standard file name extensions for many of the files that are used during development:

- **CRT** – This extension is used for SQL script files that contain CREATE TABLE statements.
- **CRX** – This extension is used for SQL script files that contain CREATE INDEX statements.
- **CRP** – This extension is used for SQL script files that contain CREATE PROCEDURE statements.
- **CRU** – This extension is used for SQL script files that can contain any kind of SQL statements. This includes CREATE TABLE, INDEX, and PROCEDURE statements.
- **CSV** – This extension is used either for “Possible Value” window declarations or for comma-delimited ASCII data files.
- **IMP** – This extension is used for SQL script files that contain INSERT INTO statements that can be used to import the data in .csv files.
- **DH** – This extension is used for Visual Basic module files that contain user-defined type definition and global variable declarations for the buffers that hold records from the database.
- **GLB** – This extension is used for Visual Basic module files that contain global variable declarations for things such as cursor and memory array handles.

The .crt, .crx, .crp, .cru, .csv, and .imp file name extensions are all recognized by a Microsoft Dynamics SL utility program called Database Update that can read these files and update the current database by using the information found in them.

Defining the Table Name and Fields

Use the procedures in “Defining New Tables” on page 35 and “Defining New Fields” on page 36 to define the table name and fields.
Generate an SQL CREATE TABLE Script, SDO, and DH File

Create the following files:

- A file that has the extension of .crt, that contains the SQL CREATE TABLE statement required to create your new table in a database.
- A file that has the extension of .dh, that contains a Visual Basic UDT declaration and global declarations for variables of that type.
- A file that has the extension of .dh.vb, that contains a Visual Basic class declaration representing a SolomonDataObject (SDO), global declarations for variables of this type, and a subroutine that contains a setaddr() call and a sql cursorex() call, for more information about the SolomonDataObject, see Solomon Data Object Files on page 10.

Make sure that the CRT file that is used to create the table in the database, the .dh file, and .dh.vb file have the identical structure as the table.

To add the DH File or DH.VB file to the project:
1. Within the Visual Basic project, select Project | Add Module.
2. Select the Existing tab.
3. Add the file.

Creating the Table in the Database

Use Database Maintenance (98.290.00) to create the new table in the database. Create a new update scenario in the dbbuild.ini or create your own configuration file following the instructions in “[Update Scenarios] Section” on page 27. Here is an example of a separate configuration file that is named NewTableA.ini that creates a table named TableA:

```
[Update Scenarios]
Update Scenario1="Create TableA"
[Update Scenario1]
AppTables=TableA.crt
AppIndexes=TableAIndex.sql
```

Creating a Unique Index on the New Table

Any table in a well designed relational database, must have at least one unique index. In other words, there must be some field, or group of fields, for which two records in a table cannot have the same values.

To create the new or customized metadata, follow these steps:

1. Create the new indexes or change existing indexes, or both, by using SQL in the appropriate Microsoft Dynamics SL application database or the Microsoft Dynamics SL system database.
2. In SQL Server Management Studio, run the new ut_build_index_metadata stored procedure one time for each table that corresponds to an index in step 1, to add the new index metadata to the index metadata tables (SLIndex and SLIndexCol). Replace MyCustomTable with the appropriate table name:
   ```
   Exec ut_build_index_metadata 'MyCustomTable'
   ```
3. Export the contents of the SLIndex table and the SLIndexCol table into a .csv file. Use the following commands to generate the .csv files, substituting values for <yourServer> and <yourDB>:
   ```
   bcp SLIndex out SLIndex.csv -S<yourServer> -d<yourDB> -T -c -t,
   bcp SLIndexCol out SLIndexCol.csv -S<yourServer> -d<yourDB> -T -c -t,
   ```

   **Note:** The comma (,) at the end of each statement is required.
4. Copy the new .csv files to the database build folders, replacing the SLIndex.csv file and the SLIndexCol.csv file that already exist in that location. Copy the files to the following locations:
   • System Indexes:
     - Microsoft Dynamics\SL\Applications\DB\Common\Sys\n   • Application Indexes:
     - Microsoft Dynamics\SL\Applications\DB\Update\App\n     - Microsoft Dynamics\SL\Applications\DB\Common\App\n
5. Create or upgrade the Microsoft Dynamics SL databases by using *Database Maintenance* (98.290.00).

**Creating and Running a Stored Procedure**

Almost without exception, if the records from a table must be used in a Microsoft Dynamics SL SDK program, a stored procedure must be created. The stored procedure will also usually have to return different result sets:

- Retrieve all records of interest
- Retrieve one and only one record

Which of these two sets the procedure will return is determined by whether wildcard characters are passed as parameters to the stored procedure. Here are some guidelines:

- The procedure should be a Select statement that selects every field from the table.
- Every field from the unique index of the table should be included as a condition in the Where clause.
- At least one of the Where clause conditions, on one or more of the index fields, should be compared to a replacement parameter.
- If a wildcard is passed to at least the last parameter, then the statement should return every record in the table.
- If actual values are passed, the statement should return one and only one record.
- There should be an Ordered By clause that matches an index of the table. Usually it will match the unique index.

In normal development, it is the responsibility of the programmer to design, create, and run the SQL script file that create the stored procedure for a table. You can run this script file using an update scenario in *Database Maintenance* (98.290.00).
Checking Code

The VB Code Inspector is a Visual Basic add-in provided by Microsoft to “inspect” Visual Basic projects using the Microsoft Dynamics SL SDK controls and functions. It enables you to find and fix certain well-defined problems in your application code. Code Inspector is installed as part of the Microsoft Dynamics SL SDK.

Using the VB Code Inspector

Code Inspector can provide significant benefits when run prior to compiling, either on demand or upon every compile of a project. By default, each time you run Visual Basic, the Code Inspector add-in is loaded and adds a VB Code Inspector menu item to the Visual Basic Add-In menu.

It is recommended that you keep the Code Inspector running all the time (the default behavior). If the Visual Basic project you are working on is not a Microsoft Dynamics SL SDK project, Code Inspector will lay dormant in the background. It only operates with Microsoft Dynamics SL SDK projects.

Although Code Inspector is loaded when you initially run Visual Basic, it does not check application code at that time. Code Inspector performs a code inspection when you explicitly request it to check code by selecting VB Code Inspector from the Add-In menu.

When Code Inspector scans code, it searches for a fixed list of well-defined issues. If Code Inspector finds a problem, it notifies you and provides the option to resolve the problem automatically. Code Inspector almost never makes any changes without first prompting you for permission to make each change. This allows you to be fully aware of any and all changes being made to a Visual Basic project.

If you choose to have Code Inspector automatically fix a problem, you still need to save the project. You also need to check the modified code into your version control system if you use such a tool.

Validating Data-Binding Values on Manually Bound Controls

Occasionally you need to bind a UI control to a data-field that does not come directly from the database. This means the user-defined type passed to the SetAddr() call does not have a corresponding table in the database. Since the user-defined type (sometimes referred to as a structure) does not exist in the database, the Microsoft Dynamics SL kernel cannot get meta-data about the user-defined type from the database.

For example, the Kernel has no idea what fields are in the user-defined type – nor does it know the datatype, length, or order of fields within the user-defined type. In this case, the meta-data must be entered manually into the Property Page associated with the FieldName property of Microsoft Dynamics SL SDK controls such as DSLMaskedText.

Code Inspector searches through the project for any Microsoft Dynamics SL SDK controls that are “unbound”: any Microsoft Dynamics SL SDK control that references a field (via the control’s FieldName property) that has been “manually specified.” When Code Inspector performs a check, it looks for Microsoft Dynamics SL SDK controls having non-zero values in any one of the following Property Page fields:

- Field Offset Value
- Declare Type
- Declare length

The specific properties that are checked include:

- FieldName
- DBNav
- Default
- PV

(These properties all contain appropriate Property Page fields to allow definition of user-defined types.)
A control is considered unbound if one or more of the following parameter values of the FieldName property is non-zero:

- Declare Length
- Declare Type
- Field Offset Value

If an unbound control is found, Code Inspector uses the value in the Struct.FieldName field on the FieldName Property Page to “look up” meta-data about that field from within a Visual Basic project. Once Code Inspector finds the meta-data, it determines if the data-binding values on the FieldName Property Page are accurate. Code Inspector attempts to calculate the correct “length” and “Field Offset Value” for the field, using information in the FieldName Struct.FieldName parameter value and the information referenced in the parameters of the corresponding SetAddr call.

If the data binding values that Code Inspector calculates are incorrect (different from those in the FieldName property), Code Inspector gives you the option of having it automatically resolve the problem by programatically entering the correct values. If Code Inspector is unable to locate meta-data about the Struct.FieldName, it notifies you of this fact but does not alter code.

**Note:** If a particular field is bound to an element of a single-dimensional array, Code Inspector expects the Struct.FieldName field on the FieldName Property Page to be formatted as follows: MyStructure.MyField(1). In other words, it expects the index of the element to which the control is bound to be enclosed within parenthesis (as opposed to something like MyStructure.MyField01 with no parentheses).

Currently, no other changes are made to any lines of Visual Basic source code in any .frm file or Visual Basic module files by Code Inspector.

**Checking for Required Controls, References, and Forms**

Each Microsoft Dynamics SL SDK project must include certain required controls, references, forms, and files. Code Inspector verifies that the appropriate Microsoft Dynamics SL references are present and, if they are missing, adds them to the project without prompting first.

**Required OCXs and Controls for Customization Manager**

The following references are required for an application to be “Customization Manager Aware”:

- Microsoft.Dynamics.SL.Controls
- Solomon.Kernel

**Changing the Default Processing Configuration**

In addition to the VB Code Inspector add-in (CodeInspector.dll), there is also a configuration file delivered with the VB Code Inspector, that is the vbcodeinspector.ini file located in the Microsoft Dynamics\SL\Applications\VT directory. This configuration file is delivered with the default processing options for VB Code Inspector already set.

If you want to change this default configuration, modify this file using any text editor and save the file. Then, re-load Visual Basic and select the VB Code Inspector Add-in from the Add-Ins menu, to display the new default configuration. This configuration will be the new default configuration; it may, in most cases, be changed in the user interface of the VB Code Inspector for a particular Visual Basic session.

The vbcodeinspector.ini file contains several sections, as detailed below:

**[VBCodelnspuctor-ProcessOptions]**

This section contains entries that allow you to define Update and Logging default options for the VB Code Inspector.

- **allow_option_change** — allows you to determine whether the user interface will allow the any changes to the default configuration.
- **logfile** — allows you to decide whether you would like a log file to be created.
logonley — determines whether the VB Code Inspector will display messages to the screen as problems are detected.

no_update — determines whether updates to the Visual Basic code will be allowed during code inspection.

append_to_existing_file — determines whether one log file will be created for all code inspections (named vbcodeinspector.log) or one file will be created for each application (named <projectname>.log).

print_config_in_logfile — determines whether the log file created will contain the configuration settings being used at the time of the Visual Basic code inspection.

customization_check — enables additional, optional checks specific to customization.

logo_check — enables additional, optional checks for Microsoft “Designed for Windows Client” logo requirements.

configuration_check — enables additional, optional checks for control property setting guidelines that may be set up by your development group.

[VBCodeInspector-WarningOptions]

This section contains entries that allow you to define additional, optional, code inspection tests that may be performed. These tests are classified as warnings because they are considered to be less critical issues. It is recommended that these warnings be enabled at some point during code inspection so the messages may be logged for further evaluation at some time.

fieldclass_min_max_check — enables the warning check for SAF.SAFFloat controls with a Fieldclass property set to 124, 125, or 126 that do not have their Min and Max properties set to all 9s.

text_property_assignment_check — enables the warning check for specific control properties (Heading and Text) being assigned to literal values.

mask_warning_check — enables the warning check for SAF.SAFMaskedText Mask property where the number of characters specified is fewer than the actual field size as it is defined to the application.

controls_outside_frame_check — enables the warning check for visible controls that are outside the boundaries of a grid container.

tabindex_in_and_out_of_frame_check — enables the warning check for controls inside a container whose tabindex property values are less than the tabindex for the container, or for controls outside the container that have tabindex property values that are greater than the container, but less than the largest tabindex property value of all controls inside the container.

sqlcurserex_use_check — enables the warning check for the use of SQLCursor instead of the preferred SQLCursorEX.

use_of_third_party_controls_check — enables the warning check for the use of third-party controls in the application that will not be available to the Template function, and Application Server processing.

invalid_property_assignment_check — enables the warning check for specific control properties being assigned explicitly and not through the use of the SetProps API.

blank_in_level_property_check — enables the warning check for a blank in the level property of a SAF control.

order_of_update_levels_check — enables the warning check for non-detail levels that are defined in the SAF.SAFUpdate control after detail levels have been defined.

use_of_a_range_of_controls_check — enables the warning check for all Microsoft Dynamics SL APIs that allow the use of a range of controls to see if the same control is identified as the beginning and ending control. The warning will appear if the same control is not the beginning and ending control. This is particularly useful when you change the tab order of controls. If you have used control ranges, these commands may be affected by changing tab order.

pv_or_dbnav_check — enables the warning check for no PV or DBNav identified as the last key field of any N(avigation) level.

no_key_field_check — enables the warning check for no key field defined for a N(avigation) or L(ookup) level.
[VBCodeInspector-RequiredReferences]
This section identifies the references that will be required for any Microsoft Dynamics SL SDK project. The references required by Microsoft Dynamics SL are entered here for illustration only. They are actually hard-coded into the VB Code Inspector, however additional required references may be added here for specific development group needs.

For example, the SQAOTest.SQAObjectTestingControl is not required for all applications; however, if your development group uses SQA for automated testing, you may want to be sure that all applications contain this reference.

The format for this section is:

ref?=ControlName,ControlFile,SearchStr1,SearchStr2

?: Is an integer incremented by 1 each time.

ControlName: Progid listed in the registry for this reference.

ControlFile: Actual filename for the reference.

SearchStr1: Library name for the actual control (can be found in the Object Browser).

SearchStr2: - (optional) Alternate Library name.

[VBCodeInspector-RequiredComponents]
This section identifies the components required by any Microsoft Dynamics SL SDK project. The components required by Microsoft Dynamics SL are entered here for illustration only. They are actually hard-coded into the VB Code Inspector. However additional required components may be added here for specific development group needs.

The format for this section is:

comp?=componentfilename

?: Is an integer incremented by 1 each time.

Componentfilename: Actual filename of the required component (path is not specified).

[VBCodeInspector-RequiredControls]
This section is used to identify the controls required by any Microsoft Dynamics SL SDK project. The controls required by Microsoft Dynamics SL are entered here for illustration, only. They are actually hard-coded into the VB Code Inspector. However, additional required controls may be added.

The format for this section is:

tcl?=ControlLibrary.Class,ControlName,Index,AllForms

?: Is an integer incremented by 1 each time.

ControlLibrary.Class: The Library and Class for the control. Use Object Browser to determine this.

ControlName: The name assigned to the control when it is pasted on the form.

Index: The index value to be assigned to the control. (-1 is the value for no index).

AllForms: An indicator of whether the control should be on all forms, or just Form1. If Allforms = -1 then the control must be on all forms. If Allforms = 0 then the control should only be on Form1.

[VBCodeInspector-RequiredProperties:ControlName]
This section may be repeated for each controlname specified in the [VBCodeInspector-RequiredControls] section. It is used to validate/assign properties for the required controls. If the Required control exists, then the properties for that control are validated against the properties assigned here. If the Required control does not exist and you decide to paste it on the form, this section will be used to assign the appropriate properties to the control.

The format for this section is:

Prop?=propname,propvalue

?: Is an integer incremented by 1 each time.
propname: Name of the control property.
propvalue: Value of that property to be assigned/validated.

[VBCodeInspector-LogoControls]
This section identifies controls whose properties will need to be checked for the Microsoft “Designed for Windows Client” logo checking. Microsoft Dynamics SL has provided the correct Logo-checking configuration in this file. There should be no reason to change the current configuration. To disable Logo checking, set logo_check=0 in the [VBCodeInspector-ProcessOptions] section.
The format for this section is:
   ctl?=ControlLibrary.Class
?: Is an integer incremented by 1 each time.
ControlLibrary.Class: ControlLibrary and Class value for the control to be checked. This value may be found using the Visual Basic Object Browser.

[VBCodeInspector-LogoProperties: ControlLibrary.Class]
This section may be repeated for each ControlLibrary.class specified in the [VBCodeInspector-LogoControls] section. Microsoft Dynamics SL has provided the correct logo-checking configuration in this file. There should be no reason to change the current configuration. To disable logo checking, set logo_check=0 in the [VBCodeInspector-ProcessOptions] section.
The format for this section is the same as for the [VBCodeInspector-RequiredProperties:ControlName] section.

[VBCodeInspector-ConfigControls]
This identifies controls whose properties should be checked for property value consistency.
The format for this section is the same as for the [VBCodeInspector-LogoControls] section.

[VBCodeInspector-ConfigProperties: ControlLibrary.Class]
This section may be repeated for each ControlLibrary.class specified in the [VBCodeInspector-ConfigControls] section. It is used to assign property values to specific control types.
The format for this section is:
   prop?=propname,propvalue,exception,inclusion
?: is an integer incremented by 1 each time.
Propname: Name of the property for that control.
Propvalue: Property value to be validated/assigned for that control.
Exception: A specific exception to the property rule. A controlname that is to be excluded from this property assignment.
Inclusion: Specific controlname that is to be assigned this property value. No other controlname of this same type will get this property assignment/validation.

The VB Code Inspector Log File
The VB Code Inspector can optionally create a log file of errors encountered during an inspection. You may choose to create one log file for each project that is inspected, or to create one log file for all code inspections. If you select an individual log file, it is given the projectname with a .LOG extension. If you choose one log file for all inspections, the name of the file is VBCodeInspector.LOG. In both cases, the log file is saved to the \Microsoft Dynamics SL\Eventlog directory.
The log file is intended to provide information regarding the code inspection that was processed. First, it prints the configuration that was used for the inspection. This portion of the log file indicates which Process, Inspection, and Warning Options were chosen, and what the Required, Logo, and Configuration values were at the time the code inspection was performed. This is especially useful if
you alter your configuration each time you run the Code Inspector and need to keep track of which 
errors occur using each configuration.

Then, as the code inspection is performed, Code Inspector logs errors as it encounters them. If Code 
Inspector makes a suggestion for a code change, that is also logged, and if you choose to accept the 
code change, a message is logged indicating that a change was made.

This way, you have an audit list of the changes the VB Code Inspector was allowed to make, and the 
other errors that still need resolution. In every instance, context information is given so that you can go 
back and verify the change to your code. When your code inspection is complete, you may want to 
delete the code inspection log. This log is not viewable in the Event Log Viewer (95.290.00) and 
therefore, cannot be deleted using that screen.
Required Fields

A new feature in 6.0 is the ability for Microsoft Dynamics SL SDK applications to have a notation for the required fields on a screen. As the application loads, the kernel determines which fields are required to be entered by the end user. The BlankErr property is checked. If the field is required, then the field’s border is shaded a “user-defined” color, which is set in User Maintenance (95.260.00).

Here are a few rules of use:

- Required fields that are disabled will not have color in the border until the control becomes enabled.
- Combo boxes never contain color, as they already have a value set.
- Check boxes never contain color, as they already have a value set.
- Option buttons never contain color, assuming they have a value set.
- If a control’s BlankErr property changes at runtime, the correct setting will be applied.
Adding Applications to the Menu System

The Microsoft Dynamics SL menu system is made of the following components:

- The parent application
- System database tables
- Maintenance screens
- Menu cache files

The Parent Application

The parent application is the main form of Microsoft Dynamics SL, where the loading of applications takes place. Module and application navigation are displayed on this form.

System Database Tables

The system database tables are:

- SLMenultem
- Modules
- Screen
- UserGrp

Menu Cache Files

These XML representations of a user’s menus are generated by the parent application, and they allow access to screens based on the user’s access rights.

Maintenance Screens

- Menu Maintenance (98.350.00) — Use this screen to create and modify menus that are associated with user groups. A menu can be exported to or imported from one Microsoft Dynamics SL installation to another using this screen. Customized menus are stored in the SLMenultem table in the system database.

- Modules Maintenance (98.320.00) — Use this screen to add your modules to the list of modules. Your entries will not be removed during future database upgrades. Modules that are designated as active will appear on Preload Screens (95.270.01) after you click the Preload button on the System Manager Access Rights Maintenance (95.270.00) screen. They will also appear in the parent application All Modules group.

- Screen Maintenance (98.330.00) — Use this screen to add your applications to the list of screens. Your entries will not be removed during a future database upgrade. Make sure to mark the screens so they show in the parent application All Modules group.
Reference

Controls

DSLDate Control

ToolBox Icon

Description

The DSLDate is a bound control which can display a date that is entered by the user or programmatically assigned to the underlying date data field at runtime. The display format of the DSLDate control is determined by the Windows Short International Date format.

Note: If you need assistance converting SAFDate controls to DSLDate, see “Appendix D: Microsoft Dynamics SL SDK Application Upgrade Utility” on page 319.

Remarks

The DSLDate control supports the following keys to affect actions in a date field:

<table>
<thead>
<tr>
<th>Key</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 through 9</td>
<td>Enter the date. Date entry follows the rules that apply to the date range. For example, entering 13 in the month segment results in a month entry of 3. The software assumes you changed 1 to 3.</td>
</tr>
<tr>
<td>Comma, period, forward slash, right arrow</td>
<td>Move to the next date segment.</td>
</tr>
<tr>
<td>BACKSPACE, left arrow</td>
<td>Move to the previous date segment.</td>
</tr>
<tr>
<td>ALT+HOME</td>
<td>Move to the first date segment.</td>
</tr>
<tr>
<td>ALT+END</td>
<td>Move to the last date segment.</td>
</tr>
<tr>
<td>F1</td>
<td>Open help.</td>
</tr>
<tr>
<td>F2</td>
<td>Display a dialog for entry of relative date values.</td>
</tr>
<tr>
<td>F3</td>
<td>Display a pop-up calendar from which you can select a date. You can also display the calendar by double right-clicking within the control.</td>
</tr>
<tr>
<td>F5 or DELETE</td>
<td>Clear the date field.</td>
</tr>
<tr>
<td>F7</td>
<td>Enter the current date.</td>
</tr>
<tr>
<td>F9</td>
<td>Display notes.</td>
</tr>
<tr>
<td>TAB</td>
<td>Move to the next tab stop.</td>
</tr>
<tr>
<td>SHIFT+TAB</td>
<td>Move to the previous tab stop.</td>
</tr>
</tbody>
</table>

Properties

<table>
<thead>
<tr>
<th>Alignment</th>
<th>Font</th>
<th><strong>Max</strong></th>
<th>TabStop</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackColor</td>
<td>ForeColor</td>
<td><strong>Min</strong></td>
<td>Tag</td>
</tr>
<tr>
<td><strong>BlankErr</strong></td>
<td>Heading</td>
<td>MouseIcon (D)</td>
<td>ToolTipText</td>
</tr>
<tr>
<td>Custom</td>
<td>Height</td>
<td>MousePointer (D)</td>
<td>Top</td>
</tr>
</tbody>
</table>
**HelpContextID** | **Name (D)** | **Trigger (D)**
---|---|---
*DragIcon* | **Index (D)** | **NoteButton** | **Visible**
*DragMode* | **InGrid (D)** | **Spin** | **WhatsThisHelpID**
**Enabled** | **Left** | **SpinIncrement** | **Width**
**FieldName (D)** | **Level (D)** | **TabIndex (D)**

* Reserved for Microsoft and should not be used.

** Property values can be narrowed at runtime but not expanded, depending on the level at which the changes are being made (All User, One User, etc.). See the “Security” section of the Customization Manager documentation.

(D) The property can be modified only at design time but not at runtime.

---

**Events**

| Chk | *DragDrop* | GotFocus | MouseDown |
---|---|---|---|
Default | *DragOver* | LostFocus | MouseMove |

*Validate

* Reserved for Microsoft and should not be used.

---

**Methods**

| AboutBox | *Move* | ShowWhatsThis |
---|---|---|
*Drag* | *SetFocus* | *ZOrder*

* Reserved for Microsoft and should not be used.

Use the **ApplSetFocus** statement instead of the **SetFocus** method.

---

**See Also**

DispField Statements, SetAddr Statement, SetDefaults Statement, SetProps Statement
DSLCheck Control

ToolBox Icon

Description

The DSLCheck is a bound control that displays an X when selected. The X disappears when the check box is cleared by clicking it again. Use this control to give the user a True/False or Yes/No option.

Remarks

The DSLCheck control functions similar to DSLOption buttons with one important exception. In particular, any number of check boxes on a form can be selected at the same time whereas only one DSLOption button can be selected within any particular group.

To display a string literal next to the DSLCheck, set the Caption property. Use the TrueText property to determine the data value that will be stored in the database when the box is checked. The FalseText property determines the data value that will be stored in the database when the box is unchecked.

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment</td>
<td>FalseText (D) Left Top</td>
</tr>
<tr>
<td>BackColor</td>
<td>FieldName (D) Level (D) Trigger (D)</td>
</tr>
<tr>
<td><strong>BlankErr</strong></td>
<td>Font MouseIcon (D) TrueText (D)</td>
</tr>
<tr>
<td>Caption</td>
<td>ForeColor MousePointer (D) <strong>Visible</strong></td>
</tr>
<tr>
<td>Custom</td>
<td>Heading Name (D) WhatsThisHelp</td>
</tr>
<tr>
<td>Default</td>
<td>Height TabIndex (D) Width</td>
</tr>
<tr>
<td>*DragIcon</td>
<td>HelpContextID TabStop</td>
</tr>
<tr>
<td>*DragMode</td>
<td>Index (D) Tag</td>
</tr>
<tr>
<td><strong>Enabled</strong></td>
<td>InGrid (D) ToolTipText</td>
</tr>
</tbody>
</table>

* Reserved for Microsoft and should not be used.
** Property values can be narrowed at runtime but not expanded, depending on the level at which the changes are being made (All User, One User, etc.). See the “Security” section of the Customization Manager documentation.

(D) The property can be modified only at design time but not at runtime.

Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chk</td>
<td>*DragOver MouseDown</td>
</tr>
<tr>
<td>Default</td>
<td>GotFocus MouseMove</td>
</tr>
<tr>
<td>*DragDrop</td>
<td>LostFocus MouseUp</td>
</tr>
</tbody>
</table>

* Reserved for Microsoft and should not be used.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AboutBox</td>
<td>*Move ShowWhatsThis</td>
</tr>
<tr>
<td>*Drag</td>
<td>*SetFocus *ZOrder</td>
</tr>
</tbody>
</table>

* Reserved for Microsoft and should not be used.

Use the ApplSetFocus statement instead of the SetFocus method.

See Also

DispField Statements, SetAddr Statement, SetDefaults Statement, SetProps Statement
DSLCombo Control

ToolBox Icon

Description
The **DSLCombo** is a bound control that combines the features of a text box and a list box. Use this control to allow the user to select an item from a pre-defined list of valid values.

Properties

<table>
<thead>
<tr>
<th>BackColor</th>
<th>Font</th>
<th>Level (D)</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BlankErr</strong></td>
<td>ForeColor</td>
<td>List (D)</td>
<td>ToolTipText</td>
</tr>
<tr>
<td>Custom</td>
<td>Heading</td>
<td>MouseIcon (D)</td>
<td>Top</td>
</tr>
<tr>
<td>Default</td>
<td>Height</td>
<td>MousePointer (D)</td>
<td>Trigger (D)</td>
</tr>
<tr>
<td>*DragIcon</td>
<td>HelpContextID</td>
<td>Name (D)</td>
<td>**Visible</td>
</tr>
<tr>
<td>*DragMode</td>
<td>Index (D)</td>
<td>NoteButton</td>
<td>WhatsThisHelp</td>
</tr>
<tr>
<td>**Enabled</td>
<td>InGrid (D)</td>
<td>TabIndex (D)</td>
<td>Width</td>
</tr>
<tr>
<td>FieldName (D)</td>
<td>Left</td>
<td>TabStop</td>
<td></td>
</tr>
</tbody>
</table>

* Reserved for Microsoft and should not be used.

** Property values can be narrowed at runtime but not expanded, depending on the level at which the changes are being made (All User, One User, etc.). See the “Security” section of the Customization Manager documentation.

(D) The property can be modified only at design time but not at runtime.

Events

<table>
<thead>
<tr>
<th>Chk</th>
<th>*DragDrop</th>
<th>GotFocus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>*DragOver</td>
<td>LostFocus</td>
</tr>
</tbody>
</table>

* Reserved for Microsoft and should not be used.

Methods

<table>
<thead>
<tr>
<th>AboutBox</th>
<th>*Move</th>
<th>ShowWhatsThis</th>
</tr>
</thead>
<tbody>
<tr>
<td>*Drag</td>
<td>*SetFocus</td>
<td>*ZOrder</td>
</tr>
</tbody>
</table>

* Reserved for Microsoft and should not be used.

Use the **ApplSetFocus** statement instead of the **SetFocus** method.

See Also
DispField Statements, SetAddr Statement, SetDefaults Statement
### DSLFloat Control

#### ToolBox Icon

![Icon](image)

#### Description

The **DSLFloat** is a bound control which can display a floating point value that is entered by the user or programmatically assigned to the underlying double precision data field at runtime.

#### Remarks

The **DSLFloat** control supports the following “hotkeys” for editing a numeric data field:

<table>
<thead>
<tr>
<th>Key</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>F5</td>
<td>Zero out the current contents of the control.</td>
</tr>
</tbody>
</table>

#### Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BackColor</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>BlankErr</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Custom</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DecimalPlaces</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Default</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*DragIcon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*DragMode</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Enabled</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FieldClass</td>
<td>(D)</td>
<td>Field name (D)</td>
</tr>
<tr>
<td>FieldName</td>
<td>(D)</td>
<td>Level (D)</td>
</tr>
<tr>
<td>Font</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Max</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ForeColor</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Min</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Height</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HelpContextID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index</td>
<td>(D)</td>
<td>Name (D)</td>
</tr>
<tr>
<td>InGrid</td>
<td>(D)</td>
<td>NoteButton</td>
</tr>
<tr>
<td><em>Reserved for Microsoft and should not be used.</em>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*DragDrop</td>
<td>*Move</td>
<td>MouseDown</td>
</tr>
<tr>
<td>*DragOver</td>
<td>MouseDown</td>
<td></td>
</tr>
<tr>
<td>Default</td>
<td>GotFocus</td>
<td>MouseMove</td>
</tr>
<tr>
<td>*Drag</td>
<td>MouseMove</td>
<td></td>
</tr>
<tr>
<td>*DragDrop</td>
<td>LostFocus</td>
<td>MouseUp</td>
</tr>
<tr>
<td>*DragIcon</td>
<td>*SetFocus</td>
<td>*ZOrder</td>
</tr>
<tr>
<td>Custom</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Visible</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FieldName</td>
<td>(D)</td>
<td>Top</td>
</tr>
<tr>
<td>Level</td>
<td>(D)</td>
<td>TabIndex</td>
</tr>
<tr>
<td>MousePointer</td>
<td>(D)</td>
<td>Tag</td>
</tr>
<tr>
<td>MouseIcon</td>
<td>(D)</td>
<td>ToolTipText</td>
</tr>
<tr>
<td>NoteButton</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Reserved for Microsoft and should not be used.</em>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*DragMode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index</td>
<td>(D)</td>
<td>Name (D)</td>
</tr>
<tr>
<td>InGrid</td>
<td>(D)</td>
<td>NoteButton</td>
</tr>
<tr>
<td>*DragIcon</td>
<td>*Move</td>
<td>MouseDown</td>
</tr>
<tr>
<td>*DragOver</td>
<td>MouseDown</td>
<td></td>
</tr>
<tr>
<td>Default</td>
<td>GotFocus</td>
<td>MouseMove</td>
</tr>
<tr>
<td>*Drag</td>
<td>MouseMove</td>
<td></td>
</tr>
<tr>
<td>*DragDrop</td>
<td>LostFocus</td>
<td>MouseUp</td>
</tr>
<tr>
<td>*DragIcon</td>
<td>*SetFocus</td>
<td>*ZOrder</td>
</tr>
<tr>
<td>Custom</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Visible</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FieldName</td>
<td>(D)</td>
<td>Top</td>
</tr>
<tr>
<td>Level</td>
<td>(D)</td>
<td>TabIndex</td>
</tr>
<tr>
<td>MousePointer</td>
<td>(D)</td>
<td>Tag</td>
</tr>
<tr>
<td>MouseIcon</td>
<td>(D)</td>
<td>ToolTipText</td>
</tr>
<tr>
<td>NoteButton</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Reserved for Microsoft and should not be used.</em>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*DragMode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index</td>
<td>(D)</td>
<td>Name (D)</td>
</tr>
<tr>
<td>InGrid</td>
<td>(D)</td>
<td>NoteButton</td>
</tr>
<tr>
<td>*DragIcon</td>
<td>*Move</td>
<td>MouseDown</td>
</tr>
<tr>
<td>*DragOver</td>
<td>MouseDown</td>
<td></td>
</tr>
<tr>
<td>Default</td>
<td>GotFocus</td>
<td>MouseMove</td>
</tr>
<tr>
<td>*Drag</td>
<td>MouseMove</td>
<td></td>
</tr>
<tr>
<td>*DragDrop</td>
<td>LostFocus</td>
<td>MouseUp</td>
</tr>
<tr>
<td>*DragIcon</td>
<td>*SetFocus</td>
<td>*ZOrder</td>
</tr>
<tr>
<td>Custom</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Visible</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FieldName</td>
<td>(D)</td>
<td>Top</td>
</tr>
<tr>
<td>Level</td>
<td>(D)</td>
<td>TabIndex</td>
</tr>
<tr>
<td>MousePointer</td>
<td>(D)</td>
<td>Tag</td>
</tr>
<tr>
<td>MouseIcon</td>
<td>(D)</td>
<td>ToolTipText</td>
</tr>
<tr>
<td>NoteButton</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Reserved for Microsoft and should not be used.</em>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*DragMode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index</td>
<td>(D)</td>
<td>Name (D)</td>
</tr>
<tr>
<td>InGrid</td>
<td>(D)</td>
<td>NoteButton</td>
</tr>
<tr>
<td>*DragIcon</td>
<td>*Move</td>
<td>MouseDown</td>
</tr>
<tr>
<td>*DragOver</td>
<td>MouseDown</td>
<td></td>
</tr>
<tr>
<td>Default</td>
<td>GotFocus</td>
<td>MouseMove</td>
</tr>
<tr>
<td>*Drag</td>
<td>MouseMove</td>
<td></td>
</tr>
<tr>
<td>*DragDrop</td>
<td>LostFocus</td>
<td>MouseUp</td>
</tr>
<tr>
<td>*DragIcon</td>
<td>*SetFocus</td>
<td>*ZOrder</td>
</tr>
<tr>
<td>Custom</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Visible</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FieldName</td>
<td>(D)</td>
<td>Top</td>
</tr>
<tr>
<td>Level</td>
<td>(D)</td>
<td>TabIndex</td>
</tr>
<tr>
<td>MousePointer</td>
<td>(D)</td>
<td>Tag</td>
</tr>
<tr>
<td>MouseIcon</td>
<td>(D)</td>
<td>ToolTipText</td>
</tr>
<tr>
<td>NoteButton</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Reserved for Microsoft and should not be used.</em>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*DragMode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index</td>
<td>(D)</td>
<td>Name (D)</td>
</tr>
<tr>
<td>InGrid</td>
<td>(D)</td>
<td>NoteButton</td>
</tr>
<tr>
<td>*DragIcon</td>
<td>*Move</td>
<td>MouseDown</td>
</tr>
<tr>
<td>*DragOver</td>
<td>MouseDown</td>
<td></td>
</tr>
<tr>
<td>Default</td>
<td>GotFocus</td>
<td>MouseMove</td>
</tr>
<tr>
<td>*Drag</td>
<td>MouseMove</td>
<td></td>
</tr>
<tr>
<td>*DragDrop</td>
<td>LostFocus</td>
<td>MouseUp</td>
</tr>
<tr>
<td>*DragIcon</td>
<td>*SetFocus</td>
<td>*ZOrder</td>
</tr>
<tr>
<td>Custom</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Visible</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FieldName</td>
<td>(D)</td>
<td>Top</td>
</tr>
<tr>
<td>Level</td>
<td>(D)</td>
<td>TabIndex</td>
</tr>
<tr>
<td>MousePointer</td>
<td>(D)</td>
<td>Tag</td>
</tr>
<tr>
<td>MouseIcon</td>
<td>(D)</td>
<td>ToolTipText</td>
</tr>
<tr>
<td>NoteButton</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Reserved for Microsoft and should not be used.</em>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*DragMode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index</td>
<td>(D)</td>
<td>Name (D)</td>
</tr>
<tr>
<td>InGrid</td>
<td>(D)</td>
<td>NoteButton</td>
</tr>
<tr>
<td>*DragIcon</td>
<td>*Move</td>
<td>MouseDown</td>
</tr>
<tr>
<td>*DragOver</td>
<td>MouseDown</td>
<td></td>
</tr>
<tr>
<td>Default</td>
<td>GotFocus</td>
<td>MouseMove</td>
</tr>
<tr>
<td>*Drag</td>
<td>MouseMove</td>
<td></td>
</tr>
<tr>
<td>*DragDrop</td>
<td>LostFocus</td>
<td>MouseUp</td>
</tr>
<tr>
<td>*DragIcon</td>
<td>*SetFocus</td>
<td>*ZOrder</td>
</tr>
</tbody>
</table>

(D) The property can be modified only at design time but not at runtime.

The valid FieldClass property values are embedded inside of the control and are viewable via a ComboBox in the Visual Basic Property Window.

#### Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chk</td>
<td>*DragOver</td>
</tr>
<tr>
<td>Default</td>
<td>GotFocus</td>
</tr>
<tr>
<td>*DragDrop</td>
<td>LostFocus</td>
</tr>
<tr>
<td><em>Reserved for Microsoft and should not be used.</em>*</td>
<td></td>
</tr>
</tbody>
</table>

#### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AboutBox</td>
<td>*Move</td>
</tr>
<tr>
<td>*Drag</td>
<td>*SetFocus</td>
</tr>
<tr>
<td><em>Reserved for Microsoft and should not be used.</em>*</td>
<td></td>
</tr>
</tbody>
</table>

Use the `ApplSetFocus` statement instead of the `SetFocus` method.

#### See Also

- `ApplSetFocus Statement`
- `DispField Statements`
- `SetAddr Statement`
- `SetDefaults Statement`
- `SetProps Statement`
**DSLGrid Control**

**ToolBox Icon**

![Image of DSLGrid icon]

**Description**
The **DSLGrid** is a bound control that can simultaneously display many records of information. Spreadsheets are commonly referred to as “grids.”

**Remarks**
The **DSLGrid** control is used to implement Detail levels.

The font of the grid cell(s) can be modified at runtime by modifying the **Font** property of the cell(s) corresponding form view control.

The foreground color of the grid cell(s) can be modified at runtime by modifying the **ForeColor** property of the cell(s) corresponding form view control.

Grid view help is driven by properties on the form view controls. If the current column’s form view partner has a help context, that value will be used. Otherwise, the help context of the form view container (for example, the frame) will be used.

The background color of the grid cell(s) can be modified at runtime by modifying the **BackColor** property of the cell(s) corresponding form view control.

To vary the BackColor on a row-by-row basis, use a **SetProp()** call on the form view control. The **SetProp()** call should be made from within the **LineGotFocus** event of the **DSLGrid**.

To modify the BackColor of an entire column, use a **MSetProp()** call on the form view control. Note that in the **MSetProp()** scenario, this call must be made before the **DSLGrid** receives focus (such as at screen load, or in a master key **Chk** event of a header level). Do not call **MSetProp()** from within **DSLGrid** events such as **LineGotFocus()**.

Grid view tooltips are driven by the **ToolTipText** property on the form view controls. If the current column’s form view partner has a value for the **ToolTipText** property, that value will be used.

The following steps outline the basic requirements to implement a single level detail screen:

**Database**

1. Create one or more database tables to hold the detail information from the grid.
2. Create a stored procedure that will be used to retrieve the database information.

**Controls**

1. Set the **Levels** property of the **DSLUpdate** control. Add an alias;D or alias;DA to the character string that is used to define the **Levels** property.
2. Place a Visual Basic frame on a Visual Basic form.
3. Paste one or more Microsoft Dynamics SL bound controls on top of this panel. These controls should be pasted with a Level Number corresponding to the position of the new level defined in step 1.
4. Place the grid on top of a Visual Basic frame.
5. Set the **DBNav** property of the **DSLGrid** control to the name of an SQL stored procedure created in step 2 of the Database requirements.
6. Set the **ColsFrozen** property of the **DSLGrid** control to 0 to allow all columns of the **DSLGrid** to scroll horizontally, or to 1 or higher to keep the left-most column(s) from scrolling horizontally.
7. If the detail records must appear in alphabetical order then the fields corresponding to the Order By clause of the **DBNav** SQL statement should be marked as key fields.
Code
1. Add code to the Form1_Load event that includes the `SetAddr()`, `SqlCursor()`, and `DetailSetup()` calls for the new level.
2. Run the application.

Properties

<table>
<thead>
<tr>
<th>BackColor (R)</th>
<th>*Enabled</th>
<th>Left (D)</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackColor (R)</td>
<td>Font (R)</td>
<td>Name (D)</td>
<td>*ToolTipText</td>
</tr>
<tr>
<td>Custom</td>
<td>ForeColor (R)</td>
<td>NoteButton</td>
<td>Top (D)</td>
</tr>
<tr>
<td>DBNav (D)</td>
<td>Height (D)</td>
<td>NoteColumn</td>
<td>Visible (R)</td>
</tr>
</tbody>
</table>
*DragIcon | *HelpContextID | TabIndex (D) | *WhatsThisHelp |
*DragMode | *Index | TabStop | Width (D) |

* Reserved for Microsoft and should not be used.
** Property values can be narrowed at runtime but not expanded, depending on the level at which the changes are being made (All User, One User, etc.). See the “Security” section of the Customization Manager documentation.
(D) The property can be modified only at design time but not at runtime.
(R) Following the property indicates that the property can be modified only at runtime but not at design time.

Events

*DragDrop | LineChk |
*DragOver | LineGotFocus |
GotFocus | LostFocus |

* Reserved for Microsoft and should not be used.

Methods

AboutBox | *Move | ShowWhatsThis |
*Drag | *SetFocus | *ZOrder |

* Reserved for Microsoft and should not be used.

Use the `ApplSetFocus` statement instead of the `SetFocus` method.

See Also

`DetailLoad Statement`, `DetailSave Statement`, `DetailSetup Functions`, `Font Property`, `MDisplay Statement`, `MGetDelHandle Function`, `MSet Statement`, `MSetProp Statement`, `DSLUpdate Control`
DSLInteger Control

ToolBox Icon

Description
The DSLInteger is a bound control which can display an integer value that is entered by the user or programmatically assigned to the underlying integer data field at runtime.

Remarks
The DSLInteger control supports the following “hotkeys” for editing an integer data field:

<table>
<thead>
<tr>
<th>Key</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>F5</td>
<td>Zero out the current contents of the control.</td>
</tr>
</tbody>
</table>

Properties

<table>
<thead>
<tr>
<th>Alignment</th>
<th>Font</th>
<th><strong>Max</strong></th>
<th>TabStop</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackColor</td>
<td>ForeColor</td>
<td><strong>Min</strong></td>
<td>Tag</td>
</tr>
<tr>
<td><strong>BlankErr</strong></td>
<td>Heading</td>
<td>MouseIcon (D)</td>
<td>ToolTipText</td>
</tr>
<tr>
<td>Custom</td>
<td>Height</td>
<td>MousePointer(D)</td>
<td>Top</td>
</tr>
<tr>
<td>Default</td>
<td>HelpContextID</td>
<td>Name (D)</td>
<td>Trigger (D)</td>
</tr>
<tr>
<td><em>DragIcon</em>*</td>
<td>Index (D)</td>
<td>NoteButton</td>
<td><strong>Visible</strong></td>
</tr>
<tr>
<td><em>DragMode</em>*</td>
<td>InGrid (D)</td>
<td>Spin</td>
<td>WhatsThisHelpID</td>
</tr>
<tr>
<td><strong>Enabled</strong></td>
<td>Left</td>
<td>SpinIncrement</td>
<td>Width</td>
</tr>
<tr>
<td>FieldName (D)</td>
<td>Level (D)</td>
<td>TabIndex (D)</td>
<td></td>
</tr>
</tbody>
</table>

(Reserved for Microsoft and should not be used.)

** Property values can be narrowed at runtime but not expanded, depending on the level at which the changes are being made (All User, One User, etc.). See the “Security” section of the Customization Manager documentation.

(D) The property can be modified only at design time but not at runtime.

Events

<table>
<thead>
<tr>
<th>Chk</th>
<th>*DragOver</th>
<th>MouseDown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>GotFocus</td>
<td>MouseMove</td>
</tr>
<tr>
<td>*DragDrop</td>
<td>LostFocus</td>
<td>MouseUp</td>
</tr>
</tbody>
</table>

(Reserved for Microsoft and should not be used.)

Methods

<table>
<thead>
<tr>
<th>AboutBox</th>
<th>*Move</th>
<th>ShowWhatsThis</th>
</tr>
</thead>
<tbody>
<tr>
<td>*Drag</td>
<td>*SetFocus</td>
<td>*ZOrder</td>
</tr>
</tbody>
</table>

(Reserved for Microsoft and should not be used.)

Use the **ApplSetFocus** statement instead of the **SetFocus** method.

See Also
DispField Statements, SetAddr Statement, SetDefaults Statement, SetProps Statement
DSLMaskedText Control

ToolBox Icon

Description
The **DSLMaskedText** is a bound control which can display text information that is entered by the user or programmatically assigned to the underlying string data field at runtime.

Remarks
The **DSLMaskedText** control supports the following "hotkeys" for editing a string data field:

<table>
<thead>
<tr>
<th>Key</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>F5</td>
<td>Blank out the current contents of the control.</td>
</tr>
<tr>
<td>F3</td>
<td>Display a list of possible values for the field. PV property must be implemented.</td>
</tr>
</tbody>
</table>

Properties

<table>
<thead>
<tr>
<th><strong>Alignment</strong></th>
<th><strong>FieldName (D)</strong></th>
<th><strong>Level (D)</strong></th>
<th><strong>Tag</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BackColor</strong></td>
<td><strong>Font</strong></td>
<td>**<strong>Mask</strong></td>
<td>ToolTipText</td>
</tr>
<tr>
<td><strong>DBNav (D)</strong></td>
<td><strong>FieldName</strong></td>
<td><strong>MouseIcon (D)</strong></td>
<td><strong>Top</strong></td>
</tr>
<tr>
<td><strong>Default</strong></td>
<td><strong>Height</strong></td>
<td><strong>Name (D)</strong></td>
<td><strong>Visible</strong></td>
</tr>
<tr>
<td>*<strong>DragIcon</strong></td>
<td><strong>Heading</strong></td>
<td><strong>MousePointer (D)</strong></td>
<td><strong>Trigger (D)</strong></td>
</tr>
<tr>
<td><strong>DBNav (D)</strong></td>
<td><strong>HelpContextID</strong></td>
<td><strong>NoteButton</strong></td>
<td><strong>WhatsThisHelp</strong></td>
</tr>
<tr>
<td><strong>FName (D)</strong></td>
<td><strong>Index (D)</strong></td>
<td><strong>PV (D)</strong></td>
<td><strong>Width</strong></td>
</tr>
<tr>
<td><strong>FieldClass (D)</strong></td>
<td><strong>Left</strong></td>
<td><strong>TabIndex (D)</strong></td>
<td><strong>Visible</strong></td>
</tr>
</tbody>
</table>

* Reserved for Microsoft and should not be used.

** Property values can be narrowed at runtime but not expanded, depending on the level at which the changes are being made (All User, One User, etc.). See the "Security" section of the Customization Manager documentation.

(D) The property can be modified only at design time but not at runtime.

The valid FieldClass property values are embedded inside of the control and are viewable via a ComboBox in the Visual Basic Property Window.

Events

<table>
<thead>
<tr>
<th><strong>Chk</strong></th>
<th>*<strong>DragOver</strong></th>
<th><strong>MouseDown</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Default</strong></td>
<td><strong>GotFocus</strong></td>
<td><strong>MouseMove</strong></td>
</tr>
<tr>
<td>*<strong>DragDrop</strong></td>
<td><strong>LostFocus</strong></td>
<td><strong>MouseUp</strong></td>
</tr>
</tbody>
</table>

* Reserved for Microsoft and should not be used.

Methods

<table>
<thead>
<tr>
<th><strong>AboutBox</strong></th>
<th>*<strong>Move</strong></th>
<th><strong>ShowWhatsThis</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>*<strong>Drag</strong></td>
<td>*<strong>SetFocus</strong></td>
<td>*<strong>ZOrder</strong></td>
</tr>
</tbody>
</table>

* Reserved for Microsoft and should not be used.

Use the **ApplSetFocus** statement instead of the **SetFocus** method.
See Also
DBNavFetch Functions, DispField Statements, PVChkFetch Functions, SetAddr Statement, SetDefaults Statement, SetProps Statement
DSLOption Control

ToolBox Icon

Description
The DSLOption button is a bound control that displays an option that can be turned on or off.

Remarks
DSLOption buttons are used as part of an option group to display multiple choices from which the user can select only one. DSLOption buttons can be grouped by drawing them inside a Visual Basic frame. To group option buttons in a frame, draw the frame first, then draw the option buttons inside. All option buttons within a container, such as a frame, are treated as a group.

While option buttons and check boxes may appear to function similarly, there is one important difference. When a user selects an option button, all other option buttons in the same group are then turned off. In contrast, multiple check boxes can be selected.

If an option button group appears within a grid, the grid uses the Heading property of the first option button as the column header.

To display a string literal next to the DSLOption button, set the Caption property. Use the TrueText property to determine the data value that will be stored in the database when the option button is selected.

The following property values are always derived from the first option button in the group, since by definition only one option button can be selected within any particular group: BlankErr, Default, FieldClass, FieldName, Heading, Level and Trigger.

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Alignment</th>
<th><strong>Enabled</strong></th>
<th>InGrid (D)</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackColor</td>
<td></td>
<td>FieldName (D)</td>
<td>Left</td>
<td>ToolTipText</td>
</tr>
<tr>
<td><strong>BlankErr</strong></td>
<td>Font</td>
<td>Level (D)</td>
<td>Top</td>
<td></td>
</tr>
<tr>
<td>Caption</td>
<td>ForeColor</td>
<td>MouseIcon (D)</td>
<td>Trigger (D)</td>
<td></td>
</tr>
<tr>
<td>Custom</td>
<td>Heading</td>
<td>MousePointer (D)</td>
<td>TrueText (D)</td>
<td></td>
</tr>
<tr>
<td>Default</td>
<td>Height</td>
<td>Name (D)</td>
<td><strong>Visible</strong></td>
<td></td>
</tr>
<tr>
<td>*DragIcon</td>
<td>HelpContextID</td>
<td>TabIndex (D)</td>
<td>WhatsThisHelp</td>
<td></td>
</tr>
<tr>
<td>*DragMode</td>
<td>Index (D)</td>
<td>TabStop</td>
<td>Width</td>
<td></td>
</tr>
</tbody>
</table>

* Reserved for Microsoft and should not be used.

** Property values can be narrowed at runtime but not expanded, depending on the level at which the changes are being made (All User, One User, etc.). See the "Security" section of the Customization Manager documentation.

(D) The property can be modified only at design time but not at runtime.

Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Chk</th>
<th>*DragOver</th>
<th>MouseDown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>GotFocus</td>
<td>MouseMove</td>
<td></td>
</tr>
<tr>
<td>*DragDrop</td>
<td>LostFocus</td>
<td>MouseUp</td>
<td></td>
</tr>
</tbody>
</table>

* Reserved for Microsoft and should not be used.
Methods

AboutBox  *Move  ShowWhatsThis
*Drag  *SetFocus  *ZOrder

* Reserved for Microsoft and should not be used.
Use the ApplSetFocus statement instead of the SetFocus method.

See Also
DispField Statements, SetAddr Statement, SetDefaults Statement, SetProps Statement
DSLUpdate Control

ToolBox Icon

Description
The DSLUpdate control is used to define logical groups of information on the screen via its Levels property as well as expose key database and/or navigational events to the application such as NewLevel, Delete and Finish.

Remarks
All applications developed must have a DSLUpdate control contained by Form1.

Properties

<table>
<thead>
<tr>
<th>Customizable (D)</th>
<th>Left</th>
<th>Name (D)</th>
<th>Top</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index (D)</td>
<td>Levels (D)</td>
<td>Tag</td>
<td></td>
</tr>
</tbody>
</table>

(D) The property can be modified only at design time but not at runtime.

Events

- Cancel
- Finish
- Update
- Delete
- NewLevel

Methods

AboutBox

Use the ApplSetFocus statement instead of the SetFocus method.

See Also

Level Property, SqlCursor Statement, SetAddr Statement
Properties

Alignment Property

**Applies To**
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCheck, DSLOption

**Description**
Returns or sets a value that determines the alignment of a CheckBox or OptionButton control, text in a control.

**Remarks**
The object is an object expression that evaluates to an object in the Applies To list. The number is an integer that specifies the type of alignment.

For the **DSLCheck** and **DSLOption** controls, an Alignment number 0 indicates that the text is left aligned and the control is right aligned. An alignment value of 1 indicates that the text is right aligned and the control is left aligned. You can display text to the right or left of **DSLOption** and **DSLCheck** controls. By default, text is left aligned.

For the **DSLMaskedText** control, the default value of 0 indicates the text is left aligned; a value of 1 indicates the text is right aligned; a value of 2 indicates the text is centered.

To modify the value of the **Alignment** property at runtime, the **SetProps** statement should be used rather than modifying the property directly in Visual Basic code. Usage of **SetProps** allows the system to track changes to property values so as to avoid conflicts with customizations.

**See Also**
SetProps Statement

BackColor Property

**Applies To**
DSLDate, DSLFloat, DSLInteger, DSLCheck, DSLCombo, DSLGrid, DSLMaskedText, DSLOption

**Description**
Specifies the background color of an object.

**Remarks**
The BackColor of grid cell(s) can be modified at runtime by modifying the **BackColor** property of the particular cell’s corresponding form view control:

- To vary the BackColor on a row-by-row basis, make a **SetProps()** call on the form view control from within the **LineGotFocus()** event of the **DSLGrid**.
- To modify the BackColor of an entire column, make an **MSetProp()** call on the form view control. In the **MSetProp()** scenario, this call must be made before the **DSLGrid** receives focus (such as at screen load, or in a master key **Chk** event of a header level). Do not call **MSetProp()** from within **DSLGrid** events such as **LineGotFocus()**.

**See Also**
LineGotFocus Event, MSetProp Statement, SetProps Statement
BlankErr Property

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCheck, DSCombo, DSLOption

Description
Determines whether or not a valid value must be entered or defaulted for the field.

Remarks
A setting of True indicates that the field requires a value. A setting of False indicates the field is optional. Disabled and/or invisible fields should not be required unless they are automatically defaulted with a valid value. If a control is marked as required by the application then it cannot be marked as optional using the Customization Manager. However if a control is marked as optional by the application then it can be marked as required by the Customization Manager.

To modify the value of the BlankErr property at runtime, the SetProps statement should be used rather than modifying the property directly in Visual Basic code. Usage of SetProps allows the system to track changes to property values so as to avoid conflicts with customizations.

See Also
Enabled Property, MSetProp Statement, SetProps Statement, Visible Property

Caption Property

Applies To
DSLCheck, DSLOption

Description
Determines the text displayed next to the control.

Remarks
To modify the value of the Caption property at runtime, the SetProps statement should be used rather than modifying the property directly in Visual Basic code. Usage of SetProps allows the system to track changes to property values so as to avoid conflicts with customizations.

See Also
SetProps Statement
ColsFrozen Property

Applies To
DSLGrid Control

Description
Sets the number of left-most columns in an DSLGrid that remain visible when the DSLGrid scrolls horizontally.

Remarks
The value of ColsFrozen defaults to 0. The maximum number of columns that can be frozen is the total number of columns in the DSLGrid. If the value specified for ColsFrozen is greater than the number of columns in the DSLGrid, then all columns will be frozen. ColsFrozen does not affect vertical scrolling.
ColsFrozen refers to the ordinal column position from left to right. For example, assume an DSLGrid contains columns A, B, C, etc., and column Z is inserted before column A at runtime. If ColsFrozen is set to 1, column Z is frozen.
The value of ColsFrozen is modifiable at runtime.

Custom Property Page

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCheck, DSLCombo, DSLOption, DSLGrid

Description
An DSL Control may have one or more properties that have an associated properties page. For example, it is common for an DSL Control to have both a BackColor property and a FieldName property assigned. Both of these properties have an associated property page. The Custom Property Page presents all relevant property pages for the specified control in a single dialog box.

Customizable Property

Applies To
DSLUpdate

Description
Determines whether an application is customizable.

Remarks
This property allows an application to identify itself as “non-customizable.” It is a Boolean property. The default value is True, which means that the application is customizable. False indicates that the application cannot be customized.
**DBNav Property**

**Applies To**
DSLMaskedText, DSLGrid

**Description**
Used to facilitate navigation through all database records in the result set of an SQL statement.

**Remarks**
When the DBNav property of an DSLMaskedText control is used, it is conceptually equivalent to the PV property except that the PV window is not applicable. Some data entry screens contain multiple key fields. If the valid values for the last key field (for example, the control that navigation is performed on) are not restricted to values from some referential table, then the DBNav property should be used instead of the PV property. In this case, the application needs to explicitly fetch the record from within its Chk event using one of the DBNavFetch functions. For information about how the user experiences changes that you make to this property, see "Using Extended Possible Values Lists" in the Quick Reference Help or user’s guide.

For example, the master table for the Payroll Employee W2 History screen is the W2Federal table. The unique index for that table contains two fields, EmpId and CalYr, representing the Employee ID and Calendar Year respectively. Consequently, the application itself also has two key fields corresponding to the unique index. The first key field is EmpID and a PV property referencing the Employee table is implemented for that control. However the CalYr field is the last master key and therefore the W2Federal record is actually fetched within its Chk event. Since Payroll does not have a referential table containing all possible calendar year values, the Possible Values window is not applicable. Therefore that particular control utilizes a combination of a DBNav property and a corresponding DBNavFetch call within its Chk event.

The only other usage of the DBNav property is for the DSLGrid control. In this case, it is used to load the grid with all of the records contained within the result set of the SQL statement or stored procedure referenced in the DBNav property.

Spreadsheets are always associated with one of two different types of detail levels: Detail or Detail (Application Loaded). The actual level type in use for any particular grid is specified in the Levels property of the DSLUpdate control. For standard Detail levels, an SQL statement or stored procedure must be entered in the DBNav property of the corresponding DSLGrid control. However for Application Loaded Detail levels, a DBNav property value is optional depending on whether or not the DetailLoad statement will be used to load the corresponding DSLGrid. If DetailLoad is utilized then an SQL statement or stored procedure must be entered in the DBNav property of the corresponding DSLGrid control. However, if the application loads the Spreadsheet via the use of MInsert calls within an SqlFetch ... SFetch type of loop then a DBNav property value is not required. In this latter case, however, the application must assume full responsibility for the update since it is taking complete control of the load operation.

The DBNav property dialog contains the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Proc/Text</td>
<td>Used to enter the SQL statement or stored procedure name whose result set identifies the correct result set for the underlying application.</td>
</tr>
<tr>
<td>Constant</td>
<td>Used in conjunction with the Parm buttons to pass a constant value as a runtime parameter to the SQL statement or stored procedure.</td>
</tr>
<tr>
<td>Wildcard</td>
<td>Used in conjunction with the Parm buttons to identify a particular parameter as being able to support SQL wildcard values.</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Struct.FieldName</td>
<td>Used in conjunction with the Parm tabs to pass the value of other data items as a runtime parameters to the SQL statement or stored procedure. The Struct.FieldName value will normally be in the “bTableA.FieldName” format. The portion of the entry that identifies the table name MUST correspond precisely to the table name string passed in a corresponding call to the SetAddr statement. For example, assume the value of TableA.FieldA should be passed as a runtime parameter to the stored procedure. In this case, the Struct.FieldName would be “bTableA.FieldA”.</td>
</tr>
<tr>
<td>Field Offset Value</td>
<td>Optional depending on whether or not the table name referenced by the Struct.FieldName is actually the name of a table in the database. See the FieldName property for more information.</td>
</tr>
<tr>
<td>Declare Type</td>
<td>Optional depending on whether or not the table name referenced by the Struct.FieldName is actually the name of a table in the database. See the FieldName property for more information.</td>
</tr>
<tr>
<td>Length</td>
<td>Optional depending on whether or not the table name referenced by the Struct.FieldName is actually the name of a table in the database. See the FieldName property for more information.</td>
</tr>
<tr>
<td>Parm Tabs</td>
<td>Refresh the various data entry controls used to define parameters with the current values of the particular parameter corresponding to the selected parm. For example, to view the values for the second parameter, select the Parm2 tab.</td>
</tr>
</tbody>
</table>

Consider the following guidelines when implementing a DBNav property:

The SQL statement or stored procedure referenced within the SQL Proc/Text field of the DBNav property must always be capable of receiving at least one parameter. However, regardless of how many parameters are expected by the SQL statement or stored procedure, the last parameter must be capable of receiving wildcard values.

Consider the following examples:

**Example 1**

Select * from TableA
where TableA.StringField LIKE @parm1
Order By StringField

**Example 2**

Select * from TableA
where TableA.StringFieldA = @parm1
    and TableA.StringFieldB LIKE @parm2
Order By StringFieldA, StringFieldB

**Example 3**

Select * from TableA
where TableA.StringFieldA = @parm1
    and TableA.IntegerFieldB BETWEEN @parm2 and @parm3
Order By StringFieldA, IntegerFieldB

The first two examples use the LIKE statement for the last string field. Because the LIKE keyword does not apply to integer fields, the BETWEEN keyword is used for integers in the third example. When the system handles "wildcard values" for a single integer parameter it will automatically pass two parameters, the first having a value corresponding to INTMIN and the last having a value corresponding to INTMAX.

When implementing a DBNav property on an DSLMaskedText, the value of the underlying control is always passed to the SQL statement or stored procedure as the LAST parameter. Consequently the last parameter does not need to be manually defined using the Parm buttons. For example, if the stored procedure has only one parameter then no parameters need to be manually defined within the DBNav property. This is due to the fact that the value of the underlying control will automatically be passed as the last parameter, which happens to be the only parameter in this particular case.
When implementing a **DBNav** property on an **DSLGrid** control, all parameters must be defined using the Parm buttons. The only exception is for “wildcard” integers in which case one integer parameter is defined but is marked as a wildcard.

The restriction clause of the SQL statement or stored procedure must be capable of retrieving one unique record. For example, assume that the **DSLGrid** is loaded with ten existing records. Furthermore, assume that the user subsequently modifies only the third record. In this case, the **DBNav** SQL statement or stored procedure must be capable of targeting only that third record.

**See Also**

Chk Event, **DBNavFetch Functions**, **DetailLoad Statement**, **DetailSave Statement**, **FieldName Property**, **PV Property**, **SetAddr Statement**

**DecimalPlaces Property**

**Applies To**

**DSLFloat**

**Description**

Determines the number of digits displayed to the right of the decimal separator.

**Remarks**

Developers should use caution when setting the **DecimalPlaces** property. In particular, when writing applications which integrate with Microsoft Dynamics SL applications, the data will ultimately be processed by Microsoft Dynamics SL. Consequently, the data will be rounded. The exact precision will vary depending on the type of number involved. The point however, is merely to remind the developer that extended precision will only have worth if all subsequent uses of such numbers are rounded correspondingly.

Neither the application itself nor custom BSL code can change this property at runtime.

However, the kernel changes the **DecimalPlaces** property in order to facilitate Flexible Decimal Precision feature, such as Float controls associated with a FieldClass.

The **DecimalPlaces** property can be modified only at design time; it cannot be modified at runtime.

**See Also**

**Max Property**, **Min Property**

**Default Property**

**Applies To**

**DSLDate**, **DSLFloat**, **DSLInteger**, **DSLMaskedText**, **DSLCheck**, **DSLCombo**, **DSLOption**

**Description**

Determines the default value for the underlying data field.

**Remarks**

The default data value for any particular data entry control can be specified via either the **Default** property or the **Default** event.

The **Default** property can be used when the default value is not contingent upon any the value of any other data item. However, if the methodology for determining a default value varies depending particular situations, then code should be written for the **Default** event. If a **Default** property is defined then the **Default** event will not be used.
The **Default** property value for an **DSLOption** button group is always derived from the first option button in the group, since by definition only one option button can be selected within any particular group.

The **Default** property dialog contains the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>Used to specify a constant data value for the underlying control. For example, if <strong>DSLCheck</strong> should always default to True, then a value of 1 would be entered (assuming that is the value of its TrueText property)</td>
</tr>
<tr>
<td>Previous Value</td>
<td>Applies only to controls related to a grid. This allows the user to enter a value once and the same value will automatically default on subsequently entered detail lines. For example, a transaction date may have a default <strong>Struct.FieldName</strong> of “bPes.Today” and Previous Value checked. This will cause the date field to initially default to the current business date. However, if the user enters a different date then the new date will default to all subsequently entered detail lines since Previous Value is checked.</td>
</tr>
<tr>
<td><strong>Struct.FieldName</strong></td>
<td>Used in cases where the value of the field should always default to the value of some other data item. This value will normally be in the “bTableName.FieldName” format. The portion of the entry that identifies the table name MUST correspond precisely to the table name string passed in a corresponding call to the <strong>SetAddr</strong> statement. For example, assume the value of TableB.FieldB should default to the value of TableA.FieldA. In this case, the <strong>Default</strong> property for the TableB.FieldB control should have a <strong>Struct.FieldName</strong> of “bTableA.FieldA”.</td>
</tr>
<tr>
<td>Field Offset Value</td>
<td>Optional depending on whether or not the table name referenced by the <strong>Struct.FieldName</strong> is actually the name of a table in the database. See the <strong>FieldName</strong> property for more information.</td>
</tr>
<tr>
<td>Declare Type</td>
<td>Optional depending on whether or not the table name referenced by the <strong>Struct.FieldName</strong> is actually the name of a table in the database. See the <strong>FieldName</strong> property for more information.</td>
</tr>
<tr>
<td>Length</td>
<td>Optional depending on whether or not the table name referenced by the <strong>Struct.FieldName</strong> is actually the name of a table in the database. See the <strong>FieldName</strong> property for more information.</td>
</tr>
</tbody>
</table>

**See Also**

Default Event, **FieldName** Property, Level_SetDefaults Statement, SetDefaults Statement, Trigger Property

**DragIcon Property**

**Applies To**

**DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLGrid, DSLCheck, DSLCombo, DSLOption**

**Description**

Reserved for Microsoft and should not be changed at design or runtime.

**DragMode Property**

**Applies To**

**DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLGrid, DSLCombo, DSLOption**
Description
Reserved for Microsoft and should not be changed at design or runtime.

Enabled Property

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCheck, DSLCombo, DSLOption

Description
Determines whether or not the user can modify the contents of the control.

Remarks
A value of True indicates that the control is enabled whereas a value of False causes the control to be disabled. Required fields should not be disabled unless they are automatically defaulted with a valid value.

To modify the value of the Enabled property at runtime, the SetProps statement should be used rather than modifying the property directly in Visual Basic code. Usage of SetProps allows the system to track changes to property values so as to avoid conflicts with customizations or other API’s such as the DisplayMode statement.

If a control is disabled by the application then it cannot be enabled using the Customization Manager. However if a control is enabled by the application then it can be disabled by the Customization Manager. The value of the Enabled property can be narrowed at runtime but not expanded, depending on the level at which the changes are being made (All User, One User, etc.). See the “Security” section of the Customization Manager documentation. The Enabled property should not be used on the DSLGrid control.

See Also
BlankErr Property, DisplayMode Statement, SetProps Statement, Visible Property

FalseText Property

Applies To
DSLCheck

Description
Determines the value of the underlying data field whenever the control is not checked.

Remarks
Normally the FalseText value for integer true/false type of fields is 0. This allows the value to more easily be tested for a false value within SQL. It also corresponds to the value of the LFALSE symbolic constant declared in Solomon.VBTools.vb. The FalseText property can be set only at design time.

See Also
TrueText Property

FieldClass Property

Applies To
DSLFloat, DSLMaskedText
Description
Associates a control with a particular class of data items having global display and/or operational characteristics.

Remarks
Within Microsoft Dynamics SL, some classes of fields have unique characteristics across the entire product line.

For example, Subaccounts can have a user-defined mask based on the segmentation defined on Flexkey Definition (21.320) in Shared Information. All controls whose underlying data field is a Subaccount have a FieldClass property value identifying them as such. When the screen containing the Subaccount field is actually displayed, the system will have automatically applied a custom mask corresponding to the globally defined Subaccount segmentation rules.

The valid FieldClass values are embedded inside of the control and are viewable by the developer via a combo box in the Visual Basic Property window. The user cannot change the FieldClass property at runtime, but can only change it at design-time via the Property window, which will explicitly present a valid list of possible values.

The following table contains all of the valid values for the FieldClass property. Each fieldclass is either of two types: Flex Key and Non-Flex Key.

<table>
<thead>
<tr>
<th>Flex Key Type Fieldclasses</th>
<th>Non-Flex Key Type Fieldclasses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subaccount Number (1)</td>
<td>Account Number (101)</td>
</tr>
<tr>
<td>Inventory Item (2)</td>
<td>Cost Type (102)</td>
</tr>
<tr>
<td>Customer ID (3)</td>
<td>Country/Region (103)</td>
</tr>
<tr>
<td>Vendor ID (4)</td>
<td>Customer Class (104)</td>
</tr>
<tr>
<td>Employee ID (109)</td>
<td>Name (Enterable) (105)</td>
</tr>
<tr>
<td></td>
<td>Deduction ID (106)</td>
</tr>
<tr>
<td></td>
<td>Depreciation ID (107)</td>
</tr>
<tr>
<td></td>
<td>Earnings Type ID (108)</td>
</tr>
<tr>
<td></td>
<td>Name (Display Only) (110)</td>
</tr>
<tr>
<td></td>
<td>Project ID (111)</td>
</tr>
<tr>
<td></td>
<td>Kit ID (112)</td>
</tr>
<tr>
<td></td>
<td>Pay Group (113)</td>
</tr>
<tr>
<td></td>
<td>Period Number (114)</td>
</tr>
<tr>
<td></td>
<td>Project Task ID (115)</td>
</tr>
<tr>
<td></td>
<td>Product Class ID (116)</td>
</tr>
<tr>
<td></td>
<td>Salesperson ID (117)</td>
</tr>
<tr>
<td></td>
<td>Site ID (118)</td>
</tr>
<tr>
<td></td>
<td>State ID (119)</td>
</tr>
<tr>
<td></td>
<td>Statement Cycle ID (120)</td>
</tr>
<tr>
<td></td>
<td>Terms ID (121)</td>
</tr>
<tr>
<td></td>
<td>Work Location ID (123)</td>
</tr>
<tr>
<td></td>
<td>Transaction Amount (124)</td>
</tr>
<tr>
<td></td>
<td>Item Quantity (125)</td>
</tr>
<tr>
<td></td>
<td>Item Price (126)</td>
</tr>
</tbody>
</table>

Flex Key Fieldclasses
The flex key fieldclasses follow, in numeric order, as shown in the previous table.
Sub Account (1)

Developer Benefits

- Inquiry on individual segments is available using CTRL+F3.
- Formatted entry and display of data, based on its segmented definition.

*Microsoft Dynamics SL Business Essentials*

- The number of segments is limited to 3.
- The total length of all segments combined can be up to 6 characters.

*Microsoft Dynamics SL Advanced Management*

- The number of segments and combined length is determined by the information entered in Shared Information on Flex Definition (21.320.00).
- The number of segments can be up to 8; the total length of all segments can be up to 24 characters.

*Application Requirements*

The control’s fieldclass property must be set to 1.

Inventory Item (2)

Developer Benefits

- Inquiry on individual segments is available by pressing CTRL+F3.
- Formatted entry and display of data, based on its segmented definition.

*Microsoft Dynamics SL Business Essentials*

- The number of segments is limited to 1
- The total length of all segments combined can be up to 30 characters.

*Microsoft Dynamics SL Advanced Management*

- The number of segments and combined length is determined by the information entered in Shared Information on Flex Definition (21.320.00).
- The number of segments can be up to 4; the total length of all segments can be up to 30 characters.

*Application Requirements*

The control’s fieldclass property should be set to 2.

Customer ID (3)

Developer Benefits

- Inquiry on individual segments is available by pressing CTRL+F3.
- Formatted entry and display of data, based on its segmented definition.

*Microsoft Dynamics SL Business Essentials*

- The number of segments is limited to 1.
- The total length of all segments combined is 15.

*Microsoft Dynamics SL Advanced Management*

- The number of segments and combined length is determined by the information entered in Shared Information on Flex Definition (21.320.00).
- The number of segments can be up to 4; the total length of all segments can be up to 15 characters.

*Application Requirements*

The control’s fieldclass property must be set to 3.
Vendor ID (4)

Developer Benefits
- Inquiry on individual segments is available by pressing CTRL+F3.
- Formatted entry and display of data, based on its segmented definition.

Microsoft Dynamics SL Business Essentials
- The number of segments is limited to 1.
- The total length of all segments combined is 15.

Microsoft Dynamics SL Advanced Management
- The number of segments and combined length are determined by the information entered in Shared Information on Flex Definition (21.320.00).
- The number of segments can be up to 4; the total length of all segments can be up to 15 characters.

Application Requirements
The control’s fieldclass property must be set to 4.

Employee ID (109)

Developer Benefits
- Inquiry on individual segments is available by pressing CTRL+F3.
- Formatted entry and display of data, based on its segmented definition.

Microsoft Dynamics SL Business Essentials
- The number of segments is limited to 1.
- The total length of all segments combined is 10.

Microsoft Dynamics SL Advanced Management
- The number of segments and combined length is determined by the information entered in Shared Information on Flex Definition (21.320.00).
- The number of segments can be up to 4; the total length of all segments can be up to 10 characters.

Application Requirements
The control’s fieldclass property must be set to 109.

Non-Flex Key Fieldclasses
The non-flex key fieldclasses follow, in numeric order, as shown in the previous table.

Note: Several of the fieldclasses provide no specific benefits at this time. They are provided for information purposes and may be used, or not.

Account Number (101)

Developer Benefits
When error checking occurs on this field, the kernel will display “System Message 786” if the account selected by the user is not an active account.

Application Requirements
- The control must have the PVNav property set, and the stored procedure specified there must return a structure exactly like the Account table as it is defined in an application database.
- The control’s fieldclass property needs to be set to 101.
Cost Type (102)
Developer Benefits
• “~” stored in the DB are converted to “?” for display on the screen.
• “?” entered on the screen are converted to “~” in the database.
Application Requirements
The control’s fieldclass property needs to be set to 102.

Country/Region (103)
Developer Benefits
None.
Application Requirements
The control’s fieldclass property needs to be set to 103.

Customer Class (104)
Developer Benefits
None.
Application Requirements
The control’s fieldclass property needs to be set to 104.

Name (Enterable) (105)
Developer Benefits
• Used only for controls that are enterable.
• Allows entry of a person’s name with the @ symbol placed in front of the portion of the name on
  which to sort; for example, John@Doe entered in the control is stored in the database as
  Doe~John to allow sorting by Last Name.
• Used in conjunction with controls that have a fieldclass of 110.
Application Requirements
The control’s fieldclass property needs to be set to 105.

Deduction ID (106)
Developer Benefits
None.
Application Requirements
The control’s fieldclass property needs to be set to 106.

Depreciation ID (107)
Developer Benefits
None.
Application Requirements
The control’s fieldclass property needs to be set to 107.

Earnings Type (108)
Developer Benefits
None.
Application Requirements
The control’s fieldclass property needs to be set to 108.
Name (Display Only) (110)

Developer Benefits
- `<Last name>`~`<First Name>` stored in the DB are converted to `<First name> <LastName>` displayed on the screen and in inquiry lists.
- Allows values entered using controls with a fieldclass of 105 to be displayed without any of the special characters.

Application Requirements
The control’s fieldclass property needs to be set to 110.

Project ID (111)

Developer Benefits
- “~” stored in the DB are converted to “?” for display on the screen.
- “?” entered on the screen are converted to “~” in the database.

Application Requirements
The control’s fieldclass property needs to be set to 111.

Kit ID (112)

Developer Benefits
None.

Application Requirements
The control’s fieldclass property needs to be set to 112.

Pay Group ID (113)

Developer Benefits
None.

Application Requirements
The control’s fieldclass property needs to be set to 113.

Period Number (114)

Developer Benefits
- The Relative Period feature is available to this control.
- An automatic mask of 99-9999 for displaying of periods.
- Additional information is written to the TI intelligent control macro about control with this fieldclass.
- Several automatic checks are done at check event time.
- The Period is checked against the GL and message 6071 is issued if the period entered is less than the XXXXX entered in GL Setup.
- Message 23 is displayed if the period entered differs from the modules current period. The module is based on the screen number which is pulled from the form's Text property. Payroll, Accounts Payable, Accounts Receivable, Project Controller, Inventory, Cash Manager, Work Order, Bank Reconciliation, and Bill of Material have their own periods. All other modules, including unrecognized/custom modules, use the General Ledger period. There is no support for custom module periods in the period field class.
- Message 22 is displayed if the period is not valid (less than 1, or greater than the defined number of fiscal periods as defined in GL Setup).

Application Requirements
The control’s fieldclass property needs to be set to 114.
Project Task ID (115)
Developer Benefits
- Inquiry on individual segments is available by pressing CTRL+F3.
- “~” stored in the database are converted to “?” for display on the screen.
- “?” entered on the screen are converted to “~” in the database.
- Formatted entry and display of data, based on its segmented definition.

*Microsoft Dynamics SL Business Essentials*
- The number of segments is limited to 1.
- The total length of all segments combined is 10.

*Microsoft Dynamics SL Advanced Management*
The number of segments and combined length is determined by the information entered in Shared Information on Flex Definition (21.320.00).

Application Requirements
- The control’s fieldclass property needs to be set to 115.
- Non-segment defined field classes.

Product Class ID (116)
Developer Benefits
None.

Application Requirements
The control’s fieldclass property needs to be set to 116.

Salesperson ID (117)
Developer Benefits
None.

Application Requirements
The control’s fieldclass property needs to be set to 117.

Site ID (118)
Developer Benefits
None.

Application Requirements
The control’s fieldclass property needs to be set to 118.

State (119)
Developer Benefits
None.

Application Requirements
The control’s fieldclass property needs to be set to 119.

Statement Cycle ID (120)
Developer Benefits
None.

Application Requirements
The control’s fieldclass property needs to be set to 120.
Terms ID (121)
Developer Benefits
None.
Application Requirements
The control’s fieldclass property needs to be set to 121.

Work Location ID (123)
Developer Benefits
None.
Application Requirements
The control’s fieldclass property needs to be set to 123.

Transaction Amount (124)
Developer Benefits
Flexible decimal precision, based on number of decimals for the base currency or the transaction currency.
Application Requirements
The control’s fieldclass property needs to be set to 124.

Item Quantity (125)
Developer Benefits
Forces decimal precision of 2 decimal places.
Application Requirements
The control’s fieldclass property needs to be set to 125.

Item Price (126)
Developer Benefits
Flexible decimal precision, based on the following setting in the Solomon.ini file (0-99):

```
[Miscellaneous]
UnitPriceOverride=2
```
Application Requirements
The control’s fieldclass property needs to be set to 126.

FieldName Property

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCheck, DSLCombo, DSLOption

Description
Facilitates proper runtime binding between the control and its underlying Visual Basic data variable by operating in conjunction with the SetAddr statement.

Remarks
The data for each individual data entry control is actually stored in an underlying Visual Basic variable. At runtime the control and its associated Visual Basic storage variable are bound together using a combination of the FieldName property of the control and a corresponding call to the SetAddr statement from within Form1_Load.
The **FieldName** property contains a Struct.FieldName value along with other more detailed information such as Field Offset Value, Declare Type and Length. At a minimum, a value must be entered into the Struct.FieldName field. This value will normally be in the “bTableName.FieldName” format. The portion of the entry that identifies the table name MUST correspond precisely to the table name string literal which is passed in a corresponding call to the `SetAddr` statement. It is not, however, required to correspond to the name of an actual table within the database.

The Field Offset Value, Declare Type and Length fields are optional depending on whether or not the table name referenced by the Struct.FieldName is actually the name of a table in the database. If the Struct.FieldName does reference a database table then SWIM can access detailed information relating to each individual field contained therein using the SQL data dictionary. If the referenced “table name” does not correspond to the name of a table in the database then values MUST be entered in the Field Offset Value, Declare Type and Length fields.

The FieldName property can be modified at design time only; it cannot be modified at runtime.

The following table contains the Declare Type and Length of several standard datatypes.

<table>
<thead>
<tr>
<th>SQL Datatype</th>
<th>Visual Basic Datatype</th>
<th>Declare Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>String</td>
<td>0</td>
<td>Length of String</td>
</tr>
<tr>
<td>Integer(2)</td>
<td>Integer</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Float</td>
<td>Double</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Date</td>
<td>Integer</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Logical</td>
<td>Integer</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>

**See Also**

SetAddr Statement

**Font Property**

**Applies To**

DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLGrid, DSLCheck, DSLCombo, DSLOption

**Description**

Used to identify a specific Font for an object.

**Remarks**

The Font of grid cell(s) can be modified at runtime by modifying the Font property of the particular cell’s corresponding form view control:

- To vary the Font on a **row-by-row** basis, make a `SetProps()` call on the form view control from within the LineGotFocus event of the **DSLGrid**.
- To modify the Font of an entire **column**, make an `MSetProp()` call on the form view control. In the `MSetProp()` scenario, this call must be made before the **DSLGrid** receives focus (such as at screen load, or in a master key **Chk** event of a header level). Do not call `MSetProp()` from within **DSLGrid** events such as **LineGotFocus()**.

**See Also**

Chk Event, LineGotFocus Event, MSetProp Statement, SetProps Statement

**ForeColor Property**

**Applies To**

DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCheck, DSLCombo, DSLOption, DSLGrid
Description
Specifies the foreground color used to display data in an object.

Remarks
The Foreground Color of grid cell(s) can be modified at runtime by modifying the ForeColor property of the particular cell's corresponding form view control:

- To vary the Foreground Color on a row-by-row basis, make a SetProps() call on the form view control from within the LineGotFocus() event of the DSLGrid.
- To modify the ForeColor of an entire column, make an MSetProp() call on the form view control. In the MSetProp() scenario, make this call before the DSLGrid receives focus (such as at screen load, or in a master key Chk event of a header level). Do not call MSetProp() from within DSLGrid events such as LineGotFocus().

See Also
LineGotFocus Event, MsetProp Statement, SetProps Statement

Heading Property

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCheck, DSLCombo, DSLOption

Description
Contains the caption for the corresponding grid column for controls actually associated with a DSLGrid.

Remarks
Column headings containing more than one line can be implemented by separating the text for each line with a comma such as “Line One, Line Two”.

To modify the value of the Heading property at runtime, the SetProps statement should be used rather than modifying the property directly in Visual Basic code. Usage of SetProps allows the system to track changes to property values so as to avoid conflicts with customizations.

If an option button group appears within a grid, the grid uses the Heading property of the first option button as the column header.

See Also
MSetProp Statement, SetProps Statement

Height Property

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCheck, DSLCombo, DSLOption, DSLGrid

Description
Determines the dimensions of the control.

Remarks
To avoid potential conflicts with customizations applied at runtime using the Customization Manager, do not modify the dimensions of a control at runtime.

The Height property of the DSLGrid control can be modified only at design time.
This property exists at both design-time and runtime. When you put an DSLGrid on a form at design-time, it will have values for the sizing properties (Left, Top, Height, Width). You can modify these values to optimize the size of the grid for design-time viewing. You can size the grid quite small and put it in the lower-right corner of the frame in which the DSLGrid resides to make it easier to see and/or manipulate all of the form view controls to which the DSLGrid will be bound.

At runtime however, do not modify this property since the kernel will seek to take total responsibility over the sizing of the DSLGrid. For example, the kernel will attempt to make the DSLGrid match the size of the underlying frame. The kernel will also resize the DSLGrid when the user resizes the form.

See Also
Left Property, Top Property, Width Property

HelpContextID Property

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCheck, DSLCombo, DSLOption, DSLUpdate

Description
Specifies an associated context number for an object.

Remarks
The HelpContextID property can be used to provide context-sensitive help for an application. You must assign the same context number to both object and to the associated help topic when you compile your help file. When a user presses the F1 key, Visual Basic automatically calls help and searches for the topic identified by the HelpContextID for the object that has the focus. If HelpContextID is set to 0, then Visual Basic looks in the HelpContextID of the object’s container, and then that object’s container, and so on. If a non-zero current context number cannot be found, the F1 key is ignored.

Do not use the HelpContextID property with the DSLGrid control. Grid view help is driven by properties on the form view controls. If the current column’s form view control has a help context, that value will be used. Otherwise, the help context of the form view container (for example, the frame) will be used.

See Also
WhatsThisHelpID Property

InGrid Property

Applies To
DSLDate, DSLFloat, DSLInteger, DSLCheck, DSLCombo, DSLMaskedText, DSLOption

Description
Determines whether or not the control is visible in grid view.

Remarks
A value of True indicates that the control is visible in grid view as a column. Otherwise it is not visible as a column.

This property should be set at the time of design, or modified through the Customization Manager. The SetProperty statement can be used to modify it at the runtime, but it will be meaningless if used after the grid is already configured. This property can also be changed using the Customization Manager.
See Also
SetProps Statement, Visible Property

Left Property

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCheck, DSLCombo, DSLOption, DSLGrid, DSLUpdate

Description
Determines the distance between the left edge of a control and the left edge of its container object.

Remarks
To avoid potential conflicts with customizations applied at runtime using the Customization Manager, do not modify the position of a control at runtime. The Left property of the DSLGrid control can be modified only at design time.

This property exists at both design-time as well as runtime. When you put an DSLGrid on a form at design-time, it will have values for the sizing properties (Left, Top, Height, Width). You can modify these values to optimize the size of the grid for design-time viewing. You can size the grid quite small and put it in the lower-right corner of the frame in which the DSLGrid resides to make it easier to see and/or manipulate all of the form view controls to which the DSLGrid will be bound.

At runtime however, do not modify this property since the kernel will seek to take total responsibility for the sizing of the DSLGrid. For example, the kernel will attempt to make the DSLGrid match the size of the underlying frame. The kernel will also resize the DSLGrid when the user resizes the form.

See Also
Height Property, Top Property, Width Property

Level Property

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCheck, DSLCombo, DSLOption

Description
Associates the control with a logical group of information contained within the application.

Remarks
The Level property is related to, but not the same as, the Levels property of the DSLUpdate control. The Levels property of the DSLUpdate control is used to define all levels for a particular screen. Each data entry control must subsequently be associated with one of these logical levels using its Level property. The Level property can be modified at design time only; it cannot be modified at runtime.

For example, assume the Levels property of the DSLUpdate control contains the following value:

```
TableA;N,TableB;D
```

In this example, TableA is the master table for first level so all controls displaying information from TableA would have a value of 0 in their Level property. Level numbers themselves always begin with zero and count upwards. Consequently all controls displaying information from TableB would have a value of 1 in their Level property.

If a particular control is one of the key fields for a particular level then a “,k” should be appended to the level number. For example, the control associated with TableA.KeyField would have a value of 0,k in its Level property.
The last key field on Navigation and Lookup levels must have either its PV or DBNav property initialized as well as a corresponding PVChkFetch or DBNavFetch call in its Chk event.

See Also
DBNav Property, Levels Property, PV Property, Chk Event, DBNavFetch Functions, PVChkFetch Functions, Level_SetDefaults Statement

Levels Property

Applies To
DSLUpdate

Description
Defines an alias name and level type for all logical groups of information on the application’s screen.

Remarks
A level is defined as a set of fields from one or more tables which form a logical group. In many cases, a one to one relationship exists between the number of levels and the number of tables whose information is displayed on the screen. For example, if a particular screen displays information from TableA, TableB and TableC then it is likely that the screen will have three levels. This simply means that are three logical groups of information being displayed on the screen.

The Levels property of the DSLUpdate control is used to define all levels for a particular screen. It should not be confused with the Level property of individual data entry controls. Each data entry control can only be associated with one of these logical levels using its Level property.

Each level within a particular screen has a corresponding name and level type. The Levels property should be a string expression in the following basic format:

AliasName0;LevelType0,AliasName1;LevelType1...,AliasName9;LevelType9

The AliasName name associated with any particular level will appear in places such as the Insert/Delete dialog, Customization Manager, Templates and Copy Special (for example, cut/copy/paste). Consequently, the AliasName should be a logical name that will have meaning to users of the application.

The LevelType defines the navigational and display characteristics of a particular level. The following table contains a list of valid level types:

<table>
<thead>
<tr>
<th>Level Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Navigation Level. Used in cases where one record at a time can be viewed such as in Customer Maintenance or Employee Maintenance. Subsequent levels inherit actions performed on a Navigation level such as New and Delete.</td>
</tr>
<tr>
<td>L</td>
<td>Lookup Level. When utilized, this type of level must be preceded by a Navigation Level. Every time the last master key of the first preceding Navigation Level is modified, the Lookup Level will cause the system to automatically retrieve a single record from a related table. Typically this type of level is used to lookup a corresponding name or description every time a new ID, such as Customer ID or Employee ID, is entered on a preceding Navigation Level.</td>
</tr>
<tr>
<td>D</td>
<td>Detail Level. Used to implement a standard DSLGrid control.</td>
</tr>
<tr>
<td>DA</td>
<td>Detail (Application Loaded) Level. Similar to the standard Detail Level except that the application loads the corresponding grid. If the DetailLoad statement is used to load the grid then SWIM will automatically handle the update operations. Otherwise the application must take responsibility for database updates for the relevant level as well as loading the grid.</td>
</tr>
<tr>
<td>Level Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>C</td>
<td>Constant Level. Used in cases where user entered key field(s) and/or database navigation are not applicable since only one record is “constantly” displayed on the screen. Typically used on Setup screens where only one record could possibly be displayed on the screen. Also used on process screens having a few data entry controls related to processing options.</td>
</tr>
</tbody>
</table>

The following table conceptually illustrates Levels property values for common database relationships:

<table>
<thead>
<tr>
<th>Table Names</th>
<th>Database Relationship</th>
<th>Levels Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>TableA</td>
<td>Header</td>
<td>TableAAlias;N</td>
</tr>
<tr>
<td>TableA</td>
<td>Detail</td>
<td>TableAAlias;D</td>
</tr>
<tr>
<td>TableA, TableB</td>
<td>Header, Detail</td>
<td>TableAAlias;N,TableBAlias;D</td>
</tr>
<tr>
<td>TableA, TableB, TableC</td>
<td>Header, Related description for Header, Detail</td>
<td>TableAAlias;N,TableBAlias;L,TableCAlias;D</td>
</tr>
<tr>
<td>TableA</td>
<td>Always only one record in the entire table.</td>
<td>TableAAlias;C</td>
</tr>
</tbody>
</table>

The following restrictions apply to the Levels property:

- A maximum of two Navigation Levels can be defined for any one application.
- A Navigation Level cannot follow a Detail Level.
- A Lookup Level, if implemented, must be preceded by a Navigation Level.

The **Levels** property can be modified at design time only; it cannot be modified at runtime.

**See Also**

Level Property, DetailLoad Statement, Level_SetDefaults Statement, SetButton Statement, SetLevelChg Statement, SqlCursor Statement, TestLevelChg Function

**List Property**

**Applies To**

DSLCombo

**Description**

Determines the fixed list of valid data values for the underlying field along with corresponding descriptions.

**Syntax**

Value;Description [, Value;Description]...

**Remarks**

Combo Box controls are implemented for fields having a fixed list of valid choices. As a user views a Combo Box control, various descriptive choices are shown which should normally have meaning to any user of the application. However, each description that the user views is actually associated with a corresponding data value. This data value is normally a single character. When the user selects ChoiceA, the value of the underlying data field is set to the data value which as been associated with ChoiceA.

The **List** property allows the developer to enter all valid data values along with corresponding viewable descriptions into a single property.

The **List** property can be modified at design time only; it cannot be modified at runtime.
Mask Property

Applies To
DSLMaskedText

Description
Determines the type and number of characters that can be entered for a particular field.

Remarks
Each character in the Mask property corresponds to one character in the displayed field. If a particular mask character is one of the supported mask types, then the corresponding valid values will be permitted for that particular edit position. Otherwise, the character is considered to be a string literal to be displayed within the field. These string literals are display only, causing the cursor to automatically “jump” over them during data entry. Furthermore, the string literals will not be stored in the resulting value of the underlying data field since they are only for display purposes.

To modify the value of the Mask property at runtime, the SetProps statement should be used, rather than modifying the property directly in Visual Basic code. Usage of SetProps allows the system to track changes to property values, avoiding conflicts with customizations.

The value of the Mask property can be narrowed at runtime but not expanded, depending on the level at which the changes are being made (All User, One User, etc.). See the “Security” section of the Customization Manager documentation.

The following table lists the supported mask types and their corresponding definition:

<table>
<thead>
<tr>
<th>Mask Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Numeric (0-9)</td>
</tr>
<tr>
<td>A</td>
<td>Alphabetic (A-Z, a-z)</td>
</tr>
<tr>
<td>V</td>
<td>Alphabetic which is converted to upper case</td>
</tr>
<tr>
<td>N</td>
<td>Alphanumeric (A-Z, a-z, 0-9)</td>
</tr>
<tr>
<td>Q</td>
<td>Alphanumeric and ? (A-Z, a-z, 0-9, ?)</td>
</tr>
<tr>
<td>W</td>
<td>Alphanumeric which is converted to upper case</td>
</tr>
<tr>
<td>X</td>
<td>ASCII 32-127 (space, letters, numbers and special characters except for * and ?)</td>
</tr>
<tr>
<td>L</td>
<td>ASCII which is converted to lower case</td>
</tr>
<tr>
<td>U</td>
<td>ASCII which is converted to upper case</td>
</tr>
<tr>
<td>M</td>
<td>Mask ASCII (Same as X but includes * and ?). Be careful about using in key fields where usage of * and ? wildcard characters could affect usage of the LIKE keyword in SQL statements</td>
</tr>
<tr>
<td>H</td>
<td>Hexadecimal (0-9, A-F)</td>
</tr>
</tbody>
</table>

See Also
SetProps Statement
Max Property

Applies To
DSLDate, DSLFloat, DSLInteger

Description
Determines the maximum valid value for the control.

Remarks
To modify the value of the Max property at runtime, the **SetProps** statement should be used rather than modifying the property directly in Visual Basic code. Usage of **SetProps** allows the system to track changes to property values so as to avoid conflicts with customizations.

The value of the Max property can be customized to a lower value using the Customization Manager. However, the Max property cannot be customized to a higher value. See the “Security” section of the Customization Manager documentation.

See Also
DecimalPlaces Property, Min Property, SetProps Statement

Min Property

Applies To
DSLDate, DSLFloat, DSLInteger

Description
Determines the minimum valid value for the control.

Remarks
To modify the value of the Min property at runtime, the **SetProps** statement should be used rather than modifying the property directly in Visual Basic code. Usage of **SetProps** allows the system to track changes to property values so as to avoid conflicts with customizations.

The value of the Min property can be customized to a higher value using the Customization Manager. However, the Min property cannot be customized to a lower value. See the “Security” section of the Customization Manager documentation.

See Also
DecimalPlaces Property, Max Property, SetProps Statement
Mouselcon Property

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCheck, DSLCombo, DSLOption

Description
Provides a custom icon that is used when the MousePointer property is set to 99.

Remarks
Use the Mouselcon property to load either cursor or icon files. Color cursor files such as those shipped with Windows NT, are displayed in black and white. To display a color cursor, use a color icon file (.ico).

The Mouselcon property provides access to custom cursors of any size, with any desired hot spot location. Visual Basic does not load animated cursor (.ani) files, even though 32-bit versions of Windows support these cursors.

This property can be modified at design time only; it cannot be modified at runtime.

See Also
MousePointer Property

MousePointer Property

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCheck, DSLCombo, DSLOption

Description
Returns or sets a value indicating the type of mouse pointer displayed when the mouse is over a particular part of an object at runtime.

Remarks
The object is a control. The value An integer specifying the type of mouse pointer displayed, as described in the Microsoft Visual Basic help. Summarized, a value of 0 causes the pointer type to be determined by the object it is over; a value of 1 is the standard mouse arrow; a value of 99 causes the custom icon specified by the Mouselcon property to be used as the mouse pointer.

This property is useful when you want to indicate changes in functionality as the mouse pointer passes over controls on a form or dialog box.

This property can be modified at design time only; it cannot be modified at runtime.

See Also
Mouselcon Property
Name Property

 Applies To
 DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLGrid, DSLCheck, DSLCombo, DSLOption, DSLUpdate

 Description
 Specifies the name used to reference the control from within code.

 Remarks
 The Customization Manager requires that all control names within a given application be unique. This is because Customization Manager saves all changes made to a particular control to the database, using the control name as a type of internal key. If an object name is duplicated, Customization Manager will be not be able to correctly resolve references to the corresponding controls.

 It is also important to note that control names should not be modified after release of the application. Customization Manager will not be able to applycustomizations referencing the “old” control name. This property can be modified at design time only; it cannot be modified at runtime.

 NoteButton Property

 Applies To
 DSLMaskedText, DSLFloat, DSLInteger, DSLDate, DSLCombo, DSLGrid

 Description
 Returns or sets a value that determines the position of the Notes/Attachments button for the control.

 Remarks
 For the DSLGrid control, this property allows the developer to specify where the notes/attachments button for the grid should appear. Possible values are:

 • 1 – bottom left (this is the default value)
 • 2 – bottom right
 • 3 – top left
 • 4 – top right
 • 5 – none

 For the DSLMaskedText, DSLFloat, DSLInteger, DSLDate, and DSLCombo controls, the possible values are:

 • 1 – left
 • 2 – right
 • 3 – none (this is the default value)

 If the value for the NoteButton property is set at the default for all controls on a level, a notes/attachments button will be placed to the right of the first key on the level, which is where notes/attachments buttons have traditionally been placed.

 If the NoteButton property is set for two controls on a level, the notes/attachments button will be placed next to the first control in tab order. Microsoft Dynamics SL SDK does not support more than one notes/attachments button per level.

 The NoteButton property can also be set for controls in a detail level. In this case, the notes/attachments button will appear next to the control with the NoteButton property set when the screen appears in form view.
See Also
NoteColumn Property, HideNoteButtons Statement

NoteColumn Property

Applies To
DSLGrid

Description
Returns or sets a value that determines if the Note column is or is not displayed in the grid.

Remarks
This property allows the developer to specify that the first column in the grid should be a note column. If a note exists for a detail line and this column is displayed, a note icon will appear in the detail line. If the user wants to edit or create a note for a detail line, they can open the note dialog by double-clicking in the note field for the particular detail line. This column is only visible in grid view. When in form view, the user will have to use the note button associated with the grid to display the note dialog.

See Also
NoteButton Property, HideNoteButtons Statement

PV Property

Applies To
DSLMaskedText

Description
Determines all possible values currently existing in the database.

Remarks
The PV property is conceptually similar to the List property of an DSLCombo control in that it identifies possible data values for the relevant control. However, the PV property is much more sophisticated in that these possible data values are actually defined as existing data items contained within the database as opposed to a hard-coded list.

For example, many screens throughout Microsoft Dynamics SL require the entry of a valid account number. Conceptually speaking, an account number is "valid" if a corresponding record can be located within the Account database table. Thus, if we wanted to determine if account number 001000 is a valid account number, we could do so by issuing the following SQL Select statement:

```
Select * From Account
Where Acct LIKE '001000'
Order by Acct
```

If no record is returned by this simple SQL statement then 001000 is not an existing account number and therefore would not be valid.

The PV property facilitates the determination of all possible values currently existing in the database by allowing the developer to specify a relevant SQL statement or stored procedure name.

In cases where the underlying application merely needs to insure that the key (for example, ID) of an existing data item is entered for a particular control, the relevant control simply needs to contain a PV property. No code is necessary in this case since the application does not need to manipulate the data record itself -- it only needs to make sure that such a record does in fact exist.

However, in cases where the application does need to manipulate the record identified via the use of the PV property, one additional step is required. In particular the application needs to explicitly fetch
the record from within its **Chk** event using one of the **PVChkBetch** functions. This will most commonly occur in the **Chk** event of the last master key of Navigation and Lookup levels, which is where the master table record containing the requisite key field values is to be fetched from the database.

The end-user will be able to actually display the Possible Values window if an entry in the PVRec table has been created with a PVId containing the same stored procedure name as is specified in the SQL Proc/Text field of the **PV** property.

The **PV** property dialog contains the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Proc/Text</td>
<td>Used to enter the SQL statement or stored procedure name whose result set identifies all possible values currently existing in the database.</td>
</tr>
<tr>
<td>Constant</td>
<td>Used in conjunction with Parms to pass a constant value as a runtime parameter to the SQL statement or stored procedure.</td>
</tr>
<tr>
<td>Wildcard</td>
<td>Used in conjunction with Parms to identify a particular parameter as being able to support SQL wildcard values.</td>
</tr>
<tr>
<td>Struct.FieldName</td>
<td>Used in conjunction with the Parms to pass the value of other data items as a runtime parameters to the SQL statement or stored procedure. The Struct.FieldName value will normally be in the &quot;bTableName.FieldName&quot; format. The portion of the entry that identifies the table name MUST correspond precisely to the table name string passed in a corresponding call to the <strong>SetAddr</strong> statement. For example, assume the value of TableA.FieldA should be passed as a runtime parameter to the stored procedure. In this case, the Struct.FieldName would be “bTableA.FieldA”.</td>
</tr>
<tr>
<td>Field Offset Value</td>
<td>Optional depending on whether or not the table name referenced by the Struct.FieldName is actually the name of a table in the database. See the <strong>FieldName</strong> property for more information.</td>
</tr>
<tr>
<td>Declare Type</td>
<td>Optional depending on whether or not the table name referenced by the Struct.FieldName is actually the name of a table in the database. See the <strong>FieldName</strong> property for more information.</td>
</tr>
<tr>
<td>Length</td>
<td>Optional depending on whether or not the table name referenced by the Struct.FieldName is actually the name of a table in the database. See the <strong>FieldName</strong> property for more information.</td>
</tr>
<tr>
<td>Parm Tabs</td>
<td>Refresh the various data entry controls used to define parameters with the current values of the particular parameter corresponding to the selected tab. For example, to view the values for the second parameter press the Parm2 tab.</td>
</tr>
</tbody>
</table>

Consider the following guidelines when implementing a **PV** property:

The SQL statement or stored procedure referenced within the SQL Proc/Text field of the **PV** property must always be capable of receiving at least one parameter. However, regardless of how many parameters are expected by the SQL statement or stored procedure, the last parameter must be capable of receiving wildcard values.

Consider the following examples:

**Example 1**

```
Select * from TableA
where TableA.StringField LIKE @parm1
Order By StringField
```

**Example 2**

```
Select * from TableA
where TableA.StringFieldA    = @parm1
   and TableA.StringFieldB LIKE @parm2
Order By StringFieldA, StringFieldB
```
Example 3

Select * from TableA
where TableA.StringFieldA = @parm1
    and TableA.IntegerFieldB BETWEEN @parm2 and @parm3
Order By StringFieldA, IntegerFieldB

The first two examples use the LIKE statement for the last string field. However, since the LIKE keyword does not apply to integer fields, the BETWEEN keyword is used for integers. When the system handles “wildcard values” for a single integer parameter it will automatically pass two parameters, the first having a value corresponding to INTMIN and the last having a value corresponding to INTMAX.

The value of the underlying control is always passed to the SQL statement or stored procedure as the LAST parameter. Consequently the last parameter does not need to be manually defined using the Parm tabs. For example, if the stored procedure has only one parameter then no parameters need to be manually defined within the PV property. This is due to the fact that the value of the underlying control will automatically be passed as the last parameter, which happens to be the only parameter in this particular case.

The Order By clause must correspond to an index (that is, it must not cause a temporary sort operation) or else the view cannot be updated.

This property can be modified at design time only; it cannot be modified at runtime. For more information about how to create and use Possible Values in the database, see “Adding and Modifying Possible Values Lists” in the Customization Help or user’s guide. For information about how the user experiences changes that you make to this property, see “Using Possible Values Lists” in the Quick Reference Help or user’s guide.

See Also

Chk Event, DBNav Property, FieldName Property, PVChkFetch Functions, SetAddr Statement

Separator Property

Applies To
DSLFloat

Description
True/False property indicating whether the thousands separator should be displayed in the control.

Remarks
When the property is set to True, numbers will display with thousand separators, as shown in the following example: 123,456.78.
When the property is set to False, numbers will display without the thousand separators, as shown in the following example: 123456.78.
The separator character is set from the Control Panel Regional Settings.

Spin Property

Applies To
DSLDate, DSLInteger

Description
True/False value indicating whether spin buttons should be displayed.
Remarks
To modify the value of the **Spin** property at runtime, the **SetProps** statement should be used rather than modifying the property directly in Visual Basic code. Usage of **SetProps** allows the system to track changes to property values so as to avoid conflicts with customizations.

See Also
SpinIncrement Property

**SpinIncrement Property**

**Applies To**
DSLDate, DSLInteger

**Description**
Specifies the amount to increment or decrement a value when a spin button is used.

Remarks
To modify the value of the **SpinIncrement** property at runtime, the **SetProps** statement should be used rather than modifying the property directly in Visual Basic code. Usage of **SetProps** allows the system to track changes to property values so as to avoid conflicts with customizations.

See Also
Spin Property

**TabIndex Property**

**Applies To**
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLGrid, DSLCheck, DSLCombo, DSLOption

**Description**
Determines the logical sequence of controls within their parent form.

Remarks
When the user presses **Tab**, the actual order of progression through the controls is determined by the value of the **TabIndex** property specified during program construction. There are however several exceptions to this general rule. In particular, focus will skip over the control in the next tab sequence if it is either disabled or invisible. Furthermore, the design time **TabIndex** property value can be modified using the Customization Manager. This allows the logical sequence of data entry for any particular screen to be customized for unique circumstances.

The **TabIndex** property is also used by API calls referencing a range of controls such as: **SetDefaults**, **SetProps** and **DispFields**. These types of API calls allow the application to specify the first and last control upon which the designated operation should be performed. All controls having a **TabIndex** between the **TabIndex** of the first control and the **TabIndex** of the last control will be included in the group of targeted controls.

This property can be modified at design time only; it cannot be modified at runtime.

See Also
DispField Statements, SetDefaults Statement, SetProps Statement
TabStop Property

**Applies To**
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCheck, DSLCombo, DSLOption, DSLGrid

**Description**
Indicates whether a user can use Tab to give the focus to an object.

**Remarks**
A value of True causes the control to be a tab stop. A value of False will cause focus to skip over the field when the user is tabbing through controls.
The TabStop property differs from the Enabled property in that even if a particular control is not a tab stop, the user can still set the focus to the control by clicking on it. If a control is disabled then it cannot have focus under any circumstances.
To modify the value of the TabStop property at runtime, the SetProps statement should be used rather than modifying the property directly in Visual Basic code. Usage of SetProps allows the system to track changes to property values so as to avoid conflicts with customizations.

**See Also**
Enabled Property, SetProps Statement

Tag Property

**Applies To**
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLGrid, DSLCheck, DSLCombo, DSLOption, DSLUpdate

**Description**
Specifies any additional data needed for an application program.

**Remarks**
To modify the value of the Tag property at runtime, the SetProps statement should be used rather than modifying the property directly in Visual Basic code. Usage of SetProps allows the system to track changes to property values so as to avoid conflicts with customizations.

ToolTipText Property

**Applies To**
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCheck, DSLCombo, DSLOption

**Description**
Returns or sets a tooltip. A tooltip is a small string of text that displays while the mouse is positioned over the associated icon on a toolbar.
Remarks
To modify the value of the ToolTipText property at runtime, the SetProps statement should be used
rather than modifying the property directly in Visual Basic code. Usage of SetProps allows the system
to track changes to property values so as to avoid conflicts with customizations.

Note: Do not use this property with the DSLGrid control.

Grid view tooltips are driven by the ToolTipText property on the form view controls. If the current
column’s form view partner has a value for the ToolTipText property, that value will be used.

Top Property

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCheck, DSLCombo, DSLGrid, DSLOption,
DSLUpdate

Description
Determines the distance between the top edge of a control and the top edge of its container object.

Remarks
To avoid potential conflicts with customizations applied at runtime using the Customization Manager,
do not modify the position of a control at runtime.

The Top property of the DSLGrid control exists at both design and runtime. However, it can be modified
only at design time. When you put an DSLGrid on a form at design-time, it will have values for the
sizing properties (Left, Top, Height, Width). You can modify these values to optimize the size of the grid
for design-time viewing. You can size the grid quite small and put it in the lower-right corner of the
frame in which the DSLGrid resides to make it easier to see and/or manipulate all of the form view
controls to which the DSLGrid will be bound.

At runtime however, do not modify this property. The kernel will seek to take responsibility for the
sizing of the DSLGrid. For example, the kernel will attempt to make the DSLGrid match the size of the
underlying frame. The kernel will also resize the DSLGrid when the user resizes the form.

See Also
Height Property, Left Property, Width Property

Trigger Property

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCheck, DSLCombo, DSLOption

Description
Used to specify a list of one or more source fields upon which the value of the current control is
dependent.

Syntax
SourceTable,Field; CalculationType; ErrorCheckType [,SourceTable,Field; CalculationType;
ErrorCheckType]...

Remarks
A single field can be a identified as a source field in the Trigger property of multiple controls.
Correspondingly, a single control can have many source fields specified in its Trigger property.
A **Trigger** is activated whenever the value of any of the source fields is changed, either by the user or when any of the corresponding controls are re-defaulted (via calls such as a **SetDefaults** or as a result of a **New** operation).

Programmatic modification of the underlying field value does not activate the **Trigger**.

The value of the underlying control will automatically be re-displayed if it is changed as a result of the **Trigger**.

The **Trigger Field Specifications** dialog contains a single text box into which the source field, calculation type and error checking option is entered for all fields upon which the value of the underlying control is dependent.

The **Source Field Name** portion of any one trigger definition specifies the data item which, when changed, activates the **Trigger**. This value should be in the “bTableName.FieldName” format. The portion of the entry that identifies the table name MUST correspond precisely to the table name string passed in a corresponding call to the **SetAddr** statement. For example, assume the value of the current control is dependent upon the value of TableA.FieldA. In this case, the **Source Field Name** would be “TableA.FieldA”.

The following table outlines valid **Calculation Type** values:

<table>
<thead>
<tr>
<th>Calculation Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>None (default option). The value of the underlying control is not changed, however, its <strong>Chk</strong> event is fired. The exact point in time at which the <strong>Chk</strong> event is fired is determined by the specified Error Checking option. This Calculation Type can be thought of as a “re-error check” type of trigger and as such can be used when FieldB should be “re-validated” anytime FieldA changes.</td>
</tr>
<tr>
<td>D</td>
<td>Default. The value of the underlying control is re-defaulted using either its <strong>Default</strong> property or <strong>Default</strong> event. If the <strong>Default</strong> event is utilized, then the code contained therein should work when any or all of the source fields have blank or 0 values.</td>
</tr>
<tr>
<td>A</td>
<td>Adjust. Used when the current control accumulates a total of other fields. Each of the fields to be totaled should be specified as a source field in the <strong>Trigger</strong> property. When a source field changes, the value of the underlying control is adjusted by the net change in the source field. This Calculation Type is normally used to total source fields that are not are associated with a <strong>DSLGrid</strong> control. To work properly, the value of the underlying field should be saved in the database along with the source fields. If the underlying control merely represents a temporary field (as opposed to a database field), it will initialize to zero even when displaying existing records having non-zero values.</td>
</tr>
<tr>
<td>C</td>
<td>Calculate. Used when the current control accumulates a total of other fields in a <strong>DSLGrid</strong>. This Calculation Type cannot be used to total fields which are not associated with an <strong>DSLGrid</strong> control. The total of the source fields is calculated during <strong>DetailLoad</strong> for existing detail lines. When a source field subsequently changes, the value of the underlying control is adjusted by the net change in the source field. It is not required for the value of the underlying control to be saved in the database since it is recalculated every time detail lines are loaded. Nevertheless it can be saved. However, if the total is in fact saved in the database, it will still be recalculated the next time the detail lines are loaded into the <strong>DSLGrid</strong>. At that point, if the new total does not match the old total stored in the database, the user will not be warned and the new total will replace the old total.</td>
</tr>
</tbody>
</table>

The following table outlines valid **Error Checking** options:

<table>
<thead>
<tr>
<th>Error Checking</th>
<th>Description</th>
</tr>
</thead>
</table>
Immediately. The Chk event of the underlying control will be run immediately after the designated operation, defined by the Calculation Type, has been completed. Consequently, the Chk event will not need to be re-run at SAVE time or when the user tabs through the field, unless its value is changed again.

Later (default option). If the underlying control is not associated with an DSLGrid control then its Chk event will be run either when the user tabs through the control or at SAVE time, whichever occurs first. However, if the underlying control IS associated with an DSLGrid then its Chk event will be run either when the user tabs through the control or focus leaves the current detail line or at SAVE time, whichever occurs first.

If the underlying field is the source field for any other Trigger property, it will be error checked immediately regardless of the Error Checking option specified in its Trigger property. This situation is referred to as a nested trigger. The reason nested trigger fields are always error checked immediately is that the underlying field of the second trigger depends on the accuracy of the value in its source field. If the Chk event of the first field is not immediately run, it is possible the underlying field of the second trigger would use a “bad” value. Correspondingly, if an error occurs during the Chk event of the first field then the second trigger will not be activated since by definition its source field contains an erroneous value. When the user subsequently enters a valid value in the first field, the second trigger will be activated.

For example, Customer ID is a trigger source for Terms ID. Terms ID is a trigger source for Discount Date. When Customer ID is changed its Chk event is run. If its Chk event finds no error, the first trigger runs and sets the value of Terms ID. Since Terms ID is also a source field for the Discount Date trigger, its Chk event is also run. Its Chk event loads the corresponding terms record and if no errors occur, the Discount Date trigger is run. This trigger sets the value of Discount Date. The Discount Date is marked for a later error check, since it is not a source field for any other triggers.

This property can be modified at design time only; it cannot be modified at runtime.

See Also
Chk Event, Default Event, Default Property, SetAddr Statement

TrueText Property

Applies To
DSLCheck, DSLOption

Description
Determines the value of the underlying data field whenever the check box / option button is selected.

Remarks
Normally the TrueText value for integer true/false type of fields (that is, check box controls) is 1. This allows the value to more easily be tested for a true value within SQL. It also corresponds to the value of the LTRUE symbolic constant declared in Solomon.VBTools.vb. However, it does not correspond to the value of True within Visual Basic, which has a value of -1.

This property can be modified at design time only; it cannot be modified at runtime.

See Also
FalseText Property

Visible Property

Applies To
DSLDate, DSLFloat, DSLInteger, DSmaskedText, DSLCheck, DSLCombo, DSLOption, DSLGrid
Description
Determines whether or not the control is visible.

Remarks
A value of True indicates that the control is visible whereas a value of False causes the control to be invisible.

Required fields should not be made invisible unless they are automatically defaulted with a valid value.

To modify the value of the Visible property at runtime, the SetProps statement should be used rather than modifying the property directly in Visual Basic code. Usage of SetProps allows the system to track changes to property values so as to avoid conflicts with customizations.

If a control is made invisible by the application then it cannot be made visible using the Customization Manager. However if a control is made visible by the application then it can be made invisible by the Customization Manager.

The following remarks relate to how the Visible property relates to the DSLGrid control. At runtime, Microsoft Dynamics SL forces the grid to be visible during initialization. Do not set this property at design time for the DSLGrid control. Likewise, do not call SetProps() on a form view control because that reveals an intent to modify the Visible property on a row-by-row type of basis, which is not supported.

The only appropriate runtime modification to the visibility of a grid component is to display or hide an entire column, based on a data-driven rule. For example, suppose the Application A contains a grid having 10 fields. Let us also suppose that fields 7, 8, 9 and 10 should not be viewable unless Module XYZ is installed and configured. Lastly, let us assume that Module XYZ is neither installed nor configured. This would mean that Application A needs to hide fields 7, 8, 9 and 10. Since these fields are associated with an DSLGrid, we would want to hide the grid columns that correspond to fields 7, 8, 9 and 10. This operation can be performed during Form_Load by calling MSetProp() for each of the fields to be hidden – each time specifying a value of False for the Visible property. Note that in the MSetProp() scenario, this call must be made before the DSLGrid receives focus (such as at screen load, or in a master key Chk event of a header level). Do not call MSetProp() from within DSLGrid events, such as LineGotFocus().

To hide an DSLGrid, set the Visible property of the underlying frame to False.

See Also
BlankErr Property, Enabled Property, MSetProp Statement, SetProps Statement

Width Property

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCheck, DSLCombo, DSLOption, DSLGrid

Description
Determines the dimensions of the control.

Remarks
To avoid potential conflicts with customizations applied at runtime using the Customization Manager, do not modify the dimensions of a control at runtime.

The following remarks relate to how the Width property functions when used with the DSLGrid control. The Width property exists at both design-time and runtime, however, it can be modified only at design time.

When you put an DSLGrid on a form at design-time, it will have values for the sizing properties (Left, Top, Height, Width). You can modify these values to optimize the size of the grid for design-time viewing. You can size the grid quite small and put it in the lower-right corner of the frame in which the
DSLGrid resides to make it easier to see and/or manipulate all of the form view controls to which the DSLGrid will be bound.

At runtime however, do not modify this property. The kernel will seek to take total responsibility for the sizing of the DSLGrid. For example, the kernel will attempt to make the DSLGrid match the size of the underlying frame. The kernel will also resize the DSLGrid when the user resizes the form.

See Also

Height Property, Left Property, Top Property
Events

Cancel Event

Applies To
DSLUpdate

Description
Occurs when the user clicks the Cancel button on the toolbar.

Syntax
Sub Update1_Cancel(Level, RetVal)

Remarks
When the user clicks the Cancel button on the toolbar, the Cancel event is called once for each level in order from LEVEL0 to LEVELn.

This event is normally used on single Constant level applications, such as Setup screens. These types of applications do not have key fields which can be re-run in order to re-fetch the currently displayed information. Consequently, the Cancel event provides the application with an opportunity to re-fetch the Constant level record (for example, the setup record).

The Cancel event uses the following arguments:

| Argument | Type  | Description                                               |
|----------|-------|---------------|----------------------------------------------------------|
| Level    | Integer | Current level being processed.                           |
| RetVal   | Integer | The corresponding message will be displayed if RetVal is modified to anything other than the ErrNoMess symbolic constant defined in Solomon.VBTools.vb. |

Example
The following example illustrates how the Payroll Setup screen has implemented the Cancel event.

Sub Update1_Cancel (level%, retval%)
Dim PRSetup_Fetch As Integer

'Initialize bPRSetup
PRSetup_Fetch = SqlFetch1(CSR_PRSetup, "PRSetup_All", bPRSetup)

If (PRSetup_Fetch = 0) Then
  'Display fields from existing PRSetup record
  Call DispFields(PNULL, PNULL, PNULL)
Else
  'Default all controls for insert mode
  Call SetDefaults(PNULL, PNULL, PNULL)
End If

End Sub
Chk Event

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCheck, DSLCombo, DSLOption

Description
Occurs, at a minimum, whenever the data field changes and loses focus.

Syntax
Sub CtrlName_Chk( [Index], ChkStrg, RetVal)

Remarks
Whenever the user modifies the value of a field, the new value often needs to be validated (for example, checked for errors). If the new data value is valid then the application may perform other related operations such as re-default or disable other fields. The Chk event is where such code normally resides since this event is called anytime the value of the field is changed.

The Chk event can also fire at other times when the user did not directly modify the value of the control. For example, navigating through existing records on a Navigation Level using the First, Last, Prev, and Next toolbar buttons is conceptually equivalent to the user entering new key field values to bring up different records. Consequently, when navigating through existing records on a Normal level, the Chk event for each key field on that Navigation Level is fired every time the user navigates from one record to the next, regardless of how the navigation operation was invoked (for example, data entry or toolbar navigation). The Chk event can also fire in relation to the Trigger property. For example, using the Trigger property, a relationship between FieldA and FieldB can be defined such that FieldB should be “re-validated” (for example, its Chk event should be called) anytime the value of FieldA changes.

The Chk event uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChkStrg</td>
<td>String</td>
<td>If the user just typed a new value for the field and pressed Tab, then ChkStrg will correspond to that new data value. If the Chk event was called for any other reason, such as navigation, triggers, etc., then the value of ChkStrg is the data value that will be assigned to the underlying field unless it is rejected within the Chk event. In this latter case, it is possible that the value of the field will be unchanged and therefore ChkStrg will reflect that fact.</td>
</tr>
<tr>
<td>RetVal</td>
<td>Integer</td>
<td>A data value can be rejected simply by modifying the RetVal parameter which is passed to the Chk event. This parameter is actually passed by reference which means that any modifications to RetVal are automatically detected by the system once program control exits the Chk event.</td>
</tr>
</tbody>
</table>
The following table outlines the possible values which can be assigned to RetVal and their corresponding effect on the system once program control exits the **Chk** event:

<table>
<thead>
<tr>
<th>RetVal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoAutoChk</td>
<td>Suppresses automatic error checking which would normally occur after the <strong>Chk</strong> event. It is typically used in the <strong>Chk</strong> event of key fields when the result of the <strong>PVChkFetch</strong> or <strong>DBNavFetch</strong> is NOTFOUND but the user should still be able to add new records. Since this return value is designed to let ChkStrg pass as valid even though a corresponding data item could not be located within the database, the value of ChkStrg will by definition be applied as the new value of the field as opposed to being rejected.</td>
</tr>
<tr>
<td>A Message Number</td>
<td>When RetVal is initialized with a specific message number, the corresponding message from the Microsoft Dynamics SL message file will automatically be displayed after the <strong>Chk</strong> event. Furthermore, the value of ChkStrg will be rejected. A common implementation of this type of return value is to set RetVal to the return value of the <strong>PVChkFetch</strong> or <strong>DBNavFetch</strong> call performed within the <strong>Chk</strong> event. This is due to the fact that the return value from these functions in the “not found” scenario corresponds to the “not found” message number. Setting RetVal to a message number is the recommended method for providing feedback to the user as to the precise reason why the value of ChkStrg is being rejected.</td>
</tr>
<tr>
<td>ErrNoMess</td>
<td>This return value is similar to a message number in the fact that it will cause the value of ChkStrg to be rejected. However the system will not display a message since the application should already have done so during the <strong>Chk</strong> event. This is useful in cases where the message requires data values for one or more replacement parameters. Consequently, the application can use the <strong>Messf</strong> statement to display the message along with the required substitution values and then subsequently set RetVal = ErrNoMess.</td>
</tr>
</tbody>
</table>

The **Chk** event for the last master key of Normal and Lookup levels must contain either a **PVChkFetch** or **DBNavFetch** call depending on whether a **PV** or **DBNav** property was implemented.

If a data entry control has a **PV** property value and no code within its Chk event, then the system will automatically validate the value of ChkStrg using the **PV** property. If the corresponding record does not exist then the value of ChkStrg will automatically be rejected and a “not found” type of message will be displayed.

**See Also**

**DBNavFetch Functions**, **Level Property**, **Messf Statement**, **PVChkFetch Functions**, **Trigger Property**

**Example**

The following example illustrates a common implementation of the **Chk** event on the last master key for a Navigation Level. This particular example was taken from the Payroll Earnings Type Maintenance screen. Notice the fact that RetVal is set to NoAutoChk. This allows new Earnings Types to be defined even though the **PVChkFetch** call returned NOTFOUND to the application.

```vba
Sub cID_Chk (chkstrg As String, retval As Integer)
    Dim EarnType_Fetch As Integer

    EarnType_Fetch = PVChkFetch1(CNULL, CSR_EarnType, chkstrg, bEarnType)

    If (EarnType_Fetch = 0) Then
        'Evaluate all properties based on bEarnType values
        Call Evaluate_Properties(FLD_ALL)
    Else
        'Properties should be re-evaluated in NewLevel event AFTER all
        'defaults have been applied
    End If
End Sub
```
End If

RetVal = NoAutoChk
End Sub

The following example illustrates the Chk event, on a non-key field control, in which a record corresponding to the value of ChkStrg must be fetched from the database for use by the application. If the record cannot be found then it is to be considered an error. This code snippet was actually taken from the Chk event of the Earnings Type field on the Timesheet Defaults subscreen of the Payroll Employee Maintenance screen. This particular application requires that the default Earnings Type for all employees must contribute to net pay. Thus it is not enough that the user enters the ID of just any Earnings Type. Rather, that Earnings Type must also be defined as contributing to net pay. If the relevant Earnings Type does not contribute to net pay then it will be rejected simply by setting RetVal to the particular Microsoft Dynamics SL message number explaining the nature of the problem.

Message number 260 is the actual message which will be displayed and its associated text in the Microsoft Dynamics SL message file reads as follows: “Earnings type must contribute to net pay, please reenter.”

Sub cDfltEarnType_Chk (chkstrg As String, retval As Integer)

RetVal = PVChkFetch1(CNULL, CSR_EarnType, chkstrg, bEarnType, Len(bEarnType))

If (RetVal = 0) Then

If (bEarnType.NetPay <> LTRUE) Then
    RetVal = 260
    End If

End If

End Sub
Default Event

 Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCheck, DSLCombo, DSLOption

 Description
Occurs anytime the control is being defaulted and a Default property has not been implemented.

 Syntax
Sub CtrlName.Default( OldValue, RetVal)

 Remarks
The default data value for any particular data entry control can be specified via either the Default property or the Default event. The Default property can be used when the default value is not contingent upon any the value of any other data item. However, if the methodology for determining a default value varies depending particular situations, then code should be written for the Default event. If a Default property is defined then the Default event will not be used.

Within the Default event, the default value of the relevant field is set by referring directly to the Visual Basic variable to which the control is bound. For example, if the control is bound to string field called “bTableA.FieldA” within its FieldName property then within the Default event the value of the field can be defaulted in a manner such as bTableA.FieldA = “StringValue”. The default data value for an DSLOption button group is always derived from the first option button in the group, since by definition only one option button can be selected within any particular group.

The Default event uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OldValue</td>
<td>String</td>
<td>Contains the data value that the underlying field had immediately prior to the Default event. This is required because the underlying field will already have its value overwritten when the Default event is called.</td>
</tr>
<tr>
<td>RetVal</td>
<td>Integer</td>
<td>A non-zero positive return value suppresses further default action, including customization defaults and trigger calls.</td>
</tr>
</tbody>
</table>

 See Also
Default Property, Level_SetDefaults Statement, SetDefaults Statement, Trigger Property
Delete Event

Applies To
DSLUpdate

Description
Occurs during the series of actions/events initiated by a DELETE operation.

Syntax
Sub Update1_Delete( Level, LevelsDone, RetVal)

Remarks
To understand when the Delete event occurs, a developer must first understand the higher level concept of what is referred to here as a DELETE operation. The operation is differentiated from the event in the fact that the Delete event comprises only a segment of a series of events implied by a DELETE operation.

A DELETE operation is initiated when the user clicks on Delete on the toolbar. The only exception is when the user deletes a detail line from an DSLGrid in which case the LineChk event is called.

A DELETE operation is comprised of the following series of actions and/or events. If the application contains more than one Navigation Level then the user will be prompted as to which of the Navigation Level records is being deleted. Processing will begin with the level the user selects and continue for each non-Lookup level in order from LEVEL0 to LEVELn.

- The Delete event is called for the level. If this is the first level to be processed than a database transaction has not yet been initiated. Consequently the application must call TranBeg if it needs to perform database update/delete operations during the first pass through the Delete event.

- If the application did not modify the value of RetVal in the preceding step then the master table for the level is deleted. The master table for any particular level is the table identified by the SetAddr call for that particular level.

Once all levels which had previously been modified have been successfully updated, the database transaction is ended and a NEW operation is automatically initiated to prepare the application for data entry.

If any errors occur during the delete of any level then the entire operation is aborted, including the database transaction.

The Delete event uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level</td>
<td>Integer</td>
<td>Current level being processed.</td>
</tr>
<tr>
<td>LevelsDone</td>
<td>Integer</td>
<td>Number of levels already processed within the context of the current DELETE operation.</td>
</tr>
<tr>
<td>RetVal</td>
<td>Integer</td>
<td>The automatic deletion of the master table for the current level, which occurs after the Delete event for that particular level, can be suppressed by setting RetVal to the NoAction symbolic constant defined in Solomon.VBTools.vb. The entire DELETE operation can be aborted by setting RetVal either to a Microsoft Dynamics SL message number or the ErrNoMess symbolic constant defined in Solomon.VBTools.vb.</td>
</tr>
</tbody>
</table>

See Also
LineChk Event, SetAddr Statement, TranBeg Statement
Example

The following example illustrates how to perform delete logic within the **Delete** event and abort the entire **DELETE operation** if any logical errors are detected. This particular code snippet relates to the Payroll Employee Maintenance screen. Notice the fact that an Employee record cannot be deleted if they have any year-to-date earnings. This is accomplished by setting RetVal to Microsoft Dynamics SL message number 259. This will not only cause the entire **DELETE operation** to be aborted but it will also cause the corresponding Microsoft Dynamics SL message to be displayed, thereby explaining the precise nature of the problem to the user. The actual message text associated with this message number is “Year-To-Date earnings must be zero.” Similarly, an Employee cannot be deleted if he/she has any existing checks or timesheet transactions. In these cases, RetVal is set to 58 which has the following corresponding Microsoft Dynamics SL message text: “This record cannot be deleted. It is in use elsewhere in the system.”

```vba
Sub Update1_Delete (level%, levelsdone%, retval%)  
    Dim SqlStr As String
    Dim PRDoc_Fetch As Integer
    Dim PRTran_Fetch As Integer

    If (level = LEVEL0) Then

        If (bEmployee.YtdEarn <> 0#) Then
            RetVal = 259
        Else
            'Determine whether or not the employee has any existing checks.
            SqlStr = "PRDoc_EmpId" + SParm(bEmployee.EmpId)
            PRDoc_Fetch = SqlFetch1(CSR_PRDoc_Del_Logic, SqlStr, bPRDoc)
            If (PRDoc_Fetch = 0) Then
                RetVal = 58
            Else
                'Determine whether or not the employee has ANY existing
                'timesheet transactions.
                SqlStr = "PRTran_EmpId_TSht" + SParm(bEmployee.EmpId) + IParm(LTRUE)
                PRTran_Fetch = SqlFetch1(CSR_PRTran_Del_Logic, SqlStr, bPRTran)
                If (PRTran_Fetch = 0) Then
                    RetVal = 58
            End If
        End If
    Else
        End If
    End If

End Sub
```

End Sub
DragDrop Event

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLGrid, DSLCheck, DSLCombo, DSLOption

Description
This event is reserved for Microsoft and should not be used.

DragOver Event

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLGrid, DSLCheck, DSLCombo, DSLOption

Description
This event is reserved for Microsoft and should not be used.

Finish Event

Applies To
DSLUpdate

Description
Occurs when the user indicates he/she is finished with the information currently displayed on the screen.

Syntax
Sub Update1_Finish (Level, Updated, RetVal)

Remarks
The Finish event is called when the user is finished with the currently displayed information so as to allow the application an opportunity to perform specific operations on or related to the data before it “leaves the user’s screen.” For example, assume the user just entered some sort of financial information and now he/she is simply going to close the application since they have completed their original task. At that point the application could warn the user that the information is out of balance and ask whether or not they want to remedy the problem. Without such a warning, the user may not notice the error until it subsequently causes some other problem.

The user is considered to be finished with the information currently displayed when they perform any one of the following operations:

- When the user clicks on the Finish button on the toolbar. In this case, the Finish event actually fires after the Update event, if a SAVE operation is even necessary (for example, if any information changed).
- When the user attempts to either enter a new item, navigate to a different item or close the screen. If information has been changed then the user will first be prompted as to whether or not they want to save their outstanding changes. If not then the screen will be refreshed with the information as it exists in the database so that accurate data is readily available when the Finish event is called.

The Finish event is called once for each level in order from LEVELn to LEVEL0. Notice that this event is called in reverse order as compared to the NewLevel, Update, Delete and Cancel events. This reverse order allows the application to report problems with the data at the most granular level first.
The application can abort the **Finish** event by merely changing the RetVal parameter. In this case, the users action is also aborted. For example, assume the user has RecordA on the screen and then clicks **Next** on the toolbar. Clearly this indicates that the user is finished viewing RecordA and now wants to view the next record, presumably RecordB. Consequently, the **Finish** event is called just prior to navigating to RecordB. If the application sets RetVal, say to a Microsoft Dynamics SL message number, then the corresponding message will be displayed when the **Finish** event for the current level exits and the navigate operation will be aborted. Since the navigate operation was aborted, the user will still be able to view RecordA.

The **Finish** event uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level</td>
<td>Integer</td>
<td>Current level being processed. The <strong>Finish</strong> event is called beginning with LEVELn (that is, the last level on the screen) and counting backwards to LEVEL0.</td>
</tr>
<tr>
<td>Updated</td>
<td>Integer</td>
<td>Currently not used.</td>
</tr>
<tr>
<td>RetVal</td>
<td>Integer</td>
<td>The <strong>Finish</strong> event can be aborted by simply setting RetVal to either a valid message number or the ErrNoMess symbolic constant defined in Solomon.VBTools.vb.</td>
</tr>
</tbody>
</table>

**GotFocus Event**

**Applies To**

DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLGrid, DSLCheck, DSLCombo, DSLOption

**Description**

Occurs whenever an object receives focus.

**Syntax**

Sub object_GotFocus

**Remarks**

The object is the control. Use this event to specify the actions that need to occur when a control first receives the focus. An object can receive the focus only if its Enabled and Visible properties are set to True.

**See Also**

Enabled Property, Visible Property
LineChk Event

Applies To
DSLGrid

Description
Occurs whenever a detail line within an DSLGrid control is either inserted, updated or deleted.

Syntax
Sub DSLGridName_LineChk( Action, RecMaintFlg, RetVal)

Remarks
Any time the user inserts, updates or deletes a detail line in an DSLGrid control, the LineChk event fires. In the insert and update cases, the event does not actually run until the user actually leaves the detail line.

This event is most often used to perform special delete logic for detail lines that the user is attempting to delete. For example, the General Ledger Chart of Accounts Maintenance screen contains a grid displaying all records from the Account table. Users can delete Account records provided that the corresponding account number is not used on any setup screens among other areas. This is actually implemented via the use of logic within the LineChk event.

Deleted records are copied from the underlying memory array to a temporary “deleted record” memory array. The resource handle to that memory array can be obtained using the MGetDelHandle function.

The LineChk event uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>Integer</td>
<td>Action being performed on the detail line.</td>
</tr>
<tr>
<td>RecMaintFlg</td>
<td>Integer</td>
<td>Current status of the detail line.</td>
</tr>
<tr>
<td>RetVal</td>
<td>Integer</td>
<td>The application can prevent record deletions by simply setting RetVal to either a valid message number or the ErrNoMess symbolic constant defined in Solomon.VBTools.vb.</td>
</tr>
</tbody>
</table>

Solomon.VBTools.vb contains the following symbolic constants defining possible Action values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERTED</td>
<td>A new detail line is being inserted.</td>
</tr>
<tr>
<td>UPDATED</td>
<td>An existing detail line is being updated.</td>
</tr>
<tr>
<td>DELETED</td>
<td>An existing detail line is being deleted.</td>
</tr>
</tbody>
</table>
Solomon.VBTools.vb contains the following symbolic constants defining possible *RecMaintFlg* values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEWROW</td>
<td>Indicates that the current detail line is just now being added. The status of the detail line will be changed to INSERTED if no errors occur during the <strong>LineChk</strong> event. If any further actions are performed on the same detail line, then when <strong>LineChk</strong> subsequently runs it will have a <em>RecMaintFlg</em> of INSERTED.</td>
</tr>
<tr>
<td>INSERTED</td>
<td>The current detail line was previously added after the <strong>DSLGrid</strong> was initially loaded but has not yet been saved to the database.</td>
</tr>
<tr>
<td>UPDATED</td>
<td>The current detail line was initially loaded into the <strong>DSLGrid</strong> but has been subsequently modified. Furthermore, the modifications to the current detail line have not yet been saved.</td>
</tr>
<tr>
<td>NOTCHANGED</td>
<td>The current detail line was initially loaded into the <strong>DSLGrid</strong> and has not been previously modified. <strong>Note</strong>: Records marked as INSERTED and UPDATED will automatically be assigned the NOTCHANGED status after the next successful SAVE operation.</td>
</tr>
</tbody>
</table>

**See Also**

LineGotFocus Event, MGetDelHandle Function, SetButton Statement

**LineGotFocus Event**

**Applies To**

**DSLGrid**

**Description**

Occurs whenever a detail line within an **DSLGrid** control receives focus.

**Syntax**

Sub **DSLGridName**.LineGotFocus( *RecMaintFlg*, *RetVal*)

**Remarks**

Any time the user moves to or inserts a detail line within an **DSLGrid** control, the **LineGotFocus** event is called immediately.

The *RecMaintFlg* parameter can subsequently be evaluated to determine whether or not the user is actually adding a new detail line. In such a case, default values can be explicitly assigned to fields within the detail record for which no corresponding control exist. For example, assume TableA is a header table and TableB is a detail record. In this case, each unique TableA record could have many unique TableB records displayed within the **DSLGrid**. However, the primary point to make here is that the first segment of the unique key for TableB would have to include a field that relates directly to TableA (for example, the join field). This type of field is a prime candidate to be defaulted in the **LineGotFocus** event since it will always have the same value (for example, the value of TableA.KeyField) and therefore creating an invisible control with a **Default** property is not really necessary.

The **LineGotFocus** event uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>RecMaintFlg</em></td>
<td>Integer</td>
<td>Status of the detail line.</td>
</tr>
<tr>
<td><em>RetVal</em></td>
<td>Integer</td>
<td>The application can prevent all of the corresponding detail level controls from being automatically defaulted when a new record is being inserted by simply setting <em>RetVal</em> to <strong>NoAction</strong> (which is a symbolic constant defined in Solomon.VBTools.vb).</td>
</tr>
</tbody>
</table>
Software Development Kit

Solomon.VBTools.vb contains the following symbolic constants defining possible *RecMaintFlg* values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEWROW</td>
<td>Indicates that the user is beginning the insertion of a new detail line. The status of the detail line will be changed to INSERTED after all fields have been error checked and no errors occur during the LineChk event.</td>
</tr>
<tr>
<td>INSERTED</td>
<td>The current detail line was successfully added after the <em>DSLGrid</em> was loaded and has not been saved to the database.</td>
</tr>
<tr>
<td>UPDATED</td>
<td>The current detail line was initially loaded into the <em>DSLGrid</em> but has been subsequently modified. Furthermore, the modifications to the current detail line have not yet been saved.</td>
</tr>
<tr>
<td>NOTCHANGED</td>
<td>The current detail line was initially loaded into the <em>DSLGrid</em> and has not been subsequently modified. <strong>Note</strong>: Records marked as INSERTED and UPDATED will automatically be assigned the NOTCHANGED status after the next successful SAVE operation.</td>
</tr>
</tbody>
</table>

**See Also**

LineChk Event, SetButton Statement

**Example**

The following example is taken from the Payroll Earnings Type Maintenance screen. This screen is a header/detail type of screen having the EarnType table as the header table and two detail records in the grid, ValEarnDed and Deduction. ValEarnDed is the master table for the detail level and the Deduction table is only joined in for a description. At any rate, notice the test for NEWROW and the corresponding work that is only performed for new detail lines.

```vbscript
Sub Spread_ValEarnDed_LineGotFocus (maintflg%, retval%)

    If (maintflg = NEWROW) Then
        'Null out secondary records on the detail line
        bDeduction = nDeduction

        'Initialize the master detail record with the key field ID from the
        'header record
        bValEarnDed.EarnTypeId = bEarnType.Id

    End If

End Sub
```

**LostFocus Event**

**Applies To**

*DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLGrid, DSLCheck, DSLCombo, DSOption*

**Description**

Occurs whenever an object loses the focus, either by user action, or by changing the focus in code using the *ApplSetFocus* statement.

**Syntax**

Sub *object*_LostFocus
See Also
ApplSetFocus Statement, GotFocus Event

MemoryLoad Event

Applies To
DSLUpdate

Remarks
This event is reserved for use by Microsoft and should not be used.

MouseDown Event

Applies To
DSLDate, DSLFloat, DSLInteger, DSLCheck, DSLMaskedText, DSLOption

Description
Occurs whenever a user presses a mouse button.

Syntax
Sub object_MouseDown(button As Integer, shift As Integer, x As Single, y As Single)

Remarks
The MouseDown event has the following parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>An object expression that evaluates to an object in the Applies To list.</td>
</tr>
<tr>
<td>button</td>
<td>An integer that identifies the button that was pressed (MouseDown) or released (MouseUp) to cause the event. The button argument is a bit field with bits corresponding to the left button (bit 0), right button (bit 1), and middle button (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Only one of the bits is set, indicating the button that caused the event.</td>
</tr>
<tr>
<td>shift</td>
<td>An integer that corresponds to the state of the Shift, Ctrl, and Alt keys when the button specified in the button argument is pressed or released. A bit is set if the key is down. The shift argument is a bit field with the least-significant bits corresponding to the Shift key (bit 0), the Ctrl key (bit 1), and the Alt key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. The shift argument indicates the state of these keys. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys are pressed. For example, if both Ctrl and Alt were pressed, the value of shift would be 6.</td>
</tr>
<tr>
<td>x, y</td>
<td>A number that specifies the current location of the mouse pointer. The x and y values are always expressed in terms of the coordinate system set by the ScaleHeight, ScaleWidth, ScaleLeft, and ScaleTop properties of the object.</td>
</tr>
</tbody>
</table>

Use a MouseDown or MouseUp event procedure to specify actions that will occur when a given mouse button is pressed or released. Unlike the Click and DblClick events, MouseDown and MouseUp events enable you to distinguish between the left, right, and middle mouse buttons. You can also write code for mouse-keyboard combinations that use the Shift, Ctrl, and Alt keyboard modifiers.
### Constant (Button) | Value | Description
--- | --- | ---
vbLeftButton | 1 | Left button is pressed.
vbRightButton | 2 | Right button is pressed.
vbMiddleButton | 4 | Middle button is pressed.

### Constant (Shift) | Value | Description
--- | --- | ---
vbShiftMask | 1 | Shift key is pressed.
vbCtrlMask | 2 | Ctrl key is pressed.
vbAltMask | 4 | Alt key is pressed.

The constants then act as bit masks you can use to test for any combination of buttons without having to figure out the unique bit field value for each combination.

You can use **MouseMove** to respond to an event caused by moving the mouse. The button argument for **MouseDown** and **MouseUp** differs from the button argument used for **MouseMove**. For **MouseDown** and **MouseUp**, the button argument indicates exactly one button per event, whereas for **MouseMove**, it indicates the current state of all buttons.

The constants then act as bit masks you can use to test for any combination of buttons without having to figure out the unique bit field value for each combination.

Use **MouseMove** to respond to an event caused by moving the mouse. The button argument for **MouseDown** and **MouseUp** differs from the button argument used for **MouseMove**. For **MouseDown** and **MouseUp**, the button argument indicates exactly one button per event, whereas for **MouseMove**, it indicates the current state of all buttons.

**See Also**

**MouseMove Event**

### MouseMove Event

**Applies To**

DSLDate, DSLFloat, DSLInteger, DSLCheck, DSLMaskedText, DSLOption

**Description**

Occurs whenever a user drags the mouse.

**Syntax**

Sub object_MouseMove(button As Integer, shift As Integer, x As Single, y As Single)

**Remarks**

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>An object expression that evaluates to an object in the Applies To list.</td>
</tr>
<tr>
<td>Button</td>
<td>An integer that corresponds to the state of the mouse buttons in which a bit is set if the button is down. The button argument is a bit field with bits corresponding to the left button (bit 0), right button (bit 1), and middle button (bit 2). These bits correspond to the values 1, 2, and 4, respectively. It indicates the complete state of the mouse buttons; some, all, or none of these three bits can be set, indicating that some, all, or none of the buttons are pressed.</td>
</tr>
</tbody>
</table>
Part | Description
--- | ---
Shift | An integer that corresponds to the state of the Shift, Ctrl, and Alt keys. A bit is set if the key is down. The shift argument is a bit field with the least-significant bits corresponding to the Shift key (bit 0), the Ctrl key (bit 1), and the Alt key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. The shift argument indicates the state of these keys. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys are pressed. For example, if both Ctrl and Alt were pressed, the value of shift would be 6.

X, y | A number that specifies the current location of the mouse pointer. The x and y values are always expressed in terms of the coordinate system set by the ScaleHeight, ScaleWidth, ScaleLeft, and ScaleTop properties of the object.

**MouseUp Event**

**Applies To**
DSLFloat, DSLInteger, DSLCheck, DSLMaskedText, DSLOption

**Description**
Occurs whenever a user releases the mouse button.

**Syntax**
Sub object _MouseUp_(button As Integer, shift As Integer, x As Single, y As Single)

**Remarks**
The MouseUp event has the following parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>An object expression that evaluates to an object in the Applies To list.</td>
</tr>
</tbody>
</table>

**Button**
An integer that identifies the button that was pressed (MouseDown) or released (MouseUp) to cause the event. The button argument is a bit field with bits corresponding to the left button (bit 0), right button (bit 1), and middle button (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Only one of the bits is set, indicating the button that caused the event.

**Shift**
An integer that corresponds to the state of the Shift, Ctrl, and Alt keys when the button specified in the button argument is pressed or released. A bit is set if the key is down. The shift argument is a bit field with the least-significant bits corresponding to the Shift key (bit 0), the Ctrl key (bit 1), and the Alt key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. The shift argument indicates the state of these keys. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys are pressed. For example, if both Ctrl and Alt were pressed, the value of shift would be 6.

**X, y**
A number that specifies the current location of the mouse pointer. The x and y values are always expressed in terms of the coordinate system set by the ScaleHeight, ScaleWidth, ScaleLeft, and ScaleTop properties of the object.

Use **MouseDown** or **MouseUp** to specify actions that will occur when a given mouse button is pressed or released. Unlike the Click and DblClick events, **MouseDown** and **MouseUp** events enable you to distinguish between the left, right, and middle mouse buttons. You can also write code for mouse-keyboard combinations that use the Shift, Ctrl, and Alt keyboard modifiers.
### Software Development Kit

#### Constant (Button)
<table>
<thead>
<tr>
<th>Constant (Button)</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbLeftButton</td>
<td>1</td>
<td>Left button is pressed.</td>
</tr>
<tr>
<td>VbRightButton</td>
<td>2</td>
<td>Right button is pressed.</td>
</tr>
<tr>
<td>VbMiddleButton</td>
<td>4</td>
<td>Middle button is pressed.</td>
</tr>
</tbody>
</table>

#### Constant (Shift)
<table>
<thead>
<tr>
<th>Constant (Shift)</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbShiftMask</td>
<td>1</td>
<td>Shift key is pressed.</td>
</tr>
<tr>
<td>vbCtrlMask</td>
<td>2</td>
<td>Ctrl key is pressed.</td>
</tr>
<tr>
<td>vbAltMask</td>
<td>4</td>
<td>Alt key is pressed.</td>
</tr>
</tbody>
</table>

The constants then act as bit masks you can use to test for any combination of buttons without having to figure out the unique bit field value for each combination.

You can use MouseMove to respond to an event caused by moving the mouse. The button argument for MouseDown and MouseUp differs from the button argument used for MouseMove. For MouseDown and MouseUp, the button argument indicates exactly one button per event, whereas for MouseMove, it indicates the current state of all buttons.

The constants then act as bit masks you can use to test for any combination of buttons without having to figure out the unique bit field value for each combination.

**See Also**
MouseMove Event

**NewLevel Event**

**Applies To**
DSLUpdate

**Description**
Occurs during the series of actions/events initiated by a NEW operation.

**Syntax**
Sub Update1_NewLevel( Level, RetVal)

**Remarks**
To understand when the NewLevel event occurs, a developer must first understand the higher level concept of what is referred to here as a NEW operation. The operation is differentiated from the event in the fact that the NewLevel event comprises only a segment of a series of events implied by a NEW operation.

A NEW operation is initiated by any one of the following occurrences:

- When ScreenInit is called from within Form1_Load.
- When the user clicks on the New button on the toolbar.
- When the user clicks on the Finish button on the toolbar. In this case, the NewLevel event actually fires after the Update and Finish events complete successfully.
- When the user clicks on the Delete button on the toolbar. After the record is successfully deleted in the Delete event, the application prepares itself to receive new information by automatically initiating a NEW operation.
- When the user navigates either prior to the first record or past the last record in a table using either the Prev or Next buttons on the toolbar.
• When the user enters a value that does not already exist in the database for one or more key fields. For example, if an application contains three key fields then a NEW operation will be initiated if the combination of all three key field values does not already exist in the database.

A NEW operation is comprised of the following series of actions and/or events for each non-detail level, beginning with the level on which the new operation was initiated. For example, if the NEW operation is initiated on LEVEL0, the levels will be processed in order from LEVEL0 to LEVELn.

• The master table for the level is blanked out. The master table for any particular level is the table identified by the SetAddr call for that particular level.

• The NewLevel event is called for the level.

• All controls on the level are defaulted.

Since defaulting for new detail level records within an DSLGrid control is performed within the LineGotFocus event, the NewLevel event is not called for detail levels.

The NewLevel event uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level</td>
<td>Integer</td>
<td>Current level being processed.</td>
</tr>
<tr>
<td>RetVal</td>
<td>Integer</td>
<td>The automatic defaulting of all controls on the current level which occurs after the NewLevel event for that particular level can be suppressed by setting RetVal to the NoAction symbolic constant defined in Solomon.VBTools.vb.</td>
</tr>
</tbody>
</table>

**See Also**

Level_SetDefaults Statement, LineGotFocus Event, ScreenInit Statement, SetAddr Statement

**Example**

The following code snippet was taken from the Payroll Employee Maintenance screen.

```vbscript
Sub Update1_NewLevel (level%, retval%)

If (level = LEVEL0) Then
  'Force ALL default values to be applied to EMPLOYEE level BEFORE Evaluate_PROPERTIES() is called.
  Call Level_SetDefaults(PNULL, PNULL, PNULL, LEVEL0)

  bEmployee.CalQtr = bPRSetup.CurrCalQtr
  bEmployee.CalYr = bPRSetup.CurrCalYr

  'Re-evaluate the properties of all controls whose property settings depend upon data values.
  Call Evaluate_PROPERTIES(FLD_ALL)

  'Set retval to keep Swim from defaulting LEVEL0 controls again.
  RetVal = NoAction

  End If

End Sub
```
QuickPrint Event

Applies To
Form1

Description
Occurs when the Quick Print button is pressed on the Applications toolbar menu.

Syntax
Sub Form1_QuickPrint(ByVal eventSender As System.Object, ByVal eventArgs As System.EventArgs)
Handles Me.QuickPrint

Remarks
By adding this event in the application it activates the Quick Print button in the Application toolbar menu.

Example
Private Sub Form1_QuickPrint(ByVal eventSender As System.Object, ByVal eventArgs As System.EventArgs) Handles Me.QuickPrint
    Dim Parmstr As String
    Dim Wherestr As String
    'The Edit Report will be printed
    Wherestr = "*pROI*" & " {vr_01810.Batch_Module} =" & _
        SParm(bBatch.Module_Renamed) & " AND " & _
        "{vr_01810.Batch_BatNbr} =" & SParm(bBatch.BatNbr)
    Parmstr = "01810/RUN " & PRMSEP & Wherestr & "/WHERE" & _
        PRMSEP & "01810" & "/FORMAT" & PRMSEP
    Call CallApplicWait( "ROI", Parmstr)
End Sub
Update Event

Applies To
DSLUpdate

Description
Occurs during the series of actions/events initiated by a SAVE operation.

Syntax
Sub Update1_Update( Level, InsertFlg, LevelsDone, LevelsLeft, RetVal)

Remarks
To understand when the Update event occurs, a developer must first understand the higher level concept of what is referred to here as a SAVE operation. The operation is differentiated from the event in the fact that the Update event comprises only a segment of a series of events implied by a SAVE operation.

A SAVE operation is initiated by any one of the following occurrences:

- When the user clicks on the Save button on the toolbar.
- When the user clicks on the Finish button on the toolbar.
- When the user answers Yes in response to the “Do you want to save your outstanding changes?” prompt. This prompt occurs anytime the user has modified data and then attempts to either enter a new item, navigate to a different item or close the screen without first saving his/her changes.

A SAVE operation is comprised of the following series of actions and/or events for each level which has been changed in order from LEVEL0 to LEVELn. For example, if the only information that changed resides on LEVEL0 then only LEVEL0 will be processed during the SAVE operation. A database transaction is started before any levels are processed.

- The Update event is called for the level.

  Note: TranEnd should absolutely never be called by the application within the Update event because the system will then be unable to roll back the entire SAVE operation!

- If the application did not modify the value of RetVal in the preceding step then the master table for the level is updated. The master table for any particular level is the table identified by the SetAddr call for that particular level.

Once all levels which had previously been modified have been successfully updated, the database transaction is ended. The Update event is then called one additional time. The Level parameter will have a value corresponding to the Finished symbolic constant defined in Solomon.VBTools.vb. At this point the level status for all levels should have a value of NOTCHANGED. Furthermore, the line status of each individual detail line within any grids should also have a value of NOTCHANGED.

If any errors occur during the update of any level then the entire operation is aborted, including the database transaction.
The **Update** event uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level</td>
<td>Integer</td>
<td>Current level being processed.</td>
</tr>
<tr>
<td>InsertFlag</td>
<td>Integer</td>
<td>True indicates a new record is being inserted. False indicates that an existing record is being updated.</td>
</tr>
<tr>
<td>LevelsDone</td>
<td>Integer</td>
<td>Number of levels already processed within the context of the current SAVE operation.</td>
</tr>
<tr>
<td>LevelsLeft</td>
<td>Integer</td>
<td>Number of levels yet to be processed within the context of the current SAVE operation. This count does not include the Finished pass through the Update event.</td>
</tr>
<tr>
<td>RetVal</td>
<td>Integer</td>
<td>The automatic updating of the master table for the current level, which occurs after the Update event for that particular level, can be suppressed by setting RetVal to the NoAction symbolic constant defined in Solomon.VBTools.vb. The entire SAVE operation can be aborted by setting RetVal either to a Microsoft Dynamics SL message number or the ErrNoMess symbolic constant defined in Solomon.VBTools.vb.</td>
</tr>
</tbody>
</table>

**See Also**

MGetLineStatus Function, SetLevelChg Statement, TestLevelChg Function
Methods

关于Box Method

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCombo, DSLCheck, DSLGrid, DSLOption, DSLUpdate

Description
Displays the About Box. The About Box is usually used to identify the version of software.

Drag Method

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCombo, DSLCheck, DSLGrid, DSLOption

Description
This method is reserved for Microsoft and should not be used.

Move Method

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCombo, DSLCheck, DSLGrid, DSLOption

Description
This method is reserved for Microsoft and should not be used.

SetFocus Method

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCombo, DSLCheck, DSLGrid, DSLOption

Description
This method is reserved for Microsoft and should not be used. Use the ApplSetFocus() API instead.
ShowWhatsThis Method

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCombo, DSLCheck, DSLGrid, DSLOption

Description
Displays a selected topic in a help file using the What’s This popup.

Syntax
object.ShowWhatsThis

Remarks
The object placeholder represents an object expression that evaluates to an object. The ShowWhatsThis method is useful for providing context-sensitive help from a context menu in your application. The method displays the topic identified by the WhatsThisHelpID property of the object specified in the syntax.

See Also
WhatsThisHelpID Property

ZOrder Method

Applies To
DSLDate, DSLFloat, DSLInteger, DSLMaskedText, DSLCombo, DSLCheck, DSLGrid, DSLOption

Description
This method is reserved for Microsoft and should not be used.
API Function Calls

ApplGetParms Function

Description
Retrieve a parameter passed by another Microsoft Dynamics SL SDK application.

Syntax
ParmValue = ApplGetParms()

Remarks
The ApplGetParms statement can be used to retrieve parameters which were originally passed by another Microsoft Dynamics SL SDK application using either the CallApplic or CallApplicWait statement. Multiple parameters can be retrieved by making successive calls to ApplGetParms.

If the calling application passed parameters via named parameter sections, using the ApplSetParmValue statement in conjunction with either the CallApplic or CallApplicWait statement, then ApplGetParms will only be able to retrieve parameters from the default Microsoft Dynamics SL SDK section. The ApplGetParmValue function is the only means by which the called application can retrieve parameters from any named parameter section other than the default Microsoft Dynamics SL SDK section.

The ApplGetParms function uses the following argument:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParmValue</td>
<td>String</td>
<td>The actual value of the next parameter to be retrieved.</td>
</tr>
</tbody>
</table>

See Also
ApplGetParmValue Function, ApplSetParmValue Statement, CallApplic Statement, CallApplicWait Statement

Example
The following code snippet illustrates how to call the Release Payroll Batch process to release a particular batch.

```vbscript
Dim ParmStr As String
ParmStr = bBatch.Module + PRMSEP + bBatch.BatNbr + PRMSEP + bBatch.EditScrnNbr + PRMSEP + bBatch.Status
Call CallApplicWait("0240000", ParmStr)
```

The following code snippet illustrates how the Release Payroll Batch process can subsequently retrieve the parameters using the ApplGetParms statement from within its Form1_Load event.

```vbscript
Dim Parm_Module As String 'FIRST command line parameter
Dim Parm_BatNbr As String 'SECOND command line parameter
Dim Parm_EditScrnNbr As String 'THIRD command line parameter
Dim Parm_Status As String 'FOURTH command line parameter
Parm_Module = ApplGetParms()
Parm_BatNbr = ApplGetParms()
Parm_EditScrnNbr = ApplGetParms()
Parm_Status = ApplGetParms()
```
ApplGetParmValue Function

Description
Retrieve a parameter passed by another Microsoft Dynamics SL SDK application.

Syntax
\[ \text{ParmValue} = \text{ApplGetParmValue} \left( \text{ParmSection}, \text{ParmName} \right) \]

Remarks
Parameters passed to a Microsoft Dynamics SL SDK application can be retrieved via one of two
different methods: \text{ApplGetParms} and \text{ApplGetParmValue}. These functions differ in that \text{ApplGetParms}
does not support multiple parameter sections whereas \text{ApplGetParmValue} does provide this more
sophisticated functionality. Consequently, \text{ApplGetParmValue} is the only means by which the called
application can retrieve parameters from any named parameter section other than the default
Microsoft Dynamics SL SDK section. For example, if the calling application sends a parameter
specifically designated as a BSL parameter, only the \text{ApplGetParmValue} function can be used to
retrieve that particular parameter since the BSL section name can be explicitly queried via the
ParmSection argument.

Named parameter sections facilitate the elimination of conflicts which can occur in the destination
program when the application itself as well as custom BSL code added via the Customization Manager
are both attempting to receive different parameters. For example, the Accounts Payable Document
Maintenance screen can optionally receive two parameters that facilitate drill-down functionality:
Reference Number and Vendor ID. The Form_Load event always calls \text{ApplGetParms} once to
determine if any parameters have been passed to the application. If even one parameter exists it is
assumed that it is the Reference Number and therefore the application calls \text{ApplGetParms} again
expecting the next parameter to be the Vendor ID. If this screen is subsequently customized by adding
calls to \text{ApplGetParms} using BSL code, an operational conflict will occur. If such an application were to
be called with only one parameter, designed to be received by the custom BSL code, it would instead
be received by the call to \text{ApplGetParms} performed by the underlying application. Consequently the
call to \text{ApplGetParms} in BSL code would not return any parameter value at all.

\text{ApplSetParmValue} and \text{ApplGetParmValue} overcome this operational conflict by facilitating the usage
of named parameter sections. Using this more sophisticated method, parameters can be passed
directly to the application itself and to custom BSL code using the two standard section names
declared in Solomon.VBTools.vb (for example, \text{PRMSECTION_VBRDT} and \text{PRMSECTION_BSL}).
Parameter sections are not, however, limited to these two standard section names. Thus, for example,
“{XYZ Section}” is a valid section name. The brackets are required since parameter sections
themselves are handled similar to section names within an .INI file.

In the previously mentioned example, a custom parameter could be sent to the Accounts Payable
Document Maintenance screen in the BSL parameter section such that only calls to
\text{ApplGetParmValue} specifically requesting BSL parameters would retrieve the parameter.
The **ApplGetParmValue** statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParmValue</td>
<td>String</td>
<td>The actual value of the parameter being retrieved.</td>
</tr>
<tr>
<td>ParmSection</td>
<td>String</td>
<td>Name of the section within the temporary parameter file from which the parameter should be retrieved. Any section name can be used such as &quot;XYZ Section,&quot; provided the calling application utilized a so named parameter section. Solomon.VBTools.vb contains three symbolic constants defining standard section names: PRMSECTION_VBRDT, PRMSECTION_BSL and PRMSECTION_TI. PRMSECTION_TI is reserved for usage in conjunction with Transaction Import. By default, the parameter will be retrieved from the section represented by PRMSECTION_VBRDT if this argument is left blank.</td>
</tr>
<tr>
<td>ParmName</td>
<td>String</td>
<td>Logical name of the parameter being retrieved. By default, parameter names are sequentially numbered (for example, PRM01, PRM02,...PRM99) if they were not explicitly named by the call to <strong>ApplSetParmValue</strong> in the calling application.</td>
</tr>
</tbody>
</table>

**See Also**

ApplGetParms Function, ApplSetParmValue Statement, CallApplic Statement, CallApplicWait Statement

**Example**

The following example will illustrate how to pass parameters to a Microsoft Dynamics SL SDK application and custom BSL code at the same time and avoid conflicts between the two.

**Code in the calling application**

```vbscript
Call ApplSetParmValue(PRMSECTION_VBRDT, "Batch Nbr", "000001")
Call ApplSetParmValue(PRMSECTION_VBRDT, "Document Nbr", "123456")
Call ApplSetParmValue(PRMSECTION_BSL, "Example Parm", "Example Parameter To BSL Code")

'Call another Microsoft Dynamics SL SDK application
CallApplicWait( "RDTAPP", "")
```

**Code in the standard Microsoft Dynamics SL SDK application (non-customized code) receiving the standard parameters.**

```vbscript
Dim Parm_BatchNbr As String
Dim Parm_DocumentNbr As String

Parm_BatchNbr = ApplGetParmValue(PRMSECTION_VBRDT, "Batch Nbr"
Parm_DocumentNbr = ApplGetParmValue(PRMSECTION_VBRDT, "Document Nbr")
```

**Basic Script code, overlaying the standard Microsoft Dynamics SL SDK application, designed to retrieve custom parameters:**

```vbscript
Dim Parm_CustomParm As String

Parm_CustomParm = ApplGetParmValue(PRMSECTION_BSL, "Example Parm")
```
ApplGetReturnParms Function

Description
Retrieve a parameter returned from a now terminated secondary application.

Syntax
\[ ParmValue = \text{ApplGetReturnParms()} \]

Remarks
If a Microsoft Dynamics SL SDK application needs to pass parameters back to the program from which it was originally called it can do so using one of the parameters to \text{ScreenExit}. When control subsequently returns to the calling application, it can issue one or more calls to \text{ApplGetReturnParms} to successively retrieve each individual parameter.

The \text{ApplGetReturnParms} function has the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParmValue</td>
<td>String</td>
<td>The actual value of the parameter being retrieved from a now terminated secondary program.</td>
</tr>
</tbody>
</table>

See Also
ScreenExit Statement

ApplInit Statement

Description
Initialize various application resources as well as initiate a dynamic link with the Microsoft Dynamics SL Parent.

Syntax
Call ApplInit

Remarks
This call is required in all applications developed with Microsoft Dynamics SL SDK.

See Also
ScreenExit Statement, ScreenInit Statement

Example
The following code illustrates the basic order of calls that are made from within the Form_Load event of Form1. Some of the calls are optional depending on the requirements of a particular application but the \text{ApplInit} call is always required.

Sub Form_Load ()

'Load application subform(s)
'Call LoadForm( SubFormName)

'Call to Initialize the Application (required in all applications)
Call ApplInit
'SetAddr call(s)
  'Call SetAddr(LEVEL0, "bTableName", bTableName, nTableName)

'SqlCursor call(s)
  'Call SqlCursor(CSR_TableName, LEVEL0)

    'Call to Initialize the Screen (required in all applications)
    Call ScreenInit

'DetailSetup call for simple grid
'MemHandle_Spread1 = DetailSetup(CSR_TableName, Spread1_TableName,\n    PNULL, bTableName, PNULL, PNULL, PNULL)

End Sub

**ApplSetFocus Statement**

**Description**
Set focus to a designated control.

**Syntax**
Call **ApplSetFocus**( TargetCtrl)

**Remarks**
**ApplSetFocus** is the preferred method to explicitly set focus to a target control. Usage of the Visual Basic .SetFocus method will cause a fatal Visual Basic error if the target control is disabled or invisible. Developers must always remember that the design time property setting of the target control cannot be guaranteed to always retain the same value at runtime. For example, the target control may be both visible and enabled in the standard application, and therefore, the .SetFocus will appear to work properly during testing. However, the end user may subsequently apply a customization which among other things disables the target control, thereby uncovering a subtle flaw in the underlying application with regards to its usage of the .SetFocus method.

The ApplSetFocus statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TargetCtrl</td>
<td>Control</td>
<td>Control to which focus should be moved.</td>
</tr>
</tbody>
</table>

**Example**
Call **ApplSetFocus**(Form1.cDiscBal)
ApplSetParmValue Statement

Description
Add one additional parameter to the list of all parameters which will be sent to the next application called by either the CallApplic or CallApplicWait statement.

Syntax
Call ApplSetParmValue(ParmSection, ParmName, ParmValue)

Remarks
There are two different methods for one Microsoft Dynamics SL SDK application to start another application: CallApplic and CallApplicWait. Regardless of which statement is used there are also two different methods for the calling program to pass parameters to the called program.

The first method is to pass the parameters to the called program using the argument to CallApplic and CallApplicWait specifically designed for this purpose. Parameters passed via this method are all grouped together and passed directly to the called application via the physical command line itself. Consequently, under this method the size and/or number of parameters is limited to the maximum command line length less the number of bytes used for internal requirements, which can vary based on the situation.

A more robust method of passing parameters is to use the ApplSetParmValue statement in conjunction with either the CallApplic or CallApplicWait statement. The principle advantage of using this method is that it allows the calling application to group parameters into named sections and explicitly label individual parameters using unique parameter names. Grouping parameters into named sections eliminates conflicts that will occur in the called program when the application itself as well as custom BSL code added via the Customization Manager are both attempting to receive different parameters. See the ApplGetParmValue function for a more detailed explanation of these potential conflicts.

The first call to ApplSetParmValue will create a temporary destination parameter file and place the first parameter in that file. By default, this temporary file will be created in the WINDOWS directory. This default can be overridden using the TempDirectory entry in the [Miscellaneous] section of the SOLOMON.INI file. Following is an example of the C:\TEMP directory specified as the parameter file directory in SOLOMON.INI:

[Miscellaneous]
TempDirectory=C:\TEMP

Subsequent calls to ApplSetParmValue will write additional parameters to the same temporary destination parameter file. The fully qualified filename of the completed parameter file will then be passed to the called program by the CallApplic and CallApplicWait statements. Once the called program has successfully loaded, it can call either ApplGetParms or ApplGetParmValue to retrieve the various parameters passed from the calling program. When the called program terminates, the temporary destination parameter file will automatically be deleted. ApplSetParmValue is only designed to facilitate parameter passing to other applications developed with Microsoft Dynamics SL SDK.
The **ApplSetParmValue** statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParmSection</td>
<td>String</td>
<td>Name of the section within the temporary parameter file to which the new parameter should be added. Any section name can be used such as “XYZ Section”. Solomon.VBTools.vb contains three symbolic constants defining standard section names: PRMSECTION_VBRDT, PRMSECTION_BSL and PRMSECTION_TI. PRMSECTION_TI is reserved for usage in conjunction with Transaction Import. By default, the new parameter will added to the section represented by PRMSECTION_VBRDT if this argument is left blank.</td>
</tr>
<tr>
<td>ParmName</td>
<td>String</td>
<td>Name assigned to the new parameter. Any name can be assigned to a parameter such as “Batch Number”. By default, the new parameter will be assigned the next sequentially numbered parameter name (for example, PRM01, PRM02....PRM99) if this argument is left blank.</td>
</tr>
<tr>
<td>ParmValue</td>
<td>String</td>
<td>The actual value of the new parameter.</td>
</tr>
</tbody>
</table>

**See Also**

ApplGetParms Function, ApplGetParmValue Function, CallApplic Statement, CallApplicWait Statement

**Example**

The following example will illustrate two different methods of calling ROI to display the Vendor List report on the screen for all Vendors having a balance greater than zero.

Pass parameters to ROI via one large parameter argument to **CallApplicWait**. This method will not work with a large WHERE clause since the entire contents of ParmStr must fit on the physical command line.

```vba
Dim ParmStr As String

ParmStr = "03670/RUN" + PRMSEP
ParmStr = ParmStr + "03670S/FORMAT" + PRMSEP
ParmStr = ParmStr + "Vendor.CurrBal > 0/WHERE" + PRMSEP
ParmStr = ParmStr + "/PSCRN"
Call CallApplicWait("ROI", ParmStr)
```

Pass parameters to ROI using **ApplSetParmValue** in conjunction with **CallApplicWait**. Using this method, the report will run properly regardless of the length of the WHERE clause.

```vba
Call ApplSetParmValue(PRMSECTION_VBRDT, "", "03670/RUN")
Call ApplSetParmValue(PRMSECTION_VBRDT, "", "03670S/FORMAT")
Call ApplSetParmValue(PRMSECTION_VBRDT, "", "Vendor.CurrBal > 0/WHERE")
Call ApplSetParmValue(PRMSECTION_VBRDT, "", "/PSCRN")
Call CallApplicWait("ROI", "")
```
AutoNbrDefault Function

Description
Set up auto-numbering.

Syntax
\[ \text{RetVal} = \text{AutoNbrDefault}(\text{Ctrl1}, \text{SqlProcName}, \text{Ctrl2}, \text{Ctrl3}) \]

Remarks
This call will set up the ability for a specified \texttt{DSLMaskedText} control representing a unique and sequentially numbered key field to be defaulted from a “last number used” field in a setup record whenever a new record is saved.

This call needs to be placed in Form_Load in a very specific location relative to the other calls. See the example for placement.

The application program must specify the keyfield control that will be defaulted with the uniquely numbered string value and must also specify the stored procedure that will retrieve the correct default information from a setup record within the database. The stored procedure should retrieve two fields of information from the setup record:

- The auto number flag associated with the field. This value must be a two byte integer having a value of 0 if auto-numbering is turned off and 1 if auto-numbering is turned on.
- The “last number used” field. This must be a string field exactly ten bytes in length. The value of this string field can be less than ten bytes but the field itself must be exactly ten bytes.

If the automatically generated number needs to be propagated to records at other levels, such as document or detail records, then the corresponding control handle(s) should be passed as Ctrl2 and Ctrl3. Both of these fields must be PNULL if no propagation is needed.

When the Update_Update event fires, the value from the setup record will be applied to records at the level currently being saved. The setup record itself will be updated as well to reflect the new “last number used” value.

The \texttt{AutoNbrDefault} function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RetVal</td>
<td>Integer</td>
<td>Zero if auto-numbering was set up successfully. Non-zero if an error occurred.</td>
</tr>
<tr>
<td>Ctrl1</td>
<td>Control</td>
<td>Keyfield control whose value for new records will be uniquely generated whole numbers in a string format.</td>
</tr>
<tr>
<td>SqlProcName</td>
<td>String</td>
<td>Name of the SQL stored procedure returning the auto number flag and the “last number used” fields from a Setup record.</td>
</tr>
<tr>
<td>Ctrl2</td>
<td>Control</td>
<td>First control into which auto-numbered values should be propagated. PNULL if no propagation is required.</td>
</tr>
<tr>
<td>Ctrl3</td>
<td>Control</td>
<td>Second control into which auto-numbered values should be propagated. PNULL if no propagation is required.</td>
</tr>
</tbody>
</table>

Example
The following code snippet illustrates how automatic Batch numbering is implemented in the Payroll Manual Check screen.

```vba
Sub Form_Load ()
    Call ApplInit
    Call SetAddr(LEVEL0, "bBatch", bBatch, nBatch)
    Call SetAddr(LEVEL1, "bPRDoc", bPRDoc, nPRDoc)
```
Call SetAddr(LEVEL2, "bEmployee", bEmployee, nEmployee)
Call SetAddr(LEVEL3, "bPRTran1", bPRTran_Earn, nPRTran_Earn)
Call SetAddr(LEVEL4, "bPRTran2", bPRTran_Ded, nPRTran_Ded)

'Miscellaneous SetAddr calls
Call SetAddr(NOLEVEL, "bCASetup", bCASetup, nCASetup)
Call SetAddr(NOLEVEL, "bCashAcct", bCashAcct, nCashAcct)
Call SetAddr(NOLEVEL, "bDeduction", bDeduction, nDeduction)
Call SetAddr(NOLEVEL, "bEarnType", bEarnType, nEarnType)
Call SetAddr(NOLEVEL, "bPayGroup", bPayGroup, nPayGroup)
Call SetAddr(NOLEVEL, "bPayPeriod", bPayPeriod, nPayPeriod)
Call SetAddr(NOLEVEL, "bPRSetup", bPRSetup, nPRSetup)

'Setaddr calls for controls whose fieldname is not a database field
Call SetAddr(NOLEVEL, "bBatchHandling", bBatchHandling, PNULL)
Call SetAddr(NOLEVEL, "bTempTotals", bTempTotals, PNULL)

'Initialize all cursors used by this screen.
Call Cursor_Init

'Load CASetup record to determine whether or not Cash Manager is
'installed and set up.
serr1 = SqlFetch1(CSR_CASetup, "CASetup_All", bCASetup)

'Load necessary PRSetup record BEFORE ScreenInit. That way any fields
' which need this record for default settings will have the values
' available when ScreenInit performs an implied NEW.
serr1 = SqlFetch1(CSR_PRSetup, "PRSetup_All", bPRSetup)

Call ScreenInit

'Setup for auto batch numbering. Note: Transactions have their BatNbr
'initialized during PRTran_Update().
serr1 = AutoNbrDefault(cBatNbrH, "PRAutoBatchNbr", cbatnbr_PRDoc, CNULL)

Mem_Handle_PRTran_Earn = DetailSetup(CSR_PRTran_Earn_DBNav, Spread_Earn, NULL, bPRTran_Earn, bEarnType, PNULL, PNULL)
Mem_Handle_PRTran_Ded = DetailSetup(CSR_PRTran_Ded_DBNav, Spread_Ded, NULL, bPRTran_Ded, bDeduction, PNULL, PNULL)

End Sub
**Button_Form_Change Statement**

**Description**
Allows an application to set the form change information triggered by a button click event.

**Syntax**
Sub **Button_Form_Change** *(ButtonName, Level, FromForm, ToForm)*

**Remarks**
This statement will pass information needed for Transaction Import to automatically generate button press statements in the control file. This call can be placed at the end of **FormLoad**.

The **Button_Form_Change** statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ButtonName</td>
<td>String</td>
<td>Name of the command button control. Its click event causes the form change.</td>
</tr>
<tr>
<td>Level</td>
<td>Integer</td>
<td>The level that forms are currently on.</td>
</tr>
<tr>
<td>FromForm</td>
<td>Integer</td>
<td>The form name the button is leaving.</td>
</tr>
<tr>
<td>ToForm</td>
<td>Integer</td>
<td>The form name the button brings up.</td>
</tr>
</tbody>
</table>

**See Also**
**Button_Level_Change Statement**

**Button_Level_Change Statement**

**Description**
Allows an application to set the level change information triggered by a button click event.

**Syntax**
Sub **Button_Level_Change** *(ButtonName, FromLevel, ToLevel)*

**Remarks**
This statement will pass information needed for Transaction Import to automatically generate button press statements in the control file. This call can be placed at the end of **FormLoad**.

The **Button_Level_Change** statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ButtonName</td>
<td>String</td>
<td>Name of the command button control. Its click event causes the level change.</td>
</tr>
<tr>
<td>FromLevel</td>
<td>Integer</td>
<td>The level number the button is leaving.</td>
</tr>
<tr>
<td>ToLevel</td>
<td>Integer</td>
<td>The level number the button brings up.</td>
</tr>
</tbody>
</table>

**See Also**
**Button_Form_Change Statement**
CallApplic Statement

Description
Run another application.

Syntax
Call CallApplic(ApplicName, ParmStr)

Remarks
This statement allows the application to call, and optionally pass parameters to, another application. Parameters passed via this method are all grouped together and passed directly to the called application via the physical command line itself. A more robust method of passing parameters is available in conjunction with the ApplSetParmVal statement.

The called application can subsequently retrieve the parameters via one of two different methods: ApplGetParms and ApplGetParmValue.

The called application will not be modal when the CallApplic statement is used. If the called application is required to be modal then the CallApplicWait statement should be utilized.

The CallApplic statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplicName</td>
<td>String</td>
<td>Name of the program to run. This name should not include the .exe file name extension.</td>
</tr>
<tr>
<td>ParmStr</td>
<td>String</td>
<td>Parameter value(s) to be sent to the called application. Multiple parameters can be passed by separating each individual parameter with the PRMSEP symbolic constant defined in Solomon.VBTools.vb.</td>
</tr>
</tbody>
</table>

See Also
ApplGetParms Function, ApplGetParmValue Function, ApplSetParmValue Statement, CallApplicWait Statement

Example
The following simple line of code illustrates how the current application can call a program named “PROGRAMB” with two parameters.

Call CallApplic("PROGRAMB", "Parm1" + PRMSEP + "Parm2")
CallApplicWait Statement

Description
Run another program as a modal application.

Syntax
Call CallApplicWait(ApplicName, ParmStr)

Remarks
This statement allows the application to call, and optionally pass parameters to, another application. Parameters passed via this method are all grouped together and passed directly to the called application via the physical command line itself. A more robust method of passing parameters is available in conjunction with the ApplSetParmValue statement.

The called application can subsequently retrieve the parameters via one of two different methods: ApplGetParms and ApplGetParmValue.

The called application will be modal when the CallApplicWait statement is used. If the called application should not be modal then the CallApplic statement should be utilized.

The CallApplicWait statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplicName</td>
<td>String</td>
<td>Name of the program to run. This name should not include the .exe file name extension.</td>
</tr>
<tr>
<td>ParmStr</td>
<td>String</td>
<td>Parameter value(s) to be sent to the called application. Multiple parameters can be passed by separating each individual parameter with the PRMSEP symbolic constant defined in Solomon.VBTools.vb.</td>
</tr>
</tbody>
</table>

See Also
ApplGetParms Function, ApplGetParmValue Function, ApplSetParmValue Statement, CallApplic Statement

Example
The following code snippet illustrates how to call the Release Payroll Batch process to release a particular batch:

```vbs```
Dim ParmStr As String

ParmStr = bBatch.Module + PRMSEP + bBatch.BatNbr + PRMSEP + PRMSEP + '
' + bBatch.EditScrnNbr + PRMSEP + bBatch.Status

Call CallApplicWait("0240000", ParmStr)
```vbs```
CallChks Statement

Description
Perform error checking for the specified fields.

Syntax
ErrRet = CallChks( Form, BeginControl, EndControl, CallApplicChkEvent, ChkDetailFields)

Remarks
This statement allows the application to perform error checking on a range of controls. By default, error checking will not occur on a control unless it has been changed since the last fetch. You can force error checks to occur regardless of whether the field changed or not by setting parameter CallApplicChk to True. Controls on Detail levels will be skipped unless parameter ChkDetailFields is True. Error checks will be performed on each control in the range, but checking will terminate when an error status is indicated on one of the controls. If all controls pass error checking, 0 is returned. Otherwise non-zero is returned.

The CallChks statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form</td>
<td>Control</td>
<td>Form handle — can be NULL to include all loaded forms.</td>
</tr>
<tr>
<td>BeginControl</td>
<td>Control</td>
<td>Handle of the first control to error check. Can be NULL to include all loaded controls for the specified form in the search.</td>
</tr>
<tr>
<td>EndControl</td>
<td>Control</td>
<td>Handle of the last control to error check. Can be NULL to include all loaded controls for the specified form in the search.</td>
</tr>
<tr>
<td>CallApplicChk</td>
<td>Integer</td>
<td>If True, calls to the Chk() event(s) regardless of whether or not the field value changed.</td>
</tr>
<tr>
<td>ChkDetailFields</td>
<td>Integer</td>
<td>If True, calls the Chk() event(s) for controls that are associated with an DSLGrid control.</td>
</tr>
<tr>
<td>Return Value</td>
<td>Integer</td>
<td>Zero if all controls were OK. Non-zero on first control found in error.</td>
</tr>
</tbody>
</table>

See Also
ApplGetParms Function, ApplGetParmValue Function, ApplSetParmValue Statement, CallApplic Statement

Example
MyErrVal=CallChks(Form1, cbatch, crefnbr, True, False)
ChkSqlException Function

Description
Obtains a value that indicates whether the application is handling a database error.

Syntax
\[ AppHandling = \text{ChkSqlException}(ErrVal) \]

Remarks
The application indicates that it will handle its own database errors by calling \textit{SqlErrException}. \textit{ChkSqlException} can be used to check the current status of whether an error is being returned to the application.

Return Value:
- LTRUE – Automatic error handling is disabled for this error.
- LFALSE – Automatic error handling is enabled for this error.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ErrVal</td>
<td>Integer</td>
<td>Error number to be checked</td>
</tr>
</tbody>
</table>

See Also
\textit{SqlErr Function, SqlErrException Statement}

Example
In an application that enables and disables automatic handling of deadlock errors in various sections of code, calling \textit{ChkSqlException} before re-enabling automatic handling can prevent \textit{SqlErrException} from displaying the 10073 error message.

```
Const DEADLOCK As Short = 1205
If ChkSqlException(DEADLOCK) = LTRUE Then
    Call SqlErrException(EXCEPTION_OFF, DEADLOCK)
End If
```
ChkCuryRateType Function

Description
Standard routine for checking the validity of a Rate Type value. Checks whether a RateType record exists for the desired Rate Type value. If a RateType record is not found, the routine returns the database error code. A return value of 0 indicates success.

Syntax

\[ IntVar = \text{ChkCuryRateType}(b\text{RateType}) \]

Remarks

ChkCuryRateType is typically called by the application to check the validity of a vendor’s or customer’s default Rate Type, before copying it to the Currency Selection Form. Since the same logic was required in a number of applications, it was provided as a standard routine, to minimize the amount of currency code in the applications.

The ChkCuryRateType statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IntVar</td>
<td>Integer</td>
<td>Any integer variable.</td>
</tr>
<tr>
<td>bRateType</td>
<td>String</td>
<td>Location of Rate Type value to be checked.</td>
</tr>
</tbody>
</table>

Example

The following example shows a portion of the Vendor Chk event for an application being enabled for multi-currency processing.

```vba
Sub cvendid_Chk (chkstrg As String, retval As Integer)
Dim maintflg As Integer

'Fetch vendor record to display Name, status and Terms
retval = pvchkfetch1(cvendid, C3, chkstrg, bVendor)

'Currency Manager
If retval = 0 Then
    'If multi-currency entry is activated...
    If bCMSsetup.MCActivated = LTRUE Then
        'If Vendor Currency ID is specified, and
        'is different from Batch Currency ID...
        If Trim$(bVendor.CuryID) <> "" And Trim$(bVendor.CuryID) <> Trim$(bCuryInfo.TranCuryId) Then
            'Give error or warning, depending upon
            'setup option.
            If bCMSsetup.APCuryOverride = LFALSE Then
                retval = 6090
                Exit Sub
            Else
                Call mess(6091)
            End If
        End If
    End If
End If
```
'If Tran Currency is different from Base Currency...
If Trim$(bCurlyInfo.TranCurlyId) <> Trim$(bPES.BaseCurlyID) Then
    'If Vendor Rate Type is specified, and
    'is different from Batch Rate Type...
    If Trim$(bVendor.CurlyRateType) <> "" And
    Trim$(bVendor.CurlyRateType) <> Trim$(bCurlyInfo.RateType) Then
        'Check whether Vendor Rate Type exists
        serr1 = ChkCurlyRateType(bVendor.CurlyRateType)
        If serr1 <> 0 Then
            'If not, exit with error
            retval = 6092
            Exit Sub
        Else
            'If it does, try to retrieve Rate Table entry
            Call CurlyRateTypeSet(bVendor.CurlyRateType)
            Call GetCurlyRate
        End If
    End If
End If
End If
End If
End If
End Sub
CurrencyField Statement

Description
Identifies Transaction Currency Controls and their corresponding Base Currency fields. The Flags parameter specifies whether or not the Base Currency field value should be recalculated before display. The information passed in the parameters is stored in the task memory of the application, and is used whenever Currency Rate Information is changed within the Currency Selection Form (opened from the Toolbar) or the Currency View is changed by pressing the Currency View button on the Toolbar.

Syntax
Call CurrencyField(hctlTranCuryFld, bBaseCuryFld, Flags)

Remarks
Add a CurrencyField call to the Form_Load event for each application control which you wish to give automatic currency calculation and display capabilities. This typically includes all controls used to enter a monetary value. The parameter information is saved by SWIM and used to redisplay monetary values when the user changes Currency Rate Information within the Currency Selection Form or “toggles” between the Transaction Currency Amounts and Base Currency Amounts, using the Currency View button.

Note: The FieldName property of the monetary control should typically be set to the database field which will hold the Transaction Currency value in the database. In record structures, the Transaction Currency field typically has a “Cury” prefix. For example, in APTran, the Transaction Currency transaction amount would be stored in CuryTranAmt. The corresponding Base Currency amount is typically stored in a similar field name without the “Cury” prefix. For example, in APTran, the Base Currency transaction amount is stored in TranAmt. Since you will usually want both the Transaction Currency amount and the Base Currency Amount stored in the database, you will typically specify the control associated with the Transaction Currency field as the first parameter in the CurrencyField call, and the location of the Base Currency field as the second parameter in the CurrencyField call.

However, it is possible to use the handle of any DSLFloat control and the location of any Double variable field as parameters.

You will usually want the Base Currency value to be automatically calculated or recalculated when the user changes the Currency Rate Information in the Currency Selection form and when the user “toggles” between Transaction Currency and Base Currency views. (Whenever the Base Currency value is recalculated, Currency Rate Information is also updated in the fields specified by CurrencyInfo for the Update Level which contains the monetary field.) For the typical case, the Flags parameter should be set to CURY_BASE_CALC.

There will be some cases when monetary fields are displayed for inquiry only, and you will not want the Base Currency values changed in any way. In those cases the Flags parameter should be set to CURY_BASE_NOCALC. The same application may need to calculate Base Currency in some situations, but not others. For example, when you retrieve an open document, you may want to be able to change monetary amounts and recalculate Base Currency values, but when you retrieve a released document, you may want to suppress changes. CurrencyFieldCalcSet is used to reset the Flags value while running.

CurrencyField calls should follow ScreenInit and DetailSetup calls. CurrencyField calls should be run, even if multi-currency processing is not enabled, to insure that database fields are properly populated with Base Currency values.
The CurrencyField statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hctlTranCuryFld</td>
<td>Control</td>
<td>Handle of control used to enter Transaction Currency value.</td>
</tr>
<tr>
<td>bBaseCuryFld</td>
<td>Double</td>
<td>Location of database field or global variable used to hold automatically-calculated Base Currency value.</td>
</tr>
<tr>
<td>Flags</td>
<td>Integer</td>
<td>Indicates whether or not Base Currency value should be recalculated before display. Set to CURY_BASE_CALC, if recalculation is desired. Set to CURY_BASE_NOCALC to suppress recalculation.</td>
</tr>
</tbody>
</table>

See Also

CurrencyInfo Statement, CuryFieldCalcSet Statement

Example

The following example shows a portion of the Form_Load event for an application being enabled for multi-currency processing.

Call ApplInit
Call setaddr(LEVEL0, "bBatch", bBatch, nBatch)
Call setaddr(LEVEL1, "bAPDoc", bAPDoc, nAPDoc)
Call setaddr(LEVEL2, "bAPTran", baptran, nAPTran)
Call setaddr(NOLEVEL, "bCuryInfo", bCuryInfo, nCuryInfo)
Call setaddr(NOLEVEL, "bCMSetup", bCMSetup, nCMSetup)

Call SqlCursor(C1, LEVEL0) 'Batch
Call SqlCursor(C2, LEVEL1) 'APDoc
Call SqlCursor(C3, LEVEL2) 'APTran

'Set up Currency Info Fields for CM
'Note - these calls need to be made even if CM is not installed.
Call CurrencyInfo(LEVEL0, bBatch, "Batch", "CuryId", "CuryMultDiv", "CuryRateType", bBatch.CuryEffDate, bBatch.CuryRate)
Call CurrencyInfo(LEVEL1, bAPDoc, "APDoc", "CuryId", "CuryMultDiv", "CuryRateType", bAPDoc.CuryEffDate, bAPDoc.CuryRate)
Call CurrencyInfo(LEVEL2, bAPTran, "APTran", "CuryId", "CuryMultDiv", NULLDATE, bAPTran.CuryRate)

'Initialize and Enable Currency Manager Controls
serr1 = sqlfetch1(c7, "CMSSetup_All", bCMSetup)
'If Currency Manager is installed and Multi-Currency is activated ...
If serr1 = 0 And bCMSetup.MCActivated = LTRUE Then
  'Activate Currency Buttons on Toolbar
  Call SetButton(CurySelButton + CuryTogButton, AllLevels, True)
  'Set default rate type
  Call CuryRateTypeSet(bCMSetup.APRtTpDflt)
'If rate type override is not allowed, 
'disable rate type field in Currency Selection form
If bCMSetup.APRtTpOverride = LFALSE Then
    Call CurySelFieldEnable(CURYSEL_RATETYPE, False)
    SaveCURYSEL_RATETYPE = CURYSEL_RATETYPE
Else
    SaveCURYSEL_RATETYPE = 0
End If
End If
ScreenInit
'Voucher Detail Grid
mhandle = detailsetup(C3, Form1.Spread1, baptran.linenbr, baptran, PNULL, PNULL, PNULL)

'Set up Currency Fields for CM
'Note - these calls need to be made even if CM is not installed.
    Call CurrencyField(Form1.cctrltot, bBatch.ctrltot, CURY_BASE_CALC)
    Call CurrencyField(Form1.corigdocamt, bAPDoc.OrigDocAmt, CURY_BASE_CALC)
    Call CurrencyField(Form1.cdiscbal, bAPDoc.DiscBal, CURY_BASE_CALC)
    Call CurrencyField(Form1.cdrtot, bBatch.DrTot, CURY_BASE_CALC)
    Call CurrencyField(Form1.cdocbal, bAPDoc.DocBal, CURY_BASE_CALC)
    Call CurrencyField(Form1.cunitprice, baptran.unitprice, CURY_BASE_CALC)
    Call CurrencyField(Form1.ctranamt, baptran.TranAmt, CURY_BASE_CALC)
CurrencyInfo Statement

Description
Saves the addresses of the application fields where Currency Rate Information will be stored for each update level. Saving the addresses allows the Currency Rate Information to be moved between the Currency Selection Form (opened from the Toolbar) and the application records, using <code>CuryInfoGet</code> and <code>CuryInfoSet</code> calls.

Syntax
Call <code>CurrencyInfo</code>(<i>Level, RecAddr, RecName, IDFldName, MDFldName, RateTypeFldName, Effdate, Rate</i>)

Remarks
Add a <code>CurrencyInfo</code> call to the Form_Load event for each Update Level which needs to update the database with Currency Selection Information. Possible fields to be updated are Currency ID, Multiply/Divide, Rate Type, Effective Date, and Rate. Some record structures only store the Currency ID, Multiply/Divide, and Rate values. An empty string and the NULLDATE constant are used as the values of the Rate Type and Effective date parameters for record structures which do not store those fields.

<code>CurrencyInfo</code> calls should be done after <code>SetAddr</code> calls, which establish levels for record structures, but before the <code>ScreenInit</code> call. <code>CurrencyInfo</code> calls should be ran even when multi-currency processing is not enabled, to insure the Currency Rate Information fields are correctly populated for Base Currency processing.

The <code>CurrencyInfo</code> statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;i&gt;Level&lt;/i&gt;</td>
<td>Integer</td>
<td>Update Level Number corresponding to record where Currency Information will be stored in the database.</td>
</tr>
<tr>
<td>&lt;i&gt;RecAddr&lt;/i&gt;</td>
<td>Any</td>
<td>Location of database record where Currency Information will be stored in the database.</td>
</tr>
<tr>
<td>&lt;i&gt;RecName&lt;/i&gt;</td>
<td>String</td>
<td>Record Name of database record where Currency Information will be stored in the database.</td>
</tr>
<tr>
<td>&lt;i&gt;IDFldName&lt;/i&gt;</td>
<td>String</td>
<td>Field Name where Currency ID value will be stored within the database record.</td>
</tr>
<tr>
<td>&lt;i&gt;MDFldName&lt;/i&gt;</td>
<td>String</td>
<td>Field Name where Multiply/Divide Operator value will be stored within the database record.</td>
</tr>
<tr>
<td>&lt;i&gt;RateTypeFldName&lt;/i&gt;</td>
<td>String</td>
<td>Field Name where Rate Type value will be stored within the database record.</td>
</tr>
<tr>
<td>&lt;i&gt;EffDate&lt;/i&gt;</td>
<td>Integer</td>
<td>Location of field where Effective Date value will be stored within the database record.</td>
</tr>
<tr>
<td>&lt;i&gt;Rate&lt;/i&gt;</td>
<td>Double</td>
<td>Location of field where Currency Rate value will be stored within the database record.</td>
</tr>
</tbody>
</table>

See Also
<code>CuryInfoGet Statement</code>, <code>CuryInfoInit Statement</code>, <code>CuryInfoSet Statement</code>
Example
The following example shows a portion of the Form_Load event for an application being enabled for multi-currency processing. Note that the Batch and APDoc records in LEVEL0 and LEVEL1 contain all of the Currency Information fields, while the APTran record in LEVEL3 only contains Currency ID, Multiply/Divide, and Rate.

Call ApplInit
Call setaddr(LEVEL0, "bBatch", bBatch, nBatch)
Call setaddr(LEVEL1, "bAPDoc", bAPDoc, nAPDoc)
Call setaddr(LEVEL2, "bAPTran", baptran, nAPTran)
Call setaddr(NOLEVEL, "bCuryInfo", bCuryInfo, nCuryInfo)
Call setaddr(NOLEVEL, "bCMSetup", bCMSetup, nCMSetup)
Call SqlCursor(C1, LEVEL0) 'Batch
Call SqlCursor(C2, LEVEL1) 'APDoc
Call SqlCursor(C3, LEVEL2) 'APTran
'Set up Currency Info Fields for CM
'Note - these calls need to be made even if CM is not installed.
Call CurrencyInfo(LEVEL0, bBatch, "Batch", "CuryId", "CuryMultDiv", J
"CuryRateType", bBatch.CuryEffDate, bBatch.CuryRate)
Call CurrencyInfo(LEVEL1, bAPDoc, "APDoc", "CuryId", "CuryMultDiv", J
"CuryRateType", bAPDoc.CuryEffDate, bAPDoc.CuryRate)
Call CurrencyInfo(LEVEL2, bAPTran, "APTran", "CuryId", "CuryMultDiv",J
"", NULLDATE, bAPTran.CuryRate)
'Initialize and Enable Currency Manager Controls
serr1 = sqlfetch1(c7, "CMSetup_All", bCMSetup)
'If Currency Manager is installed and Multi-Currency is activated .
If serr1 = 0 And bCMSetup.MCActivated = LTRUE Then
  'Activate Currency Buttons on Toolbar
  Call SetButton(CurySelButton + CuryTogButton, AllLevels, True)
  'Set default rate type
  Call CuryRateTypeSet(bCMSetup.APRtTpDflt)
  'If rate type override is not allowed,
  'disable rate type field in Currency Selection form
  If bCMSetup.APRtTpOverride = LFALSE Then
    Call CurySelFieldEnable(CURYSEL_RATETYPE, False)
    SaveCURYSEL_RATETYPE = CURYSEL_RATETYPE
  Else
    SaveCURYSEL_RATETYPE = 0
  End If
End If
Call ScreenInit
CurlyEffDateSet Statement

Description
Sets the value of the Effective Date field in the Currency Selection Form (opened from the Toolbar).

Syntax
Call CurlyEffDateSet(bEffDate)

Remarks
This routine is typically called from the Chk events of the application date fields which are potential defaults for the Effective Date in the Currency Selection Screen. Choice of default Effective date is a Currency Manager Setup option.

Note: This routine does not automatically retrieve new rate information into the Currency Selection Form. If new rate information is desired, this call must be followed by a call to GetCurlyRate.

The CurlyEffDateSet statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bEffDate</td>
<td>Integer</td>
<td>Location of field containing new Effective Date.</td>
</tr>
</tbody>
</table>

See Also
CurlyIdSet Statement, CurlyRateTypeSet Statement, GetCurlyRate Statement

Example
The following example shows the Chk event for a Document Date field. If multi-currency entry is activated and the default Effective Date selected in Currency Manager Setup is “Document Date”, the Effective Date in the Currency Selection Form is set to the Document Date.

```vba
Sub cdocdate_Chk (chkstrg As String, retval As Integer)
    Dim tempdate As Integer

    'Currency Manager
    If bCMSetup.MCAactivated = LTRUE And bCMSetup.APRateDate = "D" Then
        Call strtodate(chkstrg, tempdate)
        If datecmp(tempdate, bCurlyInfo.EffDate) <> 0 Then
            Call CurlyEffDateSet(chkstrg)
            Call GetCurlyRate
        End If
    End If
End Sub
```
CuryFieldCalcSet Statement

Description
Allows application to reset flag value which determines whether or not Base Currency value associated with a control is recalculated prior to display. The flag value is stored in the task memory of the application, and is used whenever Currency Rate Information is changed within the Currency Selection Form (opened from the Toolbar) or the Currency View is changed by pressing the Currency View button on the Toolbar.

Syntax
Call CuryFieldCalcSet(hctlTranCuryFld, Flags)

Remarks
When CurrencyField calls are made in the Form_Load event, the information contained in the CurrencyField parameters is stored in task memory for use in automatically calculating Base Currency and "toggling" between Transaction Currency and Base Currency values. The Flags parameter value sent by the CurrencyField call indicates whether the Base Currency value should be recalculated prior to display, or just displayed. If it is necessary to change the original Flags value, as the state of the displayed Document or Batch is changed, CuryFieldCalcSet is used. To allow Base Currency to be recalculated, Flags should be set to CURY_BASE_CALC. To suppress recalculation of Base Currency, Flags should be set to CURY_BASE_NOCALC. When calculation is suppressed, “toggling” to Base Currency still displays the value previously calculated and stored in the Base Currency field.

The CuryFieldCalcSet statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hctlTranCuryFld</td>
<td>Control</td>
<td>Handle of control used to enter Transaction Currency value.</td>
</tr>
<tr>
<td>Flags</td>
<td>Integer</td>
<td>Indicates whether or not Base Currency value should be recalculated before display. Set to CURY_BASE_CALC, if recalculation is desired. Set to CURY_BASE_NOCALC to suppress recalculation.</td>
</tr>
</tbody>
</table>

See Also
CurrencyField Statement

Example
The following example shows a portion of the Chk event for a Document Reference number. If the Document has been released, no changes are allowed to Document amounts. Consequently, as an Safeguard, the code also suppresses automatic recalculation of Base Currency amounts. Please note that for an open document, CuryFieldCalcSet calls are made to insure that Base Currency calculation is re-enabled, in case it had been disabled for the previously retrieved document. This logic assumes that CurrencyField calls have been made in the Form_Load event for each control for which a CuryFieldCalcSet call is made here.

Sub crefnbrh_Chk (chkstrg As String, retval As Integer)
retval = NoAutoChk
serr1 = pvchkfetch1(PNULL, C2, chkstrg, bAPDoc)
If serr1 = 0 Then
  'Currency Manager
  If bCMSetup.MCActivated = LTRUE Then
'If batch has already been released, all Currency Selection fields should have been disabled when batch was retrieved, so do nothing here.
If bBatch.Rlsed = LFALSE Then
'If batch has not been released, but document has, disable all Currency Selection fields.
If bAPDoc.Rlsed = LTRUE Then
Call CurySelFieldEnable(CURYSEL_ALL, False)
Call CuryFieldCalcSet(Form1.corigdocamt, CURY_BASE_NOCALC)
Call CuryFieldCalcSet(Form1.cdiscbal, CURY_BASE_NOCALC)
Call CuryFieldCalcSet(Form1.cdocbal, CURY_BASE_NOCALC)
Call CuryFieldCalcSet(Form1.cunitprice, CURY_BASE_NOCALC)
Call CuryFieldCalcSet(Form1.ctranamt, CURY_BASE_NOCALC)
Call CuryInfoEnable(LEVEL1, CURY_INFO_SETONLY)
Call CuryInfoEnable(LEVEL4, CURY_INFO_SETONLY)
'Otherwise disable Tran Currency ID to disallow changing currency in existing document.
Else
Call CurySelFieldEnable(CURYSEL_ALL, True)
Call CurySelFieldEnable(CURYSEL_TRANCURYID + .J
    SaveCURYSEL_RATETYPE, False)
Call CuryFieldCalcSet(Form1.corigdocamt, CURY_BASE_CALC)
Call CuryFieldCalcSet(Form1.cdiscbal, CURY_BASE_CALC)
Call CuryFieldCalcSet(Form1.cdocbal, CURY_BASE_CALC)
Call CuryFieldCalcSet(Form1.cunitprice, CURY_BASE_CALC)
Call CuryFieldCalcSet(Form1.ctranamt, CURY_BASE_CALC)
Call CuryInfoEnable(LEVEL1, CURY_INFO_SETGET)
End If
End If

'If Currency values were previously saved in doc
display them in Currency Selection form. (Under normal circumstances, there will always be Currency values in the doc, even if transaction currency was the same as base currency.)
Call CuryInfoSet(LEVEL1)
End If
CuryInfoEnable Statement

Description
Allows application to enable or disable CuryInfoGet and CuryInfoSet calls.

Syntax
Call CuryInfoEnable(Level, Flag)

Remarks
CuryInfoGet calls are used to copy Currency Rate Information values from the Currency Selection Form (opened from the Toolbar) to previously-specified fields in the database record structure used for a particular update level in the application. CuryInfoSet calls are used to copy Currency Rate Information field values from the database record structure for a particular update level into the Currency Selection Form. The database record structure fields which will hold the Currency Rate Information are specified by CurrencyInfo calls in the Form_Load event.

CuryInfoGet and CuryInfoSet calls are typically made in several different places within the application. They are also called internally from SWIM as part of other SWIM routines. If no CuryInfoEnable calls are made, the default mode will enable both CuryInfoGet and CuryInfoSet actions. When the Batch, Document, and Transaction are in certain processing states, it may be desirable to suppress the copying of Currency Rate Information to and/or from the Currency Selection Form for one or more update levels. It would be inconvenient to use conditional logic around every CuryInfoGet and/or CuryInfoSet call made in the application, and even more difficult to conditionally run the calls made within SWIM. Instead, CuryInfoEnable is used to tell SWIM to ignore CuryInfoGet, CuryInfoSet, or both. CuryInfoEnable is also used to re-enable processing of CuryInfoGet, CuryInfoSet, or both.

The CuryInfoEnable statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level</td>
<td>Integer</td>
<td>Update Level Number corresponding to record where Currency Information will be stored in the database.</td>
</tr>
</tbody>
</table>

See Also
CuryInfoGet Statement, CuryInfoSet Statement, CurrencyInfo Statement

Example
The following example shows a portion of the Chk event for a Document Reference number. If the document has been released, no changes are allowed to document amounts. Consequently, as an Safeguard, the retrieval of Currency Rate Information into the document from the Currency Selection screen, using CuryInfoGet, is also suppressed for the Document and Detail levels. Please note that CuryInfoSet calls are still allowed, since they are necessary in order to allow the Currency Selection Form to display the Currency Rate Information previously stored in the document. For an open document, CuryInfoEnable calls are made to insure that CuryInfoSet and CuryInfoGet are re-enabled, in case they had been disabled for the previously-retrieved document.

Sub crefnbrh_Chk (chkstrg As String, retval As Integer)

        retval = NoAutoChk
        serr1 = pvchkfetch1(PNULL, C2, chkstrg, bAPDoc)
If serr1 = 0 Then
   'Currency Manager
   If bCMSSetup.MCActivated = LTRUE Then

   'If batch has already been released, all Currency
   'Selection fields should have been disabled when
   'batch was retrieved, so do nothing here.
   If bBatch.Rlsed = LFALSE Then
     'If batch has not been released, but document
     'has, disable all Currency Selection fields.
   If bAPDoc.Rlsed = LTRUE Then
     Call CurySelFieldEnable(CURYSEL_ALL, False)

     Call CuryFieldCalcSet(Form1.corigdocamt, CURY_BASE_NOCALC)
     Call CuryFieldCalcSet(Form1.cdiscbal, CURY_BASE_NOCALC)
     Call CuryFieldCalcSet(Form1.cdocbal, CURY_BASE_NOCALC)
     Call CuryFieldCalcSet(Form1.cunitprice, CURY_BASE_NOCALC)
     Call CuryFieldCalcSet(Form1.ctranamt, CURY_BASE_NOCALC)
     Call CuryInfoEnable(LEVEL1, CURY_INFO_SETONLY)
     Call CuryInfoEnable(LEVEL4, CURY_INFO_SETONLY)

     'Otherwise disable Tran Currency ID to disallow
     'changing currency in existing document.
   Else
     Call CurySelFieldEnable(CURYSEL_ALL, True)
   Call CurySelFieldEnable(CURYSEL_TRANCURYID + J
     SaveCURYSEL_RATETYPE, False)

     Call CuryFieldCalcSet(Form1.corigdocamt, CURY_BASE_CALC)
     Call CuryFieldCalcSet(Form1.cdiscbal, CURY_BASE_CALC)
     Call CuryFieldCalcSet(Form1.cdocbal, CURY_BASE_CALC)
     Call CuryFieldCalcSet(Form1.cunitprice, CURY_BASE_CALC)
     Call CuryFieldCalcSet(Form1.ctranamt, CURY_BASE_CALC)
     Call CuryInfoEnable(LEVEL1, CURY_INFO_SETGET)
     Call CuryInfoEnable(LEVEL4, CURY_INFO_SETGET)
   End If
   End If

   'If Currency values were previously saved in doc
   'display them in Currency Selection form. (Under normal
   'circumstances, there will always be Currency values in
   'the doc, even if transaction currency was the same
   'as base currency.)
   Call CuryInfoSet(LEVEL1)

End If
CuryInfoGet Statement

Description
Moves Currency Rate Information from the Currency Selection Form (opened from the Toolbar) to the application database record where Currency Rate Information will be stored for a particular update level. In order to use CuryInfoGet for a particular update level, a CurrencyInfo call must have been made in Form_Load for the update level.

Syntax
Call CuryInfoGet(Level)

Remarks
CuryInfoGet calls are used to retrieve Currency Rate Information values previously defaulted into or entered into the Currency Selection Form (opened from the Toolbar) and copy the values to the Currency Rate Information fields in the database record structure used for a particular update level in the application. CuryInfoGet assumes that a CurrencyInfo call was done in the Form_Load event for each Update Level which needs to update the database with Currency Selection Information. Locations saved by the CurrencyInfo call are used by CuryInfoGet to determine where to store the retrieved values.

CuryInfoGet calls for Batch and Document levels are typically done within the Update_New event of the application. CuryInfoGet calls for detail levels are typically done within the logic for initializing a new detail line in the LineGotFocus event for the spreadsheet.

The CuryInfoGet statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level</td>
<td>Integer</td>
<td>Update Level Number corresponding to record where Currency Information will be stored in the database.</td>
</tr>
</tbody>
</table>

See Also
CuryInfoInit Statement, CuryInfoSet Statement, CurrencyInfo Statement

Example
The first example below shows a portion of the Update_New event for an application being enabled for multi-currency processing. CuryInfoGet calls for the Batch and Document levels are done here. Note that one of the CuryInfoGet calls is actually made within a subroutine called Init_Enable_Batch_Currency, which performs a group of operations required for Currency initialization at the Batch level. The call to the subroutine is made within the Level 0 case logic, and the subroutine is shown following the Update_NewLevel event code segment.

Sub Update1_NewLevel (level%, retval%)

'Set Un-Entered fields for batch and document levels

Select Case level

Case 0  'Batch

    bBatch.GLPostOpt = bAPSetup.GLPostOpt
    bBatch.PerEnt = bAPSetup.CurrPerNbr
    bBatch.JrnlType = "AP"
    bBatch.Module = "AP"
bBatch.EditScrnNbr = "03010"
bBatch.Status = "H"
bBatch.BatType = "N"
bBatchHandling.BatchHandling = "H"

'Initialize and Enable Currency Manager fields
If bCMSetup.MCActivated = LTRUE Then
    Call Init_Enable_Batch_Currency
Else
    Call CuryInfoGet(LEVEL0)
End If

Case 1  'Document

bAPDoc.PerEnt = bAPSetup.CurrPerNbr
bAPDoc.DocClass = "N"
bAPDoc.Selected = LFALSE
bAPDoc.Current = LTRUE
bAPDoc.OpenDoc = LTRUE
SaveBal = LTRUE

'Currency Manager
Call CuryInfoGet(LEVEL1)

End Select

End Sub

Sub Init_Enable_Batch_Currency ()

Call CuryFieldCalcSet(Form1.cctrltot, CURY Base CALC)
Call CuryFieldCalcSet(Form1.corigdocamt, CURY Base CALC)
Call CuryFieldCalcSet(Form1.cdiscbal, CURY Base CALC)
Call CuryFieldCalcSet(Form1.cdrtot, CURY Base CALC)
Call CuryFieldCalcSet(Form1.cdocbal, CURY Base CALC)
Call CuryFieldCalcSet(Form1.cunitprice, CURY Base CALC)
Call CuryFieldCalcSet(Form1.ctranamt, CURY Base CALC)

Call CuryInfoInit
Call CuryRateTypeSet(bCMSetup.APRtTpDflt)
Call CuryInfoGet(LEVEL0)

Call CurySelFieldEnable(CURYSEL_ALL, True)
Call CurySelFieldEnable(SaveCURYSEL_RATETYPE, False)

End Sub
The second example below shows the Spread1.LineGotFocus event for an application being enabled for multi-currency processing. The `CurlyInfoGet` call for the Detail level is done here.

```vba
Sub Spread1_LineGotFocus (maintflg%, retval%)

    If maintflg = NEWROW Then 'set detail line defaults

        'Set APTran.LineId and update APDoc.LineCntr
        baptran.lineid = bAPDoc.LineCntr + 1
        bAPDoc.LineCntr = bAPDoc.LineCntr + 1
        Call setprops(Form1, cqty, cunitprice, PROP_ENABLED, True)

        'Set Default APTran Fields not entered in the screen
        baptran.Acct = bVendor.ExpAcct
        baptran.Sub = bVendor.ExpSub
        baptran.BoxNbr = bVendor.DfltBox
        baptran.ExtraRefNbr = bAPDoc.InvcNbr
        baptran.FiscYr = Left$(bBatch.PerPost, 4)
        baptran.PerPost = bBatch.PerPost
        baptran.PerEnt = bBatch.PerEnt
        baptran.TranDate = bAPDoc.DocDate
        baptran.TranType = bAPDoc.DocType
        baptran.VendId = bVendor.VendId
        baptran.JrnlType = "AP"
        baptran.RefNbr = bAPDoc.RefNbr
        baptran.PerEnt = bAPDoc.PerEnt
        baptran.Rlsed = LFALSE
        baptran.AcctDist = LFALSE

        'Determine the Entry Type
        If bAPDoc.DocType = "VO" Or bAPDoc.DocType = "AC" Then
            baptran.DrCr = "D" 'Debit
        Else
            baptran.DrCr = "C" 'Credit
        End If

        'Currency Manager - Set detail currency info fields
        'to CURYSEL values.
        Call CurlyInfoGet(LEVEL2)
    End If

End Sub
```
CurInfoInit Statement

Description
Initializes Currency Rate Information in the Currency Selection Form (opened from the Toolbar) to Base Currency values. Transaction Currency ID is set to the Base Currency ID for the database. Rate Type is set to blanks. Effective Date is set to the current business date. Multiply/Divide is set to “M”. Rate is set to 1.0. Also sets the Currency View button image to the image associated with Base Currency.

Syntax
Call CurInfoInit

Remarks
CurInfoInit calls are used to reinitialize the Currency Rate Information values in the Currency Selection form to Base Currency values. When an application is called from the Applications menu, the Currency Selection form is automatically initialized with Base Currency values, so it is not necessary for the application to initialize the values when running. However, the application will typically need to re-initialize Currency Rate Information as part of the “New” logic for the LEVEL0 Batch or Document. If it is possible to initiate a new Batch or Document from the Batch Number or Reference Number Chk event, CurInfoInit may also need to be run there.
The CurInfoInit statement does not require any arguments.

See Also
CurInfoGet Statement, CurInfoSet Statement

Example
The following example shows a portion of the Update_NewLevel LEVEL0 case logic for an application being enabled for multi-currency processing. Note that CurInfoInit is actually called within the subroutine Init_Enable_Batch_Currency, which is used to group currency initialization logic for the Batch level, and is shown following the Update_NewLevel code segment.

Sub Update1_NewLevel (level%, retval%)

' Set Un-Entered fields for batch and document levels

Select Case level

Case 0 'Batch

bBatch.GLPostOpt = bAPSetup.GLPostOpt
bBatch.PerEnt = bAPSetup.CurrPerNbr
bBatch.Jrn1Type = "AP"
bBatch.Module = "AP"
bBatch.EditScrNbr = "03010"
bBatch.Status = "H"
bBatch.BatType = "N"
bBatchHandling.BatchHandling = "H"

'Initialize and Enable Currency Manager fields
If bCMSsetup.MCActivated = LTRUE Then
    Call Init_Enable_Batch_Currency

Else

Call CuryInfoGet(LEVEL0)

End If

Sub Init_Enable_Batch_Currency()

Call CuryFieldCalcSet(Form1.cctrltot, CURY_BASE_CALC)
Call CuryFieldCalcSet(Form1.corigdocamt, CURY_BASE_CALC)
Call CuryFieldCalcSet(Form1.cdiscbal, CURY_BASE_CALC)
Call CuryFieldCalcSet(Form1.cdrtot, CURY_BASE_CALC)
Call CuryFieldCalcSet(Form1.cdocbal, CURY_BASE_CALC)
Call CuryFieldCalcSet(Form1.cunitprice, CURY_BASE_CALC)
Call CuryFieldCalcSet(Form1.ctranamt, CURY_BASE_CALC)

Call CuryInfoInit
Call CuryRateTypeSet(bCMSetup.APRtTpDflt)
Call CuryInfoGet(LEVEL0)

Call CurySelFieldEnable(CURYSEL_ALL, True)
Call CurySelFieldEnable(SaveCURYSEL_RATETYPE, False)

End Sub

The following example below shows the Batch Number Chk event for an application being enabled for multi-currency processing. Note that CuryInfoInit is actually called within the subroutine Init_Enable_Batch_Currency, which is shown above.

Sub cbatnbrb_Chk (chkstrg As String, retval As Integer)

retval = NoAutoChk
serr1 = pvchkfetch1(PNULL, c1, chkstrg, bBatch)

If serr1 = 0 Then
'Multi-Currency Activated
If bCMSetup.MCActivated = LTRUE Then

'If batch has already been released,
'disable all Currency Selection fields.
'Also disable Base Currency Calculations
'in all fields that were originally calced.

If bBatch.Rlsed = LTRUE Then

Call CurySelFieldEnable(CURYSEL_ALL, False)
Call CuryFieldCalcSet(Form1.cctrltot, CURY_BASE_NOCALC)
Call CuryFieldCalcSet(Form1.corigdocamt, CURY_BASE_NOCALC)
Call CuryFieldCalcSet(Form1.cdiscbal, CURY_BASE_NOCALC)
Call CuryFieldCalcSet(Form1.cdrtot, CURY_BASE_NOCALC)
Call CuryFieldCalcSet(Form1.cdocbal, CURY_BASE_NOCALC)
Call CuryFieldCalcSet(Form1.cunitprice, CURY_BASE_NOCALC)

End Sub
Call CuryFieldCalcSet(Form1.ctranamt, CURY_BASE_NOCALC)
Call CuryInfoEnable(LEVEL0, CURY_INFO_SETONLY)
Call CuryInfoEnable(LEVEL1, CURY_INFO_SETONLY)
Call CuryInfoEnable(LEVEL4, CURY_INFO_SETONLY)
'Otherwise disable Tran Currency ID to disallow
'changing currency in existing batch.
Else
    Call CurySelFieldEnable(CURYSEL_ALL, True)
    Call CurySelFieldEnable(CURYSEL_TRANCURYID + J
    saveCURYSEL_RATE, False)
    Call CuryFieldCalcSet(Form1.cctritot, CURY_BASE_CALC)
    Call CuryFieldCalcSet(Form1.corigdocamt, CURY_BASE_CALC)
    Call CuryFieldCalcSet(Form1.cdiscbal, CURY_BASE_CALC)
    Call CuryFieldCalcSet(Form1.cdrtot, CURY_BASE_CALC)
    Call CuryFieldCalcSet(Form1.cdocbal, CURY_BASE_CALC)
    Call CuryFieldCalcSet(Form1.cunitprice, CURY_BASE_CALC)
    Call CuryFieldCalcSet(Form1.ctranamt, CURY_BASE_CALC)
    Call CuryInfoEnable(LEVEL0, CURY_INFO_SETGET)
    Call CuryInfoEnable(LEVEL1, CURY_INFO_SETGET)
    Call CuryInfoEnable(LEVEL4, CURY_INFO_SETGET)
End If

'If Currency values were previously saved in batch
'display them in Currency Selection form.  (Under normal
'circumstances, there will always be Currency values in
'the batch, even if transaction currency was the same
'as base currency.)
Call CuryInfoSet(LEVEL0)
End If
Else
'Batch not found - need to init record as new.
'Multi-Currency activated...
If bCMSSetup.MCActivated = LTRUE Then
    Call Init_Enable_Batch_Currency
Else
    'If M-C not activated, still need to
    'populate currency info in Batch record.
    Call CuryInfoGet(LEVEL0)
End If
End If
End Sub
CuryIdSet Statement

Description
Sets the value of the Transaction Currency ID field in the Currency Selection Form (opened from the Toolbar). Also sets the Currency View button image to the image associated with the new Transaction Currency ID.

Syntax
Call CuryIdSet(bCuryId)

Remarks
CuryIdSet is used in some non-standard applications to set the Transaction Currency ID and Currency View button image without entering the Currency Selection form.

Note: This routine does not automatically retrieve new rate information into the Currency Selection Form. If new rate information is desired, this call must be followed by a call to GetCuryRate.

The CuryIdSet statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bCuryId</td>
<td>String</td>
<td>Location of field containing new Currency ID.</td>
</tr>
</tbody>
</table>

See Also
CuryRateTypeSet Statement, CuryEffDateSet Statement, GetCuryRate Statement

Example
The following example below shows the Spread1_LineGotFocus event for an application which changes the image on the Currency View button to show the image corresponding to the Currency ID of the Account represented by each detail line.

Sub Spread1_LineGotFocus (maintflg%, retval%)

'Currency Manager - Set detail currency info fields
to CURYSEL values.
Call CuryIDSet(bCashAcct.curyid)

End Sub
CurlyInfoSet Statement

Description
Moves Currency Rate Information from the application database record where Currency Rate Information has been stored for a particular update level to the Currency Selection Form (opened from the Toolbar). Also updates the Currency View button image. In order to use CurlyInfoSet for a particular update level, a CurrencyInfo call must have been made in Form_Load for the update level.

Syntax
Call CurlyInfoSet( Level )

Remarks
CurlyInfoSet calls are used to retrieve Currency Rate Information values from the database record structure used for a particular update level in the application and copy the values to the Currency Rate Information fields in the Currency Selection Form (opened from the Toolbar). CurlyInfoSet assumes that a CurrencyInfo call was done in the Form_Load event for each Update Level which contains Currency Selection Information. Locations saved by the CurrencyInfo call are used by CurlyInfoSet to determine where to find the Currency Rate Information values in the database record structure.

CurlyInfoSet calls for Batch and Document levels are typically done within the Batch Number and Document Reference Number Chk events of the application. When an existing Batch or Document is retrieved, the Currency Selection Form is updated to show the Currency Rate Information previously saved in the database record structure. The Currency View button image is also updated to show the image corresponding to the Batch or Document Currency ID.

Note: It is a Microsoft convention that all Documents within the same Batch must have the same Currency ID. However, different Documents within the same Batch may have different Rate Types, Effective Dates, Rates, and Multiply/Divide operators. Transactions within a Document must have the same Currency Rate Information as the Document.

In a typical data entry screen, where detail lines represent Transaction records within a Document, it is not necessary to call CurlyInfoSet for each detail line, because all Transactions have the same Currency Rate Information values as the Document with which they are associated. However, if each detail line represents a Document or Batch, a CurlyInfoSet call may need to be done for each detail line as it gains focus, so that the Currency Selection Form will show the information from the focused detail. In that case, CurlyInfoSet should be called within the logic for displaying existing detail lines in the Spread1_LineGotFocus event, and within any detail control Chk event which could cause the Currency Rate Information to change.

The CurlyInfoSet statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level</td>
<td>Integer</td>
<td>Update Level Number corresponding to record where Currency Information will be stored in the database.</td>
</tr>
</tbody>
</table>

See Also
CurlyInfoGet Statement, CurlyInfoInit Statement, CurrencyInfo Statement
Example

The following example below shows a portion of the Batch Chk event for an application being enabled for multi-currency processing.

Sub cbatnbrb_Chk (chkstrg As String, retval As Integer)
    retval = NoAutoChk
    serr1 = pvchkfetch1(PNULL, c1, chkstrg, bBatch)

    If serr1 = 0 Then
        'Multi-Currency Activated
        If bCMSSetup.MCActivated = LTRUE Then
            'If batch has already been released,
            'disable all Currency Selection fields.
            'Also disable Base Currency Calculations
            'in all fields that were originally calc'd.
            If bBatch.Rlsed = LTRUE Then
                Call CurySelFieldEnable(CURYSEL_ALL, False)
                Call CuryFieldCalcSet(Form1.cctrltot, CURY_BASE_NOCALC)
                Call CuryFieldCalcSet(Form1.corigdocamt, CURY_BASE_NOCALC)
                Call CuryFieldCalcSet(Form1.cdiscbal, CURY_BASE_NOCALC)
                Call CuryFieldCalcSet(Form1.cdrtot, CURY_BASE_NOCALC)
                Call CuryFieldCalcSet(Form1.cdocbal, CURY_BASE_NOCALC)
                Call CuryFieldCalcSet(Form1.cunitprice, CURY_BASE_NOCALC)
                Call CuryFieldCalcSet(Form1.ctranamt, CURY_BASE_NOCALC)
                Call CuryInfoEnable(LEVEL0, CURY_INFO_SETONLY)
                Call CuryInfoEnable(LEVEL1, CURY_INFO_SETONLY)
                Call CuryInfoEnable(LEVEL4, CURY_INFO_SETONLY)
            'Otherwise disable Tran Currency ID to disallow
            'changing currency in existing batch.
            Else
                Call CurySelFieldEnable(CURYSEL_ALL, True)
            End If
        End If

        Call CurySelFieldEnable(CURYSEL_TRANCURYID + @SaveCURYSEL_RATETYPE, False)
        Call CuryFieldCalcSet(Form1.cctrltot, CURY_BASE_CALC)
        Call CuryFieldCalcSet(Form1.corigdocamt, CURY_BASE_CALC)
        Call CuryFieldCalcSet(Form1.cdiscbal, CURY_BASE_CALC)
        Call CuryFieldCalcSet(Form1.cdrtot, CURY_BASE_CALC)
        Call CuryFieldCalcSet(Form1.cdocbal, CURY_BASE_CALC)
        Call CuryFieldCalcSet(Form1.cunitprice, CURY_BASE_CALC)
        Call CuryFieldCalcSet(Form1.ctranamt, CURY_BASE_CALC)
        Call CuryInfoEnable(LEVEL0, CURY_INFO_SETGET)
        Call CuryInfoEnable(LEVEL1, CURY_INFO_SETGET)
        Call CuryInfoEnable(LEVEL4, CURY_INFO_SETGET)
    End If

    'If Currency values were previously saved in batch
'display them in Currency Selection form. (Under normal circumstances, there will always be Currency values in 'the batch, even if transaction currency was the same 'as base currency.)
Call CuryInfoSet(LEVEL0)
End If
Else
'Batch not found - need to init record as new. 'Multi-Currency activated...
If bCMSsetup.MCActivated = LTRUE Then
   Call Init_Enable_Batch_Currency
Else
   'If M-C not activated, still need to 'populate currency info in Batch record.
   Call CuryInfoGet(LEVEL0)
End If
End If

End If
CuryRateTypeSet Statement

Description
Sets the value of the Rate Type field in the Currency Selection Form (opened from toolbar).

Syntax
Call CuryRateTypeSet(bRateType)

Remarks
Since the default rate type varies by module, the application typically uses CuryRateTypeSet to set the default rate type to be initially shown in the Currency Selection Form. CuryRateTypeSet may also be called from the Vendor or Customer Chk event to set the rate type to the default for the Vendor or Customer.

Note: This routine does not automatically retrieve new rate information into the Currency Selection Form. If new rate information is desired, this call must be followed by a call to GetCuryRate.

The CuryRateTypeSet statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bRateType</td>
<td>String</td>
<td>Location of field containing new the rate type.</td>
</tr>
</tbody>
</table>

See Also
CuryIdSet Statement, CuryEffDateSet Statement, GetCuryRate Statement

Example
The following example shows a portion of the Form_Load event for an application being enabled for multi-currency processing. Note that CuryRateTypeSet is called to set the rate type in the Currency Selection Screen to the default rate type for AP data entry, stored in the Setup record.

Call ApplInit

Call setaddr(LEVEL0, "bBatch", bBatch, nBatch)
Call setaddr(LEVEL1, "bAPDoc", bAPDoc, nAPDoc)
Call setaddr(LEVEL2, "bAPTran", baptran, nAPTran)
Call setaddr(NOLEVEL, "bCuryInfo", bCuryInfo, nCuryInfo)
Call setaddr(NOLEVEL, "bCMSsetup", bCMSsetup, nCMSsetup)

Call SqlCursor(C1, LEVEL0) 'Batch
Call SqlCursor(C2, LEVEL1) 'APDoc
Call SqlCursor(C3, LEVEL2) 'APTran

' Set up Currency Info Fields for CM
' Note - these calls need to be made even if CM is not installed.
Call CurrencyInfo(LEVEL0, bBatch, "Batch", "CuryId", "CuryMultDiv", "CuryRateType", bBatch.CuryEffDate, bBatch.CuryRate)

Call CurrencyInfo(LEVEL1, bAPDoc, "APDoc", "CuryId", "CuryMultDiv", "CuryRateType", bAPDoc.CuryEffDate, bAPDoc.CuryRate)
Call CurrencyInfo(LEVEL2, bAPTran, "APTran", "CuryId", "CuryMultDiv", ",", NULLDATE, bAPTran.CuryRate)

'Initialize and Enable Currency Manager Controls
serr1 = sqlfetch1(c7, "CMSetup_All", bCMSetup)
'If Currency Manager is installed and Multi-Currency is activated ...
If serr1 = 0 And bCMSetup.MCActivated = LTRUE Then
  'Activate Currency Buttons on Toolbar
  Call SetButton(CurySelButton + CuryTogButton, AllLevels, True)
  'Set default rate type
  Call CuryRateTypeSet(bCMSetup.APrtTpDflt)

  'If rate type override is not allowed,
  'disable rate type field in Currency Selection form
  If bCMSetup.APrtTpOverride = LFALSE Then
    Call CurySelFieldEnable(CURYSEL_RATETYPE, False)
    SaveCURYSEL_RATETYPE = CURYSEL_RATETYPE
  Else
    SaveCURYSEL_RATETYPE = 0
  End If
End If
End If

ScreenInit
CurySelFieldEnable Statement

Description
Allows the application to control the enabling and disabling of fields within the Currency Selection Form, opened from the Toolbar.

Syntax
Call CurySelFieldEnable(FieldId, Enabled)

Remarks
CurySelFieldEnable allows the application to set the state of data entry in the Currency Selection Form to be consistent with the data entry state of the application. The fields (field groups) which can be enabled or disabled by the application are Currency ID, Rate Type, and All. The starting state for the typical application is that all Currency Selection Form fields are enabled. A CurySelFieldEnable call is typically made, along with other initialization logic, in the LEVEL0 case of the Update_NewLevel event to enable all fields in the Currency Selection Form. By Microsoft convention, all documents within a Batch must have the same Currency ID. Consequently, once the first Document of the Batch has been saved, the application may do a CurySelFieldEnable call to disable entry into the Currency ID field of the Currency Selection Form. Override of the Rate Type field in the Currency Selection Form is a Setup option. The application would typically retrieve the Setup record, and then make a CurySelFieldEnable call to disable entry into the Rate Type field, if necessary. If a Document is retrieved for inquiry only, the application would make a CurySelFieldEnable call to disable all Currency Selection Form fields. When creating or retrieving a new Document, the application would typically call CurySelFieldEnable to re-initialize Currency Selection Form entry for the next operation.

Note: It may be necessary to undo changes made for the state of the previous Document.

Some operations are done using two or more consecutive CurySelFieldEnable calls. For example, the logic for starting a new Document would need to undo any possible previous field-disabling operations made during entry of the previous Document, and then set a rate type restriction, if necessary. This would be done by first making a CurySelFieldEnable call to enable All, and then making a second CurySelFieldEnable call to disable only the Rate Type field. (The effects of consecutive calls are cumulative.)

The Currency Selection Form itself has field enabling and disabling logic which is designed to work in conjunction with the application. Upon initial entry into a screen, the values of all of the Currency Selection Form fields are automatically set to Base Currency values. As long as the Transaction Currency ID value is the same as Base Currency, no other fields in the Currency Selection Form may be changed. When the Transaction Currency and the Base Currency are the same, it is important that the SWIM currency conversion routines use a Rate of 1.0, so no changes are allowed, unless the user first changes the Transaction Currency ID. Unless the application has made a CurySelFieldEnable call which disables the Transaction Currency ID field, the user will be able to change the Transaction Currency ID field value from the Base Currency value to another Currency ID value. Entry to the other fields on the screen will then be enabled, unless the application has made a CurySelFieldEnable call which causes one or more of them to remain disabled. For example, if the application has made a CurySelFieldEnable call to disable Rate Type, the other fields will become enabled, but Rate Type will remain disabled.
The `CurySelFieldEnable` statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FieldIDs</td>
<td>Integer</td>
<td>Global Constant representing a Currency Selection Form field or group of fields to be enabled or disabled. CURYSEL_TRANCURYID — enable or disable Transaction Currency ID entry. CURYSEL_RATETYPE — enable or disable rate type entry. CURYSEL_ALL — enable or disable entry of all fields on Currency Selection Form. Multiple FieldID’s may be combined in the same call by using the + operator.</td>
</tr>
<tr>
<td>EnableFlag</td>
<td>Integer</td>
<td>True — enable fields specified by FieldID’s. False — enable fields specified by FieldID’s.</td>
</tr>
</tbody>
</table>

**Example**

The following example shows the Batch Chk event for an application being enabled for multi-currency processing. Note that if the Batch has already been released, all Currency Selection Form fields are disabled. If the Batch has not been released, all Currency Selection Form fields are enabled, except for Transaction Currency ID and possibly Rate Type. Transaction Currency ID is disabled, because by Microsoft convention, the user cannot change the currency of an existing batch. Rate Type may or may not be disabled, depending upon the value of saveCURYSEL_RATETYPE, which was previously set to CURYSEL_RATETYPE or 0 in the Form_Load event (not shown), depending upon the value of the Rate Type override option in Setup.

```vbnet
Sub cbatnbrb_Chk (chkstrg As String, retval As Integer)
    retval = NoAutoChk
    serr1 = pvchkfetch1(PNULL, c1, chkstrg, bBatch)
    If serr1 = 0 Then
        'Multi-Currency Activated
        If bCMSsetup.MCActivated = LTRUE Then
            'If batch has already been released, 'disable all Currency Selection fields.
            'Also disable Base Currency Calculations 'in all fields that were originally calced.
            If bBatch.Rlsed = LTRUE Then
                Call CurySelFieldEnable(CURYSEL_ALL, False)
                Call CuryFieldCalcSet(Form1.ccritot, CURY_BASE_NOCALC)
                Call CuryFieldCalcSet(Form1.corigdocamt, CURY_BASE_NOCALC)
                Call CuryFieldCalcSet(Form1.cdiscbal, CURY_BASE_NOCALC)
                Call CuryFieldCalcSet(Form1.cdrtot, CURY_BASE_NOCALC)
                Call CuryFieldCalcSet(Form1.cdocbal, CURY_BASE_NOCALC)
                Call CuryFieldCalcSet(Form1.cunitprice, CURY_BASE_NOCALC)
                Call CuryFieldCalcSet(Form1.ctranamt, CURY_BASE_NOCALC)
                Call CuryInfoEnable(LEVEL0, CURY_INFO_SETONLY)
                Call CuryInfoEnable(LEVEL1, CURY_INFO_SETONLY)
                Call CuryInfoEnable(LEVEL4, CURY_INFO_SETONLY)
                'Otherwise disable Tran Currency ID to disallow 'changing currency in existing batch.
            Else
                Call CurySelFieldEnable(CURYSEL_ALL, True)
            End If
        End If
    End If
End Sub
```
Call **CuryselFieldEnable**(CURYSEL_TRANCURSID + .J
    saveCURYSEL_RATETYPE, False)
Call CuryFieldCalcSet(Form1.cctrltot, CURY_BASE_CALC)
Call CuryFieldCalcSet(Form1.corigdocamt, CURY_BASE_CALC)
Call CuryFieldCalcSet(Form1.cdiscbal, CURY_BASE_CALC)
Call CuryFieldCalcSet(Form1.cdrtot, CURY_BASE_CALC)
Call CuryFieldCalcSet(Form1.cdocbal, CURY_BASE_CALC)
Call CuryFieldCalcSet(Form1.cunitprice, CURY_BASE_CALC)
Call CuryFieldCalcSet(Form1.ctranamt, CURY_BASE_CALC)
Call CuryInfoEnable(LEVEL0, CURY_INFO_SETGET)
Call CuryInfoEnable(LEVEL1, CURY_INFO_SETGET)
Call CuryInfoEnable(LEVEL4, CURY_INFO_SETGET)
End If

' If Currency values were previously saved in batch
'display them in Currency Selection form. (Under normal
'circumstances, there will always be Currency values in
'the batch, even if transaction currency was the same
'as base currency.)
Call CuryInfoSet(LEVEL0)
End If
Else
' Batch not found - need to init record as new.
'Multi-Currency activated...
If bCMSsetup.MCActivated = LTRUE Then
    Call Init_Enable_Batch_Currency
Else
    ' If M-C not activated, still need to
    'populate currency info in Batch record.
    Call CuryInfoGet(LEVEL0)
End If
End If
DateCheck Function

Description
Verify whether or not a date string in MMDDYYYY format represents a valid date.

Syntax
RetVal = DateCheck(DateString)

Remarks
The DateCheck function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RetVal</td>
<td>Integer</td>
<td>0 if the date string represents a valid date. -1 indicates a day that is not valid. -2 indicates a month that is not valid.</td>
</tr>
<tr>
<td>DateString</td>
<td>String</td>
<td>Date string to be verified. Must be in MMDDYYYY format.</td>
</tr>
</tbody>
</table>

See Also
StrToDate Statement

DateCmp Function

Description
Compare two date values.

Syntax
Cmp = DateCmp(Date1, Date2)

Remarks
To determine whether or not a date value is null use DateCmp(Date, NULLDATE). NULLDATE is global variable declared in Solomon.VBTools.vb that is properly initialized by the system at the start of all Microsoft Dynamics SL SDK applications.

The DateCmp function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cmp</td>
<td>Integer</td>
<td>&lt;0 if Date1 &lt; Date2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 if the two dates are equal.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;0 if Date1 &gt; Date2</td>
</tr>
<tr>
<td>Date1</td>
<td>Integer</td>
<td>First date value</td>
</tr>
<tr>
<td>Date2</td>
<td>Integer</td>
<td>Second date value</td>
</tr>
</tbody>
</table>
DateMinusDate Function

Description
Return the number of days between two dates.

Syntax
\[ NbrDays = \text{DateMinusDate}(Date1, Date2) \]

Remarks
The DateMinusDate function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NbrDays</td>
<td>Long</td>
<td>Number of days between Date1 and Date2 including the ending date. If Date1 &gt; Date2 then the number of days between the two dates will be a negative value.</td>
</tr>
<tr>
<td>Date1</td>
<td>Integer</td>
<td>Beginning date</td>
</tr>
<tr>
<td>Date2</td>
<td>Integer</td>
<td>Ending date</td>
</tr>
</tbody>
</table>

See Also
DatePlusDays Statement, DatePlusMonthSetDay Statement

DatePlusDays Statement

Description
Add a designated number of days to an existing date.

Syntax
Call DatePlusDays(CurrDate, NbrDaysToAdd, ResultingDate)

Remarks
The DatePlusDays statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurrDate</td>
<td>Integer</td>
<td>Starting date value.</td>
</tr>
<tr>
<td>NbrDaysToAdd</td>
<td>Integer</td>
<td>Number of days to add to CurrDate. Negative values are supported.</td>
</tr>
<tr>
<td>ResultingDate</td>
<td>Integer</td>
<td>Result of CurrDate + NbrDaysToAdd.</td>
</tr>
</tbody>
</table>

See Also
DateMinusDate Function, DatePlusMonthSetDay Statement

Example
The following example shows how the DatePlusDays statement can be used to calculate a discount date.

If (bTerms.DiscType = "D") Then
    'Calculate discount date based on number of days
Call `DatePlusDays(bAPDoc.DocDate, bTerms.DiscIntrv, bAPDoc.DiscDate)`

Else

'Calculate due date based on fixed day of the month

Call `DatePlusMonthSetDay(bAPDoc.DocDate, 1, bTerms.DiscIntrv, bAPDoc.DiscDate)`

End If

**DatePlusMonthSetDay Statement**

**Description**

Add a designated number of months to an existing date and set the day portion of the resulting date to a specific day of the month.

**Syntax**

Call `DatePlusMonthSetDay(CurrDate, NbrMthsToAdd, SetSpecificDay, ResultingDate)`

**Remarks**

The `DatePlusMonthSetDay` statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CurrDate</code></td>
<td>Integer</td>
<td>Starting date value.</td>
</tr>
<tr>
<td><code>NbrMthsToAdd</code></td>
<td>Integer</td>
<td>Number of months to add to <code>CurrDate</code>. Negative values are supported.</td>
</tr>
<tr>
<td><code>SetSpecificDay</code></td>
<td>Integer</td>
<td>Desired value for the day portion of the resulting date. In cases where <code>SetSpecificDay</code> is beyond the last valid day for the relevant month, the system automatically sets the actual day value equal to the last valid day of that month.</td>
</tr>
<tr>
<td><code>ResultingDate</code></td>
<td>Integer</td>
<td>Resulting date.</td>
</tr>
</tbody>
</table>

**See Also**

`DateMinusDate Function, DatePlusDays Statement`

**Example**

The following example shows how the `DatePlusMonthSetDay` statement can be used to calculate a discount date.

If (bTerms.DiscType = "D") Then

'Calculate discount date based on number of days

Call `DatePlusDays(bAPDoc.DocDate, bTerms.DiscIntrv, bAPDoc.DiscDate)`

Else

'Calculate due date based on fixed day of the month

Call `DatePlusMonthSetDay(bAPDoc.DocDate, 1, bTerms.DiscIntrv, bAPDoc.DiscDate)`

End If
**DateToIntlStr Function**

**Description**
Convert a specified date into the Windows short date style.

**Syntax**

\[
\text{ShortDateStr} = \text{DateToIntlStr}(\text{DateToConvert})
\]

**Remarks**

The `DateToIntlStr` function will convert a date from an SQL database format into a format governed by the Windows short date style.

`DateToIntlStr` uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShortDateStr</td>
<td>String</td>
<td>String containing the value of <code>DateToConvert</code> in a Windows short date style.</td>
</tr>
<tr>
<td>DateToConvert</td>
<td>Integer</td>
<td>Date value to be converted.</td>
</tr>
</tbody>
</table>

**See Also**

`DateToStr Function`, `DateToStrSep Function`, `IntlToStrDate Statement`, `StrToDate Statement`

**DateToStr Function**

**Description**
Convert a date value from an SQL date format into a string in MMDDYYYY format.

**Syntax**

\[
\text{DateString} = \text{DateToStr}(\text{DateToConvert})
\]

**Remarks**

The `DateToStr` function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateString</td>
<td>String</td>
<td><code>DateToConvert</code> converted to a string in MMDDYYYY format.</td>
</tr>
<tr>
<td>DateToConvert</td>
<td>Integer</td>
<td>Date value to be converted.</td>
</tr>
</tbody>
</table>

**See Also**

`DateToIntlStr Function`, `DateToStrSep Function`, `IntlToStrDate Statement`, `StrToDate Statement`

**DateToStrSep Function**

**Description**
Convert a date value from an SQL date format into a string in MM/DD/YYYY format.

**Syntax**

\[
\text{DateString} = \text{DateToStrSep}(\text{DateToConvert})
\]
Remarks

The `DateToStrSep` and `DateToStr` functions differ only in the fact that `DateToStrSep` inserts a single character separator between the month, day and year portions of the string.

The `DateToStrSep` function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DateString</code></td>
<td>String</td>
<td><code>DateToConvert</code> converted to a string in MM/DD/YYYY format.</td>
</tr>
<tr>
<td><code>DateToConvert</code></td>
<td>Integer</td>
<td>Date value to be converted.</td>
</tr>
</tbody>
</table>

See Also

`DateToIntlStr Function`, `DateToStr Function`, `IntlStrToDate Statement`, `StrToDate Statement`

DBNavFetch Functions

Description

Retrieve a composite record from the database using an SQL statement from the DBNav property of an DSLMaskedText control.

Syntax

```plaintext
RetVal = DBNavFetch1(Ctrl, Cursor, SQLParmValue, bTable1)
RetVal = DBNavFetch4(Ctrl, Cursor, SQLParmValue, bTable1, bTable2, bTable3, bTable4)
RetVal = DBNavFetch8(Ctrl, Cursor, SQLParmValue, bTable1, bTable2, bTable3, bTable4, bTable5, bTable6, bTable7, bTable8)
```

Remarks

`DBNavFetch1`, `DBNavFetch4` and `DBNavFetch8` can be used to fetch a composite record based on the SQL text from the DBNav property of the SAFMaskedText control specified in the `Ctrl` parameter. These functions are not applicable if the DBNav property does not contain either an SQL statement or stored procedure name.

All `DSLMaskedText` controls have both a PV and a DBNav property each of which can contain an SQL statement. The `DBNavFetch1` and `PVChkFetch1` functions are similar except that they use SQL statements from two different properties, namely the DBNav and PV properties. Normally the `PVChkFetch1` function will be used in conjunction with the PV property to facilitate both a Possible Values window as well as the actual fetch of a record. If the program requirements specify the need to fetch a record but not supply a Possible Values window then `DBNavFetch1` will need to be used in conjunction with the DBNav property. This situation can arise on screens containing multiple key fields where the last key field does not need a Possible Value window. An example of such a case is Employee W2 History (02.260) which has two key fields: `Employee ID` and `Calendar Year`. The requirements for the screen are that the user be forced to enter a valid employee ID based on the Employee table which should be viewable via a Possible Values window. However no table exists in Microsoft Dynamics SL to define a valid calendar year and therefore a Possible Values window is not applicable since any numeric value is valid. Once the last key field has been entered a unique record corresponding to both key fields needs to be fetched from the database. Since `Calendar Year` is the last key field, and a Possible Value window is not required for that field, the `DBNavFetch1` function was used in conjunction with the DBNav property of the calendar year control.

`DBNavFetch1` is designed for SQL statements returning data from a single table. For more advanced SQL statements having one or more table joins either `DBNavFetch4` or `DBNavFetch8` can be used.

`DBNavFetch1`, `DBNavFetch4` and `DBNavFetch8` are all designed to fetch a single composite record. For example, if an SQL statement contains eight table joins, `DBNavFetch8` does not return eight records from a single table. On the contrary it returns a single record from each of the eight tables. Consequently these functions are not used in conjunction with the DBNav property of the `DSLGrid`
control. The **DetailLoad** statement uses the DBNav property of the **DSLGrid** control to load the grid with multiple records from one or more tables.

The **DBNavFetch1** function uses the following arguments (**DBNavFetch4** and **DBNavFetch8** respectively have four and eight table structures and corresponding lengths. PNULL should be passed for unused SolomonDataObject parameters)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RetVal</td>
<td>Integer</td>
<td>0 if a record is successfully fetched. NOTFOUND is returned if no records match the restriction clause of the DBNav SQL statement.</td>
</tr>
<tr>
<td>Ctrl</td>
<td>Control</td>
<td>Control containing the DBNav property to be used as the SQL statement. Can optionally be PNULL if the call is made within the Chk event of the control whose DBNav property is being used.</td>
</tr>
<tr>
<td>Cursor</td>
<td>Integer</td>
<td>SQL database cursor.</td>
</tr>
<tr>
<td>SQLParmValue</td>
<td>String</td>
<td>Key value passed as the last parameter to the restriction clause of the DBNav SQL statement.</td>
</tr>
<tr>
<td>bTable1</td>
<td>SolomonDataObject</td>
<td>SolomonDataObject corresponding to the primary table in the DBNav SQL statement.</td>
</tr>
</tbody>
</table>

**See Also**

**DetailLoad Statement**, **PVChkFetch Functions**, **DSLMaskedText Control**, **DBNav Property**

**Example**

```vbscript
Sub cCalYr_Chk (chkstrg As String, retval As Integer)
    Dim W2Federal_Fetch As Integer

    W2Federal_Fetch = DBNavFetch1(PNULL, CSR_W2Federal, chkstrg, J
                            bW2Federal)

End Sub
```

**DecimalPlaces Statement**

**Syntax**

Call **DecimalPlaces**( **Ctrl**, **PrecisionBasedOnField**)

**Description**

Facilitates the implementation of flexible decimal precision for a **DSLFloat** control.

**Remarks**

The **DSLFloat** control contains a property called **DecimalPlaces**. The value of this property determines the number of digits displayed to the right of the decimal separator for the underlying control. In most cases the **DecimalPlaces** property is a constant value specified at design-time which does not need to be altered at runtime. However, some applications contain fields whose decimal precision can vary at runtime based on some condition or data item.

For example, some applications display monetary values such as an invoice amount or a transaction amount. These types of applications typically allow the user to specify the currency they want to use and then relevant fields, such as transaction amounts, are displayed and/or entered in that currency. The problem is that each currency can have a different decimal precision. As an illustration, US Dollars is typically rounded to two decimal places whereas the XYZ currency may be rounded to three decimal places. For the developer this means that controls used to display these types of data items need to have their decimal precision dynamically configured at runtime. In this case that translates to dynamically modifying the **DecimalPlaces** property of relevant **DSLFloat** controls based on the decimal
precision of a particular currency. Another example would be applications that display inventory item quantities and/or item prices in which the required decimal precision might vary based on the type of the inventory item currently displayed.

The **DecimalPlaces** statement simplifies the management of flexible decimal precision functionality by linking a DSLFloat control with an underlying Visual Basic variable whose value will be used to automatically reconfigure the **DecimalPlaces** property of the control at runtime.

Support for flexible decimal precision only exists for the following classes of data items (that is, FieldClass values):

<table>
<thead>
<tr>
<th>FieldClass</th>
<th>Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>124</td>
<td>Transaction Amount</td>
</tr>
<tr>
<td>125</td>
<td>Inventory Item Quantity</td>
</tr>
<tr>
<td>126</td>
<td>Inventory Item Price</td>
</tr>
</tbody>
</table>

Usage of the **DecimalPlaces** statement is only required when the precision of a particular DSLFloat control will not always match the default precision of its FieldClass.

- The default decimal precision for data items in the Transaction Amount class is based on the value of the **DecPl** field in the particular currency record that is relevant at any particular point in time (for example, either transaction currency or base currency).

- The Item Quantity and Item Price classes derive their default decimal precision from the constant value which was specified at design-time for the **DecimalPlaces** property. Item Quantity fields are typically designed with 2 decimal places in Microsoft Dynamics SL Business Essentials applications. Similarly, Item Price fields are typically designed with 3 decimal places.

The **DecimalPlaces** statement must be called for each control requiring flexible decimal precision. Multiple controls can be linked to the same underlying precision control value. The call(s) must occur in the Form_Load event of Form1.

Furthermore, the application must assume the responsibility of setting the value of the underlying precision control value (that is, **PrecisionBasedOnField**). In particular, the value of the **PrecisionBasedOnField** must be set prior to actually displaying data in any corresponding flexible precision control.

- For controls outside the context of DSLGrid, this simply means that the **PrecisionBasedOnField** should be set when the record containing the flexible precision data item is loaded. This will typically be in a key field **Chk** event.

- Controls existing within the context of a DSLGrid are handled slightly different. In this case, the table containing the underlying **PrecisionBasedOnField** will normally be joined with the table containing the flexible precision data item which is to be displayed in the DSLGrid.

The **DecimalPlaces** statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl</td>
<td>Control</td>
<td>DSLFloat control whose <strong>DecimalPlaces</strong> property needs to be modified dynamically at runtime.</td>
</tr>
<tr>
<td>PrecisionBasedOnField</td>
<td>Integer</td>
<td>Visual Basic variable whose value will be used to dynamically manage the decimal precision of the designated Ctrl at runtime.</td>
</tr>
</tbody>
</table>

**Example**

The following example examines a portion of the Form_Load event for the GL Batch Release process.

This application displays a DSLGrid containing all Batch records that are available for release. The precision associated with the Batch Control amount may vary for each detail line since each individual Batch may have been entered in a different currency.

The **DecimalPlaces** call specifies that the **DecPl** field from the currency table will be used to obtain the correct decimal precision for each detail line. Please note that the currency table also appears in the DetailSetup call in order to facilitate the retrieval of the correct currency record (and therefore the
correct DecPl value) prior to the display of each detail line. This obviously implies that the currency table is joined to the Batch table in the SQL statement associated with the DBNav property of the **DSLGrid** control. In addition, this example assumes that the cCtrlTot control utilizes the Transaction Amount FieldClass value.

```vba
Sub Form_Load ()

    Call LoadForm(ProcStatForm)

    Call ApplInit

    Call SetAddr(LEVEL0, "bBatch", bBatch, nBatch)
    Call SetAddr(NOLEVEL, "bCurrBatchSelected", bCurrBatchSelected, PNULL)

    'Call SetAddr() to get null buffers properly initialized
    Call SetAddr(NOLEVEL, "bGLSetup", bGLSetup, nGLSetup)
    Call SetAddr(NOLEVEL, "bGLTran", bGLTran, nGLTran)
    Call SetAddr(NOLEVEL, "bLedger", bLedger, nLedger)

    Call SetAddr(NOLEVEL, "bCurrncy", bCurrncy, nCurrncy)

    'Initialize global cursors
    Call CursorInit

    'Load GLSetup record
    serr1 = SqlFetch1(CSR_GLSetup, "GLSetup_All", bGLSetup)
    Call Check_Module_Setup("GL", serr1)

    Call ScreenInit

    Call DecimalPlaces(cCtrlTot, bCurrncy.DecPl)

    MemHandle = DetailSetup(CSR_Batch_DBnav, Spread1, PNULL, bBatch, bCurrncy, bCurrBatchSelected, PNULL)

**DetailLoad Statement**

**Description**

Load a designated **DSLGrid** control with information from the database using the SQL statement or stored procedure specified in its DBNav property.

**Syntax**

```vba
Call DetailLoad(DSLGridCtrl)
```

**Remarks**

Two types of detail levels (that is, spreadsheets or grids) are supported by Microsoft Dynamics SL SDK: Standard detail levels and Application Loaded detail levels. Standard and Application Loaded detail levels are respectively defined with a level type of D and DA in the Levels property of the **DSLUpdate** control. Standard detail levels are the most common type of grids. The Application Loaded detail level
is a special type of grid used when a Standard grid cannot be implemented due to various technical obstacles.

One such technical obstacle is when the SQL statement or stored procedure used to load information into the DSLGrid control requires more parameters than is supported by the DBNav property. In that case the DBNav property cannot be used and therefore the application will need to take responsibility for both loading and saving information into the relevant DSLGrid control. The DetailLoad statement cannot be used in this type of scenario.

In other cases the DBNav property can be used but the application needs to control when the information is actually loaded. Since Standard detail levels are always loaded automatically by the system, this requirement for control over the timing of the load operation can be another technical obstacle necessitating the usage of an Application Loaded detail level. In this case the application merely needs to issue a call to DetailLoad notifying the system at the precise point when the DSLGrid control to load itself with information from the database. In this case the application does not need to concern itself with responsibility for saving the data since the system will still perform that task automatically.

The DetailLoad statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSLGridCtrl</td>
<td>Control</td>
<td>DSLGrid control to be loaded with information from the database using the SQL statement or stored procedure specified in its DBNav property.</td>
</tr>
</tbody>
</table>

See Also
DBNav Property, DetailSetup Functions, MLoad Statement, DSLGrid Control

Example
An example of a screen using the DetailLoad statement is the GL Allocation Group Maintenance screen. This screen needs to control the timing of the load operation since it needs to work with the data immediately after the data is loaded but before control returns to the user. The following code snippet is from the Chk event of the header level master key field called Group ID.

Sub cGrpId_Chk (chkstrg As String, retval As Integer)
    Dim AllocGrp_Fetch As Integer
    AllocGrp_Fetch = PVChkFetch1(CNULL, CSR_AllocGrp, chkstrg, bAllocGrp)

    If (AllocGrp_Fetch = 0) Then
        'Reset properties based on allocation method of retrieved group
        Call Check_Ctrl_Props(bAllocGrp.AllocMthd)

        'Load all detail lines for both DSLGrid controls
        Call DetailLoad(cSpread_Source)
        Call DetailLoad(cSpread_Dest)

        'Initialize current group type.  MUST be called AFTER both grids have been loaded since it operates on data presently existing in the two DSLGrid controls.
        Call CurrGroupType_Init
    End If
    retval = NoAutoChk
End Sub
DetailSave Statement

Description
Update the database with the contents of the designated DSLGrid control.

Syntax
Call DetailSave(Cursor, DSLGridCtrl, TablesUpdating)

Remarks
DetailSave is used in extremely rare cases in which the application either needs to control precisely when the database is actually updated with the contents of an DSLGrid control or which individual tables should be updated. Usage of DetailSave is not required for either Standard or Application Loaded detail levels (that is, D and DA level types respectively in the Levels property of the DSLUpdate control).

When the Update event fires for any type of detail level, the actual update to the database does not occur until after control returns back to the system for that level. In such a case, the DetailSave call could be issued during the Update event for the relevant detail level and then RetVal should be set to the NoAction symbolic constant defined in Solomon.VBTools.vb. By setting RetVal to NoAction, the system will be notified to avoid saving data from the DSLGrid control again, as it normally would, after control returns back to the system for that particular detail level.

By default, the system saves data for the master table associated with any type of detail level. The master table for a level is the table identified in the SetAddr call for that particular level. If an application has a grid containing information from more than one table (for example, a joined view) and also needs to update both tables with modifications made by the user, then the DetailSave statement will need to be utilized in the Update event of the DSLUpdate control as previously discussed.

The DetailSave statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cursor</td>
<td>Integer</td>
<td>SQL database cursor associated with the DSLGrid control. This should be the same cursor passed to the DetailSetup function when the DSLGrid was originally initialized.</td>
</tr>
<tr>
<td>DSLGridCtrl</td>
<td>Control</td>
<td>DSLGrid control whose information is to be saved to the database.</td>
</tr>
<tr>
<td>TablesUpdating</td>
<td>String</td>
<td>Name of each table, in the specified cursor's view, to be updated with information from the designated DSLGridCtrl. Multiple table names should be separated by commas.</td>
</tr>
</tbody>
</table>

See Also
DetailLoad Statement, DetailSetup Functions, SetAddr Statement, DSLUpdate Control
DetailSetup Functions

Description
Initialize a Detail level by binding together all associated components, such as controls, database cursor and table structures at runtime.

Syntax
MemHandle = DetailSetup(Cursor, DSLGridCtrl, AutoLineNbrFld, bTable1, bTable2, bTable3, bTable4)
MemHandle = DetailSetup8(Cursor, DSLGridCtrl, AutoLineNbrFld, bTable1, bTable2, bTable3, bTable4, bTable5, bTable6, bTable7, bTable8)

Remarks
A Detail data entry level in a Microsoft Dynamics SL SDK application is composed of several different components which are bound together at runtime via either the DetailSetup or DetailSetup8 function. For example, each detail data entry level has a DSLGrid control to manage the user interface for multiple detail lines. Since the data displayed in the DSLGrid is actually stored in a memory array the array needs to be opened at runtime. At any point in time the application needs to have table structure variables capable of holding data for the current record in the grid and/or its underlying memory array. An DSLGrid control with its underlying memory array and table structure variables is nevertheless ineffectual without the ability to load and save data, which necessitates a connection to the database through a cursor. The DetailSetup and DetailSetup8 functions automatically open a memory array and attach all these components together with it at runtime.

The DetailSetup function uses the following arguments (DetailSetup8 has eight SolomonDataObjects. PNULL should be passed for unused SolomonDataObject parameters).

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Unique ID of the memory array automatically opened for storage of detail lines associated with the DSLGrid.</td>
</tr>
<tr>
<td>Cursor</td>
<td>Integer</td>
<td>SQL database cursor used to load and save records associated with the DSLGridCtrl. This cursor must have already been allocated using SqlCursor before the DetailSetup call is made.</td>
</tr>
<tr>
<td>DSLGridCtrl</td>
<td>Control</td>
<td>DSLGrid control.</td>
</tr>
<tr>
<td>AutoLineNbrFld</td>
<td>Integer</td>
<td>If the detail lines are kept unique via the use of a line number key then a reference to the line number field should be passed via this argument. SWIMAPI will manage line numbers at runtime by automatically incrementing the value of the field referenced in this argument. This field must be a member of the table structure referenced in the bTable1 argument. If detail lines are kept unique via the use of a data entry key (for example, line numbers are not used) then PNULL should be passed as the value for this argument.</td>
</tr>
<tr>
<td>bTable1</td>
<td>SolomonDataObject</td>
<td>SolomonDataObject corresponding to the primary table in the Spreadsheet and underlying memory array.</td>
</tr>
<tr>
<td>bTable2</td>
<td>SolomonDataObject</td>
<td>Second SolomonDataObject. PNULL if the grid only contains one table structure.</td>
</tr>
<tr>
<td>bTable3</td>
<td>SolomonDataObject</td>
<td>Third SolomonDataObject. PNULL if the grid contains less than three table structures.</td>
</tr>
<tr>
<td>bTable4</td>
<td>SolomonDataObject</td>
<td>Fourth SolomonDataObject. PNULL if the grid contains less than four table structures.</td>
</tr>
</tbody>
</table>
These calls are only required for screens implementing a Detail level (that is, a D or DA level in the Update.Levels property). In these cases, the `DetailSetup` and `DetailSetup8` calls must be made within the Form_Load event of Form1. Additionally the call must occur after `ScreenInit` and any optional `AutoNbrDefault` calls.

**See Also**

`DetailLoad Statement`, `DetailSave Statement`, `DetailSetupExtend Function`, `MGetDelHandle Function`, `ScreenInit Statement`, `DSGrid Control`, `DSUpdate Control`

**Example**

The following code illustrates a `DetailSetup` call in the context of a two level Batch / Transaction screen. In this case `bPRTran.LineNbr` line number values will be automatically managed by SWIMAPI.

```vbscript
Sub Form_Load()

    Call ApplInit

    'SetAddr calls for primary table on each screen level
    Call setaddr(LEVEL0, "bBatch", bBatch, nBatch)
    Call setaddr(LEVEL1, "bPRTran", bPRTran, nPRTran)

    'Miscellaneous SetAddr calls
    Call setaddr(NOLEVEL, "bEarnType", bEarnType, nEarnType)
    Call setaddr(NOLEVEL, "bEmployee", bEmployee, nEmployee)

    'SqlCursor calls for primary table on each screen level
    Call SqlCursor(CSR_Batch, LEVEL0) 'BATCH - 'N'ormal level
    Call SqlCursor(CSR_PRTran, LEVEL1) 'PRTRAN - 'D'etail level

    'Miscellaneous SqlCursor calls
    Call SqlCursor(CSR_EarnType, NOLEVEL)
    Call SqlCursor(CSR_Employee, NOLEVEL)

    Call ScreenInit

    'Setup for auto batch numbering
    Serr1 = AutoNbrDefault(cBatNbrH, "PRAutoBatchNbr", cBatNbrD, CNULL)

    MemHandle_PRTran = `DetailSetup`(CSR_PRTran, Spread_PRTran, bPRTran.AddressOf("LineNbr"), bPRTran, bEmployee, bEarnType, PNULL)

End Sub
```
DetailSetupExtend Function

Description
Bind an additional table that is not referenced by the DSLGrid. DBNav stored procedure to a grid already initialized via the use of the DetailSetup or DetailSetup8 function.

Syntax
RetVal = DetailSetupExtend(DSLGridCtrl, bTable1)

Remarks
This call allows additional tables to be added to an existing grid. It must be noted that the only difference between DetailSetup() table structures and DetailSetupExtend() table structures is that records from tables bound via the latter method are not retrieved via the DSLGridCtrl.DBNav SQL statement or stored procedure. For example, examine the following call to setup a hypothetical grid:

MemHandle = DetailSetup( CSR_DBNav, Spread_PRTran, PNULL, bPRTran, bNonDBMemBuffer, PNULL, PNULL)

In this example the data represented by bNonDBMemBuffer is not to be returned from the database, as its name implies. Thus, for example, the DSLGridCtrl DBNav statement could be something similar to:

Select * from PRTran where ....

The key to highlight in this Select statement is that data is only being retrieved from the PRTran table. In other words, NonDBMemBuffer appears nowhere in the entire SQL statement. If a developer were to attempt to implement a grid with the preceding DetailSetup call and its associated DBNav SQL statement, an error would occur. In particular, SWIMAPI would object to the fact that the size of the data referred to in the DBNav stored procedure (for example, the size of a single PRTran record) does not match the total length of all SolomonDataObjects passed in the DetailSetup call. Normally such an error message indicates an error in the application, and as such will elicit a grateful response from the developer for highlighting the exact nature of the problem. However, occasionally developers may run across a case where they really do need information from the database to coexist in a grid with data that does not come from the database. DetailSetupExtend was created to facilitate this type of functionality.

This API allows the developer to add tables to the grid with the understanding that data from such table(s) is not referred to by the DBNav SQL statement. Consequently, SWIMAPI will not include the length of any tables bound via any DetailSetupExtend call in the “length of a single detail line” for the purpose of comparing with the size of a composite data record referred to by the DBNav SQL statement. Rather it will only compare the size of a composite data record referred to by the DSLGridCtrl DBNav SQL statement with the total length of all tables in the DetailSetup or DetailSetup8 call.

The DetailSetupExtend function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RetVal</td>
<td>Integer</td>
<td>0 if extension of SpreadSheet and its underlying memory array was successful.</td>
</tr>
<tr>
<td>DSLGridCtrl</td>
<td>Control</td>
<td>DSLGrid control.</td>
</tr>
<tr>
<td>bTable1</td>
<td>SolomonDataObject</td>
<td>Supplementary SolomonDataObject to be added to the SpreadSheet and its underlying memory array.</td>
</tr>
</tbody>
</table>

See Also
DetailSetup Functions, DSLGrid Control
DispField Statements

Description
Display the value of the underlying data field(s) corresponding to the designated control(s).

Syntax
Call DispField(Control)
Call DispFields(Form, FirstControl, LastControl)

Remarks
Each DSL data control is associated with an underlying Visual Basic variable via a combination of its FieldName property and an associated SetAddr call. The system will automatically redisplay the new value of relevant controls anytime the system is the entity modifying the value of the underlying Visual Basic variable, such as when a new record is loaded. However, when the application directly modifies the value of a Visual Basic variable underlying an DSL data control, then it may also need to call one of the DispField statements to display the new value in the relevant control.

DispField and DispFields are functionally equivalent except that the latter statement can be used to display a range of controls based on their TabIndex property order.

The DispFields statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form</td>
<td>Control</td>
<td>Form containing all controls between FirstControl and LastControl inclusive. PNULL can be passed to include all loaded forms.</td>
</tr>
<tr>
<td>FirstControl</td>
<td>Control</td>
<td>First control whose underlying data value is to be displayed. PNULL can be passed to include all controls on the designated Form.</td>
</tr>
<tr>
<td>LastControl</td>
<td>Control</td>
<td>Last control whose underlying data value is to be displayed. PNULL can be passed to include all controls on the designated Form.</td>
</tr>
</tbody>
</table>

See Also
FieldName Property, MDisplay Statement, SetAddr Statement, TabIndex Property

Example
The following code snippet from the Payroll Earnings Type Maintenance screen illustrates how the DispField statement should be used after the Visual Basic variable underlying a particular data control has been modified programmatically. The code is from the Chk event of the Earnings Type combo box control.

Sub cType_Chk (chkstrg As String, retval As Integer)

    'Value must be in buffer before properties can be re-evaluated
    bEarnType.Type = chkstrg

    'Benefit Class ID only applies to earnings types which use benefits.
    If (bEarnType.Type <> "B") Then
        bEarnType.BenClassId = ""
        Call DispField(cBenClassId)
    End If

Else
'Earnings Types which do use benefits must contribute to 'Net Pay 'with a 1.0 multiplier. This is an assumption 'the 02.40 code which actually books the liability to the 'General Ledger. 

bEarnType.NetPay = LTRUE
bEarnType.PayRateMult = 1#

Call DispField(cNetPay)
Call DispField(cPayRateMult)

End If

'Re-evaluate all properties which are conditional on earnings type
Call Evaluate_Properties(FLD_BENCLASSID)
Call Evaluate_Properties(FLD_NETPAY)
Call Evaluate_Properties(FLD_PAYRATEMULT)

End Sub

DispForm Statement

Description
Display a designated subform.

Syntax
Call DispForm(SubFormName, CenterIt)

Remarks
DispForm will cause the designated subform to be displayed modally, meaning that no other form from the same application can receive focus until the subform is hidden via a call to HideForm.

Form1 is always displayed automatically by the system. Consequently, this call is only necessary for subforms. However the subform must have previously been loaded in the Form1_Load event using the LoadForm statement.

The DispForm statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SubFormName</td>
<td>Control</td>
<td>Form to be displayed modally.</td>
</tr>
<tr>
<td>CenterIt</td>
<td>Integer</td>
<td>TRUE is the subform is to be centered on the screen. FALSE if the form should be displayed at its design time coordinates.</td>
</tr>
</tbody>
</table>

See Also
HideForm Statement, LoadForm Statement

Example
Sub cPayrollTables_Click ()
    Call DispForm(F0229002, True)
End Sub
**DisplayMode Statement**

**Description**
Toggle “display only” mode.

**Syntax**
Call `DisplayMode(Mode)`

**Remarks**
This call can be used to easily put an application into “display only” mode using a single line of code rather than being forced to disable each individual control.

Once `DisplayMode` has been called, all calls to `SetProps` referencing the Enabled property are essentially ignored. The advantage of this type of architecture is that it reduces code complexity. In particular, the application can issue calls to the `SetProps` statement without surrounding each and every call with a conditional statement (such as “If DisplayMode = False”).

When an application is placed into “display only” mode, several of the Parent toolbar buttons will also be affected. In particular, the Insert, Delete and Save buttons for all levels will be disabled after their current setting has been saved. When `DisplayMode` is subsequently called to take the application out of “display only” mode, the original settings for these Parent toolbar buttons will be restored.

All calls to `DisplayMode` are ignored if the application is automatically placed into “display only” mode by the system due to access rights.

The `DisplayMode` statement uses the following argument:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>Integer</td>
<td>True to place the application into “display only” mode. False to take an the application out of “display only” mode.</td>
</tr>
</tbody>
</table>

**See Also**
`DisplayModeSetProps Statement`, `SetButton Statement`, `SetProps Statement`

**DisplayModeSetProps Statement**

**Description**
Modify a control property regardless of whether or not the application is in display mode.

**Syntax**
Call `DisplayModeSetProps(Form, FirstControl, LastControl, PropertyToModify, NewPropertyValue)`

**Remarks**
This statement is functionally equivalent to the `SetProps` statement with the exception that calls to `DisplayModeSetProps` will modify property values without regard to whether the application has been placed in display only mode via the `DisplayMode` statement.

`DisplayModeSetProps` should only be used on those rare occasions when an application uses the `DisplayMode` statement to implement display only functionality but at the same time needs to manipulate the properties of one or more controls. `SetProps` cannot be used in these specific instances since such calls are overridden anytime the application is in display only mode. In the vast majority of cases this will be an advantage since the application does not need to contain code determining whether or not it is in display only mode before deciding which `SetProps` calls to make and which ones not to make. On the contrary the application can make a single call to `DisplayMode` which will automatically override all subsequent calls to `SetProps` until `DisplayMode` is called again to
get out of display only mode. If the property of one or more controls needs to be manipulated while the
application is in display mode, then the **DisplayModeSetProps** statement should be utilized.

**DisplayModeSetProps** uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form</td>
<td>Control</td>
<td>Form containing all controls from FirstControl to LastControl inclusive. PNULL can be used to include all forms within the application.</td>
</tr>
<tr>
<td>FirstControl</td>
<td>Control</td>
<td>First control to be assigned a new property value for the PropertyToModify.</td>
</tr>
<tr>
<td>LastControl</td>
<td>Control</td>
<td>Last control to be assigned a new property value. A particular property can be modified for all controls on the Form by passing PNULL for both the FirstControl and LastControl arguments.</td>
</tr>
<tr>
<td>PropertyToModify</td>
<td>String</td>
<td>Name of the property whose value is to be modified.</td>
</tr>
<tr>
<td>NewPropertyValue</td>
<td>String or Integer</td>
<td>New value to be assigned to the designated property of all applicable controls. <strong>Note:</strong> The type of data actually required for this argument depends on which property is being modified.</td>
</tr>
</tbody>
</table>

**See Also**
DisplayMode Statement, SetProps Statement

**DParm Function**

**Description**
Convert a date into an SQL parameter string.

**Syntax**

```
SQLParmStr = DParm(DateToConvert)
```

**Remarks**
The **DParm** function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLParmStr</td>
<td>String</td>
<td><strong>DateToConvert</strong> converted into an SQL parameter string.</td>
</tr>
<tr>
<td>DateToConvert</td>
<td>Integer</td>
<td>Date value to convert.</td>
</tr>
</tbody>
</table>

**See Also**
FParm Function, IParm Function, SParm Function
ExportCustom Function

Description
Export one or more customizations to an ASCII text file.

Syntax
RetVal = ExportCustom(KeySegScreenNbr, KeySegUserID, KeySegBegLevel, KeySegEndLevel, KeySegLanguageId, KeySegCpnyName, OutputFile, AppendToFile)

Remarks
Customizations can be copied to other databases by first exporting them from the source database to an ASCII text file using the ExportCustom function. The resulting export file can subsequently be imported into a destination database using the ImportCustom function. The Export Customizations (91.500) and Import Customizations (91.510) application screens utilize these two functions to perform their export and import work.

Each call to ExportCustom will export all customizations whose unique key corresponds to the various KeySeg parameters.

The following are valid values for KeySegBegLevel and KeySegEndLevel:
- 100 — Language
- 200 — Supplemental Product
- 300 — All Users
- 400 — One User
- 500 — Self

See Also
ImportCustom Function
The ExportCustom function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RetVal</td>
<td>Integer</td>
<td>A non-zero return value indicates an error occurred during the export.</td>
</tr>
<tr>
<td>KeySegScreenNbr</td>
<td>String</td>
<td>Screen ID of the Microsoft Dynamics SL SDK application to which the customization being exported applies. For example, if the customization applies to the Sales Order and Memo (05.260.00), then 05260 should be passed. SQL wildcard characters are supported.</td>
</tr>
<tr>
<td>KeySegUserID</td>
<td>String</td>
<td>If the level of the source customization is Self then User ID should contain the ID of the user who created the customization. Correspondingly if the level of the source customization is One User then User ID should contain the ID of the user to which the customization applies. SQL wildcard characters are supported.</td>
</tr>
<tr>
<td>KeySegBegLevel</td>
<td>Integer</td>
<td>Microsoft Dynamics SL allows various levels of customization to exist for any particular screen. For example, two customizations could exist for the Sales Order and Memo (05.260.00). One customization could be for All Users and the other could be for One User named THOMAS. Both customizations are for the same screen. The only difference is the Customization Level. KeySegBegLevel should contain the beginning (for example, lowest) level number of all the customizations which are to be exported for the specified KeySegScreenNbr.</td>
</tr>
<tr>
<td>Argument</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>KeySegEndLevel</td>
<td>Integer</td>
<td>Ending (highest) level number of all the customizations which are to be exported for the specified KeySegScreenNbr. If only a single customization is to be exported then KeySegBegLevel and KeySegEndLevel should both contain the precise level number of the particular customization which is to be exported.</td>
</tr>
<tr>
<td>KeySegLanguageId</td>
<td>String</td>
<td>Language Id of the source customization. SQL wildcard characters are supported.</td>
</tr>
<tr>
<td>KeySegCpnyName</td>
<td>String</td>
<td>Multiple customizations can exist within the Supplemental Product customization level, differentiated only by their Company Name. KeySegCpnyName facilitates the export of a particular customization at the Supplemental Product level by allowing a unique Company Name to be specified. SQL wildcard characters are supported.</td>
</tr>
<tr>
<td>OutputFile</td>
<td>String</td>
<td>Fully qualified name of the file to which the customization(s) are to be exported. The standard file extension for customization export files is CST.</td>
</tr>
<tr>
<td>AppendToFile</td>
<td>Integer</td>
<td>True if the customization should be appended to the OutputFile. False if the contents of OutputFile should be overwritten.</td>
</tr>
</tbody>
</table>

**Example**

The following example illustrates a simple process that will export all of the Custom records contained in the Mem_Custom memory array.

**Note:** The initialization of the Mem_Custom array, such as opening and loading the memory array, is purposely not included so as to focus on the `ExportCustom` function call.

```vba
Dim Mem_Fetch As Integer
Dim Mem_MaintFlg As Integer
Dim Nbr_Selected_Recs_Processed As Integer
Dim File_Append_Flag As Integer
Dim Error_Encountered As Integer

Call status(StartProcess, False, "", DISP_ONLY)
Mem_Fetch = mfirst(Mem_Custom, Mem_MaintFlg)
While ((Mem_Fetch = 0) And (Error_Encountered = 0))
  'In the event the output file already exists, the first customization should replace its entire contents. All customizations subsequently exported should be appended to the output file.
  If (Nbr_Selected_Recs_Processed = 0) Then
    File_Append_Flag = False
  Else
    File_Append_Flag = True
  End If

  'Export the current customization
  Error_Encountered = ExportCustom(bCustom.ScreenId, 
                                    bCustom.UserId, bCustom.Sequence, bCustom.Sequence, 
                                    bCustom.LanguageId, bCustom.Company_Name, "CUSTOM.CST", 
                                    File_Append_Flag)
```
'Update counter controlling file append

Nbr_Selected_Recs_Processed = Nbr_Selected_Recs_Processed + 1

Mem_Fetch = mnext(Mem_Custom, Mem_MaintFlg)

Wend

Call status(EndProcess, False, "", DISP_ONLY)

**FPAdd Function**

**Description**
Add two double-precision floating-point values together with a designated rounding precision.

**Syntax**

\[ Result = \text{FPAdd}(Db1, Db2, \text{Precision}) \]

**Remarks**
Error conditions occurring during the addition operation, such as an overflow error, will be handled automatically by the system. These types of errors will cause the appropriate error message to be either displayed on the screen or written to the process status log depending on the context in which the error occurred. After an error condition has been properly reported, the application will be terminated.

The **FPAdd** function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result</td>
<td>Double</td>
<td>Return value</td>
</tr>
<tr>
<td>Db1</td>
<td>Double</td>
<td>First double value</td>
</tr>
<tr>
<td>Db2</td>
<td>Double</td>
<td>Second double value</td>
</tr>
<tr>
<td>Precision</td>
<td>Integer</td>
<td>Rounding precision.</td>
</tr>
</tbody>
</table>

**Note:** The precision parameter can be an explicit precision value as well as one of the following symbolic constants defined in Solomon.VBTools.vb:

- **MONEY** — Monetary value
- **INV_UNIT_QTY** — Inventory unit quantity
- **UNITS** — Work units such as hours worked in Payroll
- **INV_UNIT_PRICE** — Inventory unit price
- **PERCENT** — Percentage value

**See Also**
FPDiv Function, GetPrecision Function, FPMult Function, FPRnd Function, FPSub Function

**Example**
The following example illustrates how to add a fixed amount to a double-precision floating-point value and let the system handle all rounding issues.

\[ b\text{Employee.YTDEarn} = \text{FPAdd}(b\text{Employee.YTDEarn}, 1000.12, \text{MONEY}) \]
FParm Function

Description
Convert a double-precision floating-point value into an SQL parameter string.

Syntax
SQLParmStr = FParm(DblToConvert)

Remarks
The FParm function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLParmStr</td>
<td>String</td>
<td>DblToConvert converted into an SQL parameter string.</td>
</tr>
<tr>
<td>DblToConvert</td>
<td>Double</td>
<td>Double-precision floating-point value to convert.</td>
</tr>
</tbody>
</table>

See Also
DParm Function, IParm Function, SParm Function

FPDiv Function

Description
Divide one double-precision floating-point value by another with a designated rounding precision.

Syntax
Result = FPDiv(Dbl1, Dbl2, Precision)

Remarks
This function will divide the value of Dbl1 by Dbl2 and return the result.

Error conditions occurring during the division operation, such as division by zero, will be handled automatically by the system. These types of errors will cause the appropriate error message to be either displayed on the screen or written to the process status log depending on the context in which the error occurred. After an error condition has been properly reported, the application will be terminated.

The FPDiv function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result</td>
<td>Double</td>
<td>Return value</td>
</tr>
<tr>
<td>Dbl1</td>
<td>Double</td>
<td>First double value</td>
</tr>
<tr>
<td>Dbl2</td>
<td>Double</td>
<td>Second double value</td>
</tr>
<tr>
<td>Precision</td>
<td>Integer</td>
<td>Rounding precision.</td>
</tr>
</tbody>
</table>

Note: The precision parameter can be an explicit precision value as well as one of the following symbolic constants defined in Solomon.VBTools.vb:

- MONEY — Monetary value
- INV_UNIT_QTY — Inventory unit quantity
- UNITS — Work units such as hours worked in Payroll
- INV_UNIT_PRICE — Inventory unit price
- PERCENT — Percentage value
See Also
FPAdd Function, GetPrecision Function, FPMult Function, FPRnd Function, FPSub Function

Example
The following example illustrates how to divide a double-precision floating-point value by a fixed amount and let the system handle all rounding issues:

```
WeeklySalary = FPDiv(bEmployee.StdSlry, 52, MONEY)
```

FPMult Function

Description
Multiply two double-precision floating-point values together to a designated rounding precision.

Syntax
```
Result = FPMult(Dbl1, Dbl2, Precision)
```

Remarks
Error conditions occurring during the multiplication operation, such as an overflow error, will be handled automatically by the system. These types of errors will cause the appropriate error message to be either displayed on the screen or written to the process status log depending on the context in which the error occurred. After an error condition has been properly reported, the application will be terminated.

The FPMult function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result</td>
<td>Double</td>
<td>Return value</td>
</tr>
<tr>
<td>Dbl1</td>
<td>Double</td>
<td>First double value</td>
</tr>
<tr>
<td>Dbl2</td>
<td>Double</td>
<td>Second double value</td>
</tr>
<tr>
<td>Precision</td>
<td>Integer</td>
<td>Rounding precision.</td>
</tr>
</tbody>
</table>

Note: The precision parameter can be an explicit precision value as well as one of the following symbolic constants defined in Solomon.VBTools.vb:

- **MONEY** — Monetary value
- **INV_UNIT_QTY** — Inventory unit quantity
- **UNITS** — Work units such as hours worked in Payroll
- **INV_UNIT_PRICE** — Inventory unit price
- **PERCENT** — Percentage value

See Also
FPAdd Function, FPDiv Function, GetPrecision Function, FPRnd Function, FPSub Function

Example
The following example illustrates how to multiply a double-precision floating-point value by a fixed amount and let the system handle all rounding issues:

```
bEmployee.StdSlry = FPMult(WeeklySalary, 52, MONEY)
```
FPRnd Function

Description
Round a double-precision floating-point value to a designated rounding precision.

Syntax
\[ \text{Result} = \text{FPRnd}(\text{DblToRound}, \text{Precision}) \]

Remarks
Error conditions occurring during the rounding operation, such as an overflow error, will be handled automatically by the system. These types of errors will cause the appropriate error message to be either displayed on the screen or written to the process status log depending on the context in which the error occurred. After an error condition has been properly reported, the application will be terminated.

The FPRnd function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result</td>
<td>Double</td>
<td>Return value</td>
</tr>
<tr>
<td>DblToRound</td>
<td>Double</td>
<td>Value to be rounded</td>
</tr>
<tr>
<td>Precision</td>
<td>Integer</td>
<td>Rounding precision.</td>
</tr>
</tbody>
</table>

Note: The precision parameter can be an explicit precision value as well as one of the following symbolic constants defined in Solomon.VBTools.vb:

- MONEY — Monetary value
- INV_UNIT_QTY — Inventory unit quantity
- UNITS — Work units such as hours worked in Payroll
- INV_UNIT_PRICE — Inventory unit price
- PERCENT — Percentage value

See Also
FPAdd Function, FPDiv Function, GetPrecision Function, FPMult Function, FPSub Function

Example:
Dim Result As Double

'Perform multiplication in straight Visual Basic code (as opposed to using FPMult)
'so the application can detect an overflow error and handle it 'appropriately.
    Result = bAPTran.Qty * bAPTran.UnitPrice

'Handle overflow errors
    If (Result >= OVERFLOW) Then
        'Display the following error message: "Last operation exceeds
        'field size, please reenter"
        Call Mess(1)
    Else
'Round to a monetary value
    bAPTran.TranAmt = FPRnd(Result, MONEY)
End If

**FPSub Function**

**Description**
Subtract one double-precision floating-point value from another with a designated rounding precision.

**Syntax**

\[
\text{Result} = \text{FPSub}(\text{Dbl1}, \text{Dbl2}, \text{Precision})
\]

**Remarks**
This function will subtract the value of \( \text{Dbl2} \) from \( \text{Dbl1} \) and return the result.

Error conditions occurring during the subtraction operation, such as an overflow error, will be handled automatically by the system. These types of errors will cause the appropriate error message to be either displayed on the screen or written to the process status log depending on the context in which the error occurred. After an error condition has been properly reported, the application will be terminated.

The **FPSub** function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result</td>
<td>Double</td>
<td>Return value</td>
</tr>
<tr>
<td>Dbl1</td>
<td>Double</td>
<td>First double value</td>
</tr>
<tr>
<td>Dbl2</td>
<td>Double</td>
<td>Second double value</td>
</tr>
<tr>
<td>Precision</td>
<td>Integer</td>
<td>Rounding precision.</td>
</tr>
</tbody>
</table>

**Note:** The precision parameter can be an explicit precision value as well as one of the following symbolic constants defined in Solomon.VBTools.vb:

- **MONEY** — Monetary value
- **INV_UNIT_QTY** — Inventory unit quantity
- **UNITS** — Work units such as hours worked in Payroll
- **INV_UNIT_PRICE** — Inventory unit price
- **PERCENT** — Percentage value

**See Also**
FPAdd Function, FPDiv Function, GetPrecision Function, FPMult Function, FPRnd Function

**Example**
The following example illustrates how to subtract a fixed amount from a double-precision floating-point value and let the system handle all rounding issues:

\[
b\text{Employee.YTDEarn} = \text{FPSub}(b\text{Employee.YTDEarn}, 1000.12, \text{MONEY})
\]
FtoA Function

Description
Convert a double-precision floating-point value into an ASCII string with the designated rounding precision.

Syntax

\[
\text{Str} = \text{FtoA( DblToConvert, Precision)}
\]

Remarks
The FtoA function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Str</td>
<td>String</td>
<td>Returned string</td>
</tr>
<tr>
<td>DblToConvert</td>
<td>Double</td>
<td>Double-precision floating-point value to convert.</td>
</tr>
<tr>
<td>Precision</td>
<td>Integer</td>
<td>Rounding precision.</td>
</tr>
</tbody>
</table>

Note: The precision parameter can be an explicit precision value as well as one of the following symbolic constants defined in Solomon.VBTools.vb:

- MONEY — Monetary value
- INV_UNIT_QTY — Inventory unit quantity
- UNITS — Work units such as hours worked in Payroll
- INV_UNIT_PRICE — Inventory unit price
- PERCENT — Percentage value

See Also
FPRnd Function, GetPrecision Function

GetCuryRate Statement

Description
Retrieves Currency Rate information from the Currency Rate table in the database, based upon the Transaction Currency ID, Base Currency ID, Rate Type, and Effective Date values in the Currency Selection Form fields at the time of the call. Updates the Rate, Rate Reciprocal, and Multiply/Divide fields of the Currency Selection Form from the record retrieved from the database. The record with a matching Transaction Currency ID, Base Currency ID, and Rate Type, and an Effective Date on or before the specified Effective Date will be retrieved, if it exists. If a Currency Rate table entry is not found, an error message is displayed. If new rate information is successfully retrieved, all Base Currency values for all monetary fields specified for multi-currency processing are recalculated, based upon the new rate.

Syntax

Call GetCuryRate

Remarks
New rate information is typically retrieved into the Currency Selection form when data entered into the application causes one of the key fields used to establish previous Currency Rate Information to change. For example, if the user enters a vendor ID into a document, and the application logic re-defaults the Rate Type field in the Currency Selection Form to the default value for the Vendor, the
Reference

185

application will typically need to retrieve the Currency Rate Information corresponding to the new rate
type.
The GetCuryRate statement does not require any arguments.

Example
The following example shows the Vendor Chk event for an application being enabled for multi-currency
processing. If the Vendor record contains a valid rate type, the rate type in the Currency Selection
Form is re-defaulted to the Vendor default value, and new Currency Rate Information is retrieved using
the GetCuryRate statement. Note that CuryRateTypeSet is called before GetCuryRate. Since
GetCuryRate uses the current contents of the Currency Selection Form as key values to look up the
new rate, it is important that any necessary updates to the key values in the Currency Selection Form
be performed before calling GetCuryRate.
Sub cvendid_Chk (chkstrg As String, retval As Integer)
Dim maintflg As Integer
'Fetch vendor record to display Name, status and Terms
retval = pvchkfetch1(cvendid, C3, chkstrg, bVendor)
'Currency Manager
If retval = 0 Then
'If multi-currency entry is activated...
If bCMSetup.MCActivated = LTRUE Then
'If Vendor Currency ID is specified, and
'is different from Batch Currency ID...
If Trim$(bVendor.CuryID) <> "" And Trim$(bVendor.CuryID) <> ↵
Trim$(bCuryInfo.TranCuryId) Then
'Give error or warning, depending upon
'setup option.
If bCMSetup.APCuryOverride = LFALSE Then
retval = 6090
Exit Sub
Else
Call mess(6091)
End If
End If
'If Tran Currency is different from Base Currency...
If Trim$(bCuryInfo.TranCuryId) <> Trim$(bPES.BaseCuryID)
'Then If Vendor Rate Type is specified, and
'is different from Batch Rate Type...
If Trim$(bVendor.CuryRateType) <> "" And ↵
Trim$(bVendor.CuryRateType) <> Trim$(bCuryInfo.RateType) ↵
Then
'Check whether Vendor Rate Type exists
serr1 = ChkCuryRateType(bVendor.CuryRateType)
If serr1 <> 0 Then
'If not, exit with error
retval = 6092
Exit Sub


Else
    'If it does, try to retrieve Rate Table entry
    Call CuryRateTypeSet(bVendor.CuryRateType)
    Call GetCuryRate
    End If
End If
End If
End If
End If
End If
End If
End Sub

GetModulePeriod Function

Description
Retrieve the current fiscal period of a designated Microsoft Dynamics SL module.

Syntax

\[
\text{CurrFiscalPeriod} = \text{GetModulePeriod}(\text{SIVModuleNbr})
\]

Remarks
There are two ways to determine the current fiscal period of a module. The first and most obvious method is to retrieve the setup record for the appropriate module directly from the database. An easier method, however, is to simply call the \text{GetModulePeriod} function specifying the relevant module number.

The \text{GetModulePeriod} function has the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{CurrFiscalPeriod}</td>
<td>String</td>
<td>Current fiscal period of the relevant module.</td>
</tr>
<tr>
<td>\text{SIVModuleNbr}</td>
<td>String</td>
<td>Number of a specific module whose current period is being requested. The module number string must always be two bytes (for example, “01” instead of “1” for General Ledger).</td>
</tr>
</tbody>
</table>

See Also

\text{PeriodMinusPeriod Function, PeriodPlusPerNum Function}
GetPrecCury Function

Description
Return the rounding precision associated with the designated Currency ID.

Syntax
\[ Precision = \text{GetPrecCury}(CurrencyID) \]

Remarks
Each currency defined within Microsoft Dynamics SL can use a different decimal precision. Consequently, all floating point calculations on currency related fields, such as transaction amounts, should round their result to match the rounding precision of the relevant currency. The \text{GetPrecCury} function facilitates this objective by allowing applications to easily obtain the rounding precision associated with a particular currency.

In applications where only one transaction currency and one base currency are in effect at any given time, the preferred method for specifying the precision parameter in a floating point calculation is to use either the TRANCURY or BASECURY constant defined in Solomon.VBTools.vb. However, in applications where multiple transaction currencies and/or base currencies are displayed or processed, the precision must be obtained for the particular currency represented in the calculation.

The \text{GetPrecCury} function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision</td>
<td>Integer</td>
<td>Rounding precision utilized by the designated currency. If CurrencyID could not be found in the database then PREC_OVERFLOW will be returned.</td>
</tr>
<tr>
<td>CurrencyID</td>
<td>String</td>
<td>Unique ID of the currency for which a rounding precision is desired.</td>
</tr>
</tbody>
</table>

Example
Sub TranTotal_Update (ByVal Acct As String, ByVal Subacct As String, ByVal CuryID As String, ByVal BaseCuryID As String, ByVal CuryCreditAmt As Double, ByVal CuryDebitAmt As Double, ByVal CreditAmt As Double, ByVal DebitAmt As Double, CuryTranCRTot As Double, CuryTranDBTot As Double, TranCrTot As Double, TranDBTot As Double)

'Currency Manager - use currency field reference for test
If ((CuryCreditAmt <> 0#) And (CuryDebitAmt <> 0#)) Then
  'Tran can NEVER have both a credit AND debit amount - as we would not
  'know for sure which value to use in our system.
  Call Status(MSG TRANC_S1_S2_CANT_CR_AND_DR_AMT, True, SParm(Acct) + SParm(Subacct), LOG_AND_DISP)
Else

  If (CuryCreditAmt <> 0#) Then
    'Credit Transaction
    TranCrTot = FPadd(TranCrTot, CreditAmt, GetPrecCury(BaseCuryID))
    CuryTranCRTot = FPadd(CuryTranCRTot, CuryCreditAmt, GetPrecCury(CuryID))
  Else

  Else
'Assume Debit Transaction
TranDBTot = FPadd(TranDBTot, DebitAmt, GetPrecCury(BaseCuryID))
CuryTranDBTot = FPadd(CuryTranDBTot, CuryDebitAmt, GetPrecCury(CuryId))

End If

End If

End

See Also
DecimalPlaces Statement, FPAdd Function, FPDiv Function, FPMult Function, FPRnd Function, FPSub Function, Get Precision Function

GetPrecision Function

Description
Return the rounding precision for the designated rounding constant.

Syntax
RoundingPrecision = GetPrecision (RoundingConstant)

Remarks
This function allows the application to obtain the actual rounding precision value associated with one of the rounding constants defined in Solomon.VBTools.vb.
The GetPrecision function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RoundingPrecision</td>
<td>Integer</td>
<td>Number of decimal places to which the designated constant is rounded.</td>
</tr>
<tr>
<td>RoundingConstant</td>
<td>Integer</td>
<td>The following valid values are defined as symbolic constants in Solomon.VBTools.vb:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MONEY — Monetary value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INV_UNIT_QTY — Inventory unit quantity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UNITS — Work units such as hours worked in Payroll</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INV_UNIT_PRICE — Inventory unit price</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PERCENT — Percentage value</td>
</tr>
</tbody>
</table>

See Also
FPAdd Function, FPDiv Function, FPMult Function, FPRnd Function, FPSub Function
GetSWIMDefaultPrintInfo Function

Description
Retrieve the default print information and settings.

Syntax
result = GetSWIMDefaultPrintInfo( printInfo )

Remarks
The GetSWIMDefaultPrintInfo function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result</td>
<td>Boolean</td>
<td>Return value indicating success or failure of function call.</td>
</tr>
<tr>
<td>printInfo</td>
<td>PInfo</td>
<td>PInfo structure containing print information and settings.</td>
</tr>
</tbody>
</table>

PInfo Structure
The PInfo structure describes all of the settings used to create report output and send it to a printer, disk file, or preview window.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeviceName</td>
<td>String * 256</td>
<td>Name of printer</td>
</tr>
<tr>
<td>DriverName</td>
<td>String * 256</td>
<td>Name of print driver</td>
</tr>
<tr>
<td>PrintPort</td>
<td>String * 256</td>
<td>Port of printer</td>
</tr>
<tr>
<td>PrintDestinationName</td>
<td>String * 256</td>
<td>Name of destination file for exported report</td>
</tr>
<tr>
<td>PrintFileType</td>
<td>String * 2</td>
<td>File type code (documented in standard and advanced reporting guide under RIPARAM RI_FILETYPE)</td>
</tr>
<tr>
<td>PrintToFile</td>
<td>Integer</td>
<td>1=Printing to file, 0=not printing to file</td>
</tr>
<tr>
<td>PrintIncludeCodes</td>
<td>Integer</td>
<td>No longer used</td>
</tr>
<tr>
<td>DevMode</td>
<td></td>
<td>Fields specified in PInfo to avoid alignment problem. See Windows definition of DEVMODE structure for description of values/settings.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dmDeviceName</td>
<td>String * 32</td>
<td></td>
</tr>
<tr>
<td>dmSpecVersion</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>dmDriverVersion</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>dmSize</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>dmDriverExtra</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>dmFields</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>dmMyFiller</td>
<td>Integer</td>
<td>Used to avoid alignment problems</td>
</tr>
<tr>
<td>dmOrientation</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>dmPaperSize</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>dmPaperLength</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>dmPaperWidth</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>dmScale</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>dmCopies</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>dmDefaultSource</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>dmPrintQuality</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>dmColor</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>dmDuplex</td>
<td>Integer</td>
<td></td>
</tr>
</tbody>
</table>
## Software Development Kit

### Setting

<table>
<thead>
<tr>
<th>Setting</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dmYResolution</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>dmTTOption</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>dmCollate</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>dmFormName</td>
<td>String * 32</td>
<td></td>
</tr>
<tr>
<td>dmLogPixels</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>dmBitsPerPel</td>
<td>Long</td>
<td></td>
</tr>
<tr>
<td>dmPelsWidth</td>
<td>Long</td>
<td></td>
</tr>
<tr>
<td>dmPelsHeight</td>
<td>Long</td>
<td></td>
</tr>
<tr>
<td>dmDisplayFlags</td>
<td>Long</td>
<td></td>
</tr>
<tr>
<td>dmDisplayFrequency</td>
<td>Long</td>
<td></td>
</tr>
<tr>
<td>dmICMMethod</td>
<td>Long</td>
<td></td>
</tr>
<tr>
<td>dmICMIntent</td>
<td>Long</td>
<td></td>
</tr>
<tr>
<td>dmMediaType</td>
<td>Long</td>
<td></td>
</tr>
<tr>
<td>dmDitherType</td>
<td>Long</td>
<td></td>
</tr>
<tr>
<td>dmICCManufacturer</td>
<td>Long</td>
<td></td>
</tr>
<tr>
<td>dmICCModel</td>
<td>Long</td>
<td></td>
</tr>
<tr>
<td>dmPanningWidth</td>
<td>Long</td>
<td></td>
</tr>
<tr>
<td>dmPanningHeight</td>
<td>Long</td>
<td></td>
</tr>
</tbody>
</table>

### FontInfo

<table>
<thead>
<tr>
<th>Setting</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fiFontName</td>
<td>String * 32</td>
<td>Font name</td>
</tr>
<tr>
<td>fiFontSize</td>
<td>Integer</td>
<td>Font size</td>
</tr>
<tr>
<td>fiBold</td>
<td>Integer</td>
<td>1=bold, 0=normal</td>
</tr>
<tr>
<td>fiItalic</td>
<td>Integer</td>
<td>1=italic, 0=normal</td>
</tr>
</tbody>
</table>

### WindowsDefault

<table>
<thead>
<tr>
<th>Setting</th>
<th>Integer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WindowsDefault</td>
<td>Integer</td>
<td>Send to Windows default printer</td>
</tr>
</tbody>
</table>

### PrinterOrientation

<table>
<thead>
<tr>
<th>Setting</th>
<th>Integer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrinterOrientation</td>
<td>Integer</td>
<td>1=Use printer orientation, 0=Use report orientation</td>
</tr>
</tbody>
</table>

### See Also

- GetSWIMPrintInfo Function, SetSWIMPrintInfo Function
GetSWIMPrintInfo Function

Description
Retrieve the current print information and settings.

Syntax
result = GetSWIMPrintInfo( printInfo )

Remarks
The GetSWIMPrintInfo function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result</td>
<td>Boolean</td>
<td>Return value indicating success or failure of function call.</td>
</tr>
<tr>
<td>printInfo</td>
<td>PInfo</td>
<td>PInfo structure containing print information and settings.</td>
</tr>
</tbody>
</table>

See Also
GetSWIMDefaultPrintInfo Function, SetSWIMPrintInfo Function, PInfo Structure

GetSysDate Statement

Description
Retrieve the current system date.

Syntax
Call GetSysDate(Date)

Remarks
The GetSysDate statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>Integer</td>
<td>Date variable to be initialized to the current system date.</td>
</tr>
</tbody>
</table>

See Also
GetSysTime Statement

GetSysTime Statement

Description
Retrieve the current system time.

Syntax
Call GetSysTime(Time)

Remarks
The GetSysTime statement uses the following arguments:
<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Integer</td>
<td>Time variable to be initialized to the current system time.</td>
</tr>
</tbody>
</table>

**See Also**

*GetSysDate Statement*

**Grid_Sortable Statement**

**Description**

Informs the system that the specified Grid control is to be treated as a user-sortable grid. This means that the end-user of the screen will be able to sort the information on the grid by clicking on the column header.

**Syntax**

Call `Grid_Sortable(LEVEL0, Spread1)`

**Remarks**

The `Grid_Sortable` statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level</td>
<td>Integer, 0 - 10</td>
<td>Specifies the Screen level number for the controls on the grid that is going to be sortable.</td>
</tr>
<tr>
<td>Spread</td>
<td>Control</td>
<td><code>DSLGrid</code> control.</td>
</tr>
</tbody>
</table>

The Grid control specified in the `Spread1` parameter must abide by the following conditions:

- There can be no key fields defined on the controls that make up the form view of the Grid control.
- When this statement is run, all the fields on the Grid will be disabled. All Update buttons will be disabled for the level specified in the first parameter. Insert, Update, and Delete operations will not be allowed for the level once this statement runs.
HideForm Statement

Description
Hide a designated subform previously displayed via a call to DispForm.

Syntax
Call HideForm(SubFormName)

Remarks
This function is typically used in the click event of the OK or Cancel button of the designated subform. The HideForm statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SubFormName</td>
<td>Control</td>
<td>Form to be hidden.</td>
</tr>
</tbody>
</table>

See Also
DispForm Statement, LoadForm Statement

Example
Sub cOK_PayrollTables_Click ()

    Call HideForm(F0229002)

End Sub

HideNoteButtons Statement

Description
Hides or displays the Notes/Attachments button for the specified level.

Syntax
Call HideNoteButtons(LevelNbr, HideFlag)

Remarks
The HideNoteButtons statement should be called to hide the notes/attachments button for a certain level. Specifying True for the second parameter hides the button. Specifying False displays the button (if one exists) for the level specified in the first parameter. This statement can also be called using the AllLevels constant to hide the notes/attachments buttons for all the levels in the application.

See Also
NoteButton Property, NoteColumn Property
ImportCustom Function

Description
Import customizations from an ASCII customization export file.

Syntax
RetVal = ImportCustom(ImportFile, ConflictResolution, ErrorHandling)

Remarks
Customizations can be imported directly into a database from a customization export file which previously by either the Export Customizations (91.500) screen or the ExportCustom function.

Each call to ImportCustom will import all customizations contained within the ImportFile. In the event an error occurs during the import operation a description of the error will be written to the process status log.

The ImportCustom function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RetVal</td>
<td>Integer</td>
<td>Zero if no errors occurred. Non-zero if one or more errors occurred (regardless of which error handling option was used).</td>
</tr>
<tr>
<td>ImportFile</td>
<td>String</td>
<td>Fully qualified name of the file from which the customization(s) are to be imported.</td>
</tr>
<tr>
<td>ConflictResolution</td>
<td>Integer</td>
<td>Option specifying how conflicts with existing customizations should be resolved. The following are valid values along with a corresponding description: 0 — Overwrite Existing Customization 1 — Reject New Customization 2 — Merge Both Customizations</td>
</tr>
<tr>
<td>ErrorHandling</td>
<td>Integer</td>
<td>Option specifying how errors detected in the ImportFile should be handled. The following are valid values along with a corresponding description: 0 — Ignore Syntax Errors 1 — Reject Entire File</td>
</tr>
</tbody>
</table>

Conflict Resolution Notes
While processing any particular Import File the system may detect that a customization already exists in the position where a new customization is destined for storage. For example, suppose that a customization with the following keys already exists in the database: Screen ID: 05260, Customization Level: One User, User ID: THOMAS and a blank Company Name. If the Import File contains a customization with these same key field values then an import conflict will occur. In this case there are several different methods of Conflict Resolution:

• Using the Overwrite option, the new customization will overwrite the current customization.

• However, if you are certain that you never want to overwrite any existing customizations, then you can specify the Reject Customization option. In the event of a conflict, this option will cause the new customization to be rejected.

• The most sophisticated option allows you to merge new customizations together with existing customizations. This is an extremely powerful feature which allows two customizations to be merged into one new customization down to the property level. For example, assume that the position of ControlA has been customized so that its on-screen position varies from the standard screen. Now, suppose that we want to import a customization in which a reference will be made to ControlA. In particular the new customization is going to disable ControlA. If the sophisticated Merge option is utilized, then the result will be that ControlA will have both its screen position and enabled properties customized! The only case in which a Merge cannot be successfully carried out
by the system is in the case where a conflict occurs at the property or BSL code procedure level. In our example, if the new customization being imported also customizes the screen position of ControlA, then a conflict at the property level will result. In such a case the new customization will take precedence over the existing customization.

**Error Handling Notes**

These options control how the system should respond to errors which may occur during the import operation. An Import Error occurs anytime a syntax error is found to exist in the Import File currently being processed. For example, each customization in the Import File must begin with “Begin Customization” and goes on to specify other key values. If the ASCII text in the Import File is “Begin Customization”, then an Import Error will result since the keyword Begin is misspelled.

There are two different methods of error handling:

- The Reject Entire File option will cause the entire Import File to be rejected if any Import Errors occur while importing any of the customizations contained therein. Since an Import File can contain many customizations this option facilitates an “all or nothing” type of import operation.
- An alternate method of handling Import Errors is to Ignore them. This option should be used with caution. The usefulness of this option is primarily during development of sophisticated customizations containing advanced Basic Script code. The ability to Ignore errors essentially provides the ability to compile the entire customization, including all Basic Script code, and receive a list of all errors in the entire customization in a single attempt. You can subsequently fix all of the errors and import the customization again this time using the Reject Entire File method of handling Import Errors. If an error actually occurs during the import process and you have chosen to Ignore errors the resulting customization may not operate properly. Suffice it to say if you receive any error during the import process you should make appropriate corrections to the Import File and import the customization again before attempting to actually use the screen being customized.

In addition to importing new customizations, the `ImportCustom` function can also be used to automatically delete existing customizations. The Import File simply needs to contain a line beginning with “Delete” followed by all the key field values necessary to identify a unique customization. For example, `Delete Screen: 05260 Sequence: 500 UserId: "SYSADMIN" CompanyName: ""`

You will notice a field called Sequence in the preceding example. Sequence is the technical term for customization level. The following are valid values for Sequence:

- **100** — Supplemental Product
- **200** — Language
- **300** — All Users
- **400** — One User
- **500** — Self

Within the Import File a Delete line can occur anywhere a Begin Customization line can occur. Thus a Delete line cannot occur within a Begin Customization and End Customization block of text. When you examine the contents of an Import File you will notice that its composition is very similar to a Visual Basic ASCII form file.

**See Also**

`ExportCustom Function`
Example

Begin Customization with various key fields identifying a unique customization.

    Begin ControlType ControlName
        Customized Properties
    End

    Begin ControlType ControlName
        Customized Properties
    End

    Begin Macro Text
        Sub ProcedureName( )
        End Sub
    End

End Customization

Begin Customization    ...Various key fields identifying a unique customization...
End Customization
.
.
Delete    ...Various key fields identifying a unique customization...

The following code fragment illustrates how to import all customizations contained in a file called CUSTOM.CST. Conflicts with existing customizations will be resolved by specifying that the new customization should overwrite the existing customization. All customizations contained within the CUSTOM.CST import file will be rejected if any syntax errors are encountered.

    Dim Record_Count    As Integer

    Call Status(StartProcess, False, "", DISP_ONLY)

    Record_Count = ImportCustom("CUSTOM.CST", 0, 1)

    Call Status(EndProcess, False, "", DISP_ONLY)
**IncrStrg Statement**

**Description**
Increment a string representation of a whole number value.

**Syntax**
Call **IncrStrg**(StringNbr, Length, Increment)

**Remarks**
The **IncrStrg** statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StringNbr</td>
<td>String</td>
<td>String whose current whole number is to be incremented.</td>
</tr>
<tr>
<td>Length</td>
<td>Integer</td>
<td>Size of StringNbr. It is not required that this value equal the full size of StringNbr. For example, if the string can actually hold 10 bytes but currently the developer only desires to use 6 byte values then a value of 6 can be passed.</td>
</tr>
<tr>
<td>Increment</td>
<td>Integer</td>
<td>Amount by which StringNbr is to be incremented.</td>
</tr>
</tbody>
</table>

**Example**
Dim BatNbrLength As Integer

BatNbrLength = bGLSetup.LastBatNbr.Trim.Length

'Increment last batch number to the next sequential value (within the 'size of batch numbers actually being used - e.g. BatNbrLength).
Call **IncrStrg**(bGLSetup.LastBatNbr, BatNbrLength, 1)

**InitLocalizationSubsystem Function**

**Description**
Called to initiate local language user interface support in Visual Basic programs where language translations are implemented using Microsoft Resource files. Determines whether the text of the program user interface will be loaded from language-translated resources, contained in a language-specific resource-only .DLL file, or from the standard U.S. English resources compiled into the program file. Saves a handle to the resources for later use by functions that apply resource values to the program user interface.

**Syntax**
Boolvar = InitLocalizationSubsystem(ScreenName)

**Remarks**
The Microsoft Resource file approach to local language user interface support must be used in only programs that cannot be accessed by Customization Manager. To use this function, LLI.BAS and LLI.DLL must be added to the Visual Basic project, and all language-specific text that is visible as part of the user interface must be moved to a Microsoft Resource file and replaced by resource IDs in the main code.

The first 5 characters of the Resource file name must be the same as the first 5 characters of the Visual Basic program file name. The 6th through 8th characters must contain the 3-letter Microsoft locale ID for U.S. English, which is “EMU.” When the program file name is fewer than 5 characters,
prior to appending the locale ID, zeros must be appended in the remaining positions to fill the first 5 characters.

System-related text, such as the numeric portion of the screen name, found in the Form1.Text must not be moved to the Resource file. The InitLocalizationSubsystem function must be called, prior to any other logic, in the Form1.Form_Load event.

The InitLocalizationSubsystem function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolvar</td>
<td>Boolean</td>
<td>Any Boolean variable</td>
</tr>
<tr>
<td>ScreenName</td>
<td>String</td>
<td>Visual Basic program file name, including extension</td>
</tr>
</tbody>
</table>

See Also
LoadStr Function, Localize Statement

Example
The following example shows the beginning logic of the Form_Load event for an application being enabled for local language user interface support, using the resource file method.

InitLocalizationSubsystem("2400000.exe")
ScreenTitle = LoadStr(IDS_CURRENCYSELECTION)
ScreenTitle = ScreenTitle + " " + SCREENNUMBER
With Form1
  .Text = ScreenTitle
  Call Localize(.ltocurrencyid)
  Call Localize(.lratetype)
  Call Localize(.lratereciprocal)
  .
  Call Localize(.cbuttoncancel)
EndWith

Call ApplInit

IntlStrToDate Statement

Description
Convert a date string from the Windows short date style into an SQL database date format.

Syntax
Call IntlStrToDate(DateStrToConvert, SQLDate)

Remarks
IntlStrToDate can be used to convert a string formatted according to the Windows short date style into a format suitable for storage in a date field within the SQL database.

The IntlStrToDate statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateStrToConvert</td>
<td>String</td>
<td>Date string to be converted. This string must be in the Windows short date format.</td>
</tr>
<tr>
<td>SQLDate</td>
<td>Integer</td>
<td>Converted date value.</td>
</tr>
</tbody>
</table>
**IParm Function**

**Description**
Convert an integer into an SQL parameter string.

**Syntax**

```
SQLParmStr = IParm(IntToConvert)
```

**Remarks**

The **IParm** function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLParmStr</td>
<td>String</td>
<td><em>IntToConvert</em> converted into an SQL parameter string.</td>
</tr>
<tr>
<td>IntToConvert</td>
<td>Integer</td>
<td>Integer value to convert.</td>
</tr>
</tbody>
</table>

**See Also**

DParm Function, FParm Function, SParm Function

**Example**

These examples assume the following SQL statement was used to create a stored procedure called `GLTran_Module_BatNbr_LineNbr`

```
Select * from GLTran
where Module  = @parm1
and BatNbr   = @parm2
and LineNbr between @parm3beg and @parm3end
order by Module, BatNbr, LineNbr;
```

This code snippet illustrates how the previously defined stored procedure can be used to fetch a single transaction having a LineNbr of 84 in GL Batch #000123.

```
SqlStr = "GLTran_Module_BatNbr_LineNbr" + SParm("GL") + SParm("000123") + IParm(84) + IParm(84)
GLTran_Fetch = SqlFetch1(CSR_GLTran, SqlStr, bGLTran)
```

This code snippet illustrates the previously defined stored procedure can be used to fetch all transactions in GL Batch #000123.

```
SqlStr = "GLTran_Module_BatNbr_LineNbr" + SParm("GL") + SParm("000123") + IParm(INTMIN) + IParm(INTMAX)
GLTran_Fetch = SqlFetch1(CSR_GLTran, SqlStr, bGLTran)
While ( GLTran_Fetch = 0)
    GLTran_Fetch = SFetch1( CSR_GLTran, bGLTran))
Wend
```
**IsAdministrator Function**

**Description**
Determine if the user is a member of the Administrators group. If `ParmStr` is empty or blank, the user who is currently logged on is used.

**Syntax**
```
Admin = IsAdministrator("bob")
```

**Remarks**
The `IsAdministrator` function uses the following argument:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParmStr</td>
<td>String</td>
<td>Microsoft Dynamics SL user id</td>
</tr>
</tbody>
</table>

Return Value:
- True – The user is a member of the Administrators group.
- False – The user is not a member of the Administrators group.

**ISAppAutomating Function**

**Description**
Returns a flag indicating whether or not the application is automated through the Solomon Object Model.

**Remarks**
Return Value:
- False – The application is not automated.
- True – The Application is automated.

**IS_AppServer Function**

**Description**
Returns a flag indicating whether or not the application is being run by Application Server.

**Remarks**
Return Value:
- Zero – application is not being run by Application Server.
- Non-Zero – Application Server is running the application.
IS_TI Function

Description
Returns a flag indicating whether or not the application is being automated by Transaction Import.

Remarks
Return Value:
- Zero – application is not being automated by Transaction Import.
- Non-Zero – Transaction Import is automating the application.

IsMultiCompany Function

Description
Returns a flag indicating whether or not MultiCompany is enabled in the current system.

Remarks
Return Value:
- Zero – MultiCompany is not enabled in the current system.
- Non-Zero – MultiCompany is enabled in the current system.

Level_SetDefaults Statement

Description
Set all controls on a particular level to their default value using either their Default Property or Default Event code.

Syntax
Call Level_SetDefaults(Form, FirstControl, LastControl, LevelNbr)

Remarks
Level_SetDefaults is functionally equivalent to the SetDefaults statement except that the former can be used to explicitly default all controls having a particular level number in their Level property.

The Level_SetDefaults statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form</td>
<td>Control</td>
<td>PNUL should be passed.</td>
</tr>
<tr>
<td>FirstControl</td>
<td>Control</td>
<td>PNUL should be passed.</td>
</tr>
<tr>
<td>LastControl</td>
<td>Control</td>
<td>PNUL should be passed.</td>
</tr>
<tr>
<td>LevelNbr</td>
<td>Control</td>
<td>Level number for which all relevant controls should be defaulted.</td>
</tr>
</tbody>
</table>

See Also
Default Event, Default Property, SetDefaults Statement

Example
The Payroll Earnings Type Maintenance screen uses the following code to force default values to be applied during the NewLevel event as opposed to after the NewLevel event. This allows that particular application to reset various property values based on actual default data values, including customized default values. Conceptually speaking, this can be useful when Field B should be disabled depending on the value of Field A. In such a case the decision as to whether or not to disable Field B cannot be made until the value of Field A is actually set.
Sub Update1_New (level%, retval%)
    If (level = LEVEL0) Then
        'Force ALL default values to be applied to EARNTYPE level
        'BEFORE Evaluate_Properties() is called.
        Call Level_SetDefaults(PNULL, PNULL, PNULL, LEVEL0)
        Call Evaluate_Properties(FLD_ALL)
        'Set retval to keep the system from setting default values
        'again for LEVEL0.
        RetVal = NoAction
    End If
End Sub
LoadForm Statement

Description
Load a subform.

Syntax
Call LoadForm(SubFormName)

Remarks
All subforms contained within a particular application must be loaded prior to the ApplInit call in the Form1_Load event using the LoadForm statement.

The call to LoadForm does not cause the subform to be displayed. However, once a subform has been successfully loaded it can be subsequently displayed using the DispForm statement.

The LoadForm statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SubFormName</td>
<td>Control</td>
<td>Form to be loaded.</td>
</tr>
</tbody>
</table>

See Also
ApplInit Statement, DispForm Statement, HideForm Statement

Example
The following code snippet illustrates how the subforms for the Payroll Employee Maintenance screen are loaded in the Load event of Form1.

Sub Form_Load ()
Dim PRSetup_Fetch As Integer

Call LoadForm(F0225001) 'Timesheet Defaults
Call LoadForm(F0225002) 'Miscellaneous Info
Call LoadForm(F0225003) 'Pay Information
Call LoadForm(F0225004) 'Employee Deductions (EarnDed grid)
Call LoadForm(F0225005) 'Employee Benefits (BenEmp grid)

Call ApplInit

Call SetAddr(LEVEL0, "bEmployee", bEmployee, nEmployee)
Call SetAddr(LEVEL1, "bWorkLoc", bWorkLoc, nWorkLoc)
Call SetAddr(LEVEL2, "bEarnType", bEarnType, nEarnType)
Call SetAddr(LEVEL3, "bPayGroup", bPayGroup, nPayGroup)
Call SetAddr(LEVEL4, "bEarnDed", bEarnDed, nEarnDed)
Call SetAddr(LEVEL5, "bBenEmp", bBenEmp, nBenEmp)

Call SetAddr(LEVEL0, "bPRDoc", bPRDoc, nPRDoc)
Call SetAddr(LEVEL0, "bPRSetup", bPRSetup, nPRSetup)
Call SetAddr(LEVEL0, "bPRTran", bPRTran, nPRTran)
Call SetAddr(LEVEL0, "bDeduction", bDeduction, nDeduction)
Call SetAddr(LEVEL0, "bBenefit", bBenefit, nBenefit)
Call SqlCursor(CSR_Employee, LEVEL0) 'Navigation Level
Call SqlCursor(CSR_WorkLoc, LEVEL1)  'Lookup Level
Call SqlCursor(CSR_EarnType, LEVEL2)  'Lookup Level
Call SqlCursor(CSR_PayGroup, LEVEL3)  'Lookup Level
Call SqlCursor(CSR_EarnDed_DBNav, LEVEL4)  'Detail Level
Call SqlCursor(CSR_BenEmp_DBNav, LEVEL5)  'Detail Level

Call SqlCursor(CSR_PRSetup, NOLEVEL)
Call SqlCursor(CSR_Deduction, NOLEVEL)
Call SqlCursor(CSR_Benefit, NOLEVEL)
Call SqlCursor(CSR_Trns_Benefit, NOLEVEL)

Call SqlCursor(CSR_PRDoc_Del_Logic, NOLEVEL)
Call SqlCursor(CSR_PRTran_Del_Logic, NOLEVEL)

'Fetch PRSetup BEFORE ScreenInit so any controls which default to
'PRSetup will default properly on the implied New at ScreenInit time.
PRSetup_Fetch = SqlFetch1(CSR_PRSetup, "PRSetup_All", bPRSetup)

Call ScreenInit

MemArray_EmpDeduction = DetailSetup(CSR_EarnDed_DBNav, F0225004..
Spread_EarnDed    PNULL, bEarnDed, bDeduction PNULL, PNULL)

MemArray_EmpBenefit = DetailSetup(CSR_BenEmp_DBNav,F0225005..
Spread_BenEmp, PNULL, bBenEmp, bBenefit, PNULL, PNULL)

End Sub
LoadStr Function

Description
Called in Visual Basic programs in which language translations are implemented using Microsoft Resource files. Loads a string resource element into a string variable.

Syntax
Stringvar = LoadStr(ResourceID)

Remarks
This function must be preceded by the `InitLocalizationSubsystem` function. See the reference for `InitLocalizationSubsystem` for information on the requirements for enabling local language user interface support in programs using the Microsoft Resource file approach.

The `LoadStr` function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stringvar</td>
<td>String</td>
<td>Any string variable</td>
</tr>
<tr>
<td>ResourceID</td>
<td>Long</td>
<td>Resource ID number (or constant representing the Resource ID number)</td>
</tr>
</tbody>
</table>

See Also
`InitLocalizationSubsystem Function, Localize Statement`

Example
The following example shows the beginning logic of the Form_Load event for an application being enabled for local language user interface support, using the resource file method.

IDS_CURRENCYSELECTION is a constant representing the Resource ID number of the string containing the screen title text “Currency Selection.” SCREENNUMBER is a constant containing the screen number string “(24.000.00).” The screen number is not moved to the resource file, because it contains system-related text that should not be translated.

```
InitLocalizationSubsystem("2400000.exe")
ScreenTitle = LoadStr(IDS_CURRENCYSELECTION)
ScreenTitle = ScreenTitle + " " + SCREENNUMBER
With Form1
    .Text = ScreenTitle
    Call Localize(.ltocuryid)
    Call Localize(.lratetype)
    Call Localize(.lratereciprocal)
    .
    .
    Call Localize(.cbuttoncancel)
EndWith
Call ApplInit
```
Localize Statement

Description
Called in Visual Basic programs in which language translations are implemented using Microsoft Resource files. Loads resource values into the language-translatable properties of a control.

Syntax
Localize(Ctrl)

Remarks
This statement must be preceded by the InitLocalizationSubsystem function. See the reference for InitLocalizationSubsystem for information on the requirements for enabling local language user interface support in programs using the Microsoft Resource file approach.

For any control, the Localize statement looks for the following potentially translatable properties:

- Caption
- List
- TabCaption()
- Help
- Heading
- Text
- List()
- ColumnText

If the property contains a Resource ID as its value, the Resource ID will be replaced by the corresponding resource string.

String values for control properties not handled by the Localize function can be loaded from resources by using the LoadStr function.

The Localize function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl</td>
<td>Control</td>
<td>Control whose property value(s) must be loaded from resources</td>
</tr>
</tbody>
</table>

See Also
InitLocalizationSubsystem Function, LoadStr Function
Example
The following example shows the beginning logic of the Form_Load event for an application being enabled for local language user interface support, using the resource file method:

```vbscript
InitLocalizationSubsystem("2400000.exe")
ScreenTitle = LoadStr(IDS_CURRENCYSELECTION)
ScreenTitle = ScreenTitle + " " + SCREENNUMBER
With Form1
    .Text = ScreenTitle
    Call Localize(.ltocuryid)
    Call Localize(.lratetype)
    Call Localize(.lratereciprocal)
    
    Call Localize(.cbuttoncancel)
EndWith

Call ApplInit

MArrayCnt Function

Description
Return the number of records in a designated memory array.

Syntax

\[
NumRecs = \text{MArrayCnt}(\text{MemHandle})
\]

Remarks
The MArrayCnt function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumRecs</td>
<td>Integer</td>
<td>Number of records in the designated memory array.</td>
</tr>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Memory array resource handle.</td>
</tr>
</tbody>
</table>

See Also
MOpen Functions, DetailSetup Functions

Example
This code snippet came from the Update event of Journal Entry Screen. Note that the MemHandle variable was the return value of the DetailSetup call issued in the Form1_Load event.

```vbscript
Sub Update1_Update (level%,insertflg%,levelsdone%,levelsleft%,RetVal%)

    If (level = Level0) Then
        'If the batch level changed then we will just
        'go ahead and presume that either Batch.JrnlType
        'and/or Batch.PerPost changed. In such a case ALL of the
        'GLTrans need to have their respective fields updated
        'with the new batch value(s) IF the
```
"batch has ALREADY been released.

If ((bBatch.Status = "U") And (MarrayCnt(MemHandle) > 0)) Then
    Call Mset(cJrnlTypeD, bBatch.JrnlType)
    Call Mset(cFiscYr, Left$(bBatch.PerPost, 4))
    Call Mset(cPerPostD, bBatch.PerPost)

    If (TestLevelChg(Level1) = NOTCHANGED) Then
        Call SetLevelChg(Level1, UPDATED)
    End If

Else
    'Don't set the JrnlType/FiscYr/PerPost of
    'transactions if the batch status
    'is H/B since they will get set automatically
    'by the 01.400 release process.

End If

End If

End Sub

**MCallChks Statement**

**Description**
Callable by an application for doing row-by-row error checking for an DSLGrid.

**Syntax**
Call **MCallChks**(MemArray, BeginControl, EndControl)

**Remarks**
The **MCallChks** function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemArray</td>
<td>Integer</td>
<td>Memory array handle</td>
</tr>
<tr>
<td>BeginControl</td>
<td>Control</td>
<td>Handle of the first control to error check. PNULL wildcard is not supported.</td>
</tr>
<tr>
<td>EndControl</td>
<td>Control</td>
<td>Handle of the last control to error check. PNULL wildcard is not supported.</td>
</tr>
<tr>
<td>Return Value</td>
<td>Integer</td>
<td>Zero if all controls were OK. Non-zero on first control found in error.</td>
</tr>
</tbody>
</table>
MClear Statement

Description
Delete all records from the designated memory array.

Syntax
Call MClear(MemHandle)

Remarks
The MClear statement can be used to clear an existing memory array of its contents. The array will stay allocated and can be subsequently re-used.
The MClear statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Memory array resource handle.</td>
</tr>
</tbody>
</table>

See Also
MOpen Functions, DetailSetup Functions

MClose Statement

Description
Close an existing memory array.

Syntax
Call MClose(MemHandle)

Remarks
MClose can be used to close a memory array previously opened using one of the MOpen functions. Memory arrays created automatically by the DetailSetup functions should not be closed by the application.

The MClose statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Memory array resource handle.</td>
</tr>
</tbody>
</table>

See Also
MOpen Functions
Example
This example illustrates how to close a memory array no longer needed by the application.

Dim Mem_Account As Integer
Dim CSR_Account As Integer
Dim SqlStr As String
Dim Account_Fetch As Integer

'Open memory array to hold Chart of Accounts
Mem_Account = MOpen( TRUE, bAccount, Len(bAccount), PNULL, 0, PNULL, 0, PNULL, 0)

'Allocate cursor
Call SqlCursor( CSR_Account, NOLEVEL)

'Initialize cursor with a SQL statement and immediately fetch first 'record
SqlStr = "Select * from Account order by Acct"
Account_Fetch = SqlFetch1(CSR_Account, SqlStr, bAccount)

'Read through all subsequent Account records, inserting each 'one into the memory array.
While( Account_Fetch = 0)

    'Insert current Account record into the memory array
    Call MInsert( Mem_Account)

    'Fetch the next Account record
    Account_Fetch = SFetch1(CSR_Account, bAccount)
Wend

'Close the memory array
Call MClose( Mem_Account)

MDelete Function

Description
Delete the current record from the designated memory array.

Syntax
RecFetch = MDelete(MemHandle, RecMaintFlg)

Remarks
The current record in a memory array can be deleted by using the MDelete function. After the record is deleted, the system will automatically navigate to the next memory array record since there must always be a current record in memory arrays, assuming of course that one or more records exist. Consequently, the return value and corresponding record status apply to the next record after the deleted record.
When this call is used on a memory array associated with a **DSLGrid** control (for example, memory arrays opened automatically via the **DetailSetup** function), an **MDisplay** call will be necessitated to properly synchronize the **DSLGrid** appearance with the memory array.

The **MDelete** function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecFetch</td>
<td>Integer</td>
<td>0 if the next record is successfully fetched. NOTFOUND is returned if no subsequent records exist in the specified memory array (for example, when the last record itself is deleted). This does not mean that no additional records exist in the memory array. Rather it simply means that no records exist after the record just deleted.</td>
</tr>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Memory array resource handle.</td>
</tr>
<tr>
<td>RecMaintFlg</td>
<td>Integer</td>
<td>Status of the memory array record (assuming it was successfully fetched). Solomon.VBTools.vb contains the following symbolic constants defining possible memory array record status values: INSERTED, UPDATED and NOTCHANGED</td>
</tr>
</tbody>
</table>

**See Also**

**DetailSetup Functions, MDisplay Statement, MInsert Statement, MUpdate Statement**

### **MDisplay Statement**

**Description**

Display the current contents of the designated memory array in its corresponding **DSLGrid** control.

**Syntax**

Call **MDisplay**(*MemHandle*)

**Remarks**

Each **DSLGrid** control is associated with an underlying memory array that is opened automatically by the system during the **DetailSetup** call. Anytime data within this memory array is modified directly by the application, as opposed to by the user, the **DSLGrid** control must be subsequently redisplayed.

When **MDisplay** is called, the current memory array record will be displayed at the top of the **DSLGrid** control. Thus, for example, if the application wants the first memory array record to display at the top of the **DSLGrid** control it should initially call **MFirst** to move to the first record and then call **MDisplay**. The **MDisplay** statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Memory array resource handle. This memory array must be associated with a <strong>DSLGrid</strong> control.</td>
</tr>
</tbody>
</table>

**See Also**

**DetailSetup Functions**
Example
The following example illustrates how the Release Payroll Batches process redisplays the **DSLGrid** control associated with the MemHandle memory array after processing has completed.

Sub cBegProcessing_Click ()

    Dim RecFound   As Integer
    Dim MemMaintFlg    As Integer
    Dim Nbr_Of_Batches_Processed As Integer

    Call Status(StartProcess, False, "", 0)

    'Explicitly initialize processing counter to zero BEFORE calling
    'ProcValidBatch() for the FIRST time.
    Nbr_Of_Batches_Processed = 0

    RecFound = MFirst(MemHandle, MemMaintFlg)
    While (RecFound = 0)

        If (bCurrBatchSelected = True) Then
            'Process the selected batch
            Call ProcValidBatch(Nbr_Of_Batches_Processed)

            'Delete current and get next memory array batch record
            RecFound = MDelete(MemHandle, MemMaintFlg)
        Else
            'Current batch is not selected so get the next batch
            'from the memory array.
            RecFound = MNext(MemHandle, MemMaintFlg)
        End If

    End While

    Call Status(EndProcess, False, "", 0)

    'Redisplay the grid with the modified contents of the memory array.
    RecFound = MFirst(MemHandle, MemMaintFlg)
    Call MDisplay(MemHandle)

End Sub
Mess Statement

Description
Displays a message from the Microsoft Dynamics SL message file and waits for the user to choose a button.

Syntax
Call Mess(MsgNumber)

Remarks
When Microsoft Dynamics SL is installed, an ASCII text file called Messages.csv is copied to the \Microsoft Dynamics SL directory. This file contains all messages relating to the Microsoft Dynamics SL product, including all independently developed applications created with Microsoft Dynamics SL SDK. Each message has, among other things, a message number. A particular message can be displayed to the screen by merely passing its associated message number to the Mess statement.

The Mess statement should be used if the actual text of the message contains replaceable parameters.

The MessResponse function can be used to determine which button was chosen by the user to close the message box.

The standard Visual Basic MsgBox statement should not be used in applications developed with the Microsoft Dynamics SL SDK in order to avoid conflicts with other Microsoft Dynamics SL utilities such as Cut/Copy/Paste and Transaction Import. These utilities have built-in sophistication to respond to messages from the underlying application during the particular operation being performed such as paste or import. However, this automated functionality does not apply to messages displayed using the standard Visual Basic MsgBox statement. The MessBox statement has been provided to facilitate similar functionality to the standard Visual Basic MsgBox statement with the exception that MessBox does not conflict with other Microsoft Dynamics SL utilities.

The Mess statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MsgNumber</td>
<td>Integer</td>
<td>Number of the message from the Microsoft Dynamics SL message file that is to be displayed.</td>
</tr>
</tbody>
</table>

Each record (message) contained within the MESSAGES.CSV file contains the following fields separated by a comma:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Number</td>
<td>Message number which is required by most of the message API’s such as Mess.</td>
</tr>
<tr>
<td>Category</td>
<td>0 for all Microsoft Dynamics SL messages. All independently developed applications must also use 0.</td>
</tr>
<tr>
<td>Language</td>
<td>0 for English.</td>
</tr>
<tr>
<td>Type</td>
<td>For future use. Currently set the field to 0.</td>
</tr>
<tr>
<td>Box Type</td>
<td>See detailed explanation of Box Type values below.</td>
</tr>
<tr>
<td>Record Type</td>
<td>S for messages created and maintained by Microsoft. Independent developers should not use S for their new messages since they could be deleted or modified by Microsoft.</td>
</tr>
<tr>
<td>Unattended Default Button</td>
<td>This value is used by Transaction Import to respond to messages displayed by the underlying application during the import operation.</td>
</tr>
<tr>
<td>Message Text</td>
<td>Actual text of the message. Replaceable parameters appear as “%s”. For example: My first name is %s and my last name is %s.</td>
</tr>
</tbody>
</table>
The Box Type field within the MESSAGES.CSV file, for standard Microsoft Dynamics SL messages, can have the following values:

<table>
<thead>
<tr>
<th>Box Type Value</th>
<th>Icon</th>
<th>Buttons</th>
<th>Default Button</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>16</td>
<td>Stop</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>48</td>
<td>Exclamation</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>64</td>
<td>Information</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>36</td>
<td>Question</td>
<td>Yes / No</td>
<td>Yes</td>
</tr>
<tr>
<td>292</td>
<td>Question</td>
<td>Yes / No</td>
<td>No</td>
</tr>
<tr>
<td>33</td>
<td>Question</td>
<td>OK / Cancel</td>
<td>OK</td>
</tr>
</tbody>
</table>

**See Also**

MessBox Statement, Messf Statement, MessResponse Function

**Example**

The Payroll Employee Maintenance screen allows the user to enter the number of personal exemptions claimed by any particular employee on the Miscellaneous Information sub screen. Anytime this value is changed the user is prompted, via a message, as to whether or not the new total number of personal exemptions is to be utilized by each individual deduction for calculation purposes.

Message number 739 is the actual message displayed and its associated text in the Microsoft Dynamics SL message file reads as follows: “Do you want to update the employee’s deductions with the new exemption value?” This particular message will also display two buttons in the message box, a Yes button and a No button. The MessResponse function is subsequently called to determine which of these two buttons the user actually selected to close the message box.

```vba
Sub cDfltPersExmpt_Chk (chkstrg As String, retval As Integer)
    Dim MemArray_NbrRecs As Integer

    MemArray_NbrRecs = MArrayCnt(MemArray_EmpDeduction)

    'If the memory array has any records in it then prompt the user
    'whether or not he/she wants to update the number of PERSONAL
    'EXEMPTIONS on ALL existing employee deductions.
    If (MemArray_NbrRecs > 0) Then
        Call Mess( 739)

        If (MessResponse() = IDYES) Then
            Call MSet(F0225004.cNbrPersExmpt, chkstrg)
            Call MDisplay(MemArray_EmpDeduction)
        End If
    End If
End Sub
```
MessBox Statement

Description
Displays a message and waits for the user to choose a button.

Syntax
Call MessBox(Msg, Type, Title)

Remarks
The standard Visual Basic MsgBox statement should not be used in applications developed with the Microsoft Dynamics SL SDK in order to avoid conflicts with other Microsoft Dynamics SL utilities such as Cut/Copy/Paste and Transaction Import. These utilities have built-in sophistication to respond to messages from the underlying application during the particular operation being performed such as paste or import. However, this automated functionality does not apply to messages displayed using the standard Visual Basic MsgBox statement. The MessBox statement has been provided to facilitate similar functionality to the standard Visual Basic MsgBox statement with the exception that MessBox does not conflict with other Microsoft Dynamics SL utilities.

The MessResponse function can be used to determine which button was chosen by the user to close the message box.

The MessBox statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Msg</td>
<td>String</td>
<td>Message text to be displayed.</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
<td>Numeric value controlling the icon style, buttons to be displayed as well as which button is the default button.</td>
</tr>
<tr>
<td>Title</td>
<td>String</td>
<td>Text to display in the title bar of the message dialog box.</td>
</tr>
</tbody>
</table>

The MsgBox type is a numeric value that describes:

a) Which buttons appear in the window when the message is displayed on the screen.
b) What Icon appears in the window.
c) Which button is the default button on the screen.

Note: A, b, and c have associated numeric values. When you combine them, they form the Message Box Type value. The possible values for a, b, and c are outlined here.

<table>
<thead>
<tr>
<th>Table 1 Buttons (a)</th>
<th>Buttons Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OK</td>
</tr>
<tr>
<td>1</td>
<td>OK and Cancel</td>
</tr>
<tr>
<td>2</td>
<td>Abort, Retry, and Ignore</td>
</tr>
<tr>
<td>3</td>
<td>Yes, No, and Cancel</td>
</tr>
<tr>
<td>4</td>
<td>Yes and No</td>
</tr>
<tr>
<td>5</td>
<td>Retry and Cancel</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2 Icon Value (b)</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>Display the Stop Sign Icon</td>
</tr>
<tr>
<td>32</td>
<td>Display the Question Mark Icon</td>
</tr>
<tr>
<td>48</td>
<td>Display the Exclamation Point Icon</td>
</tr>
<tr>
<td>64</td>
<td>Display the Information Icon</td>
</tr>
</tbody>
</table>
Table 3

<table>
<thead>
<tr>
<th>Default Button Value (c)</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>First button is default</td>
</tr>
<tr>
<td>256</td>
<td>Second button is the default</td>
</tr>
<tr>
<td>512</td>
<td>Third button is the default</td>
</tr>
</tbody>
</table>

See Also

Mess Statement, Messf Statement, MessResponse Function

Example

Call MessBox("Project is over budget. Do you wish to continue?",308,"Over Budget")

Will produce a message that resembles the following:

Referring to our tables, the Yes/No options are worth 4. The exclamation point icon is worth 48. Having the second button (No) be the default button is worth 256. Add them all up and you have 308 which is used as the second parameter.
**Messf Statement**

**Description**
Formats a message from the Microsoft Dynamics SL message file with replaceable parameters and then displays it and waits for the user to choose a button.

**Syntax**
Call `Messf(MsgNumber, Parm1Str, Parm2Str, Parm3Str, Parm4Str, Parm5Str, Parm6Str)`

**Remarks**
When Microsoft Dynamics SL is installed, an ASCII text file called `Messages.csv` is copied to the Microsoft Dynamics SL directory. This file contains all messages relating to the Microsoft Dynamics SL product, including all independently developed applications created with the Microsoft Dynamics SL SDK. Each message has, among other things, a message number. The message can also contain up to six replaceable parameters by placing a %s at the appropriate point(s) within the actual message text. A particular message can be subsequently displayed to the screen by merely passing its associated message number to the `Messf` statement along with data values for each replaceable parameter.

The `Mess` statement should be used if the actual text of the message does not contain replaceable parameters.

The `MessResponse` function can be used to determine which button was chosen by the user to close the message box.

The standard Visual Basic `MsgBox` statement should not be used in applications developed with Microsoft Dynamics SL SDK in order to avoid conflicts with other Microsoft Dynamics SL utilities such as Cut/Copy/Paste and Transaction Import. These utilities have built-in sophistication to respond to messages from the underlying application during the particular operation being performed such as paste or import. However, this automated functionality does not apply to messages displayed using the standard Visual Basic `MsgBox` statement. The `MessBox` statement has been provided to facilitate similar functionality to the standard Visual Basic `MsgBox` statement with the exception that `MessBox` does not conflict with other Microsoft Dynamics SL utilities.

The `Messf` statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MsgNumber</code></td>
<td>Integer</td>
<td>Number of the message from the Microsoft Dynamics SL message file that is to be displayed.</td>
</tr>
<tr>
<td><code>Parm1Str</code></td>
<td>String</td>
<td>Data value for the first replaceable parameter.</td>
</tr>
<tr>
<td><code>Parm2Str</code></td>
<td>String</td>
<td>Data value for the second replaceable parameter. Blank if the message text contains only one replaceable parameter.</td>
</tr>
<tr>
<td><code>Parm3Str</code></td>
<td>String</td>
<td>Data value for the third replaceable parameter. Blank if the message text contains less than three replaceable parameters.</td>
</tr>
<tr>
<td><code>Parm4Str</code></td>
<td>String</td>
<td>Data value for the fourth replaceable parameter. Blank if the message text contains less than four replaceable parameters.</td>
</tr>
<tr>
<td><code>Parm5Str</code></td>
<td>String</td>
<td>Data value for the fifth replaceable parameter. Blank if the message text contains less than five replaceable parameters.</td>
</tr>
<tr>
<td><code>Parm6Str</code></td>
<td>String</td>
<td>Data value for the sixth replaceable parameter. Blank if the message text contains less than six replaceable parameters.</td>
</tr>
</tbody>
</table>
Each record (message) contained within the MESSAGES.CSV file contains the following fields separated by a comma:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Number</td>
<td>Message number which is required by most of the message API's such as Messf.</td>
</tr>
<tr>
<td>Category</td>
<td>0 for all Microsoft Dynamics SL messages. All independently developed applications must also use 0.</td>
</tr>
<tr>
<td>Language</td>
<td>0 for English.</td>
</tr>
<tr>
<td>Type</td>
<td>For future use. Currently set the field to 0.</td>
</tr>
<tr>
<td>Box Type</td>
<td>See detailed explanation of Box Type values below.</td>
</tr>
<tr>
<td>Record Type</td>
<td>S for messages created and maintained by Microsoft. Independent developers should not use S for their new messages since they could be deleted or modified by Microsoft.</td>
</tr>
<tr>
<td>Unattended Default Button</td>
<td>This value is used by Transaction Import to respond to messages displayed by the underlying application during the import operation.</td>
</tr>
<tr>
<td>Message Text</td>
<td>Actual text of the message. Replaceable parameters appear as “%s”. For example: My first name is %s and my last name is %s.</td>
</tr>
</tbody>
</table>

The Box Type field within the MESSAGES.CSV file, for standard Microsoft Dynamics SL messages, can have the following values. Other valid Box Type values can be used in custom messages. See the MessBox Statement documentation for a complete description:

<table>
<thead>
<tr>
<th>Box Type Value</th>
<th>Icon</th>
<th>Buttons</th>
<th>Default Button</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>16</td>
<td>Stop</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>48</td>
<td>Exclamation</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>64</td>
<td>Information</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>36</td>
<td>Question</td>
<td>Yes / No</td>
<td>Yes</td>
</tr>
<tr>
<td>292</td>
<td>Question</td>
<td>Yes / No</td>
<td>No</td>
</tr>
<tr>
<td>33</td>
<td>Question</td>
<td>OK / Cancel</td>
<td>OK</td>
</tr>
</tbody>
</table>

See Also
FtoA Function, Mess Statement, MessBox Statement, MessResponse Function

Example
The Payroll Manual Check screen uses the following code to warn the user of the fact that a Batch is out of balance.

Message number 818 is the actual message displayed and its associated text in the Microsoft Dynamics SL message file reads as follows: “Batch is out of balance by %s. Do you want to edit?”. This particular message will also display two buttons in the message box, a Yes button and a No button. The MessResponse function is subsequently called to determine which of these two buttons the user actually selected to close the message box.

'Make sure that the batch itself is in balance with the documents.

    Batch_Out_Of_Bal_Amt = FPSub(bBatch.CtrlTot, bBatch.DrTot, MONEY)

    If (Batch_Out_Of_Bal_Amt <> 0#) Then

        Call Messf(818, FtoA(Batch_Out_Of_Bal_Amt, MONEY), "", "", "", "","")
If (MessResponse() = IDYES) Then
    'User decided to edit the batch - so abort the Finish
    retval = ErrNoMess
    'Set focus on the Batch Control Total field
    Call ApplSetFocus(cCtrlTot)
End If

MessGetText Function

Description
Returns the message text associated with a particular message number.

Syntax
TextStr = MessGetText(MsgNumber)

Remarks
The MessGetText function uses the following argument:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TextStr</td>
<td>String</td>
<td>Text of the designated MsgNumber.</td>
</tr>
<tr>
<td>MsgNumber</td>
<td>Integer</td>
<td>Number of the message from the Microsoft Dynamics SL message file that is to be displayed.</td>
</tr>
</tbody>
</table>

See Also
Mess Statement, Messf Statement
MessResponse Function

Description
Returns the button chosen by the user to close the last message box displayed using the Mess, Messf or MessBox statements.

Syntax
```
ButtonId = MessResponse()
```

Remarks
The MessResponse function returns one of the following symbolic constants declared in Solomon.VBTools.vb:

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDOK</td>
<td>OK button selected.</td>
</tr>
<tr>
<td>IDYES</td>
<td>Yes button selected.</td>
</tr>
<tr>
<td>IDNO</td>
<td>No button selected.</td>
</tr>
<tr>
<td>IDCANCEL</td>
<td>Cancel button selected.</td>
</tr>
<tr>
<td>IDABORT</td>
<td>Abort button selected.</td>
</tr>
<tr>
<td>IDRETRY</td>
<td>Retry button selected.</td>
</tr>
<tr>
<td>IDIGNORE</td>
<td>Ignore button selected.</td>
</tr>
</tbody>
</table>

See Also
Mess Statement, Messf Statement

Example
The Payroll Manual Check screen uses the following code to warn the user of the fact that a Batch is out of balance.

Message number 818 is the actual message displayed and its associated text in the Microsoft Dynamics SL message file reads as follows: “Batch is out of balance by %s. Do you want to edit?”. This particular message will also display two buttons in the message box, a Yes button and a No button. The MessResponse function is subsequently called to determine which of these two buttons the user actually selected to close the message box.

```
'Make sure that the batch itself is in balance with the documents.
Batch_Out_Of_Bal_Amt = FPSub(bBatch.CtrlTot, bBatch.DrTot, MONEY)
If (Batch_Out_Of_Bal_Amt <> 0#) Then
    Call Mess( 818, FtoA(Batch_Out_Of_Bal_Amt, MONEY), "", ",", ",", ",", ",")
    If (MessResponse() = IDYES) Then
        'User decided to edit the batch - so abort the
        Finish
        retval = ErrNoMess
    'Set focus on the Batch Control Total field
    Call ApplSetFocus(cCtrlTot)
End If
End If
```
MExtend Function

Description
Extends an existing memory array with additional data buffers; returns zero if successful, non-zero for a failure.

Syntax
Function MExtend(ByVal MemHandle As Short, ByRef bTable As SolomonDataObject) As Short

Remarks
This function is most useful when a memory array requires more than eight data buffers. MOpen8 allows you to declare an array with a maximum of eight data buffers. If more are required, this function will allow you to add up to eight additional buffers to the array.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemHandle</td>
<td>Short</td>
<td>Unique handle to the memory array</td>
</tr>
<tr>
<td>bTable</td>
<td>SolomonDataObject</td>
<td>Data buffer to add to the array</td>
</tr>
</tbody>
</table>

See Also
MOpen

Example
This example illustrates how to add additional data buffers to a memory array.

Dim MemArrayHandle As Short
Dim retval As Short

' open memory array with the following buffers
MemArrayHandle = MOpen8(True, _
          bAccount, _
          bGLTran, _
          bBatch, _
          bg1setup, _
          bFLEXDEF, _
          bCashAcct, _
          bPCSetup, _
          bPJPENT)

'Add additional buffers to the memory array
retval = MExtend(MemArrayHandle, bPJPROJ)
retval = MExtend(MemArrayHandle, bPJTEXT)
MFirst Function

Description
Move to the first record in a designated memory array.

Syntax
RecFetch = MFirst(MemHandle, RecMaintFlg)

Remarks
MFirst moves to the first record of a specified memory array and copies the contents of the array record into the data structure(s) previously specified in either the MOpen or DetailSetup function call used to originally open the relevant memory array.

When this call is used on a memory array associated with an DSLGrid control (that is, memory arrays opened automatically via the DetailSetup function), an MDisplay call will be necessitated to properly synchronize the DSLGrid appearance with the memory array.

The MFirst function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecFetch</td>
<td>Integer</td>
<td>0 if a record is successfully fetched. NOTFOUND is returned if no records exist in the specified memory array.</td>
</tr>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Memory array resource handle.</td>
</tr>
<tr>
<td>RecMaintFlg</td>
<td>Integer</td>
<td>Status of the memory array record (assuming it was successfully fetched).</td>
</tr>
</tbody>
</table>

See Also
MLast Function, MNext Function, MPrev Function

MGetDelHandle Function

Description
Returns the resource handle of the memory array used to temporarily hold detail lines deleted from the designated DSLGrid control.

Syntax
DelMemHandle = MGetDelHandle(DSLGridCtrl)

Remarks
Each DSLGrid control is associated with two underlying memory arrays that are opened automatically by the system during the DetailSetup call. The primary memory array is used to hold the records which are actually visible in the grid. The resource handle to this primary memory array is actually returned by the DetailSetup functions. However, another array is also created to temporarily hold records which the user has deleted until a SAVE operation is performed and the deletions are actually committed to the database. The resource handle to this memory array can be retrieved using the MGetDelHandle function.

Once the resource handle for the deleted record memory array has been retrieved, it can be used by the application to loop through records deleted by the user using calls such as MFirst and MNext.
The **MGetDelHandle** function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DelMemHandle</td>
<td>Integer</td>
<td>Resource handle for the memory array holding records deleted from the designated <strong>DSLGrid</strong>.</td>
</tr>
<tr>
<td>DSLGridCtrl</td>
<td>Control</td>
<td><strong>DSLGrid</strong> control.</td>
</tr>
</tbody>
</table>

**See Also**

*DetailSetup Functions*

### MGetLineStatus Function

**Description**

Returns the line status of the current record in the designated memory array.

**Syntax**

```vba
RecMaintFlg = MGetLineStatus(MemHandle)
```

**Remarks**

The **MGetLineStatus** function allows the application to retrieve the status of the current memory array record at any time.

The **MGetLineStatus** function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecMaintFlg</td>
<td>Integer</td>
<td>Status of the current memory array record. Solomon.VBTools.vb contains the following symbolic constants defining possible memory array record status values: INSERTED, UPDATED and NOTCHANGED</td>
</tr>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Memory array resource handle.</td>
</tr>
</tbody>
</table>

**See Also**

*MSetLineStatus Function*

### MGetRowNum Function

**Description**

Returns the row / record number of the current record in the designated memory array.

**Syntax**

```vba
CurrRecNbr = MGetRowNum(MemHandle)
```

**Remarks**

The **MGetRowNum** function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurrRecNbr</td>
<td>Integer</td>
<td>Row / record number of the current record.</td>
</tr>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Memory array resource handle.</td>
</tr>
</tbody>
</table>

**See Also**

*MArrayCnt Function, MSetRow Statement*
Minsert Statement

Description
Insert a new record into a designated memory array.

Syntax
Call Minsert(MemHandle)

Remarks
Minsert is used to add a new record to a memory array. This is accomplished by copying the contents of all data structures previously associated with the designated memory array into a new memory array record. Data structures are associated with a memory array in either the MOpen or DetailSetup function call used to originally open the relevant memory array. The new memory array record will have a line status of INSERTED.

When this call is used on a memory array associated with a DSLGrid control (for example, memory arrays opened automatically via the DetailSetup function), an MDisplay call will be necessitated to properly synchronize the DSLGrid appearance with the memory array.

The Minsert statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Memory array resource handle.</td>
</tr>
</tbody>
</table>

See Also
DetailSetup Functions, MDelete Function, MOpen Functions, MSetLineStatus Function, MUpdate Statement

Example
This example illustrates how records can be inserted into a memory array under program control.

```vba
Dim Mem_Account As Integer
Dim CSR_Account As Integer
Dim SqlStr As String
Dim Account_Fetch As Integer

'Open memory array to hold Chart of Accounts
Mem_Account = MOpen( TRUE, bAccount, Len(bAccount), PNULL, 0, PNULL, 0, PNULL, 0)

'Allocate cursor
Call SqlCursor( CSR_Account, NOLEVEL)

'Initialize cursor with a SQL statement and immediately fetch first record
SqlStr = "Select * from Account order by Acct"
Account_Fetch = SqlFetch1(CSR_Account, SqlStr, bAccount)

'Read through all subsequent Account records, inserting each one into the memory array.
While( Account_Fetch = 0)
    'Insert current Account record into the memory array
    Call Minsert( Mem_Account)
    'Fetch the next Account record
```
Account_Fetch = SFetch1(CSR_Account, bAccount)

MKey Statement

Description
Define a key field for a previously opened memory array.

Syntax
Call MKey(MemHandle, KeySegmentNbr, TableDotFieldName, Ascending)

Remarks
Occasionally a program will need the ability to easily locate a particular record within a memory array based on one or more key field values. The MKeyFind function can be used to accomplish this goal assuming the sort order for the memory array has been previously defined. Memory arrays associated with an DSLGrid control automatically have their sort order initialized by the DetailSetup function based on the key field control(s) contained within the grid (for example, notated by a “,k” in the levels property of the controls). All other memory arrays must have their sort order explicitly defined via one of several different methods. Each of the methods to define a key field, such as MKey, MKeyFld, MKeyHctl and MKeyOffset, vary primarily in the way they acquire detailed information on a key field such as datatype, size and byte offset within a user-defined datatype.

The MKey statement is the simplest and most common of all these methods to define a memory array key field. MKey is so simple because the system will automatically determine the requisite key field information for the TableDotFieldName based both on the SetAddr call for the relevant table and its corresponding data dictionary information in the database. The two restrictions of the MKey method are that it can only be used for fields whose table exists in the database (as opposed to a memory variable existing only within Visual Basic code, for example) and a SetAddr call must have already been issued for the relevant table.

Multi-segment keys can be defined by successive calls to MKey with different KeySegmentNbr argument values.

The MKey statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Unique handle to a previously opened memory array.</td>
</tr>
<tr>
<td>KeySegmentNbr</td>
<td>Control</td>
<td>Memory array key segment whose key field is being defined. The first key segment number is always zero. Multi-segment keys must have contiguous key segment values such as 0 and 1 as opposed to 0 and 3. The maximum allowable number of key segments is five.</td>
</tr>
<tr>
<td>TableDotFieldName</td>
<td>String</td>
<td>Name of the designated key field in a Table.FieldName format such as “Account.Acct”.</td>
</tr>
<tr>
<td>Ascending</td>
<td>Integer</td>
<td>True if the key segment should be sorted ascending. False to implement a descending sort sequence for the key segment currently being defined.</td>
</tr>
</tbody>
</table>

See Also
MKeyFind Function, MKeyFld Statement, MKeyHctl Statement, MKeyOffset Statement, MOpen Functions, MSort Statement
Example
This example illustrates how to open a memory array and define multiple key fields.

```vba
Dim Mem_ValEarnDed As Integer
Call SetAddr(NOLEVEL, "bValEarnDed", bValEarnDed, nValEarnDed)
Mem_ValEarnDed = MOpen(True, bValEarnDed, Len(bValEarnDed), PNULL, 0,
NULL, 0, PNULL, 0, 0)

'Set up use of MKeyFind() for memory array
Call MKey(Mem_ValEarnDed, 0, "bValEarnDed.EarnTypeId", True)
Call MKey(Mem_ValEarnDed, 1, "bValEarnDed.DedId", True)
```

MKeyFind Function

Description
Find a specific record in a sorted memory array based on designated key field values.

Syntax

```vba
RecFetch = MKeyFind(MemHandle, KeySeg1Val, KeySeg2Val, KeySeg3Val, KeySeg4Val, KeySeg5Val)
```

Remarks
Occasionally a program will need the ability to easily locate a particular record within a memory array based on one or more key field values. The **MKeyFind** function can be used to accomplish this goal assuming the sort order for the memory array has been previously defined. Memory arrays associated with a **DSLGrid** control automatically have their sort order initialized by the **DetailSetup** function based on the key field control(s) contained within the grid (for example, notated by a ",k" in the levels property of the controls). All other memory arrays must have their sort order explicitly defined via one of several different methods. Each of the methods to define a key field, such as **MKey**, **MKeyFld**, **MKeyHctl** and **MKeyOffset**, vary primarily in the way they acquire detailed information on a key field such as datatype, size and byte offset within a user-defined datatype.

If a record whose key fields exactly match the **KeySeg?Val** arguments does not exist, the system positions to the closest match. It will however still return a NOTFOUND to the application.

The **MKeyFind** function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RecFetch</strong></td>
<td>Integer</td>
<td>0 if a record is successfully fetched. NOTFOUND is returned if an exact match cannot be located.</td>
</tr>
<tr>
<td><strong>MemHandle</strong></td>
<td>Integer</td>
<td>Memory array resource handle.</td>
</tr>
<tr>
<td><strong>KeySeg1Val</strong></td>
<td>Integer, Double or String</td>
<td>Desired value for the first key segment.</td>
</tr>
<tr>
<td><strong>KeySeg2Val</strong></td>
<td>Integer, Double or String</td>
<td>Desired value for the second key segment. PNULL if the memory array only has one key segment.</td>
</tr>
<tr>
<td><strong>KeySeg3Val</strong></td>
<td>Integer, Double or String</td>
<td>Desired value for the third key segment. PNULL if the memory array has less than three key segments.</td>
</tr>
<tr>
<td><strong>KeySeg4Val</strong></td>
<td>Integer, Double or String</td>
<td>Desired value for the fourth key segment. PNULL if the memory array has less than four key segments.</td>
</tr>
<tr>
<td><strong>KeySeg5Val</strong></td>
<td>Integer, Double or String</td>
<td>Desired value for the fifth key segment. PNULL if the memory array has less than five key segments.</td>
</tr>
</tbody>
</table>

See Also
**MKey**, **MKeyFId**, **MKeyHctl**, **MKeyOffset**, **MOpen**, **MSetRow**, **MSort**
Example
This example illustrates how to open a memory array and load the entire chart of accounts into the newly-created array and then find a specific account record.

Dim Mem_Account As Integer
Dim CSR_Account As Integer
Dim SqlStr As String
Dim Account_Fetch As Integer

'Open memory array to hold Chart of Accounts
Mem_Account = MOpen( TRUE, bAccount, Len(bAccount), PNULL, 0, J
   PNULL, 0, PNULL, 0)

'Set up use of MKeyFind() for memory array
Call MKey(Mem_Account, 0, "bAccount.Acct", True)

'Allocate cursor
Call SqlCursor( CSR_Account, NOLEVEL)

'Initialize cursor with a SQL statement and immediately fetch first record
SqlStr = "Select * from Account order by Acct"
Account_Fetch = SqlFetch1(CSR_Account, SqlStr, bAccount)

'Read through all subsequent Account records, inserting each one into the memory array.
While( Account_Fetch = 0)
   'Insert current Account record into the memory array
   Call MInsert( Mem_Account)
   'Fetch the next Account record
   Account_Fetch = SFetch1(CSR_Account, bAccount)
Wend

'Find the memory array record for a specific account
Account_Fetch = MKeyFind( Mem_Account, "2020", PNULL, PNULL, J
   PNULL, PNULL)
MKeyFld Statement

Description
Define a key field for a previously opened memory array.

Syntax
Call MKeyFld(MemHandle, KeySegmentNbr, TableDotFieldName, bTable, Ascending)

Remarks
Occasionally a program will need the ability to easily locate a particular record within a memory array based on one or more key field values. The MKeyFind function can be used to accomplish this goal assuming the sort order for the memory array has been previously defined. Memory arrays associated with an DSLGrid control automatically have their sort order initialized by the DetailSetup function based on the key field control(s) contained within the grid (for example, notated by a ”,k” in the levels property of the controls). All other memory arrays must have their sort order explicitly defined via one of several different methods. Each of the methods to define a key field, such as MKey, MKeyFld, MKeyHctl and MKeyOffset, vary primarily in the way they acquire detailed information on a key field such as datatype, size and byte offset within a user-defined datatype.

The MKeyFld method is similar to the MKey method except that it does not require a SetAddr call for the relevant table.

Multi-segment keys can be defined by successive calls to MKeyFld with different KeySegmentNbr argument values.

The MKeyFld statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Unique handle to a previously opened memory array.</td>
</tr>
<tr>
<td>KeySegmentNbr</td>
<td>Integer</td>
<td>Memory array key segment whose key field is being defined. The first key segment number is always zero. Multi-segment keys must have contiguous key segment values such as 0 and 1 as opposed to 0 and 3. The maximum allowable number of key segments is five.</td>
</tr>
<tr>
<td>TableDotFieldName</td>
<td>String</td>
<td>Name of the designated key field in a Table.FieldNName format such as “Account.Acct”.</td>
</tr>
<tr>
<td>bTable</td>
<td>User-defined datatype</td>
<td>Memory array table structure containing the designated key field. This table structure must also have been previously passed to the MOpen call.</td>
</tr>
<tr>
<td>Ascending</td>
<td>Integer</td>
<td>True if the key segment should be sorted ascending. False to implement a descending sort sequence for the key segment currently being defined.</td>
</tr>
</tbody>
</table>

See Also
MKey Statement, MKeyFind Function, MKeyHctl Statement, MKeyOffset Statement, MOpen Functions, MSort Statement
Example

This example illustrates how to open a memory array and define multiple key fields.

```vba
Dim Mem_ValEarnDed As Integer
Mem_ValEarnDed = MOpen(True, bValEarnDed, Len(bValEarnDed), PNULL, 0, J
   PNULL, 0, PNULL, 0)
'Set up use of MKeyFind() for memory array
Call MKeyFld(Mem_ValEarnDed, 0, "bValEarnDed.EarnTypeId", .J
   bValEarnDed, True)
Call MKeyFld(Mem_ValEarnDed, 1, "bValEarnDed.DedId", .J
   bValEarnDed, True)
```

**MKeyHctl Statement**

**Description**

Define a key field for a previously opened memory array.

**Syntax**

```
Call MKeyHctl(MemHandle, KeySegmentNbr, KeyFieldControl, Ascending)
```

**Remarks**

Occasionally a program will need the ability to easily locate a particular record within a memory array based on one or more key field values. The MKeyFind function can be used to accomplish this goal assuming the sort order for the memory array has been previously defined. Memory arrays associated with a DSLGrid control automatically have their sort order initialized by the DetailSetup function based on the key field control(s) contained within the grid (for example, notated by a ",k" in the levels property of the controls). All other memory arrays must have their sort order explicitly defined via one of several different methods. Each of the methods to define a key field, such as MKey, MKeyFld, MKeyHctl and MKeyOffset, vary primarily in the way they acquire detailed information on a key field such as datatype, size and byte offset within a user-defined datatype.

The MKeyHctl method acquires information on the designated key field from a control having the key field itself in its FieldName property.

Multi-segment keys can be defined by successive calls to MKeyHctl with different KeySegmentNbr and KeyFieldControl argument values.

The MKeyHctl statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Unique handle to a previously opened memory array.</td>
</tr>
<tr>
<td>KeySegmentNbr</td>
<td>Integer</td>
<td>Memory array key segment whose key field is being defined. The first key segment number is always zero. Multi-segment keys must have contiguous key segment values such as 0 and 1 as opposed to 0 and 3. The maximum allowable number of key segments is five.</td>
</tr>
<tr>
<td>KeyFieldControl</td>
<td>Control</td>
<td>Name of the control whose FieldName property refers to the designated key field in a Table.FieldName format such as &quot;Account.Acct&quot;.</td>
</tr>
<tr>
<td>Ascending</td>
<td>Integer</td>
<td>True if the key segment should be sorted ascending. False to implement a descending sort sequence for the key segment currently being defined.</td>
</tr>
</tbody>
</table>

**See Also**

MKey Statement, MKeyFind Function, MKeyFld Statement, MKeyOffset Statement, MOpen Functions, MSort Statement
MKeyOffset Statement

Description
Define a key field for a previously opened memory array.

Syntax
Call MKeyOffset(MemHandle, KeySegmentNbr, bTable, KeyFldByteOffset, KeyFldDataType, KeyFldDataLength, Ascending)

Remarks
Occasionally a program will need the ability to easily locate a particular record within a memory array based on one or more key field values. The MKeyFind function can be used to accomplish this goal assuming the sort order for the memory array has been previously defined. Memory arrays associated with an DSLGrid control automatically have their sort order initialized by the DetailSetup function based on the key field control(s) contained within the grid (for example, notated by a “,k” in the levels property of the controls). All other memory arrays must have their sort order explicitly defined via one of several different methods. Each of the methods to define a key field, such as MKey, MKeyFld, MKeyHctl and MKeyOffset, vary primarily in the way they acquire detailed information on a key field such as datatype, size and byte offset within a user-defined datatype.

The MKeyOffset method is the most flexible method of defining memory array key fields but it is also the most detailed to code. It is designed to facilitate the definition of a key field that does not exist in the database and therefore has no correlated data dictionary information in the database. This situation can occur if one of the user-defined datatypes in a memory array is only declared within Visual Basic and does not exist within the database. In such a case, the system has no way of determining the byte offset from the beginning of the structure for any particular field, the field datatype nor the length of the field. The MKeyOffset statement allows the developer to explicitly pass all of this detailed information relating to the designated key field since it does not exist in the SQL data dictionary.

Multi-segment keys can be defined by successive calls to MKeyOffset with different KeySegmentNbr argument values.

The MKeyOffset statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Unique handle to a previously opened memory array.</td>
</tr>
<tr>
<td>KeySegmentNbr</td>
<td>Integer</td>
<td>Memory array key segment whose key field is being defined. The first key segment number is always zero. Multi-segment keys must have contiguous key segment values such as 0 and 1 as opposed to 0 and 3. The maximum allowable number of key segments is five.</td>
</tr>
<tr>
<td>bTable</td>
<td>User-defined datatype</td>
<td>Memory array table structure containing the designated key field. This table structure must also have been previously passed to the MOpen call.</td>
</tr>
<tr>
<td>KeyFldByteOffset</td>
<td>Integer</td>
<td>This argument is designed to help the system locate the first byte of the designated key field. The system will already know the memory location of the first byte of the entire user-defined datatype via the bTable argument. The byte offset tells the system how far the first byte of the designated key field is offset from the first byte of the entire user-defined datatype. If the designated key field is the first field in the user-defined datatype then a value of zero should be passed.</td>
</tr>
</tbody>
</table>
### KeyFldDataType
- **Type:** Integer
- **Description:** Specifies the datatype of the designated key field. The following datatype constants are declared in Solomon.VBTools.vb:
  - DATA_TYPE_STRING
  - DATA_TYPE_FLOAT
  - DATA_TYPE_INTEGER
  - DATA_TYPE_DATE
  - DATA_TYPE_TIME
  - DATA_TYPE_LOGICAL

### KeyFldDataLength
- **Type:** Integer
- **Description:** Size of the designated key field. For example, `bTable.KeyFld.Length`.

### Ascending
- **Type:** Integer
- **Description:** True if the key segment should be sorted ascending. False to implement a descending sort sequence for the key segment currently being defined.

### See Also
- MKey Statement, MKeyFind Function, MKeyFld Statement, MKeyHctl Statement, MOpen Functions, MSort Statement

### Example
The following example illustrates a memory array containing only selected fields from the Employee table that is nevertheless sorted by employee ID. By only storing selected Employee fields in the memory array, much less memory will be consumed for each record within the memory array. Since not all fields in the Employee database table are contained within the Employee_SelFld user-defined datatype, the data dictionary information in the SQL database corresponding to the standard Employee table is not usable by the system for purposes of determining the required key field information. Consequently, **MKeyOffset** must be utilized to implement sorting on the Employee ID key field.

Code to declare the user-defined datatype containing only selected fields from the Employee table. Notice that the **Name** field is being deliberately declared before the **EmpId** field so as to further illustrate the complete flexibility of **MKeyOffset**.

```vbnet
type Employee_SelFld
    Name  As String * 30
    EmpId As String * 10
End Type
```

```vbnet
Global bEmployee_SelFld  As Employee_SelFld
```

Code to open a memory array for the **bEmployee_SelFld** user-defined datatype and define Employee ID as the key field.

```vbnet
Dim Mem_Employee_SelFld  As Integer
Mem_Employee_SelFld = MOpen( TRUE, bEmployee_SelFld, Len(bEmployee_SelFld), P_NULL, 0, P_NULL, 0, P_NULL, 0)
Call MKeyOffset(Mem_Employee_SelFld, 0, bEmployee_SelFld, 30, DATA_TYPE_STRING, bEmployee_SelFld.EmpId.Length, True)
```
Code to load the memory array with relevant selected fields for all employees in the database. Notice that the order of the fields in the SQL Select statement correspond to the order of the fields in the Employee_SelFld user-defined datatype.

```
Dim CSR_Employee_SelFld  As Integer
Dim SqlStr     As String
Dim Employee_SelFld_Fetch  As Integer

'Allocate a cursor
   Call SqlCursor(CSR_Employee_SelFld, NOLEVEL)

'Initialize cursor with a SQL statement and immediately fetch
'the first record
   SqlStr = "Select Name, EmpId from Employee Order By EmpId"
   Employee_SelFld_Fetch = SqlFetch1(CSR_Employee_SelFld, SqlStr, bEmployee_SelFld)

'Read through all subsequent Employee records, inserting each one into
'the memory array.
   While(Employee_SelFld_Fetch = 0)

      'Insert current Employee record into the memory array
         Call MInsert( Mem_Employee_SelFld)

      'Fetch the next Employee record
         Employee_SelFld_Fetch = SFetch1(CSR_Employee_SelFld, bEmployee_SelFld)

   Wend
```
**MLast Function**

**Description**
Move to the last record in a designated memory array.

**Syntax**

\[
\text{RecFetch} = \text{MLast}(\text{MemHandle, RecMaintFlg})
\]

**Remarks**

`MLast` moves to the last record of a specified memory array and copies the contents of the array record into the data structure(s) previously specified in either the `MOpen` or `DetailSetup` function call used to originally open the relevant memory array.

When this call is used on a memory array associated with a `DSLGrid` control (for example, memory arrays opened automatically via the `DetailSetup` function), an `MDisplay` call will be needed to properly synchronize the grid appearance with the memory array.

The `MLast` function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>RecFetch</code></td>
<td>Integer</td>
<td>0 if a record is successfully fetched. NOTFOUND is returned if no records exist in the specified memory array.</td>
</tr>
<tr>
<td><code>MemHandle</code></td>
<td>Integer</td>
<td>Memory array resource handle.</td>
</tr>
<tr>
<td><code>RecMaintFlg</code></td>
<td>Integer</td>
<td>Status of the memory array record (assuming it was successfully fetched). Solomon.VBTools.vb contains the following symbolic constants defining possible memory array record status values: INSERTED, UPDATED and NOTCHANGED.</td>
</tr>
</tbody>
</table>

**See Also**

`MFirst Function`, `MNext Function`, `MPrev Function`

**MLoad Statement**

**Description**
Load a memory array with all records returned from the database by an SQL statement.

**Syntax**

Call `MLoad(MemHandle, Cursor)`

**Remarks**

There are two ways to load data directly from the database into a memory array. The most obvious method is to insert one record at a time into the memory array until no additional records are returned from the database. A simpler method is to load the entire array via a single call to the `MLoad` statement. The only requirement is that the `Cursor` passed as a parameter to the `MLoad` statement must already be initialized with an SQL Select statement or stored procedure so it is ready to begin returning data. The cursor can be initialized by passing the SQL Select statement or stored procedure, along with any necessary parameters, to the `SQL` statement.
The **MLoad** statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Memory array resource handle.</td>
</tr>
<tr>
<td>Cursor</td>
<td>Integer</td>
<td>SQL database cursor. This cursor must have already been initialized with an SQL Select statement or stored procedure.</td>
</tr>
</tbody>
</table>

**See Also**

```
DetailLoad Statement, Sql Statement
```

**Example**

This example illustrates how to open a memory array and load the entire chart of accounts into the newly-created array using a single call to the **MLoad** statement.

```vba
Dim Mem_Account As Integer
Dim CSR_Account As Integer
Dim SqlStr   As String

'Open memory array to hold Chart of Accounts
Mem_Account = MOpen( TRUE, bAccount, Len(bAccount), PNULL, 0, PNULL, 0, PNULL, 0)

'Allocate cursor
Call SqlCursor( CSR_Account, NOLEVEL)

'Initialize cursor with a SQL statement
SqlStr = "Select * from Account order by Acct"
Call Sql(CSR_Account, SqlStr)

'Load the memory array with all Account records
Call MLoad( Mem_Account, CSR_Account)
```

**MNext Function**

**Description**

Move to the next record in a designated memory array.

**Syntax**

```
RecFetch = MNext(MemHandle, RecMaintFlg)
```

**Remarks**

**MNext** moves to the next record of a specified memory array and copies the contents of the array record into the data structure(s) previously specified in either the **MOpen** or **DetailSetup** function call used to originally open the relevant memory array.
When this call is used on a memory array associated with an DSLGrid control (for example, memory arrays opened automatically via the DetailSetup function), an MDisplay call will be necessitated to properly synchronize the DSLGrid appearance with the memory array.

The MNext function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecFetch</td>
<td>Integer</td>
<td>0 if a record is successfully fetched. NOTFOUND is returned if the current record is already the last record in the specified memory array as well as when no records exist.</td>
</tr>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Memory array resource handle.</td>
</tr>
<tr>
<td>RecMaintFlg</td>
<td>Integer</td>
<td>Status of the memory array record (assuming it was successfully fetched). Solomon.VBTools.vb contains the following symbolic constants defining possible memory array record status values: INSERTED, UPDATED and NOTCHANGED.</td>
</tr>
</tbody>
</table>

See Also
MFirst Function, MLast Function, MPrev Function

MOpen Functions

Description
Open a new memory array and return a corresponding unique memory array number.

Syntax
MemHandle = MOpen(DelRetToSystem, bTable1, bTable2, bTable3, bTable4)
MemHandle = MOpen8(DelRetToSystem, bTable1, bTable2, bTable3, bTable4, bTable5, bTable6, bTable7, bTable8)

Remarks
A memory array must be opened before insert, update, delete or memory array navigation operations can be performed on it. The memory array handle returned by MOpen and MOpen8 is used when performing these types of operations on memory arrays.

MOpen allocates memory for up to four table structures whereas MOpen8 can handle up to eight different SolomonDataObjects for each memory array record.

MOpen is only used to open memory arrays which are not associated with an DSLGrid control. Memory arrays which correspond to grids are opened automatically by either the DetailSetup or DetailSetup8 function.

The MOpen function uses the following arguments (MOpen8 has eight SolomonDataObjects. PNULL should be passed for unused SolomonDataObject parameters)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Unique handle to the newly created memory array. If a new memory array was successfully opened this value will be &gt;= 0.</td>
</tr>
<tr>
<td>DelRetToSystem</td>
<td>Control</td>
<td>Controls the handling of system memory with respect to deleted memory array lines. True causes memory consumed by deleted memory array records to be returned to system memory. False causes the memory to be retained. Normally True should be used.</td>
</tr>
<tr>
<td>bTable1</td>
<td>SolomonDataObject</td>
<td>First SolomonDataObject of memory array.</td>
</tr>
<tr>
<td>bTable2</td>
<td>SolomonDataObject</td>
<td>Second SolomonDataObject of memory array. PNULL if the memory array only contains one table structure.</td>
</tr>
<tr>
<td>bTable3</td>
<td>SolomonDataObject</td>
<td>Third SolomonDataObject of memory array. PNULL if the memory array contains less than three table structures.</td>
</tr>
</tbody>
</table>
The table structures passed to **MOpen** do not have to be actual database tables. They can be either a simple data item, such as a double, or any user-defined data type created with the Visual Basic Type statement.

**See Also**

MClear Statement, MClose Statement, MDelete Function, MDisplay Statement, MFirst Function, MInsert Statement, MKey Statement, MKeyFind Function, MLast Function, MNext Function, MPrev Function, MUpdate Statement

**Example**

This example illustrates how to open a memory array and load the entire chart of accounts into the newly created array.

```vba
Dim Mem_Account As Integer
Dim CSR_Account As Integer
Dim SqlStr As String
Dim Account_Fetch As Integer

'Open memory array to hold Chart of Accounts
Mem_Account = MOpen( TRUE, bAccount, PNULL, PNULL, PNULL)

'Allocate cursor
Call SqlCursor( CSR_Account, NOLEVEL)

'Initialize cursor with a SQL stored procedure and immediately fetch first record
SqlStr = "Select * from Account order by Acct"
Account_Fetch = SqlFetch1(CSR_Account, SqlStr, bAccount)

'Read through all subsequent Account records, inserting each one into the memory array.
While( Account_Fetch = 0)
    'Insert current Account record into the memory array
    Call MInsert( Mem_Account)

    'Fetch the next Account record
    Account_Fetch = SFetch1(CSR_Account, bAccount)
Wend
```

**MPrev Function**

**Description**

Move to the previous record in a designated memory array.

**Syntax**

```vba
RecFetch = MPrev(MemHandle, RecMaintFlg)
```
Remarks

**MPrev** moves to the previous record of a specified memory array and copies the contents of the array record into the data structure(s) previously specified in either the **MOpen** or **DetailSetup** function call used to originally open the relevant memory array.

When this call is used on a memory array associated with a **DSLGrid** control (for example, memory arrays opened automatically via the **DetailSetup** function), an **MDisplay** call will be necessitated to properly synchronize the **DSLGrid** appearance with the memory array.

The **MPrev** function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecFetch</td>
<td>Integer</td>
<td>0 if a record is successfully fetched. NOTFOUND is returned if the current record is already the first record in the specified memory array as well as when no records exist.</td>
</tr>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Memory array resource handle.</td>
</tr>
<tr>
<td>RecMaintFlg</td>
<td>Integer</td>
<td>Status of the memory array record (assuming it was successfully fetched). Solomon.VBTools.vb contains the following symbolic constants defining possible memory array record status values: INSERTED, UPDATED and NOTCHANGED.</td>
</tr>
</tbody>
</table>

See Also

**MFirst Function, MLast Function, MNext Function**

**MSet Statement**

Description

Explicitly set the value of a particular control for every record in its corresponding **DSLGrid** control.

Syntax

Call **MSet**(Control, NewDataValue)

Remarks

The **MSet** statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>Control</td>
<td>Control which is bound to the Record.FieldName whose value is to be changed for every record in the relevant <strong>DSLGrid</strong>. <strong>Note:</strong> This statement is only for use with controls associated with an <strong>DSLGrid</strong> control.</td>
</tr>
<tr>
<td>NewDataValue</td>
<td>String</td>
<td>New data value for the Record.FieldName associated with the designated Control. The data value must be in a string format.</td>
</tr>
</tbody>
</table>

See Also

**MSetProp Statement**

**MSetLineStatus Function**

Description

Set the line status of the current record in the designated memory array.

Syntax

```plaintext
RetVal = MSetLineStatus(MemHandle, NewLineStatus)
```
Remarks
Each record within a memory array has its own line status such as INSERTED, UPDATED or NOTCHANGED. The system automatically modifies the line status based on the type of activity last performed on any particular memory array record. For example, if a record is inserted into a memory array using MInsert, then the status of that new memory array record will automatically be set to INSERTED. Occasionally, however, the application may need to assign a specific line status to a particular memory array record. In such cases the MSetLineStatus function can be used to carry out this task.

One common usage of this function is when a memory array associated with an DSLGrid control is being loaded from the database under application control. In these cases records are being inserted into the memory array via successive MInsert calls. However since the data is in no way modified between the time it is fetched and the time it is inserted into the memory array the line status of the resultant memory array record is forced to change from INSERTED to NOTCHANGED.

The MSetLineStatus function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RetVal</td>
<td>Integer</td>
<td>-1 if a memory array handle that is not valid is passed.</td>
</tr>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Memory array resource handle.</td>
</tr>
<tr>
<td>RecMaintFlg</td>
<td>Integer</td>
<td>New status of the current memory array record. Solomon.VBTools.vb contains the following symbolic constants defining possible memory array record status values: INSERTED, UPDATED and NOTCHANGED.</td>
</tr>
</tbody>
</table>

See Also
MGetLineStatus Function, MInsert Statement, MUpdate Statement

Example
This example illustrates how to load a memory array from the database and at the same time force the line status of all newly-inserted memory array records to be NOTCHANGED.

```vbnet
Dim Mem_Account As Integer
Dim CSR_Account As Integer
Dim SqlStr  As String
Dim Account_Fetch As Integer

'Open memory array to hold Chart of Accounts
    Mem_Account = MOpen( TRUE, bAccount, Len(bAccount), PNULL, 0, PNULL, 0, PNULL, 0)

'Allocate cursor
    Call SqlCursor( CSR_Account, NOLEVEL)

'Initialize cursor with a SQL statement and immediately fetch first record
    SqlStr = "Select * from Account order by Acct"
    Account_Fetch = SqlFetch1(CSR_Account, SqlStr, bAccount)

'Read through all subsequent Account records, inserting each one into the memory array.
    While( Account_Fetch = 0)
        'Insert current Account record into the memory array
            Call MInsert( Mem_Account)
        'Since the record ALREADY exists in the database, reset the line status of the current memory array record so that the
```
'application will be able to detect whether or not any calls
to MUpdate were subsequently made for the current record.
Call MSetLineStatus( Mem_Account, NOTCHANGED)

'Fetch the next Account record
Account_Fetch = SFetch1(CSR_Account, bAccount)
Wend

MSetProp Statement

Description
Set the value of a particular property for both the designated form view control as well as its associated DSLGrid control.

Syntax
Call MSetProp(Control, PropertyName, NewPropValue)

Remarks
At runtime, each column of an DSLGrid control is associated with an underlying (form view) Microsoft Dynamics SL SDK custom control. The MSetProp statement can be used to modify a property setting for both the underlying control as well as the relevant column in the associated grid. For example, an entire column can be disabled using the MSetProp statement.

If the application wants to modify property values on a line by line basis, as opposed to an entire column, then it will need to manage the property settings using calls to the SetProps statement from within the LineGotFocus event of each individual detail line.

Note: MSetProp cannot be called from within the FormLoad event of a Visual Basic application. MSetProp relies on the grid control to be completely configured before changing its properties. The grid is not completely configured until the form is displayed on the screen. This display occurs after the FormLoad event has occurred.

The MSetProp statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>Control</td>
<td>Control whose designated property value is to be modified both in form view as well as grid view.</td>
</tr>
<tr>
<td>PropertyName</td>
<td>String</td>
<td>Name of the property whose value is to be modified.</td>
</tr>
<tr>
<td>NewPropValue</td>
<td>String or Integer</td>
<td>New property value. The actual datatype varies based on the property being referenced.</td>
</tr>
</tbody>
</table>

See Also
BlankErr Property, Enabled Property, Heading Property, Mask Property, Min Property, Max Property, SetProps Statement, DSLGrid Control, Visible Property

Example
The Release Payroll Batches process screen contains an DSLGrid control which automatically displays all Payroll batches that are ready to be released. The DSLGrid control only exists to facilitate the users need to specify which batches should be released. Consequently, for example, the Insert, Save, and Delete buttons on the toolbar do not need to be enabled since the user is not allowed insert new batches nor update existing batches. This is accomplished using the SetButton statement. However, since disabling all of these buttons causes all controls within the DSLGrid to be disabled, the application needs to subsequently re-enable the one control used to select individual batches for processing.
'Disable buttons so user will not be able to take any action on batches in the grid
   Call SetButton(DeleteButton + InsertButton + SaveButton + \n   CancelButton, AllLevels, False)

'Re-enable the control used to Select/Deselect batches.
   Call MSetProp(cCurrBatSelected, PROP_ENABLED, True)

MSetRow Statement

Description
Set the current row / record number of a designated memory array.

Syntax
Call MSetRow(MemHandle, NewCurrRecNbr)

Remarks
An application can jump to a specific memory array record via either the MKeyFind function or the MSetRow statement. MSetRow jumps to a specific record number whereas MKeyFind locates the record with designated key field values.

The MSetRow statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Memory array resource handle.</td>
</tr>
<tr>
<td>NewCurrRecNbr</td>
<td>Integer</td>
<td>Row / record number of the desired memory array record.</td>
</tr>
</tbody>
</table>

See Also
MGetRowNum Function, MKeyFind Function

MSort Statement

Description
Sort data contained within an existing memory array based upon predefined key fields.

Syntax
Call MSort(MemHandle)

Remarks
This statement will sort all existing records within the designated memory array based upon key fields previously defined using one of the MKey, MKeyFld, MKeyHctl or MKeyOffset statements.

If the data within the memory array is initially loaded and maintained in the proper order, a call to MSort will not be necessary. MSort only needs to be called if one or more records within the memory array are not in an order consistent with the previously defined key fields. When the data is out of order, MKeyFind will not work properly since it assumes that the data within the memory array is consistent with the predefined sort sequences.

Memory arrays associated with a DSLGrid control require an MDisplay call to redisplay the grid, subsequent to calling MSort.
See Also
MDisplay Statement, MKey Statement, MKeyFind Function, MKeyFld Statement, MKeyHctl Statement, MKeyOffset Statement, MOpen Functions

MUpdate Statement

Description
Update the current memory array record of a designated memory array with new data values.

Syntax
Call MUpdate(MemHandle)

Remarks
MUpdate is used to update an existing record of a specified memory array. This is accomplished by copying the new contents of all data structures previously associated with the designated memory array over the existing memory array record. Data structures are associated with a memory array in either the MOpen or DetailSetup function call used to originally open the relevant memory array. If the memory array record had a line status of INSERTED prior to the MUpdate call then the line status will remain INSERTED. In all other cases, the memory array record will have a line status of UPDATED.

The MUpdate statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Memory array resource handle.</td>
</tr>
</tbody>
</table>

See Also
DetailSetup Functions, MDelete Function, MInsert Statement, MOpen Functions, MSetLineStatus Function

NoteCopy Function

Description
Find the source sNote record using the SourceNoteid, then create a new sNote record with the same Note text.

Syntax
DestinationNoteId = NoteCopy(SourceNoteId, DestRecordType)

Remarks
NoteCopy returns an integer that is the new noteid for the sNote that was created. If NoteCopy returns a 0, then it was not successful (possibly a note did not exist for the source record).

A transaction needs to be open at the time that NoteCopy is called so that the sNote record that is created can be saved. The save of the sNote record happens within the NoteCopy function, but the save of the destination record does not. Once the noteid is returned and the destination.noteid field is filled, it is up to the developer to determine whether the record will need to be saved explicitly or whether SWIM will save it.

The NoteCopy function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SourceNoteId</td>
<td>Integer</td>
<td>Identifier for the source sNote record.</td>
</tr>
<tr>
<td>DestRecordType</td>
<td>String</td>
<td>The name of the destination record for the Note.</td>
</tr>
</tbody>
</table>

Example
This example is taken from code that is creating a new shipper from an existing one. If a note exists on the existing shipper, it is copied to the new shipper record.

If (bSOShipHeader.NoteID = 0) Then
    bSOShipHeader.NoteID = NoteCopy(bSOHeader.NoteID, "SOShipHeader")
End If

### PasteTemplate Function

**Description**
Paste information from the designated template into the current application.

**Syntax**

\[
RetVal = \text{PasteTemplate}(\text{TemplateID})
\]

**Remarks**

The PasteTemplate statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RetVal</td>
<td>Integer</td>
<td>Zero if no errors occurred. The NOTFOUND symbolic constant, declared in Solomon.VBTools.vb, is returned if the TemplateID does not exist.</td>
</tr>
<tr>
<td>TemplateID</td>
<td>String</td>
<td>ID of the template whose information is to be pasted into the current Microsoft Dynamics SL SDK application screen.</td>
</tr>
</tbody>
</table>

**See Also**

SaveTemplate Statement

### PeriodCheck Function

**Description**
Verify whether or not a period string in YYYYPP format represents a valid fiscal period.

**Syntax**

\[
RetVal = \text{PeriodCheck}(\text{PeriodString})
\]

**Remarks**

The PeriodCheck function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RetVal</td>
<td>Integer</td>
<td>Value of the period number portion of PeriodString if the string represents a valid fiscal period. Otherwise a value of -1 will be returned if the string is not valid. A period string is not valid if the period portion is less than one or greater than the number of valid fiscal periods defined in the GLSetup record.</td>
</tr>
<tr>
<td>PeriodString</td>
<td>String</td>
<td>Period string to be verified. Must be in YYYYPP format.</td>
</tr>
</tbody>
</table>
**PeriodMinusPeriod Function**

**Description**
Return the number of periods between two fiscal periods.

**Syntax**
\[ \text{NbrPeriods} = \text{PeriodMinusPeriod}(\text{PerNbr1}, \text{PerNbr2}) \]

**Remarks**
The `PeriodMinusPeriod` function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NbrPeriods</td>
<td>Integer</td>
<td>Number of periods between <code>PerNbr1</code> and <code>PerNbr2</code>. If <code>PerNbr1</code> &gt; <code>PerNbr2</code> then the number of fiscal periods between the two periods will be a negative value.</td>
</tr>
<tr>
<td><code>PerNbr1</code></td>
<td>String</td>
<td>Beginning fiscal period. Must be in YYYYPP format.</td>
</tr>
<tr>
<td><code>PerNbr2</code></td>
<td>String</td>
<td>Ending fiscal period. Must be in YYYYPP format.</td>
</tr>
</tbody>
</table>

**Note:** This function will use the number of periods in a fiscal year, as specified in the GLSetup record, in order to derive an accurate result.

**See Also**
PeriodPlusPerNum Function

**PeriodPlusPerNum Function**

**Description**
Add a designated number of periods to an existing fiscal period.

**Syntax**
\[ \text{ResultingPerNbr} = \text{PeriodPlusPerNum}(\text{CurrPerNbr}, \text{NbrPeriodsToAdd}) \]

**Remarks**
The `PeriodPlusPerNum` function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResultingPerNbr</td>
<td>String</td>
<td>Result of <code>CurrPerNbr</code> + <code>NbrPeriodsToAdd</code>.</td>
</tr>
<tr>
<td><code>CurrPerNbr</code></td>
<td>String</td>
<td>Starting fiscal period. Must be in YYYYPP format.</td>
</tr>
<tr>
<td><code>NbrPeriodsToAdd</code></td>
<td>Integer</td>
<td>Number of fiscal periods to add to <code>CurrPerNbr</code>. Negative values are supported.</td>
</tr>
</tbody>
</table>

**Note:** This function will use the number of periods in a fiscal year, as specified in the GLSetup record, in order to derive an accurate result.

**See Also**
PeriodMinusPeriod Function
PVChk Function

Description
Perform possible value error checking for the designated control.

Syntax
\[ RetVal = \text{PVChk}(Ctrl, Cursor, SQLParmValue) \]

Remarks
PVChk performs error checking on the specified control using the stored procedure or SQL text defined in the PV property of the control. The cursor is allocated if uninitialized.

The difference between this function and the \text{PVChkFetch} functions is that this function will not return the record to the application for display.

The PVChk function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RetVal</td>
<td>Integer</td>
<td>0 if a record is successfully fetched. NOTFOUND is returned if no records match the restriction clause of the PV SQL statement.</td>
</tr>
<tr>
<td>Ctrl</td>
<td>Control</td>
<td>Control handle. Can be PNULL if calling PVChk from the control's Chk event.</td>
</tr>
<tr>
<td>Cursor</td>
<td>Integer</td>
<td>SQL database cursor.</td>
</tr>
<tr>
<td>SQLParmValue</td>
<td>String</td>
<td>Key value passed as the last parameter to the restriction clause of the PV SQL statement.</td>
</tr>
</tbody>
</table>

See Also

DSLMaskedText Control, PVChkFetch Functions, PV Property

PVChkFetch Functions

Description
Retrieve a composite record from the database using an SQL statement from the PV property of an DSLMaskedText control.

Syntax

\[
\begin{align*}
RetVal &= \text{PVChkFetch1}(Ctrl, Cursor, SQLParmValue, bTable1) \\
RetVal &= \text{PVChkFetch4}(Ctrl, Cursor, SQLParmValue, bTable1, bTable2, bTable3, bTable4) \\
RetVal &= \text{PVChkFetch8}(Ctrl, Cursor, SQLParmValue, bTable1, bTable2, bTable3, bTable4, bTable5, bTable6, bTable7, bTable8)
\end{align*}
\]

Remarks

Each DSLMaskedText control has a PV property which can contain an SQL statement or stored procedure name. These functions can be used to fetch a composite record instance based on the SQL text from the PV property of the control specified in the Ctrl parameter. These functions are not applicable if the PV property does not contain either an SQL statement or stored procedure name.

PVChkFetch1 is designed for SQL statements returning data from a single table. For more advanced SQL statements having one or more table joins either \text{PVChkFetch4} or \text{PVChkFetch8} can be used.

The PVChkFetch1 function uses the following arguments (PVChkFetch4 and PVChkFetch8 respectively have four and eight SolomonDataObjects and corresponding lengths. PNULL should be passed for unused SolomonDataObject parameters.)
### Reference 245

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RetVal</td>
<td>Integer</td>
<td>0 if a record is successfully fetched. NOTFOUND is returned if no records match the restriction clause of the PV SQL statement.</td>
</tr>
<tr>
<td>Ctrl</td>
<td>Control</td>
<td>Control containing the PV property to be used as the SQL statement. Can optionally be PNULL if the call is made within the Chk event of the control whose PV property is being used.</td>
</tr>
<tr>
<td>Cursor</td>
<td>Integer</td>
<td>SQL database cursor.</td>
</tr>
<tr>
<td>SQLParmValue</td>
<td>String</td>
<td>Key value passed as the last parameter to the restriction clause of the PV SQL statement.</td>
</tr>
<tr>
<td>bTable1</td>
<td>SolomonDataObject</td>
<td>Table structure corresponding to the primary table in the PV SQL statement.</td>
</tr>
</tbody>
</table>

### See Also

DBNavFetch Functions, PVChk Function, DSLMaskedText Control, PV Property

### Example

The following example illustrates the usage of `PVChkFetch1` in the Chk event of a Payroll Work Location ID control. Since PNULL is passed for the control parameter the SQL statement in the PV property of the cWrkLocId control itself is used. The ID entered by the user is passed to the Chk event as chkstrg. By sending this value to `PVChkFetch1` it will be used as the last parameter to the restriction clause of the PV SQL statement.

```vbscript
Sub cWrkLocId_Chk (chkstrg As String, retval As Integer) Handles cWrkLocId.ChkEvent
    Dim WorkLocation_Fetch As Integer
    WorkLocation_Fetch = PVChkFetch1(PNULL, CSR_WorkLocation, chkstrg, bWorkLoc)
    RetVal = NoAction
End Sub
```

### SaveTemplate Statement

**Description**
Save information from the current application to a designated template.

**Syntax**

Call `SaveTemplate(TemplateID, Description, AppliesToUserID, IncludeLowerLevels, StartingLevelNbr)`

**Remarks**

The Template feature makes it possible to store data from the current screen and subsequently paste that data into the same screen at a later time. These timesaving templates can be saved to the database programmatically using the `SaveTemplate` statement as well as via the Template menu item on the Edit menu. Each template can contain complete transactions and entities or individual fields selected by the user. Relative date and period features allow a template to paste data relative to the current date and fiscal period. Templates can be designated as private to a specific user or marked public for availability to all users. Templates are stored in the system database and therefore they are independent of any particular application database.

Unless otherwise specified, all date and period values pasted from a template will be equal to the Microsoft Dynamics SL business date, located on the File menu, and the current period for the module. To override this default action, the user entering data for the template must specify a new relative date or period value for each desired field. This is done immediately before saving a template. Specifying a relative date or period value for a field contained in grid detail lines will change the
template value of that field for all detail lines. Relative values can be defined by selecting the pertinent
date or period field and pressing F2 to start the Relative Date or Relative Period screen, whichever is
appropriate.

The **SaveTemplate** statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| TemplateID        | String| ID of the template being created or updated. If a template with the
designated TemplateID already exists, it will be overwritten. The
TemplateID can be up to 30 characters.                                      |
| Description       | String| Description of the template.                                                |
| AppliesToUserID   | String| Microsoft Dynamics SL User ID to which the template applies. By
default, the template will be Public if AppliesToUserID is left blank. |
| IncludeLowerLevels| Integer| False if only data for the StartingLevelNbr is to be saved to the
template. Otherwise data for lower levels, in addition to
StartingLevelNbr, will also be saved to the template.                   |
| StartingLevelNbr  | Integer| Number of the first application level for which data is to be saved
to the template. For example, on Batch / Document / Detail screens, a value of zero could be passed to start the save with the Batch level. Solomon.VBTools.vb contains two symbolic constants which can also be passed:
CcpSelectedFields — Only those fields currently highlighted by the
user will be saved to the template.
CcpAllLevels — Data from all application levels will be saved to the
template regardless of the IncludeLowerLevels argument value.             |

**See Also**

PasteTemplate Function

**ScreenExit Statement**

**Description**

Terminate the dynamic link with the Parent, log out of the database and terminate the application.

**Syntax**

Call **ScreenExit**(OpCode, ParmStr)

**Remarks**

The **ScreenExit** call is required in the Unload event of Form1 for all applications developed with the
Microsoft Dynamics SL SDK.
The **ScreenExit** statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>String</td>
<td>Operation Code. If another application should be run after the current application terminates then the name of the next application can be specified using this argument. Parameters can be sent to the next application using the <strong>ParmStr</strong> argument. If the current application was originally called by another application then return values can be passed back to the originating application by passing the <strong>APPLICRETURNPARMS</strong> symbolic constant defined in <strong>Solomon.VBTools.vb</strong>. The actual return value(s) are passed via the <strong>ParmStr</strong> argument. Normally this argument will be blank, in which case the value of <strong>ParmStr</strong> is ignored, the application is terminated and no further action is taken.</td>
</tr>
<tr>
<td>ParmStr</td>
<td>String</td>
<td>Parameter value(s) to be sent to the next application or returned to the calling application. Multiple parameters can be passed by separating each individual parameter with the <strong>PRMSEP</strong> symbolic constant defined in <strong>Solomon.VBTools.vb</strong>.</td>
</tr>
</tbody>
</table>

**See Also**

ApplInit Statement, ScreenInit Statement

**ScreenInit Statement**

**Description**

Perform all screen resource initializations that must occur after a dynamic link with the Parent has been established and relevant resources have been allocated by the application.

**Syntax**

Call **ScreenInit**

**Remarks**

The **ScreenInit** call is required in all applications developed with the Microsoft Dynamics SL SDK. Similar to the **ApplInit** statement, the **ScreenInit** statement initializes application resources. However, the initializations occurring during **ScreenInit** are much more extensive. For example, it is during this call that customizations are loaded, triggers are set up, access rights are checked, an implied NEW operation is fired as well as many other common initialization tasks.

Since database I/O occurs during the **ScreenInit** call, it must be called after **ApplInit**. Additionally, it is required that the **ScreenInit** call occur after all **SetAddr** and **SqlCursor** calls have been made for each level corresponding to the Levels property of the **DSLUpdate** control.

**See Also**

ApplInit Statement, ScreenExit Statement, SetAddr Statement, SqlCursor Statement, DSLUpdate Control

**Example**

The following code illustrates the basic order of calls that are made from within the Form_Load event of Form1. Some of the calls are optional depending on the requirements of a particular application but the **ScreenInit** call is always required.

```vbnet
Sub Form_Load ()
' Load application subform(s)
    'Call LoadForm( SubFormName)
' Call to Initialize the Application (required in all applications)
```
Call ApplInit
'SetAddr call(s)
   'Call SetAddr(LEVEL0, "bTableName", bTableName, nTableName)
'SqlCursor call(s)
   'Call SqlCursor(CSR_TableName, LEVEL0)
'Call to Initialize the Screen (required in all applications)
   Call ScreenInit
'DetailSetup call for simple grid
   'MemHandle_Spread1 = DetailSetup(CSR_TableName, Spread1_TableName, ., bTableName, ., ., .)
End Sub

SDelete Statement

Description
Delete the current record from a designated table within an existing SQL view.

Syntax
Call SDelete(Cursor, TablesDeletingFrom)

Remarks
A value of "*." can be passed as a table name, meaning that all current records in the existing view will be deleted. Please note that this call requires that the current record already be fetched via such functions as SqlFetch1 or SFetch1 so that the designated cursor actually has a current record.

The SDelete statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cursor</td>
<td>Integer</td>
<td>SQL database cursor.</td>
</tr>
<tr>
<td>TablesDeletingFrom</td>
<td>String</td>
<td>Name of each table, in the specified cursor’s view, from which the current record is to be deleted. Multiple table names should be separated by commas.</td>
</tr>
</tbody>
</table>

See Also
SDeleteAll Function, SFetch Functions, SqlFetch Functions

SDeleteAll Function

Description
Delete all records from the designated table(s) within a predefined view.

Syntax
RetVal = SDeleteAll(Cursor, TablesDeletingFrom)

Remarks
Deletes records from some or all of the tables in a view based on the current restrictions. A view must have already been initialized for the specified cursor via functions such as SQL, SFetch1 or SqlFetch4. If no restriction is specified when the cursor is initialized, then this call will remove all records from the designated table(s) in the view.
The **SDeleteAll** function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RetVal</td>
<td>Integer</td>
<td>SQL database cursor.</td>
</tr>
<tr>
<td>Cursor</td>
<td>Integer</td>
<td>SQL database cursor.</td>
</tr>
<tr>
<td>TablesDeletingFrom</td>
<td>String</td>
<td>Name of each table, in the specified cursor’s view, from which all records</td>
</tr>
<tr>
<td></td>
<td></td>
<td>within the current view are to be deleted. Multiple table names should be</td>
</tr>
<tr>
<td></td>
<td></td>
<td>separated by commas. A value of &quot;<em>.</em>&quot; can be used to delete records from</td>
</tr>
<tr>
<td></td>
<td></td>
<td>all tables within the view.</td>
</tr>
</tbody>
</table>

**See Also**

SDelete Statement, SFetch Functions, SQL Statement, SqlFetch Functions

**Example**

Delete all vendors having a zero balance.

```vba
Dim SqlStr As String
'Initialize the cursor with a SQL statement
SqlStr = "Select * from Vendor where CurrBal = 0 and FutureBal = 0"
Call Sql(CSR_Vendor_Del, SqlStr)

'Delete all records matching the restriction clause of the SQL statement
'Delete all records matching the restriction clause of the SQL statement
'used to initialize the CSR_Vendor_Del cursor.
Call SDeleteAll(CSR_Vendor_Del, "*.*")
```

**SetAddr Statement**

**Description**

Associate a table name together with a Visual Basic variable and an optional screen level number.

**Syntax**

Call **SetAddr**(LevelNbr, TableNameStr, bTableName, nTableName)

**Remarks**

The **SetAddr** statement facilitates proper runtime binding between a Visual Basic data variable and relevant data entry controls, including controls created using the Customization Manager.

Although SWIM has access to extensive information about any particular data entry control, one vital piece of information cannot be determined merely by looking at the control itself. In particular, SWIM needs to know where the data for the control is stored in memory. Since the data is actually stored in an underlying Visual Basic variable, SWIM has no means of directly determining where that Visual Basic variable is stored in memory. The **SetAddr** statement is used by the application to resolve this problem.

To facilitate the explanation of the linkage between the **SetAddr** statement and corresponding data entry controls, consider the following user-defined datatype and a corresponding **SetAddr** call for the Account database table:

```
Type Account
    Acct    As String * 10
    Active  As Integer
    ConsolAcct   As String * 10
    CuryId   As String * 4
    Descr    As String * 30
End Type
```
The `SetAddr` call itself associates the value of the `TableNameStr` argument together with the memory location of the first byte of the SolomonDataObject passed via the `bTableName` argument. If the relevant table is a database table, SWIM can access detailed information relating to each individual field contained therein using the SQL data dictionary. For example, SWIM can determine the name, data type, maximum size as well as the relative placement (byte offset, for example) of each individual field within the table. After the `SetAddr` call in the above example, SWIM would know that the first byte of `bVBVarAccount` is at a particular location in memory, hereafter referred to as location M, and furthermore that the first byte of `bVBVarAccount.Acct` is offset zero bytes from location M as well as the fact that Acct is ten bytes long. Similarly, it would also know that the first byte of `bVBVarAccount.Active` is offset ten bytes from location M, and is two bytes long since it is an integer. Since the value of “`bAccount`” is passed as the `TableNameStr` argument it is the string literal associated with memory location M. Anytime SWIM encounters “`bAccount.<SomeFieldName>`” in a `FieldName` property it will have all of this detailed information readily available so it can access the corresponding data at the appropriate memory location. The same concept applies when SWIM encounters “`bAccount.<SomeFieldName>`” in any of the `DBNav`, `Default`, `PV` or `Trigger` properties.

As previously mentioned, the detailed information acquired by SWIM as a result of the `SetAddr` call can be directly linked to the `FieldName` property of data entry controls. The `FieldName` property contains a `Struct.FieldName` value along with other more detailed information such as field offset value, declare type and length. Once they have been fully initialized, these values facilitate the linkage between the control and the associated memory location where the data value of the control is stored. In the vast majority of cases the detailed field information is initialized automatically by SWIM in using information acquired via a corresponding `SetAddr` call and the SQL data dictionary. Usage of an unbound control is the only case where the developer must fill in the detailed field information manually since it will not exist in the SQL data dictionary.

The `Struct.FieldName` portion of the `FieldName` property must always be populated with a string value in the “`bTableName.Fieldname`” format. Using the above example, a control for the Acct field would have a value of “`bAccount.Acct`” for the `Struct.FieldName` portion of its `FieldName` property. Similarly the Active control would have a value of “`bAccount.Active`” for the `Struct.FieldName` portion of its `FieldName` property.

The first portion of this `Struct.FieldName` string, “`bAccount`” in our example, is used to link the control with a particular `SetAddr` call that had the same value for its `TableNameStr` argument. Once that initial linkage is made the exact memory location can be determined automatically by correlating the last portion of the `Struct.FieldName` string, “Acct” or “Active” in our example, with the detailed field information acquired as a result of the relevant `SetAddr` call.

Thus, in general the `SetAddr` call facilitates the linkage of a “`bTableName.Fieldname`” type of string literal with a precise location in memory for the corresponding data value.

The `SetAddr` statement uses the following arguments:
**Reference 251**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LevelNbr</td>
<td>Integer</td>
<td>Level Number. The Levels property of the DSLUpdate control is used to define all of the levels for a particular screen. If the table referred to by a particular SetAddr call is the master table for one of those levels then the corresponding level number should be passed via this argument. Symbolic constants have been defined in Solomon.VBTools.vb for LEVEL0 through LEVEL9. Note that level numbers begin with LEVEL0. If the table is not a master table for any level on the screen then the NOLEVEL symbolic constant should be passed.</td>
</tr>
<tr>
<td>TableNameStr</td>
<td>String</td>
<td>Table name string literal which can subsequently be used by SWIM to link the table name portion of a “bTableName.FieldName” type of string literal with a precise location in memory. By convention this value should begin with a “b” such as “bAccount” for the Account table. <strong>Note:</strong> This value does not have to correlate to a database table. In these cases, the system will assume it refers to an unbound data buffer. All references to unbound “table names” in DBNav, Default, FieldName and PV properties must be accompanied with manually entered detailed field information since the system will be unable to access this information from the SQL data dictionary.</td>
</tr>
<tr>
<td>bTableName</td>
<td>SolomonDataObject</td>
<td>SolomonDataObject whose datatype corresponds to the table referred to in the TableNameStr argument. For example, if “bAccount” is passed then the SolomonDataObject passed via this argument must be a user-defined datatype whose structure precisely corresponds to the Account table in the database.</td>
</tr>
<tr>
<td>nTableName</td>
<td>SolomonDataObject</td>
<td>SolomonDataObject whose datatype corresponds to the table referred to in the TableNameStr argument. If TableNameStr does not correlate to a database table then PNULL must be passed as the value for this argument. This value will be properly initialized with null values as long as the relevant table is a database table. Any structure of the relevant datatype can then be easily blanked out using the bTableName = nTableName methodology.</td>
</tr>
</tbody>
</table>

**See Also**

FieldName Property, DSLUpdate Control

**Example**

The following example illustrates how to perform SetAddr calls for a moderately complex screen containing multiple levels such as the Payroll Employee Maintenance screen. Notice that some of the SetAddr calls use the NOLEVEL flag, indicating that these tables either do not have any corresponding data entry controls or that these tables are merely not the master table for any level. In any case, detailed field level information will be stored for each of these tables. Additionally, the null structure for each of the tables will still be properly initialized.

```vbscript
Sub Form_Load ()
    Call LoadForm(F0225001) 'Timesheet Defaults
    Call LoadForm(F0225002) 'Miscellaneous Info
    Call LoadForm(F0225003) 'Pay Information
    Call LoadForm(F0225004) 'Employee Deductions
    Call LoadForm(F0225005) 'Employee Benefits

    Call ApplInit

    Call SetAddr(LEVEL0, "bEmployee", bEmployee, nEmployee)
    Call SetAddr(LEVEL1, "bWorkLoc", bWorkLoc, nWorkLoc)
End Sub
```
Call SetAddr(LEVEL2, "bEarnType", bEarnType, nEarnType)
Call SetAddr(LEVEL3, "bPayGroup", bPayGroup, nPayGroup)
Call SetAddr(LEVEL4, "bEarnDed", bEarnDed, nEarnDed)
Call SetAddr(LEVEL5, "bBenEmp", bBenEmp, nBenEmp)

Call SetAddr(NOLEVEL, "bPRDoc", bPRDoc, nPRDoc)
Call SetAddr(NOLEVEL, "bPRSetup", bPRSetup, nPRSetup)
Call SetAddr(NOLEVEL, "bPRTran", bPRTran, nPRTran)
Call SetAddr(NOLEVEL, "bDeduction", bDeduction, nDeduction)
Call SetAddr(NOLEVEL, "bBenefit", bBenefit, nBenefit)

'Allocate one cursor for each level on the screen
Call SqlCursor(CSR_Employee, LEVEL0)  'Employee Navigation
Call SqlCursor(CSR_WorkLoc, LEVEL1)   'WorkLoc Lookup Level
Call SqlCursor(CSR_EarnType, LEVEL2)  'EarnType Lookup Level
Call SqlCursor(CSR_PayGroup, LEVEL3)  'PayGroup Lookup Level
Call SqlCursor(CSR_EarnDed_DBNav, LEVEL4)  'EarnDed Detail Level
Call SqlCursor(CSR_BenEmp_DBNav, LEVEL5)  'BenEmp Detail Level

'Allocate cursors not explicitly associated with
'any particular screen level
Call SqlCursor(CSR_PRSetup, NOLEVEL)
Call SqlCursor(CSR_Deduction, NOLEVEL)
Call SqlCursor(CSR_Benefit, NOLEVEL)
Call SqlCursor(CSR_Trns_Benefit, NOLEVEL)

Call SqlCursor(CSR_PRDoc_Del_Logic, NOLEVEL)
Call SqlCursor(CSR_PRTran_Del_Logic, NOLEVEL + SqlList)

Call ScreenInit

MemArray_EmpDeduction = DetailSetup(CSR_EarnDed_DBNav, F0225004.Spread_EarnDed, PNULL, bEarnDed, bDeduction, PNULL, PNULL)

MemArray_EmpBenefit = DetailSetup(CSR_BenEmp_DBNav, F0225005.Spread_BenEmp, PNULL, bBenEmp, bBenefit, PNULL, PNULL)

End Sub
**SetAutoNbrFlag Statement**

**Description**
Toggle the auto-numbering feature on and off.

**Syntax**
Call `SetAutoNbrFlag(Control, ActiveFlag)`

**Remarks**
This statement is used in cases where a data entry field may control the use of auto numbering.
The `SetAutoNbrFlag` statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>Control</td>
<td>Auto number control to activate/inactivate.</td>
</tr>
<tr>
<td>ActiveFlag</td>
<td>Integer</td>
<td>True to activate auto-numbering, False to inactivate.</td>
</tr>
</tbody>
</table>

**See Also**
AutoNbrDefault Function

**Example**

```vba
Sub cmodule_Chk (chkstrg As String, RetVal As Integer)
    If (chkstrg = "GL") Then
        'Turn on auto numbering because GL batch types are being entered.
        Call SetAutoNbrFlag(cBatNbrH, True)
    Else
        Call SetAutoNbrFlag(cBatNbrH, False)
    End If

End Sub
```

**SetButton Statement**

**Description**
Toggle the Enabled property of parent toolbar buttons.

**Syntax**
Call `SetButton(ToolBarButton, LevelNbr, EnabledPropVal)`

**Remarks**
Usage of the `SetButton` statement allows the application to turn toolbar buttons on and off.
Calls to `SetButton` cannot be made prior to the `ApplInit` call in the Form1_Load event.
The `SetButton` statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ToolBarButton</code></td>
<td>Integer</td>
<td>Toolbar button(s) to be toggled on / off. The following symbolic constants are defined in Solomon.VBTools.vb for usage in this argument: InsertButton, SaveButton, DeleteButton, CancelButton, NextButton, PreviousButton, FirstButton, LastButton, CurySelButton, CuryTogButton. Multiple buttons can be referenced by adding two or more of these symbolic constants together.</td>
</tr>
<tr>
<td><code>LevelNbr</code></td>
<td>Integer</td>
<td>Level number to which the toolbar button modification applies. For example, the InsertButton can be enabled for LEVEL0 but disabled for LEVEL1. All levels can be referenced using the AllLevels symbolic constant defined in Solomon.VBTools.vb.</td>
</tr>
<tr>
<td><code>EnabledPropVal</code></td>
<td>Integer</td>
<td>TRUE if the designated button(s) should be enabled. FALSE to disable the relevant button(s).</td>
</tr>
</tbody>
</table>

**See Also**

**DisplayMode Statement**

**Example**

The Release Payroll Batches process screen contains an `DSLGrid` control which automatically displays all Payroll batches that are ready to be released. The `DSLGrid` control only exists to facilitate the user’s need to specify which batches should be released. Consequently, for example, the Insert and Save buttons on the toolbar do not need to be enabled since the user is not allowed insert new batches nor update existing batches.

The following code is used to disable toolbar buttons that the user should not be allowed to press while the Release Payroll Batches application has focus.

```
'Disable buttons so user will not be able to take any action on batches in the grid
   Call SetButton(DeleteButton + InsertButton + SaveButton + CancelButton, 
                  _
                  AllLevels, False)
' Re-enable the control used to Select/Deselect batches.
   Call MSetProp(cCurrBatSelected, PROP_ENABLED, True)
```

**SetDefaults Statement**

**Description**

Set one or more controls to their default value using either their Default Property or Default Event code.

**Syntax**

```
Call SetDefaults(Form, FirstControl, LastControl)
```

**Remarks**

Each Microsoft Dynamics SL SDK data control has both a `Default` property as well as a `Default` event. Any particular data control can use one of these two methods to define a default data value for itself. The system uses these methods any time a particular control is to be initialized to its default value. An exhaustive discussion of all the times when this occurs is beyond the scope of the `SetDefaults` statement. However, one such time a control is set to its default value is when the application
explicitly directs the system to do so via usage of the **SetDefaults** statement in reference to the relevant control.

The **SetDefaults** statement can be used to default a range of controls based on their TabIndex property order. All controls whose TabIndex falls between *FirstControl* and *LastControl* will be defaulted. The **Level_SetDefaults** statement is functionally equivalent except it can be used to explicitly default all controls having a particular level number in their Level property.

Since **SetDefaults** implies a change in the data value of the designated controls, the system will “mark” the controls as requiring error checking. The system does not, however, immediately perform the requisite error checking (it does not immediately fire the Chk event). The error checking is nevertheless guaranteed to occur prior to any updates to the database.

**Note:** When an application needs to null out a particular field, perhaps because the field is no longer applicable, it should explicitly do so programmatically and then redisplay the relevant control using the **DispField** statement. After the control has been redisplayed it can then be disabled using the **SetProps** statement. The **SetDefaults** statement should not be used in these cases even if the relevant control has no Default Property value and no code within Default Event. A developer may well wonder why this caution would be expressed since the field will in fact be nulled out when no Default Property or Event code exists and therefore the application will appear to work properly during testing. However, developers must always be conscious of the fact that the end user may subsequently apply a customization which among other things defines some default value other than null for the relevant control. Such a customization would reveal a subtle flaw in the underlying application with regards to its usage of the **SetDefaults** statement to null out a particular data field.

The following code conceptually illustrates how to properly null out and disable a control which is no longer applicable:

```plaintext
Record.Field = NULL (0 for numeric datatypes, "" for string datatype)
Call DispField(cField)
Call SetProps(Form1, cField, cField, Enabled, False)
```

The **SetDefaults** statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form</td>
<td>Control</td>
<td>Form containing all controls between <em>FirstControl</em> and <em>LastControl</em> inclusive. PNULL can be passed to include all loaded forms.</td>
</tr>
<tr>
<td>FirstControl</td>
<td>Control</td>
<td>First control whose data value is to be defaulted. PNULL can be passed to include all controls on the designated Form.</td>
</tr>
<tr>
<td>LastControl</td>
<td>Control</td>
<td>Last control whose data value is to be defaulted. PNULL can be passed to include all controls on the designated Form.</td>
</tr>
</tbody>
</table>

**See Also**

Default Event, Default Property, **DispField** Statements, **Level_SetDefaults** Statement, **SetProps** Statement

**SetKeysEnabledOnly** Statement

**Description**

Enable/disable non-keyfield Microsoft Dynamics SL SDK custom controls.

**Syntax**

Call **SetKeysEnabledOnly**(Form, FirstControl, LastControl, Action)

**Remarks**

The **SetKeysEnabledOnly** statement uses the following arguments:
<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form</td>
<td>Control</td>
<td>Form containing all controls between FirstControl and LastControl inclusive. PNULL can be passed to include all loaded forms.</td>
</tr>
<tr>
<td>FirstControl</td>
<td>Control</td>
<td>First control to be enabled/disabled. PNULL can be passed to include all controls on the designated Form.</td>
</tr>
<tr>
<td>LastControl</td>
<td>Control</td>
<td>Last control to be enabled/disabled. PNULL can be passed to include all controls on the designated Form.</td>
</tr>
<tr>
<td>Action</td>
<td>Integer</td>
<td>True if non-keyfields within the designated control range should be disabled. False if non-keyfields within the designated control range should be restored to their original enabled state.</td>
</tr>
</tbody>
</table>

**See Also**

DisplayMode Statement, SetProps Statement

**SetLevelChg Statement**

**Description**

Set the update status of a specific level.

**Syntax**

Call `SetLevelChg(LevelNbr, Status)`

**Remarks**

Each update level, as defined by the Levels property of the DSLUpdate control, has a corresponding level status flag that is automatically maintained by the system. The purpose of the level status flag is to facilitate the optimization of database updates performed in response to Parent toolbar buttons. In general, these flags allow the system to only perform database updates for update levels which have in fact changed. If no information has changed then no information needs to be saved.

As previously mentioned, these update flags are automatically maintained by the system. When an existing record is loaded the flag is set to NOTCHANGED. If any non-keyfield is subsequently modified then the level status flag for the corresponding level is set to UPDATED. When a new record is being entered, the level status flag is set to INSERTED.

The `SetLevelChg` statement allows the application to override the current value a the status flag for a particular level. This can be useful if a data value is modified programmatically and therefore the system needs to be notified that something has changed so the corresponding information will actually be saved when the user presses the Save toolbar button.

The `SetLevelChg` statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LevelNbr</td>
<td>Integer</td>
<td>Level whose status flag is to be explicitly set.</td>
</tr>
<tr>
<td>Status</td>
<td>Integer</td>
<td>Level status flag. The following valid values are defined as symbolic constants in Solomon.VBTools.vb: INSERTED, UPDATED, NOTCHANGED.</td>
</tr>
</tbody>
</table>

**See Also**

TestLevelChg Function

**Example**

The Payroll Earnings Type Maintenance screen contains a button to automatically populate the grid with all Deductions. This amounts to inserting records into the grid (for example, into its underlying memory array) under program control. Since the data is not entered by the user, the system needs to
be notified that information at the grid level (for example, LEVEL1 in this case) has been programmatically updated and therefore needs to be saved. However, such notification only needs to occur if the system is not already aware that data has changed.

'If any records were inserted into the memory array then we need to make 'sure that the level status for the detail level is something other than 'NOTCHANGED so the system will know that something needs to be saved.

        If (AnyRecsInserted = True) Then

            If (TestLevelChg(LEVEL1) = NOTCHANGED) Then
                Call SetLevelChg(LEVEL1, UPDATED)
            End If
        End If

SetProps Statement

Description
Set the value of a particular property for the designated control(s).

Syntax
Call SetProps(Form, FirstControl, LastControl, PropertyName, NewPropValue)

Remarks
To set new property values for controls, the SetProps statement should be used rather than modifying the properties directly in Visual Basic code. Usage of SetProps allows the system to track changes to property values so as to avoid conflicts with customizations and / or other API's such as the DisplayMode statement.

For example, assume FieldA is disabled directly by the application using the FieldA.Enabled = False methodology. Since this property modification is made directly using Visual Basic code as opposed to the SetProps statement, the system has no knowledge of the fact that FieldA is now disabled. This can cause conflicts with the Customization Manager since some types of customizations do not apply to controls that the underlying application has disabled. The same is true for other properties as well.

Furthermore by setting properties directly, the effectiveness of statements such as DisplayMode is eliminated. Once DisplayMode has been called, all calls to SetProps referencing the Enabled property, are essentially ignored. The advantage of this type of architecture is that it reduces code complexity. In particular, the application can issue calls to the SetProps statement without surrounding each and every call with a conditional statement (such as “If DisplayMode = False”).
The `SetProps` statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Form</code></td>
<td>Control</td>
<td>Form containing all controls between <code>FirstControl</code> and <code>LastControl</code> inclusive. PNULL can be passed to include all loaded forms.</td>
</tr>
<tr>
<td><code>FirstControl</code></td>
<td>Control</td>
<td>First control whose designated property value is to be modified. PNULL can be passed to include all controls on the designated <code>Form</code>.</td>
</tr>
<tr>
<td><code>LastControl</code></td>
<td>Control</td>
<td>Last control whose designated property value is to be modified. PNULL can be passed to include all controls on the designated <code>Form</code>.</td>
</tr>
<tr>
<td><code>PropertyName</code></td>
<td>String</td>
<td>Name of the property whose value is to be modified. <strong>Note:</strong> All of the designated controls must have this property.</td>
</tr>
<tr>
<td><code>NewPropValue</code></td>
<td>String or Integer</td>
<td>New property value. The actual datatype varies based on the property being referenced.</td>
</tr>
</tbody>
</table>

The following valid values for the `PropertyName` argument are defined as symbolic constants:

<table>
<thead>
<tr>
<th>Symbolic Constant</th>
<th>Valid Datatype</th>
<th>Valid Data Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROP_BLANKERR</td>
<td>Integer</td>
<td>TRUE / FALSE</td>
</tr>
<tr>
<td>PROP_CAPTION</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>PROP_CUSTLIST</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>PROP_ENABLED</td>
<td>Integer</td>
<td>TRUE / FALSE</td>
</tr>
<tr>
<td>PROP_HEADING</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>PROP_MASK</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>PROP_MIN</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>PROP_MAX</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>PROP_TABSTOP</td>
<td>Integer</td>
<td>TRUE / FALSE</td>
</tr>
<tr>
<td>PROP_VISIBLE</td>
<td>Integer</td>
<td>TRUE / FALSE</td>
</tr>
</tbody>
</table>

**See Also**

BlankErr Property, DisplayMode Statement, DisplayModeSetProps Statement, Enabled Property, Heading Property, Mask Property, Min Property, Max Property, MSetProp Statement, TabStop Property, Visible Property

**Example**

The Payroll Deduction Maintenance screen allows deductions to be based on one of several different values such as earnings or even another deduction. If a DeductionA is based on another DeductionB then DeductionB must be entered by the user in the Base Deduction ID field. However, in all other cases the Base Deduction ID field is not applicable and therefore should be disabled.

The following code is used to modify the BlankErr and Enabled properties of the Base Deduction ID control.

```vbscript
If (bDeduction.BaseType = "D") Then
    'Current deduction IS based on another deduction.
    Call SetProps(Form1, Form1.cBaseid, Form1.cBaseid, "Enabled", True)
    Call SetProps(Form1, Form1.cBaseid, Form1.cBaseid, "BlankErr", True)
Else
    'Current deduction is NOT based on another deduction.
```
Call `SetProps(Form1, Form1.cBaseid, Form1.cBaseid, PROP_ENABLED, False)`
Call `SetProps(Form1, Form1.cBaseid, Form1.cBaseid, PROP_BLANKERR, False)`
End If

SetRestart Statement

Description
Identify the single cursor used to loop through multiple sets of high-level data items, within the context of a process, each one of which is autonomous and therefore is processed in a separate transaction.

Syntax
Call `SetRestart(Cursor)`

Remarks
In the context of Microsoft Dynamics SL, a process is an application that performs a logical set of operations on one or more sets of data in order to transform each data set from one business stage to another without user interaction. Thus at a conceptual level, a process transforms one set of data and then begins the same transformation on another set of data which is completely independent of the first set of data.

For example, consider the General Ledger Posting process. This process loops through Batch records, each of which is completely independent of all other Batch records, and posts the General Ledger transactions associated with each individual Batch.

Since a process operates on data it must therefore also be able gracefully manage faulty data. In order for a process to be fault tolerant it will need not only the ability to write error messages to a log but it will also have to abort database updates which occurred prior to the error in order to preserve the logical integrity of the database. After an error has been properly managed, the process will need to either move on to the next set of data to be processed or as a last resort the process itself may need to be terminated.

When errors of various sorts occur, such as when a particular referential record cannot be found or an out of balance condition is detected, the `Status` statement can be used to report the precise nature of the fatal error to the Event Log. Fatal errors that occur on the restart cursor itself, such as an I/O error or a corrupt stored procedure will cause the application to terminate since by definition it cannot even process the next record in its outermost processing loop. Fortunately, fatal errors on the restart cursor itself are extremely rare. All other fatal cause the system to automatically go into “abort mode”. Abort mode can simply be defined as a mode in which the system is essentially ignoring all database calls since the current transaction is going to be aborted anyway (as opposed to successfully committed). For example, if the current Batch cannot be posted for some reason then report the problem and move on to the next Batch. Abort mode is basically a method whereby the system makes it easier for the application to move on to the next autonomous set of data (Batch in this case). In these cases, code complexity is reduced since the application does not have to surround each database call with conditional statements which are conceptually equivalent to “If (EverythingOK = True) Then ...”. As previously stated this is due to the fact that database calls are essentially ignored as long as the application is in abort mode. Nevertheless this leads directly to the question relating to the precise point at which the application comes back out of abort mode.

If a restart cursor is active, the application should come out of abort mode when processing begins for the next logically independent set of data. In the previously cited example this would translate to the point at which the next Batch is fetched from the database. The `SetRestart` statement is designed to identify the single cursor that the application uses to fetch the successive sets of logically independent data (the Batch cursor in this example).

The restart cursor automatically loses its “restart” connotation when it reaches the end of its result set. This occurs when a NOTFOUND is returned to the application in response to a fetch operation on the restart cursor.
If a restart cursor is not active, the application should come out of abort mode as soon as it calls `TranEnd` for the faulty transaction.

The `SetRestart` statement uses the following argument:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cursor</td>
<td>Integer</td>
<td>The single SQL database cursor that the application uses to fetch the successive sets of logically independent data for processing.</td>
</tr>
</tbody>
</table>

**See Also**

Status Statement, TranBeg Statement, TranEnd Statement

**Example**

The following code snippet from the General Ledger Posting process illustrates how a restart cursor on the Batch cursor is implemented using the `SetRestart` statement.

```plaintext
Call SetRestart(Post_CSR_Batch)
Call TranBeg(True)

BatchFetch = SqlFetch1(Post_CSR_Batch, SqlStr, bBatch)
While (BatchFetch = 0)
    Call Status(SaveGoodArgVals, False, Trim$(PostControlInfo.
    Module_Caption) + $ bBatch.Module) + NL +
    Trim$(PostControlInfo.Batch_Caption) +
    SParm(bBatch.BatNbr), DISP_ONLY)
    'Initialize all of the BatchFiscal* structures. This call must be
    'before the call to IsBatchOKToPost().
    Call InitAllBatchFiscals

    OkToPost = IsBatchOKToPost(MsgNbr)
    If (OkToPost = False) Then
        Call Status(MsgNbr, True, "", LOG_AND_DISP)
    End If

    'If posting to any prior fiscal year/period then issue a
    '-warning (i.e. 'not fatal'). Make sure that this test is after the
    'check for whether or not it is even OK to post at all.
    If (BatchFiscal.Year < CurrFiscal.Year) Then
        Call Status(MSG_WARN_POSTING_TO_PAST_FISC_YR, False, "",
        LOG_AND_DISP)
    ElseIf (BatchFiscal.Per < CurrFiscal.Per) Then
        Call Status(MSG_WARN_POSTING_TO_PAST_FISC_PER, False, "",
        LOG_AND_DISP)
```

```plaintext
```

```plaintext
```
End If

'Initialize totals to zero prior to PostAllTrans() since that
'procedure uses incremental totaling.
  TranCrTotal = 0#
  TranDrTotal = 0#
  CuryTranCrTotal = 0#
  CuryTranDrTotal = 0#

'Post all GLTrans associated with current batch
  Call PostAllTrans(TranCrTotal, TranDrTotal, CuryTranCrTotal, CuryTranDrTotal)

'Determine whether or not the batch is out of balance with its
'transactions
  If (bBatch.BatType <> "J") Then
    If bBatch.CuryID = bGLSetup.BaseCuryID Then
      If ((bBatch.CrTot <> TranCrTotal) Or (bBatch.DrTot <> TranDrTotal)) Then
        Call Status(MSG_BATCH_OUT_OF_BAL_NOT_POSTED, True, "", LOG_AND_DISP)
      End If
    Else
      If ((bBatch.CuryCrTot <> CuryTranCrTotal) Or (bBatch.CuryDrTot <> CuryTranDrTotal)) Then
        Call Status(MSG_BATCH_OUT_OF_BAL_NOT_POSTED, True, "", LOG_AND_DISP)
      End If
    End If
  Else
    'Adjustment batches are allowed to post without regard
    'to whether or not they are in balance.
  End If 'bBatch.BatType

'Update batch record
  bBatch.Status = "P"
  Call SUpdate1(Post_CSR_Batch, "Batch", bBatch)

Call TranEnd
Call TranBeg(True)
*Get next batch record on the RESTART cursor
  BatchFetch = SFetch1(Post_CSR_Batch, bBatch)

Wend
Call TranEnd
.
.
.

SetStatusBarText

Description
Sets the text to appear in the text pane of the status bar and, optionally, the tooltip text to be displayed for the status bar text. If no tooltip text is specified, the tooltip displays the same text as the status bar text.

Syntax
Call SetStatusBarText(Text, Tooltip)

Remarks
The SetStatusBarText statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>String</td>
<td>The text to be displayed in the text pane of the status bar, up to 48 characters.</td>
</tr>
<tr>
<td>Tooltip</td>
<td>String</td>
<td>The tooltip to be displayed for the status bar text. If an empty string is specified, the string for the status bar text is used.</td>
</tr>
</tbody>
</table>

Example
This example displays in the status bar the quantity available of an inventory item. Notice for the text of the message, the code uses a message from messages.csv so it can easily be translated to any language.

QtyAvail = gADGPlan.GetQtyAvailToday(pInvtID, pSiteID)

'Inventory Item %s has a quantity of %s available
msg = GetMessageText(15376, Trim$(pInvtID), Format$(QtyAvail))

Call SetStatusBar(msg, "")
SetSWIMPrintInfo Function

Description
Replace the current print information and settings with those in the application’s PInfo structure.

Syntax
result = SetSWIMPrintInfo( printInfo )

Remarks
The SetSWIMPrintInfo function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result</td>
<td>Boolean</td>
<td>Return value indicating success or failure of function call</td>
</tr>
<tr>
<td>printInfo</td>
<td>PInfo</td>
<td>PInfo structure containing print information and settings</td>
</tr>
</tbody>
</table>

See Also
GetSWIMDefaultPrintInfo Function, GetSWIMPrintInfo Function

SetTI_Alias_Level Statement

Description
Sets a different level for a control for use during processing by transaction import.

Syntax
Call SetTI_Alias_Level(Control, Level)

Remarks
This function is used to specify a different level for a control when importing data with transaction import. It is used when it is necessary for transaction import to process the control at a different level than the level specified in the control’s Level property.

The SetTI_Alias_Level statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>Control</td>
<td>Control whose level you want to change for purposes of importing data using Transaction Import.</td>
</tr>
<tr>
<td>Level</td>
<td>Integer</td>
<td>Level that Transaction Import should use for this control</td>
</tr>
</tbody>
</table>
SFetch Functions

Description
Used to retrieve a composite record from the database based on some pre-defined SQL statement or stored procedure.

Syntax

\[
\text{RetVal} = \text{SFetch1}(\text{Cursor}, \text{bTable1})
\]
\[
\text{RetVal} = \text{SFetch4}(\text{Cursor}, \text{bTable1}, \text{bTable2}, \text{bTable3}, \text{bTable4})
\]
\[
\text{RetVal} = \text{SFetch8}(\text{Cursor}, \text{bTable1}, \text{bTable2}, \text{bTable3}, \text{bTable4}, \text{bTable5}, \text{bTable6}, \text{bTable7}, \text{bTable8})
\]

Remarks
In order to fetch information from the server it must first know what tables, records and fields are being queried from a particular cursor. Consequently the cursor must first be initialized with either an SQL statement or stored procedure via the use of the Sql statement or the SqlFetch1, SqlFetch4 or SqlFetch8 functions. Once the database view has been established these functions will retrieve the next sequential record in the view consistent with the Order By clause of the SQL statement used to initialize the view. After the last record in the view has been returned all subsequent calls to SFetch1, SFetch4 and SFetch8 will return NOTFOUND.

SFetch1 is designed for SQL statements returning data from a single table. For more advanced SQL statements having one or more table joins either SFetch4 or SFetch8 can be used.

The SFetch1 function uses the following arguments (SFetch4 and SFetch8 respectively have four and eight SolomonDataObjects. PNULL should be passed for unused SolomonDataObject parameters).

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RetVal</td>
<td>Integer</td>
<td>0 if a record is successfully fetched. NOTFOUND is returned if no additional records exist in the current view.</td>
</tr>
<tr>
<td>Cursor</td>
<td>Integer</td>
<td>SQL database cursor.</td>
</tr>
<tr>
<td>bTable1</td>
<td>SolomonDataObject</td>
<td>SolomonDataObject corresponding to the primary table in the SQL statement.</td>
</tr>
</tbody>
</table>

Note: SGroupFetch1, SGroupFetch4 or SGroupFetch8 must be used if the SQL statement used to initialize the cursor contained one or more of the following components:

- Group aggregate functions (such as Count and Sum)
- DISTINCT keyword
- GROUP BY clause
- HAVING clause
- Subqueries

See Also
Sql Statement, SqlFetch Functions, SGroupFetch Functions
Example
The following code reads through all records in the Account table. Since the Account_All stored procedure only selects data from a single table (for example, the Account table) \texttt{SFetch1} would be adequate. However in this example, \texttt{SFetch4} is actually used to illustrate how to pass PNULL,0 for unused table structure arguments.

```vbs
Dim CSR_Account As Integer
Dim SqlStr   As String
Dim AcctFetch As Integer

'Allocate cursor
Call SqlCursor( CSR_Account, NOLEVEL)

'Initialize cursor with SQL stored procedure and fetch first record
SqlStr = "Account_All" + sparm(SQLWILDSTRING)
AcctFetch = SqlFetch4(CSR_Account, SqlStr, bAccount, PNULL, PNULL, PNULL)

'Read through all Account records
While (AcctFetch = 0)
    AcctFetch = SFetch4(CSR_Account, bAccount, PNULL, PNULL, PNULL, 0)
Wend
```

\textbf{SGroupFetch Functions}

\textbf{Description}
Used to retrieve a composite record from the database based on some pre-defined SQL statement or stored procedure containing one or more group aggregate functions and/or clauses.

\textbf{Syntax}

\begin{align*}
\text{RetVal} &= \textit{SGroupFetch1}(\text{Cursor, bTable1}) \\
\text{RetVal} &= \textit{SGroupFetch4}(\text{Cursor, bTable1, bTable2, bTable3, bTable4}) \\
\text{RetVal} &= \textit{SGroupFetch8}(\text{Cursor, bTable1, bTable2, bTable3, bTable4, bTable5, bTable6, bTable7, bTable8})
\end{align*}

\textbf{Remarks}
In order to fetch information from the server it must first know what tables, records and fields are being queried from a particular cursor. Consequently the cursor must first be initialized with either an SQL statement or stored procedure via the use of the \texttt{Sql} statement. \texttt{SGroupFetch1}, \texttt{SGroupFetch4} or \texttt{SGroupFetch8} are only designed for cases where the SQL statement used to initialize the cursor contains one or more of the following:

\begin{itemize}
    \item Group aggregate functions (such as Count and Sum)
    \item DISTINCT keyword
    \item GROUP BY clause
    \item HAVING clause
\end{itemize}
Subqueries
The logically equivalent SFetch1, SFetch4 and SFetch8 functions should be used if the SQL statement does not contain any of the above referenced items.

Once the database view has been established these functions will retrieve the next sequential record or data value in the view consistent with the Order By or Group By clause of the SQL statement used to initialize the view. After the last record or data value in the view has been returned all subsequent calls to SGroupFetch1, SGroupFetch4 and SGroupFetch8 will return NOTFOUND.

SGroupFetch1 is designed for SQL statements returning data from a single table. For more advanced SQL statements having one or more table joins either SGroupFetch4 or SGroupFetch8 can be used.

The SGroupFetch1 function uses the following arguments (SGroupFetch4 and SGroupFetch8 respectively have four and eight SolomonDataObjects. PNULL should be passed for unused SolomonDateObject parameters):

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RetVal</td>
<td>Integer</td>
<td>0 if a record or data value is successfully fetched. NOTFOUND is returned if no additional records exist in the current view.</td>
</tr>
<tr>
<td>Cursor</td>
<td>Integer</td>
<td>SQL database cursor.</td>
</tr>
<tr>
<td>bTable1</td>
<td>SolomonDataObject</td>
<td>SolomonDataObject corresponding to the primary table or data value in the SQL statement.</td>
</tr>
</tbody>
</table>

Note: The type and size of the data returned can vary when the SQL statement contains one or more group aggregates. For the COUNT group aggregate, the data will always be returned as a 4-byte integer (for example, Long Visual Basic datatype). The MIN and MAX group aggregates always return the same data type and length as the field on which the aggregate is based. The SUM and AVG group aggregates always return 8-byte floating point values (for example, Double Visual Basic datatype).

See Also
Sql Statement, SFetch Functions

Example
This example calculates the sum of the POReceipt.RcptQtyTot in batch 000001 using the SUM group aggregate and stores the result in a local variable called RetQtyTotal. Note that the SUM is the data that is actually fetched and therefore it is stored in RetQtyTotal rather than FetchRetVal.

```vba
Dim FetchRetVal as Integer
Dim RetQtyTotal As Double

'Initialize the cursor with a SQL statement containing the SUM group aggregate
Call sql(CSR_POReceipt, "Select SUM(RcptQtyTot) From POReceipt Where POReceipt.BatNbr = '000001' Order By POReceipt.RcptNbr")

'Actually retrieve the SUM from the database server
FetchRetVal = SGroupFetch1(CSR_POReceipt, RetQtyTotal)
```
SInsert Statements

Description
Insert one record into each specified table within an existing database view.

Syntax
Call **SInsert1** (Cursor, TablesInsertingInto, bTable1)
Call **SInsert4** (Cursor, TablesInsertingInto, bTable1, bTable2, bTable3, bTable4)
Call **SInsert8** (Cursor, TablesInsertingInto, bTable1, bTable2, bTable3, bTable4, bTable5, bTable6, bTable7, bTable8)

Remarks
New records can be programmatically inserted directly into an existing database table via the use of the **SInsert1**, **SInsert4** and **SInsert8** statements. In order to insert information into the database, the server must first know what tables are being referenced by a particular cursor. Consequently the cursor must first be initialized with either an SQL statement or stored procedure via the use of the **Sql** statement or the **SqlFetch1**, **SqlFetch4** or **SqlFetch8** functions. Once the database view has been established these functions will insert one new record into each table referenced in the **TablesInsertingInto** argument using data from corresponding table structure arguments.

**SInsert1** is designed for SQL statements referencing a single table. In this case the **TablesInsertingInto** is always the name of the single table actually referenced. For more advanced SQL statements having one or more table joins either **SInsert4** or **SInsert8** can be used. The referencing of more than one table does not automatically force the insertion of a record into every table in the view anytime **SInsert4** or **SInsert8** is used on the corresponding cursor. A single record will only be inserted into each table explicitly specified in the **TablesInsertingInto** argument so long as each table name so specified is also referenced in the SQL statement which was used to initialize the current view. Thus, for example, if TableA and TableB are the only two tables referenced in the SQL statement used to initialize the current view, then a value of TableXYZ would not be valid for the **TablesInsertingInto** argument.

The limit for each field or column is 800 characters.

The **SInsert1** function uses the following arguments (**SInsert4** and **SInsert8** respectively have four and eight **SolomonDataObjects**. **PNULL** should be passed for unused **SolomonDataObject** parameters.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cursor</td>
<td>Integer</td>
<td>SQL database cursor.</td>
</tr>
<tr>
<td>TablesInsertingInto</td>
<td>String</td>
<td>Name of each table, in the specified cursor’s view, into which a new record is to be inserted. Multiple table names should be separated by commas.</td>
</tr>
<tr>
<td>bTable1</td>
<td><strong>SolomonDataObject</strong></td>
<td><strong>SolomonDataObject</strong> corresponding to the primary table in the SQL statement. Data in this class object will be inserted into its corresponding database table if the name of the said database table is explicitly specified in the <strong>TablesInsertingInto</strong> argument.</td>
</tr>
</tbody>
</table>

See Also
**Sql Statement, SqlFetch Functions**

Example
This example illustrates the usage of **SInsert1** to insert a copy of each record in the Account table into a table called AccountCopy. The AccountCopy table does not exist in a standard application database. Nevertheless to facilitate this example it is assumed that a table named AccountCopy does in fact exist in the application database and that it also has the exact same field and index characteristics as the Account table.
Dim CSR_Account As Integer
Dim CSR_AccountCopy_Insert As Integer
Dim SqlStr As String
Dim AcctFetch As Integer

'Allocate database cursors
    Call SqlCursor(CSR_Account, NOLEVEL)
    Call SqlCursor(CSR_AccountCopy_Insert, NOLEVEL)

'Initialize cursor used to insert copies of Account records with an
'appropriate SQL statement so it is ready to receive SInsert1() calls.
    Call Sql(CSR_AccountCopy_Insert, "Select * from AccountCopy Order By Acct")

'Begin a database transaction since all updates to the database must
'occur within a transaction.
    Call TranBeg(True)

'Read through all Account records and insert a copy of each one into the
'AccountCopy table
    SqlStr = "Select * from Account Order by Acct"
    AcctFetch = SqlFetch1(CSR_Account, SqlStr, bAccount)
    While (AcctFetch = 0)
        Call SInsert1(CSR_AccountCopy_Insert, "AccountCopy", bAccount)
        AcctFetch = SFetch1(CSR_Account, bAccount)
    Wend

'End the database transaction to commit all newly inserted
'records to the database.
    Call TranEnd
SParm Function

Description
Convert a string into an SQL parameter string.

Syntax
SQLParmStr = SParm(StrToConvert)

Remarks
The SParm function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLParmStr</td>
<td>String</td>
<td>StrToConvert converted into an SQL parameter string.</td>
</tr>
<tr>
<td>StrToConvert</td>
<td>String</td>
<td>String value to convert.</td>
</tr>
</tbody>
</table>

See Also
DParm Function, FParm Function, IParm Function

Example
These examples assume the following SQL statement was used to create a stored procedure called Employee_EmpId

```
Select * from Employee where EmpId LIKE @parm1 order by EmpId
```

This code snippet illustrates how to use the Employee_EmpId stored procedure to fetch the employee record for employee #000581.

```
SqlStr = "Employee_EmpId" + SParm("000581")
Employee_Fetch = SqlFetch1(CSR_Employee, SqlStr, bEmployee)
```

This code snippet illustrates how to use the Employee_EmpId stored procedure to fetch all employee records by using a wildcard value for the EmpId parameter.

```
SqlStr = "Employee_EmpId" + SParm(SQLWILDSTRING)
Employee_Fetch = SqlFetch1(CSR_Employee, SqlStr, bEmployee)
While ( Employee_Fetch = 0)
    Employee_Fetch = SFetch1( CSR_Employee, bEmployee)
Wend
```

This code snippet illustrates how to use the Employee_EmpId stored procedure to fetch all employee records beginning and ending with zero by using a combination of string literals and single character wildcard values for the EmpId parameter.

```
SQLParmStr = "0" + SQLWILDCCHAR + SQLWILDCCHAR + SQLWILDCCHAR + SQLWILDCCHAR + "0"
SqlStr = "Employee_EmpId" + SParm(SQLParmStr)
Employee_Fetch = SqlFetch1(CSR_Employee, SqlStr, bEmployee)
While ( Employee_Fetch = 0)
    Employee_Fetch = SFetch1( CSR_Employee, bEmployee)
Wend
```
Sql Statement

Description
Initialize a new database view.

Syntax
Call Sql(Cursor, SqlStr)

Remarks
Takes the specified SQL text, compiles it, and then runs it. If fetch operations are required then one of the SFetch functions must be called. If the SQL statement is performing an Update Set, Delete From or Insert into operation, no subsequent fetch operations are required. If the SQL statement references one or more parameters, the SqlSubst and SqlExec functions must also be called. The Sql statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cursor</td>
<td>Integer</td>
<td>SQL database cursor.</td>
</tr>
<tr>
<td>SqlStr</td>
<td>String</td>
<td>SQL statement or stored procedure to be used in initializing a new database view. If a stored procedure is used then all parameters must be sequentially appended in order of occurrence within the original Create Procedure statement using calls to SqlSubst.</td>
</tr>
</tbody>
</table>

See Also
SFetch Functions, SqlSubst Statement, SqlExec Statement, SqlFetch Functions

Example
Dim SqlStr As String
' Set AP Documents which are marked as current to non-current
    SqlStr = "Update APDoc Set Current = 'False' where APDoc.DocBal = 0 .J
           and PerEnt <=" + sparm(NextPerNbr) + " and Current = 'True''"
    Call Sql(C_APClose, Trim$(SqlStr))
' Process posted AP batches
    SqlStrg = "Select * from Batch Where Module = 'AP' and Status = 'P' .J
                   Order By BatNbr"
    Call Sql(CSR_Batch, SqlStrg)
serr1 = SFetch1(c1, bBatch)
    While serr1 = 0
       ' Process the batch
           ....
       ' Get the next batch
           serr1 = SFetch1(CSR_Batch, bBatch)
    Wend
SqlCursor Statement

Description
Allocate a new database cursor.

Syntax
Call SqlCursor(Cursor, Flags)

Remarks
All read/write communication between an application and the database must occur through a
database cursor. A cursor is basically a database resource used to track low level information required
to implement SQL database read/write operations. For example, a cursor tracks the SQL statement
used to initialize the current view, what individual fields were selected, the current record within the
view as well as other more detailed information.

Each level within a screen must have a corresponding cursor allocated within the Form_Load event of
Form1 to facilitate database read/write activity at that level. Additionally, many of the SQL API calls
within the Microsoft Dynamics SL SDK, such as Sql, SFetch1, SqlFetch1 and SUupdate1 require a
database cursor as one of their arguments.

Each application can allocate a maximum of 36 cursors. Cursors no longer needed by the application
can optionally be freed via the use of the SqlFree statement. All cursors are automatically released by
the system when the application terminates (for example, when ScreenExit is called).

The SqlCursor statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cursor</td>
<td>Integer</td>
<td>Variable to be initialized with a resource handle for an SQL database cursor.</td>
</tr>
<tr>
<td>Flags</td>
<td>Integer</td>
<td>One or more special flags notating what and/or how the cursor will be used. At a minimum the Flags parameter must contain one of the LEVEL0 through LEVEL9 or NOLEVEL symbolic constants defined in Solomon.VBTools.vb. Cursors not explicitly associated with a particular level number should be allocated using the NOLEVEL flag.</td>
</tr>
</tbody>
</table>

The following symbolic constants can optionally be passed as Flags (by adding them to the required
LEVEL0 through LEVEL9 or NOLEVEL symbolic constants):

- SqlList — Read / Write operations performed on the cursor will automatically be buffered to
  improve performance. If an application updates even one record on a buffered cursor, it must
  update all records read with that cursor within the same database transaction. Failure to comply
  with this requirement will result in a sparse update error.

- SqlSystemDb — All database operations will be directed to the system database as opposed to the
  application database.

See Also
SFetch Functions, SInsert Statements, Sql Statement, SqlFetch Functions, SqlFree Statement,
SUupdate Statements

Example
The following example illustrates how to allocate multiple cursors for a moderately complex screen
containing multiple levels such as the Payroll Employee Maintenance screen. Notice that some of the
cursors are allocated using the NOLEVEL flag, indicating that these cursors are not explicitly reserved
for database read/write activity associated with any particular level on the screen.

Sub Form_Load ()
    Call loadform(F0225001)  
        'Timesheet Defaults
    Call loadform(F0225002)  
        'Miscellaneous Info
Call loadform(F0225003) 'Pay Information
Call loadform(F0225004) 'Employee Deductions
Call loadform(F0225005) 'Employee Benefits

Call ApplInit

Call SetAddr(LEVEL0, "bEmployee", bEmployee, nEmployee)
Call SetAddr(LEVEL1, "bWorkLoc", bWorkLoc, nWorkLoc)
Call SetAddr(LEVEL2, "bEarnType", bEarnType, nEarnType)
Call SetAddr(LEVEL3, "bPayGroup", bPayGroup, nPayGroup)
Call SetAddr(LEVEL4, "bEarnDed", bEarnDed, nEarnDed)
Call SetAddr(LEVEL5, "bBenEmp", bBenEmp, nBenEmp)
Call SetAddr(NOLEVEL, "bPRDoc", bPRDoc, nPRDoc)
Call SetAddr(NOLEVEL, "bPRSetup", bPRSetup, nPRSetup)
Call SetAddr(NOLEVEL, "bPRTran", bPRTran, nPRTran)
Call SetAddr(NOLEVEL, "bDeduction", bDeduction, nDeduction)
Call SetAddr(NOLEVEL, "bBenefit", bBenefit, nBenefit)

'Allocate one cursor for each level on the screen
Call SqlCursor(CSR_Employee, LEVEL0) 'Employee Navigation Level
Call SqlCursor(CSR_WorkLoc, LEVEL1) 'WorkLoc Lookup Level
Call SqlCursor(CSR_EarnType, LEVEL2) 'EarnType Lookup Level
Call SqlCursor(CSR_PayGroup, LEVEL3) 'PayGroup Lookup Level
Call SqlCursor(CSR_EarnDed_DBNav, LEVEL4) 'EarnDed Detail Level
Call SqlCursor(CSR_BenEmp_DBNav, LEVEL5) 'BenEmp Detail Level

'Allocate cursors not explicitly associated with any particular screen 'level
Call SqlCursor(CSR_PRSetup, NOLEVEL)
Call SqlCursor(CSR_Deduction, NOLEVEL)
Call SqlCursor(CSR_Benefit, NOLEVEL)
Call SqlCursor(CSR_Trns_Benefit, NOLEVEL)
Call SqlCursor(CSR_PRDoc_Del_Logic, NOLEVEL)
Call SqlCursor(CSR_PRTran_Del_Logic, NOLEVEL + SqlList)

Call ScreenInit

MemArray_EmpDeduction = DetailSetup(CSR_EarnDed_DBNav,.J
F0225004.Spread_EarnDed, PNULL, bEarnDed,bDeduction, PNULL, PNULL)

MemArray_EmpBenefit = DetailSetup(CSR_BenEmp_DBNav,.J
F0225005.Spread_BenEmp, PNULL, bBenEmp, bBenefit, PNULL, PNULL)

End Sub
SqlCursorEx

Description
Allocate a new database cursor.

Syntax
Call SqlCursorEx(Cursor, Flags, CursorName, ReferencedTableNames, UpdateTableNames)

Remarks
All read/write communication between an application and the database must occur through a
database cursor. A cursor is basically a database resource used to track low level information required
to implement SQL database read/write operations. For example, a cursor tracks the SQL statement
used to initialize the current view, what individual fields were selected, the current record within the
view as well as other more detailed information.

Each level within a screen must have a corresponding cursor allocated within the Form_Load event of
Form1 to facilitate database read/write activity at that level. Additionally, many of the SQL API calls
within the Microsoft Dynamics SL SDK, such as Sql, SFetch1, SqlFetch1 and SUpdate1 require a
database cursor as one of their arguments. If the cursor handle passed to one of these SQL API calls
has not been previously allocated then it will automatically be allocated during the call. However it is
important to be aware of the fact that all cursors not explicitly allocated, via an SqlCursorEx call, are
automatically allocated as read-only cursors.

Each application can allocate a maximum of 36 cursors. Cursors no longer needed by the application
can optionally be freed via the use of the SqlFree statement. All cursors are automatically released by
the system when the application terminates (for example, when ScreenExit is called).

The Microsoft Dynamics SL kernel supports the reuse of cursors. In the Microsoft SQL Server
environment, use SqlFree() to free a cursor and release the memory it used. In the Scalable SQL or
Pervasive.SQL environment, SqlFree() releases the cursor, but does not release the memory used by
the cursor until the database session is closed. This behavior can cause out-of-memory conditions on
some large screens. Therefore, use SqlFree() in the Scalable SQL or Pervasive.SQL environment with
care.

You should use SqlFree() on the same cursor for a different SQL statement because SqlFree(): makes
code more readable and logical; clears out the attributes associated with the old cursor; forces
attribute specification for the new cursor; releases any locks associated with a cursor; and releases all
memory allocated for the cursor.

If you do not use an SqlFree() call between Sql() calls using the same cursor, and each Sql() call issues
a different SQL statement, the attributes and table names specified in the SqlCursorEx() call apply to all
SQL statements issued with this cursor.

Every reuse of a cursor must be in the same database for which the SqlCursorEx() call was issued. If
SqlCursorEx() specifies the SqlSystemDb flag, then all database calls made with this cursor are run
against the system database.

When you specify ReferencedTableNames and UpdateTableNames (the fourth and fifth parameters) in
the SqlCursorEx() call, these table names will apply to every SQL statement that is used with the
cursor. To specify an SQL statement that does not include the same table names as those specified in
the original SqlCursorEx() call, you must call SqlFree() to free the cursor and then issue a new
SqlCursorEx() statement.
The **SqlCursorEx** statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cursor</strong></td>
<td>Integer</td>
<td>Variable to be initialized with a resource handle for an SQL database cursor.</td>
</tr>
<tr>
<td><strong>Flags</strong></td>
<td>Integer</td>
<td>One or more optimization flags notating what and/or how the cursor will be used. At a minimum the Flags parameter must contain one of the LEVEL0 through LEVEL9 or NOLEVEL symbolic constants defined in Solomon.VBTools.vb. Cursors not explicitly associated with a particular level number should be allocated using the NOLEVEL flag.</td>
</tr>
<tr>
<td><strong>CursorName</strong></td>
<td>String</td>
<td>Alias name associated with the cursor. This value is used solely to enhance the readability of diagnostic messages.</td>
</tr>
<tr>
<td><strong>ReferencedTableNames</strong></td>
<td>String</td>
<td>Comma delimited list of table names that will be referenced by the cursor. This list should be thought of as being applicable so long as the cursor remains allocated.</td>
</tr>
<tr>
<td><strong>UpdateTableNames</strong></td>
<td>String</td>
<td>Comma delimited list of all tables which may be updated by the cursor. The principle usage of this list is to facilitate performance optimizations related to tables in the cursor's view that will never be updated. If no table names are specified then by default it will be assumed that all of the referenced tables will be updated at some point. Any table name appearing in UpdateTableNames must also be specified in ReferencedTableNames. This list of table names should not be confused with the table names passed to any of the SInsert, SUpdate or SDelete statements. The names passed to those statements identify the actual tables updated in a particular database operation — which may be a subset of the UpdateTableNames argument.</td>
</tr>
</tbody>
</table>

**Note:** The following optimization flags are implemented as symbolic constants in Solomon.VBTools.vb. They can optionally be passed via the Flags argument by adding them to the required LEVEL0 through LEVEL9 or NOLEVEL symbolic constant:

- **SqlReadOnly** — This flag indicates that the cursor is used exclusively for read operations, not for record inserts, updates, or deletions. An error is generated if you attempt an Supdate(), Sdelete(), or Sinsert() call on a cursor that contains this flag. If a cursor is not explicitly declared by the application using either SqlCursor or SqlCursorEx(), the Microsoft Dynamics SL kernel declares and allocates the cursor automatically. The cursor automatically declared on Sql() or SqlFetch() calls will specify this flag if the operation is contained within a transaction (that is, a TranBeg() statement has been issued). Rows fetched with this cursor type are not exclusively locked, regardless whether they are fetched within a transaction. The “shared” database connection is used to run the operation associated with this cursor. If this is the only cursor flag specified (other than the level identifier), the cursor API will use buffering to fetch multiple records with one database call.
  
  **API Type:** Advanced Cursor
  
  **Example Uses:** A process that needs to accumulate the sum of a floating point field and store the result in a different record; if the summing needs to take place within a transaction, this flag is ideal.

- **SqlFastReadOnly** — Like the SqlReadOnly flag, this flag indicates that the cursor is used exclusively for read operations. All database operations occurring on this type of cursor are serviced directly from Microsoft SQL Server’s faster low-level API. Cursors with this flag should not be used when accessing a table that may have insert, update, or delete operations performed on it from a different cursor in the same database transaction. The operation is run using a connection that is explicitly used for read-only operations. Therefore, records fetched with this cursor type are not exclusively locked, regardless whether they are fetched within a transaction.
  
  **API Type:** Low-level
Example Uses: The application needs to load a display-only grid with values from many General Ledger Transaction (GLTran) records.

- SqlList — This flag indicates that read/write operations performed on the cursor are automatically buffered to improve performance. If an application updates even one row on a buffered cursor, it must update all rows read using that cursor within the same database transaction; otherwise, a sparse update error occurs.

  With buffering, as many records (or rows) are returned from the server as will fit in 30K of memory space. Buffering is not available for the low-level API. The kernel uses buffering as the default for all fetches that use the Advanced Cursor API, thus greatly improving performance.

  In the Microsoft SQL Server environment, all database operations are buffered whenever possible, regardless whether this flag has been specified.

  API Type: Advanced Cursor

  Example Uses: There is no use for this flag in the Microsoft SQL Server environment.

- SqlNoList — This flag suppresses buffering of read/write operations. All database operations are buffered by default, whenever possible, regardless whether the SqlList flag has been specified. Consequently, on the Microsoft SQL Server platform, buffering must be explicitly suppressed if for some reason the application does not want a particular cursor to be buffered. In some cases, this default buffering is automatically suppressed due to low-level restrictions.

  API Type: Has no effect.

  Example Uses: If you know there will never be more than one record fetched (as with Setup records, for example), this flag would be used to tell the kernel not to perform buffering calculations.

- SqlNoSelect — This flag indicates that the SQL statement to be performed on the cursor can be any valid Microsoft SQL Server statement except a Select statement. No Sfetch(), Sinsert(), Supdate(), or Sdelete() calls can be performed on this cursor. The most common use for this type of cursor is to process Update and Delete Microsoft SQL Server statements. The kernel releases the dbproc after running the SQL Server statement, to allow the kernel to use the “shared” database connection to perform this operation.

  All stored procedures that do not contain a Select statement should use a cursor with this flag specified. Microsoft Dynamics SL Release and Posting processes use this cursor flag.

  API Type: Low-level

  Example Uses: The application needs to issue a large stored procedure that contains multiple update and select statements, as well as conditional logic or other T-SQL logic.

- SqlSingleRow — This flag is used with cursors that never process more than one composite record after each Select statement. It is designed primarily to facilitate optimization on the Microsoft SQL Server platform. All database operations occurring on an SqlSingleRow cursor are serviced directly from Microsoft SQL Server’s faster low-level API.

  SFetch calls on cursors of this type should not be separated from the associated SQL call by any other database operations. The simplest method to satisfy this requirement is to use the SqlFetch function. An application using this flag must use the cursor immediately for single-row lookup or insertion so that the kernel can release the dbproc.

  When an application inserts a record instead of fetching one, the Sinsert() call must immediately follow the Sql() call. Performing SqlFetch() followed by Sinsert() is not supported. Instead, issue the Sql() call followed by the Sinsert() call. Because of this requirement, the kernel can use the “shared” database connection to perform this operation.

  If an application attempts to specify this flag on a non-lookup level cursor, an error is generated.

  API Type: Low-level

  Example Uses: The application needs to fetch the GLSetup record to get the current period, but has no other use for it. As another example, an application needs to insert one record, but do nothing else with that record.
Software Development Kit

- **SqlSystemDb** — This flag indicates that all database operations for the specified cursor are directed to the system database. By default, all database operations for the cursor are performed on the application database. When using this flag in addition to other cursor optimization flags, you can separate the flags using + or |. For example, to declare a cursor for reading only reading the User table in the System database, the SqlCursorEx statement would use the following syntax:

  ```vbs
  Call SqlCursorEx(c1, NOLEVEL + SqlSystemDb + SqlFastReadOnly, _
  "UserCursor","User","")
  ```

  **Note:** For those optimization flags that invoke Microsoft SQL Server's low-level API (SqlFastReadOnly, SqlNoSelect, and SqlSingleRow), database operations for the cursor optimized with one of these flags on the SQL Server platform bypass the use of SQL Server's advanced cursor API. As a result, database operations are serviced directly from SQL Server's low-level API, thus resulting in enhanced performance. However, when using this low-level API, the cursor can encounter contention with other cursors in the same application. (This is not the case when all operations are serviced by SQL Server's advanced cursor API.) SQL Server statements processed with a cursor using the SqlFastReadOnly, SqlNoSelect, or SqlSingleRow flag must include all fields for every table referenced by the SQL statement. For example, partial-record Select statements (“Select BatNbr from Batch” as opposed to “Select * from Batch”) are not supported by cursors allocated with these flags specified.

**SqlErr Function**

**Description**

Obtain the return value for the SQL operation last performed.

**Syntax**

```vbs
RetVal = SqlErr()
```

**Remarks**

This function can be used after any SQL call that is declared as a statement (as opposed to a function), in order to obtain the return value. For example, the call `SInsert1` is declared as a subroutine and does not return a value to the application. In most cases, the application does not need to check the return from this call, since by default SWIM traps all return codes except 0 and the NOTFOUND symbolic constant. However, in special cases, where the `SqlErrException` statement is used to give the application more control over error handling, the application will need to obtain the return code and this function is used for that purpose.

The `SqlErr` function returns one of the following integer global constants declared in Solomon.VBTools.vb:

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUPLICATE</td>
<td>The record last updated or inserted caused a duplicate error as defined by one or more unique indexes on the relevant table.</td>
</tr>
</tbody>
</table>

**See Also**

`SqlErrException Statement`
Example
This example illustrates how to insert a uniquely numbered Batch record into the database. The example assumes that a database transaction is already active when the illustrated procedure is called. SqlErrException and SqlErr are used to detect duplicate batch number error without causing Swim to abort the transaction. The sample procedure receives two parameters:

- **BatchStruct** — A Batch record which is to be saved to the database, having all relevant fields ALREADY initialized EXCEPT the batch number itself.
- **AutoNbr_SqlStr** — The name of an “auto number” stored procedure which will fetch AutoBat and LastBatNbr fields (in that order) from one of the setup records.

```vbnet
Sub BATCH_AUTONBR_INSERT (BatchStruct As Batch, ByVal AutoNbr_SqlStr As String)
    Dim AutoNbrFetch    As Integer

    'Allocate cursor resources
    Call SqlCursor(CSR_AutoNbr, NOLEVEL)
    Call SqlCursor(CSR_Batch_AutoNbr_Insert, NOLEVEL)

    'Setup cursor with stored procedure so it will be able to
    'run an SInsert1()
    Call Sql(CSR_Batch_AutoNbr_Insert, "Batch_Module_BatNbr" + J
    sparm("") + sparm(""))

    'Fetch the necessary fields for auto batch numbering from the Setup
    'record specified by AutoNbr_SqlStr
    AutoNbrFetch = SqlFetch1(CSR_AutoNbr, AutoNbr_SqlStr, AutoNbr)

    'Turn ON exception error checking for DUPLICATE error condition so
    'Swim will not go into abort mode if a duplicate batch number happens
    'to already exist.
    Call SqlErrException(EXCEPTION_ON, DUPLICATE)

    Do
        'Increment  AutoNbr.LastNbrUsed to next sequential value
        '(within the size of batch numbers actually being used).
        Call incrstrg(AutoNbr.LastNbrUsed, 6, 1)

        BatchStruct.BatNbr = AutoNbr.LastNbrUsed

        'Attempt to insert batch record with new batch number
        Call SInsert1(CSR_Batch_AutoNbr_Insert, "Batch",
        BatchStruct)

        Loop While (SqlErr() = DUPLICATE)

        'Write changes to Setup record back to database
        Call SUpdatel(CSR_AutoNbr, "*.*", AutoNbr)

        'Turn OFF exception error checking for DUPLICATE.
```

```vbnet
'... (continued as in the document)...
```
Call SqlErrException(EXCEPTION_OFF, DUPLICATE)

'Free up cursor resources
Call SqlFree(CSR_AutoNbr)
Call SqlFree(CSR_Batch_AutoNbr_Insert)
Call SqlFree(CSR_Batch_AutoNbr_Insert)

End Sub

SqlErrException Statement

Description
Toggle automatic error handling logic for one or more database error codes.

Syntax
Call SqlErrException (ToggleFlag, ErrorToExcept)

Remarks
By default, all error codes except 0 and NOTFOUND are trapped within SWIM and are not returned to the application. To alter this behavior an application can use the SqlErrException function to tell SWIM not to trap certain errors and instead return them to the application. This statement is used in conjunction with the SqlErr function.

Note that the application is responsible for toggling the exception back off once it is no longer needed.

The SqlErrException statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ToggleFlag</td>
<td>Integer</td>
<td>Used to tell SWIM to turn the error exception on or off. The EXCEPTION_ON and EXCEPTION_OFF symbolic constants are defined in Solomon.VBTools.vb as the only two valid values.</td>
</tr>
<tr>
<td>ErrorToExcept</td>
<td>Integer</td>
<td>Error code to be returned to the application rather than handled automatically by the system. If only duplicate record errors should be excepted from automatic error handling logic then the DUPLICATE symbolic constant defined in Solomon.VBTools.vb should be passed. The RETURN_ALL_ERRVALS symbolic constant defined in Solomon.VBTools.vb should be passed if all errors are to be returned to the application.</td>
</tr>
</tbody>
</table>

See Also
SqlErr Function

Example
This example illustrates how to insert a uniquely numbered Batch record into the database. The example assumes that a database transaction is already active when the illustrated procedure is called. SqlErrException and SqlErr are used to detect duplicate batch number error without causing Swim to abort the transaction. The sample procedure receives two parameters:

- **BatchStruct** — A Batch record which is to be saved to the database, having all relevant fields ALREADY initialized EXCEPT the batch number itself.

- **AutoNbr_SqlStr** — The name of an “auto number” stored procedure which will fetch AutoBat and LastBatNbr fields (in that order) from one of the setup records.
Sub BATCH_AUTONBR_INSERT (BatchStruct As Batch, ByVal AutoNbr_SqlStr As String)
    Dim AutoNbrFetch As Integer
    'Allocate cursor resources
    Call SqlCursor(CSR_AutoNbr, NOLEVEL)
    Call SqlCursor(CSR_Batch_AutoNbr_Insert, NOLEVEL)
    'Setup cursor with stored procedure so it will be able to run an
    'SInsert1()
    Call Sql(CSR_Batch_AutoNbr_Insert, "Batch_Module_BatNbr" + "" +
             sparm("") + sparm(""))
    'Fetch the necessary fields for auto batch numbering from the Setup
    'record specified by AutoNbr_SqlStr
    AutoNbrFetch = SqlFetch1(CSR_AutoNbr, AutoNbr_SqlStr, AutoNbr)
    'Turn ON exception error checking for DUPLICATE error condition so
    'Swim will not go into abort mode if a duplicate batch number happens
    'to already exist.
    Call SqlErrException(EXCEPTION_ON, DUPLICATE)
    Do
        'Increment  AutoNbr.LastNbrUsed to next sequential value
        '(within the size of batch numbers actually being used).
        Call incrstrg(AutoNbr.LastNbrUsed, 6, 1)
        BatchStruct.BatNbr = AutoNbr.LastNbrUsed
        'Attempt to insert batch record with new batch number
        Call SInsert1(CSR_Batch_AutoNbr_Insert, "Batch", J
                      BatchStruct)
    Loop While (SqlErr() = DUPLICATE)
    'Write changes to Setup record back to database
    Call SUpdate1(CSR_AutoNbr, ".*.", AutoNbr)
    'Turn OFF exception error checking for DUPLICATE.
    Call SqlErrException(EXCEPTION_OFF, DUPLICATE)
    'Free up cursor resources
    Call SqlFree(CSR_AutoNbr)
    Call SqlFree(CSR_Batch_AutoNbr_Insert)
End Sub

SqlExec Statement

Description
Run an inline SQL statement.

Syntax
Call SqlExec(Cursor)
Remarks
Run a dynamic SQL statement on a cursor previously initialized with calls to the Sql and SqlSubst statements (in that order).
The SqlExec statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cursor</td>
<td>Integer</td>
<td>SQL database cursor</td>
</tr>
</tbody>
</table>

See Also
Sql Statement, SqlSubst Statement

SqlFetch Functions

Description
Used to initialize a new database view and immediately retrieve a composite record.

Syntax
RetVal = SqlFetch1(Cursor, SqlStr, bTable1)
RetVal = SqlFetch4(Cursor, SqlStr, bTable1, bTable2, bTable3, bTable4)
RetVal = SqlFetch8(Cursor, SqlStr, bTable1, bTable2, bTable3, bTable4, bTable5, bTable6, bTable7, bTable8)

Remarks
In order to fetch information from the database a database view must first be initialized specifying what tables, fields and restriction criteria are to be utilized. Secondly an actual request for data from an existing view must be sent to the server. Each of the SqlFetch1, SqlFetch4 and SqlFetch8 functions effectively perform both of these operations in a single call which would otherwise require a combination of two calls (for example, Sql and SqlFetch1). In looping situations where a program needs to sequentially read through multiple records in a view, these functions are convenient for initializing the view and immediately fetching the first record. However they should not be used to fetch subsequent records since the view is being re-established each time SqlFetch1, SqlFetch4 or SqlFetch8 is called and therefore they will always only fetch the first record in the view. In these cases, SFetch1, SFetch4 or SFetch8 can be used to fetch subsequent records.

SqlFetch1 is designed for SQL statements returning data from a single table. For more advanced SQL statements having one or more table joins either SqlFetch4 or SqlFetch8 can be used.

The SqlFetch1 function uses the following arguments (SqlFetch4 and SqlFetch8 respectively have four and eight SolomonDataObjects. PNULL should be passed for unused SolomonDataObject parameters)
**SGroupFetch1, SGroupFetch4 or SGroupFetch8** must be used if the SQL statement used to initialize the cursor contains one or more of the following components:

- Group aggregate functions (such as Count and Sum)
- DISTINCT keyword
- GROUP BY clause
- HAVING clause
- Subqueries

**See Also**

Sql Statement, SParm Function, IParm Function, FParm Function, DParm Function, SGroupFetch Functions

**Example**

The following code reads through all records in the Account table. It is assumed that the Account_All stored procedure was created using the following text “Select * from Account where Acct LIKE @parm1 Order By Acct”. In following example, all records in the Account table will meet the restriction clause since a wildcard value (for example, SQLWILDSTRING) is used for the one and only parameter to the stored procedure. In this case a wildcard value can properly be passed for the value of @parm1 since this parameter is associated with the LIKE keyword. Furthermore, since the Account_All stored procedure only selects data from a single table (for example, the Account table) **SqlFetch1** would be adequate. However in this example, **SqlFetch4** is actually used to illustrate how to pass PNULL,0 for unused table structure arguments.

```vba
Dim CSR_Account As Integer
Dim SqlStr As String
Dim AcctFetch As Integer

'Allocate cursor
    Call SqlCursor( CSR_Account, NOLEVEL)

'Initialize cursor with a SQL stored procedure and immediately
'fetch first record
SqlStr = "Account_All" + SParm(SQLWILDSTRING)
AcctFetch = SqlFetch4(CSR_Account, SqlStr, bAccount, PNULL, PNULL, PNULL)

'Read through all subsequent Account records
    While (AcctFetch = 0)
        AcctFetch = SFetch4(CSR_Account, bAccount, PNULL, PNULL, PNULL)
    Wend
```
SqlFree Statement

Description
Free a database cursor no longer needed by the application.

Syntax
Call SqlFree(Cursor)

Remarks
The SqlFree statement is used to deallocate cursors previously allocated with the SqlCursor statement. The usage is this statement is entirely optional. It is normally used when a cursor is no longer needed by an application whose total number of cursors is very close to the maximum limit. All cursors are automatically released by the system when the application terminates (for example, when ScreenExit is called).

The SqlFree statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cursor</td>
<td>Integer</td>
<td>Resource handle for the SQL database cursor to be deallocated.</td>
</tr>
</tbody>
</table>

See Also
SqlCursor Statement

SqlNoWait Statement

Description
Implement asynchronous query processing on Microsoft SQL Server.

Syntax
Call SQLNoWait (Cursor, SQLStr)

Remarks
By design, when Microsoft SQL Server processes a stored procedure, control is not passed back to the calling application until the stored procedure has completed. For lengthy stored procedures, the processing status timer is not updated, nor can the user cancel the process by clicking Cancel. The SQLNoWait statement allows processing of a large SQL statement or stored procedure on the server to start, and allows the Process Status Window timer to be updated while running the SQL statement.

When the application issues an SQLNoWait() call, control is not returned to the application until the SQL statement has been run.

If the user clicks Cancel during processing, the process is cancelled. When the user clicks OK for the “Process has been cancelled” message in the Process Status Window, the application and running of the stored procedure or SQL statement is terminated.

This statement should only be used within a Microsoft Dynamics SL SDK process that has already issued a Status( StartProcess...) call. If no process has been started, BulletProof error 10247 is generated by the kernel.

Any stored procedure that is called using SQLNoWait must have SET NOCOUNT ON as its first command.
The SQLNoWait statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cursor</td>
<td>Integer</td>
<td>SQL database cursor.</td>
</tr>
<tr>
<td>SQLStr</td>
<td>String</td>
<td>SQL statement or stored procedure to be used in initializing a new database view. If a stored procedure is used then all parameters must be sequentially appended in order of occurrence within the original Create Procedure statement.</td>
</tr>
</tbody>
</table>

Example

Private Sub AsyncAPITest_Click()
    Dim SQLStmt As String
    Dim serr As Integer
    ' Call the AsyncTest stored procedure
    SQLStmt = "AsyncTest"
    ' Start the process
    Call Status(StartProcess, False, ",", 0)
    Call Status(0, False, "Processing Asynchronously", DISP_ONLY)
    Call sqlNoWait(c2, SQLStmt)
    serr = SqlErr()
    ' End The process
    Call Status(EndProcess, False, ",", 0)
End Sub

SqlRowCntrs Statement

Description
SqlRowCntrs informs the kernel to use update counters from the current memory array row, and to use these update counters to perform database contention checking.

Syntax
Call SqlRowCntrs(Handle, Cursor)

Remarks
SqlRowCntrs is used in the Update_Update event whenever the application program wishes to bypass default database update behavior that is performed by the kernel. It is used when the application program is performing updates from a DSLGrid application on a line-by-line basis.

The SqlRowCntrs statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemHandle</td>
<td>Integer</td>
<td>Memory array resource handle</td>
</tr>
<tr>
<td>Cursor</td>
<td>Integer</td>
<td>SQL database cursor used in previous fetch call</td>
</tr>
</tbody>
</table>

See Also
MFirst Function, MNext Function, DetailLoad Statement, SQLFetch Functions
Example

Sub Update1_Update(level%, insertflg%, levelsdone%, levelsleft%, retval%)

    Dim MemRecFound%
    Dim MaintFlg%
    Dim DocFound%

    'Begin the transaction
    Call Tranbeg(True)

    'Move to first row in the grid
    MemRecFound = mfirst(MemHandle, MaintFlg)

    While MemRecFound = 0

        Select Case MaintFlg

            'Determine the maintenance status of the current row
            Case UPDATED, INSERTED
                Call sqlRowCntrs(MemHandle, C3)
                DocFound = sqlfetch1(C3, ProcName, Hold_APDoc)
                If DocFound = 0 Then
                    If bAPDoc.Status = "A" Then
                        'Document is Active and can be marked as selected
                        bAPDoc.Selected = LTRUE
                        Call supdate1(C3, "APDoc", bAPDoc)
                    End If
                End If
            End Case

        End Select

        ' Fetch next row in memory array
        MemRecFound = mnext(MemHandle, MaintFlg)
    Wend

    Call Tranend

    'Set so that kernel does not process
    retval = NoAction

End Sub
SqlSubst Statement

Description
Specify a data value for a substitution parameter in an inline SQL statement previously run via the Sql function.

Syntax
Call SqlSubst(Cursor, ParmName, ParmValue)

Remarks
SqlSubst allows you to specify values for one or more substitution variables in an inline SQL statement that is run via the Sql statement. It is not to be used for a stored procedure.

This statement must be called after the Sql statement and before the SqlExec statement. It can be called any number of times before SqlExec is called, depending on how many parameter values there are to substitute into the SQL statement.

The SqlSubst statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cursor</td>
<td>Integer</td>
<td>SQL database cursor previously initialized with an inline SQL statement containing one or more parameters.</td>
</tr>
<tr>
<td>ParmName</td>
<td>String</td>
<td>Name of the SQL statement parameter (not including the @ character).</td>
</tr>
<tr>
<td>ParmValue</td>
<td>String</td>
<td>Data value to be used for the designated parameter.</td>
</tr>
</tbody>
</table>

See Also
Sql Statement, SqlExec Statement

Example
Dim CSR_AcctHist as Integer
Dim SqlStr as String
Dim AcctHist_Fetch as Integer

'Build an inline Sql Statement
SqlStr = "Select * From AcctHist"
SqlStr = SqlStr + " Where AcctHist.Acct = @parm1"
SqlStr = SqlStr + " and AcctHist.Sub = @parm2"
SqlStr = SqlStr + " and AcctHist.FiscYr = @parm3"
SqlStr = SqlStr + " Order By Acct, Sub, FiscYr"

'Initialize cursor with dynamic sql statement.
Call Sql(CSR_AcctHist, SqlStr)

'Substitute data values for each parameter
Call SqlSubst(CSR_AcctHist, "parm1", "1010")
Call SqlSubst(CSR_AcctHist, "parm2", "000000")
Call SqlSubst(CSR_AcctHist, "parm3", "1998")

'Run the completed statement
Call SqlExec(CSR_AcctHist)

'Fetch the record
AcctHist_Fetch = SFetch1(CSR_AcctHist, bAcctHist)
Status Statement

Description
Report process status information to either the Process Status Window or the Event Log or both.

Syntax
Call Status(OpCode, FatalError, InfoString, Destination)

Remarks
The Status statement is foundational to all processes developed with Microsoft Dynamics SL SDK. This statement is used to formally start a process, report progress and/or error information during the process as well as formally end the process. Once a process has completed, the user can review information relating to events which occurred during the process in the Event Log which was generated by the application using the Status statement.

Whenever the Status statement is used to report data values along with a corresponding caption, it is recommended that the text of a Label control be used for the reported caption and that the data value be formatted with the SParm statement if it is a string value. For example, if a particular Batch Number is to be reported then a Label control should be created having a Text property of “Batch Number:”. The actual name of the Label control itself is not as important but perhaps this one could be named Ip_Batch where the “Ip” portion of the name could signify that the Label is used during the process. At any rate a particular Batch Number can subsequently be reported with code similar to: Ip_Batch.Text + SParm(bBatch.BatNbr). The exact syntax obviously will vary depending on whether a message is being utilized that can handle replacement parameters. In that case the code would assume the form of: Ip_Batch.Text + NL + SParm(bBatch.BatNbr). The advantage of utilizing Label controls for string literals, such as text values, is that it facilitates customization of the actual reported text using the Customization Manager. Thus, for example, this methodology makes it extremely easy for a particular process to be customized for a particular language without the need to modify string literals in the underlying source code.

The Status statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>Integer</td>
<td>Operation code. This argument must be either zero or a message number corresponding to a message in the Microsoft Dynamics SL message file or one of the following symbolic constants defined in Solomon.VBTools.vb: StartProcess — Displays the Process Status Window and creates an Event Log, SaveGoodArgVals — Identifies the unique key values (for example, good argument values), along with corresponding captions, for the high level entity currently being processed such as the current Batch or Document. The values are actually passed in the InfoString argument. Normally good argument values only need to be displayed on the Process Status Window (for example, DISP_ONLY) since these unique key values will automatically be written to the Event Log in the event of an error. EndProcess — Closes the Event Log and displays the name of the Event Log only if any information was actually reported during the process. Lastly, the OK button is displayed on the Process Status Window and then the system waits for the user to respond. StopProcess — Similar to EndProcess except that the OK button is not displayed, and therefore a user response is not required, unless information was actually reported during the process and consequently the user needs to be notified of the existence of an Event Log. A value of zero will operate the same as a message number except that SWIM will not refer to the Microsoft Dynamics SL message file for any message text. This allows applications to send Label captions (Label.Text) directly to the log without the requirement of an actual message number.</td>
</tr>
<tr>
<td>Argument</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>FatalError</td>
<td>Integer</td>
<td>True / False whether a fatal error is being reported. Fatal errors cause the system to go into abort mode for transactions which are abortable (for example, TranBeg( True). The application will automatically come out of abort mode on the next TranEnd call if no restart cursor is in effect. If a restart cursor is in effect then the application will come out of abort mode when the next fetch occurs on the restart cursor. If the transaction is not abortable then a fatal error will cause the application to terminate. This argument is ignored for all OpCode values except zero and message numbers.</td>
</tr>
<tr>
<td>InfoString</td>
<td>String</td>
<td>Information to be reported. If OpCode is zero then the only reported information is the value of InfoString. If OpCode is a message number then InfoString can be used to pass data values for replaceable parameters, if any, contained within the designated message. Multiple parameters are supported by separating each corresponding data value with the NL symbolic constant defined in Solomon.VBTools.vb. This argument is ignored for all OpCode values except zero, message numbers and SaveGoodArgVals.</td>
</tr>
<tr>
<td>Destination</td>
<td>Integer</td>
<td>Controls where the status information should be reported. The following valid values are defined as symbolic constants in Solomon.VBTools.vb: DISP_ONLY — Only report the information to the Process Status Window. LOG_ONLY — Only report the information to the Event Log. LOG_AND_DISP — Report the information to both the Process Status window and the Event Log. This argument is ignored for all OpCode values except zero, message numbers and SaveGoodArgVals. A value of 0 may be passed in cases where Destination is not applicable.</td>
</tr>
</tbody>
</table>

See Also

SetRestart Statement, TranBeg Statement, TranEnd Statement

Example

The Payroll Check Update process uses the following code to begin the process, report errors during the process and lastly to end the process.

Two error messages can potentially be reported during the process depending on various logical problems detected in data. Message #832 is assumed to have the following associated text in the Microsoft Dynamics SL message file: “Batch status is invalid for processing.” Lastly, message #6003 is assumed to have the following associated text: “Warning - %s %s is not found.” Notice that the example code illustrates how to pass data values to the two replaceable parameters in this particular message.

```vbnet
Dim SqlStr As String  'Command string to send to Sql functions
Dim Batch_Fetch As Integer 'Return value from SqlFetch( Batch)
Dim Hold_Batch As Batch   'Temp buffer to hold batch values for voiding
Dim Any_Checks_kept As Integer 'True/False whether any of the checks which were KEPT (e.g. as opposed to voided or deleted). This will help us determine what sort of update to make to processing all of the checks.
```
Dim Any_Checks_VOIDED As Integer 'True/False whether any of the checks
   ' which were
processed were VOIDED. Used for
   ' essentially the
same type of purpose as
   ' Any_Checks_KEPT
Dim ParmStr As String 'Parameters to send to 02.40
   ' Release PR

Call Status(StartProcess, False, "", 0)

   Call TranBeg(True)
   'Refetch the batch which was just created.
   strSql = "Batch_Module_BatNbr" + SParm("PR") +
         SParm(bBatch.BatNbr)
   Batch_Fetch = SqlFetch1(CSR_Batch, strSql, bBatch)

   Call Status(SaveGoodArgVals, False, lp_Batch.Text + J
         sparm(bBatch.BatNbr), DISP_ONLY)

   If (Batch_Fetch = 0) Then

   'Make absolutely sure that the batch has NOT already been kept
   If (bBatch.Status = "K") Then

      Select Case (bCheckHandling_All.Str)
      Case "D"
         Call Process_Checks_Delete_All(bBatch.BatNbr)
      Case "S"
         Call Process_Checks_Selected(bBatch, J
         Any_Checks_KEPT = False
         Any_Checks_VOIDED = False
         Any_Checks_KEPT = False
         Any_Checks_VOIDED = Process_Checks_Void_All(bBatch)
      Case Else   ' "K"
         'There is absolutely no need to process the documents since we already
         'know that we are going to keep them all and they are already attached
         'to the current batch.
         Any_Checks_KEPT = True
         Any_Checks_VOIDED = False
      End Select

      Any_Checks_KEPT = False
      Any_Checks_VOIDED = False
If any checks at all were either KEPT or VOIED
then we 'will update the batch to be a "real" batch as
opposed to 'VOIDING the batch.

If ((Any_Checks_KEPT = True) Or
    (Any_Checks VOIDED = True)) Then
    bBatch.Descr = ""  'Clear
    bBatch.PerPost = bPerPost.Str

    If (Any_Checks_KEPT = True) Then
        bBatch.Status = "B"
    Else
        'All checks were voided - so
    End If

    bBatch.Rlsed = LTRUE
    bBatch.Status = "C"
End If

out UserID
out Checking Account
out Checking SubAcct

Else
    'Re-initialize batch as a
    voided batch

    Hold_Batch = bBatch

    bBatch = nBatch
    bBatch.BatNbr = 

    Hold_Batch.BatNbr

    bBatch.Module = 

    Hold_Batch.Module

    bBatch.PerPost = bPerPost.Str
    bBatch.Rlsed = LTRUE
    bBatch.Status = "V"
End If

Call SUpdate1(CSR_Batch, "Batch", bBatch)

Else
    'Batch status is not valid for processing.
    Call Status(832, True, "", LOG_AND_DISP)
End If  'bBatch.Status

Else
    'Warning - %s %s is not found.
    Call Status(6003, True, lp_Batch.Text + NL + J
    SParm(bBatch.BatNbr), LOG_AND_DISP)
End If 'Batch_Fetch

Call TranEnd

'RElease batch to which any KEPT checks were attached.
   If ((TranStatus() = 0) And (Any_ChecksKEPT = True)) Then
      'Call 02.400 to release the current batch
      ParmStr = bBatch.Module + PRMSEP + bBatch.BatNbr +
      PRMSEP + bBatch.EditScrnNbr + PRMSEP + bBatch.Status
      Call CallApplicWait("0240000", ParmStr)
   End If

Call Status(EndProcess, False, ",", 0)

If (Trim$(bRptRuntime.BatNbr) <> ") Then
   'If this program is called by ROI then the user is ONLY allowed to
   'process THE batch for which printing was just completed.
   'Consequently there is absolutely nothing else for the user
   'to do from within this process.
   Call ScreenExit("", ")
Else
   'Re-initialize screen
   bBatch = nBatch

   Call SetDefaults(PNULL, PNULL, PNULL)
   Call DispFields(PNULL, PNULL, PNULL)
   Call Evaluate_Properties(FLD_ALL)
   Call ApplSetFocus(cCheckHandling_All)
End If

**StrToDate Statement**

**Description**
Convert a date value from a string in MMDDYYYY format into an SQL date format.

**Syntax**
Call **StrToDate**(DateStringToConvert, **SQLDate**)  
The **StrToDate** statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DateStringToConvert</strong></td>
<td>String</td>
<td>String in MMDDYYYY format.</td>
</tr>
<tr>
<td><strong>SQLDate</strong></td>
<td>Integer</td>
<td>Converted date value.</td>
</tr>
</tbody>
</table>

**See Also**
DateToIntl Function, DateToStr Function, DateToStrSep Function, IntlStrToDate Statement
StrToTime Statement

Description
Convert a time value from a string in HHMMSShh format into an SQL time format.

Syntax
Call StrToTime(TimeStringToConvert, SQLTime)

The StrToTime statement uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeStringToConvert</td>
<td>String</td>
<td>String in HHMMSShh format.</td>
</tr>
<tr>
<td>SQLTime</td>
<td>Integer</td>
<td>Converted time value.</td>
</tr>
</tbody>
</table>

See Also
TimeToStr Function

SUpdate Statements

Description
Update one record of each specified table within an existing database view.

Syntax
Call SUpdate1(Cursor, TablesUpdating, bTable1)
Call SUpdate4(Cursor, TablesUpdating, bTable1, bTable2, bTable3, bTable4)
Call SUpdate8(Cursor, TablesUpdating, bTable1, bTable2, bTable3, bTable4, bTable5, bTable6, bTable7, bTable8)

Remarks
Existing records can be programmatically updated directly via the use of the SUpdate1, SUpdate4 and SUpdate8 statements. However, before a record can be updated it must first be fetched using one of the SFetch1, SFetch4 and SFetch8 or SqlFetch1, SqlFetch4 and SqlFetch8 functions. The fetch operation which precedes the update must be made using the same database view / cursor on which the update will occur. For example, if the fetch occurs on Cursor A then the update must also occur on Cursor A as opposed to some unrelated Cursor XYZ. Nevertheless, once the database view has been established these functions will update the current record in the view for each table referenced in the TablesUpdating argument using data from corresponding table structure arguments.

SUpdate1 is designed for SQL statements referencing a single table. In this case the TablesUpdating is always the name of the single table actually referenced. For more advanced SQL statements having one or more table joins either SUpdate4 or SUpdate8 can be used. The referencing of more than one table does not automatically force the current record of every table within the view to be updated anytime SUpdate4 or SUpdate8 is used on the corresponding cursor. The current record of a particular table in the view will only be updated if its corresponding table name is explicitly specified in the TablesUpdating argument so long as each table name so specified is also referenced in the SQL statement which was used to initialize the current view. Thus, for example, if TableA and TableB are the only two tables referenced in the SQL statement used to initialize the current view, then a value of TableXYZ would not be valid for the TablesUpdating argument.

The SUpdate1 function uses the following arguments (SUpdate4 and SUpdate8 respectively have four and eight SolomonDataObjects. PNULL should be passed for unused SolomonDataObject parameters)
## Software Development Kit

### Argument

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cursor</td>
<td>Integer</td>
<td>SQL database cursor.</td>
</tr>
<tr>
<td>TablesUpdating</td>
<td>String</td>
<td>Name of each table, in the specified cursor’s view, whose current record is to be updated. Multiple table names should be separated by commas.</td>
</tr>
<tr>
<td>bTable1</td>
<td>SolomonDataObject</td>
<td>Table structure corresponding to the primary table in the SQL statement. Data from this structure will be used to overwrite existing data in the corresponding current record if the name of the said database table is explicitly specified in the <code>TablesUpdating</code> argument.</td>
</tr>
</tbody>
</table>

**Note:** Database updates occurring on cursors allocated by `SqlCursorEx` using the `SqlList` flag (for example, buffered cursors) have two unique requirements. First, the application must update all records it reads, using a buffered cursor, if it updates even one record. Failure to comply with this requirement will result in a sparse update error. Secondly, the application should not modify the view on the cursor after updates have occurred until after the transaction has ended. For example, if the application is reading and updating Table A records on buffered Cursor A then Cursor A should not be used for any other purpose until after the database transaction has ended. If no updates are made using Cursor A then this requirement does not apply.

### See Also

SFetch Functions, SqlFetch Functions, SqlCursor Statement

### Example

This simple example will update all GLTran records in General Ledger Batch 000001 as being released. The release of GLTran records actually entails additional application logic which is not directly relevant to the illustration of the `SUpdate4` statement and therefore it has been removed from this example.

This example will assume the GLTran_Module_BatNbr_LineNbr stored procedure was originally created with the following SQL statement:

```sql
Select * from GLTran
where Module  = @parm1
and BatNbr  = @parm2
and LineNbr between @parm3beg and @parm3end
order by Module, BatNbr, LineNbr
```

Since the above SQL statement only retrieves data from a single table (for example, the GLTran table) `SUpdate1` would be adequate. However in this example, `SUpdate4` is actually used to illustrate how to pass `PNULL`, 0 for unused table structure arguments.

```vbnet
Dim CSR_GLTran  As Integer
Dim SqlStr      As String
Dim GLTranFetch As Integer

'Allocate a database cursor. A buffered cursor (e.g. SqlList) can be
'used to speed up performance since ALL GLTran records read within the
'database transaction will also be updated.
Call SqlCursor( CSR_GLTran, NOLEVEL + SqlList)

'Begin a database transaction since all updates to the database must
'occur within a transaction.
Call TranBeg(True)

'Initialize SqlStr with a stored procedure and associated parameters
```
which can be used to fetch all GLTran records in GL Batch 000001.

```
SqlStr = "GLTran_Module_BatNbr_LineNbr" + sparm("GL") + J
         + sparm("000001") + iparm(INTMIN) + iparm(INTMAX)
```

'Initialize cursor with a SQL stored procedure and immediately fetch 'first record.

```
GLTranFetch = SqlFetch4(CSR_GLTran, SqlStr, bGLTran, J
         PNUL, PNUL, PNUL)
```

While (GLTranFetch = 0)

  'Release current transaction
  bGLTran.Posted = "U"
  bGLTran.Rlsed = LTRUE

  'Update the record last fetched on CSR_GLTran (e.g. the current GLTran 'record) with the modified contents of bGLTran
  Call SUpdate4(CSR_GLTran, "GLTran", bGLTran, J
         PNUL, PNUL, PNUL)

  'Load next transaction record
  GLTranFetch = SFetch4(CSR_GLTran, bGLTran, J
         PNUL, PNUL, PNUL)

Wend

'End the database transaction to commit all updates to the database.

Call TranEnd

---

**SwimGetProfile Function**

**Description**

Return the value of a specific entry within the designated INI file.

**Syntax**

```
INIEntryValue = SwimGetProfile(INISection, INIEntry, DefaultValue, MaxEntrySize, INIFileName)
```

**Remarks**

The SwimGetProfile function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIEntryValue</td>
<td>String</td>
<td>Returned value for the designated entry within the INIFileName.</td>
</tr>
<tr>
<td>INISection</td>
<td>String</td>
<td>Name of the section within the INIFileName containing the INIEntry.</td>
</tr>
<tr>
<td>INIEntry</td>
<td>String</td>
<td>Name of the entry whose associated value is to be returned.</td>
</tr>
<tr>
<td>DefaultValue</td>
<td>String</td>
<td>Default value to be returned if the designated entry cannot be located.</td>
</tr>
</tbody>
</table>
See Also
SwimWriteProfile Function

SwimWriteProfile Function

Description
Write a new value for a specific entry within the designated INI file.

Syntax
RetVal = SwimWriteProfile(INISection, INIEntry, INIEntryValue, INIFileName)

Remarks
The SwimGetProfile function uses the following arguments:

See Also
SwimGetProfile Function

TestLevelChg Function

Description
Return the current update status flag for a specific level.

Syntax
Status = TestLevelChg(LevelNbr)

Remarks
Each update level, as defined by the Levels property of the DSLUpdate control, has a corresponding level status flag that is automatically maintained by the system. The purpose of the level status flag is to facilitate the optimization of database updates performed in response to Parent toolbar buttons. In general, these flags allow the system to only perform database updates for update levels which have in fact changed. If no information has changed then no information needs to be saved.
As previously mentioned, these update flags are automatically maintained by the system. When an existing record is loaded the flag is set to NOTCHANGED. If any non-keyfield is subsequently modified then the level status flag for the corresponding level is set to UPDATED. When a new record is being entered, the level status flag is set to INSERTED.

The **TestLevelChg** function allows the application to access the current value of this flag for a specific level. The current level status flag can be overridden by the application using the **SetLevelChg** statement.

The **TestLevelChg** function uses the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>Integer</td>
<td>Current value of the level status flag for the designated LevelNbr. The following possible values are defined as symbolic constants in Solomon.VBTools.vb: INSERTED, UPDATED, NOTCHANGED.</td>
</tr>
<tr>
<td>LevelNbr</td>
<td>Integer</td>
<td>Level whose status flag is to be returned.</td>
</tr>
</tbody>
</table>

**See Also**

**SetLevelChg Statement**

**Example**

The Payroll Earnings Type Maintenance screen contains a button to automatically populate the grid with all Deductions. This amounts to inserting records into the grid (for example, into its underlying memory array) under program control. Since the data is not entered via the user interface by the user, the system needs to be notified that information at the grid level (LEVEL1 in this case) has been programmatically updated and therefore needs to be saved. However, such notification only needs to occur if the system is not already aware that data has changed.

'If any records were inserted into the memory array then we need to make sure that the level status for the detail level is something other than NOTCHANGED so the system will know that something needs to be saved.

    If (AnyRecsInserted = True) Then
        If (TestLevelChg(LEVEL1) = NOTCHANGED) Then
            Call SetLevelChg(LEVEL1, UPDATED)
        End If
    End If

**TimeToStr Function**

**Description**

Convert a time value from SQL time format into a string in HHMMSshh format.

**Syntax**

```
TimeString = TimeToStr(TimeToConvert)
```

**Remarks**

The **TimeToStr** function uses the following arguments:
<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeString</td>
<td>String</td>
<td>(TimeToConvert) converted to a string in HHMMSShh format.</td>
</tr>
<tr>
<td>TimeToConvert</td>
<td>Integer</td>
<td>Time value to be converted.</td>
</tr>
</tbody>
</table>

**See Also**

StrToTime Statement

**TranAbort Statement**

**Description**

Abort the current database transaction.

**Syntax**

Call TranAbort

**Remarks**

The TranAbort statement allows the application to abort a database transaction which was initiated using the TranBeg statement.

Calling TranAbort directly is not, however, the recommended method to abort transactions. If the transaction to be aborted is an update operation for an application screen then the recommended method is for the application to set RetVal in the Update event of the DSLUpdate control to either an error message number or the ErrNoMess symbolic constant defined in Solomon.VBTools.vb. On the other hand, if the abort is to occur during a process then a fatal error message should be written to the Event Log using the Status statement. This will also have the effect of aborting the current database transaction.

**See Also**

Status Statement, TranBeg Statement, TranEnd Statement, Update Event

**TranBeg Statement**

**Description**

Begin a database transaction.

**Syntax**

Call TranBeg(IsAbortable)

**Remarks**

All updates to the database must occur within a database transaction.

All updates to the database will not actually be committed to the database until the transaction is ended via the TranEnd statement.

If any errors occur during the database transaction then the system will automatically abort (roll back) all updates occurring during the transaction as opposed to committing them to the database. If the transaction is an update operation for an application screen then it will be aborted when the application to sets RetVal in the Update event of the DSLUpdate control to either an error message number or the ErrNoMess symbolic constant defined in Solomon.VBTools.vb. On the other hand, if the transaction occurs during a process then it will be aborted when a fatal error message is reported using the Status statement.

The TranBeg statement uses the following argument:
<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsAbortable</td>
<td>Integer</td>
<td>True if the transaction is abortable (which is virtually always the case). Otherwise False. Errors occurring during non-abortable transactions will cause the application to immediately terminate.</td>
</tr>
</tbody>
</table>

See Also
Status Statement, TranEnd Statement, TranStatus Function, Update Event

TranEnd Statement

Description
End the current database transaction and commit all updates to the database.

Syntax
Call TranEnd

Remarks
If any errors occurred during the database transaction then the system will automatically abort (rollback) all updates which occurred during the transaction as opposed to committing them to the database. If the transaction is an update operation for an application screen then it will be aborted when the application sets RetVal in the Update event of the DSLUpdate control to either an error message number or the ErrNoMess symbolic constant defined in Solomon.VBTools.vb. On the other hand, if the transaction occurs during a process then it will be aborted when a fatal error message is reported using the Status statement.

See Also
Status Statement, TranBeg Statement, TranStatus Function, Update Event

TranStatus Function

Description
Returns the status of the current or last database transaction.

Syntax

`IntegerErrVal = TranStatus()`

Remarks
If a database transaction is currently not open then the status of the last performed database transaction is returned.
A return value of zero indicates that the transaction was successful.
A non-zero return value indicates a fatal error occurred during the transaction and therefore it will be, or was, aborted. In this case, the actual return value itself is the number of the error message describing the nature of the problem.

See Also
Status Statement, TranBeg Statement, TranEnd Statement, Update Event
Appendix A: Toolset Limitations

Microsoft Dynamics SL SDK Limitations

The table below lists the known limitations of the toolset:

<table>
<thead>
<tr>
<th>Description</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Levels</td>
<td>10</td>
</tr>
<tr>
<td>Number of Navigation Levels</td>
<td>2</td>
</tr>
<tr>
<td>Number of Key Field Controls per Level</td>
<td>5</td>
</tr>
<tr>
<td>Number of <strong>DSLGrids</strong></td>
<td>10</td>
</tr>
<tr>
<td>Number of tables that can be associated with any <strong>DSLGrid</strong></td>
<td>16</td>
</tr>
<tr>
<td>Number of SetAddr() calls</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Number of cursors available to the application</td>
<td>36</td>
</tr>
<tr>
<td>Number of memory arrays</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Number of keys that can be implemented for a single memory array</td>
<td>5</td>
</tr>
<tr>
<td>Number of triggers per control</td>
<td>8</td>
</tr>
<tr>
<td>Number of database tables</td>
<td>1000</td>
</tr>
<tr>
<td>Number of parameters allowed for a PV or DBNav</td>
<td>8</td>
</tr>
<tr>
<td>Number of cursors available to an application per database</td>
<td>50</td>
</tr>
<tr>
<td>Number of characters per field or column</td>
<td>800</td>
</tr>
</tbody>
</table>
Appendix B: Requirements for System Table Views Stored Procedures

Security measures require that existing and new stored procedures in the application database that reference system table views (such as VS_Company or VS_AcctSub) have additional special logic. The stored procedure will have permission to the view only if you add the following line to the stored procedure:

```sql
WITH EXECUTE AS '07718158D19D4f5f9D23B55DBF5DF1'
```

Without this line in a stored procedure that references one of the system table views, permission error 229 will occur.

**Example:**

```sql
CREATE Procedure RQ_AcctSub_Acct
    @CpnyID varchar(10),
    @UserID varchar(47),
    @Acct varchar(10)
WITH EXECUTE AS '07718158D19D4f5f9D23B55DBF5DF1'
as
    select *
    from vs_AcctSub
    where CpnyID = @CpnyID
    and Acct in (Select Acct from RQUserAcct where UserID = @UserID)
    and Acct like @Acct
    and Active = 1
    order by Acct, Sub
```
Appendix C: Extending Doc Share’s Capabilities

Introduction
You can use the Doc Share feature to create SharePoint sites and document libraries that give organizations the ability to share a variety of documents. The information in this appendix will assist you in extending the Doc Share feature.

This information will help you:
- Add a new entity type – see page 304.
- Add a document type to a new instance type – see page 306.
- Add a new document type to an existing instance type – see page 308.
- Use the Web service and wrapper functions to modify a SharePoint site – see page 310.

Common Doc Share Terms
The following terms are used frequently in Doc Share design:
- **Entity type** — Database record type. The standard Doc Share record types are Customer, Vendor, and Project.
- **Entity instance** — Database record occurrence for the entity. For example, a record for a customer is an entity record of the Customer instance.
- **Entity document type** — Information that you upload for an entity. For example, a statement for a Customer instance.

Database Tables for Doc Share
The database tables explained below are essential to Doc Share design:
- **WSPInstance** — Holds configuration information generated when you create a new entity type. These entity type configuration settings are stored in the application database. They give you the ability to turn off site creation and document uploading without losing the entity type’s setup information.
- **WSPDoc** — Associates document types with entity types. The records hold configuration settings for a specific document type, which give you the ability to turn off document uploading without losing the settings.
- **WSPObjExtension** — Holds specific configuration information for an entity instance, including the root site URL and site name setup. Its contents are derived from the WSPInstance record.
- **WSPPubDocLib** — Holds specific document configuration information for an entity instance, including the root site URL and site name setup. Its contents are derived from WSPInstance and WSPDoc records.
Adding an Entity Type

To create a new entity type for Doc Share, you need to produce a WSPInstance record in the application database. The sample statement below generates a record for the SalesPerson entity type.

- SLTypeID — The value in SLTypeID must be unique. The code example uses a high number to allow for additions in the future.
- SLTypeDesc – Values in SLTypeDesc appear on the Entity Type possible values list in System Manager's SharePoint Site Configuration (98.360.00) screen.

After you create the WSPInstance record, you will need to write code that enables Doc Share to use the new entity type.

Note: You can modify entity types on the SharePoint Site Configuration (98.360.00) screen.

Example

```
Insert into wspInstance (  
     [Crtd_DateTime]   ,[Crtd_Prog]     ,[Crtd_User]  
    ,[defaultCreationOpt]  
    ,[docLibName]     ,[docLibOrSite]   ,[docLibTemplate]  
    ,[inheritPermissions]  
    ,[Lupd_DateTime]   ,[Lupd_Prog]     ,[Lupd_User]  
    ,[rootSiteUrl]     
    ,[S4Future01]     ,[S4Future02]     ,[S4Future03]     ,[S4Future04]  
    ,[S4Future05]     ,[S4Future06]     ,[S4Future07]     ,[S4Future08]  
    ,[SPtemplate]     ,[SLTypeID]        ,[SLTypeDesc]  
    ,[spdocLibTemplate]       ,[Status]    ,[subSiteNamePrefix]  
    ,[templateFile]  
  ) values ( 0  --[Crtd_DateTime]  
    ,'9836000'     --[Crtd_Prog]  
    ,''            --[Crtd_User]  
    ,3             --[defaultCreationOpt]  
    ,''            --[docLibName]  
    ,''            --[docLibOrSite]  
    ,''            --[docLibTemplate]  
    ,''            --[inheritPermissions]  
    ,''            --[Lupd_DateTime]  
    ,'9836000'     --[Lupd_Prog]  
    ,''            --[Lupd_User]  
    ,''            --[rootSiteUrl]  
    ,''            --[S4Future01]  
    ,''            --[S4Future02]  
    ,0             --[S4Future03]  
    ,0             --[S4Future04]  
    ,0             --[S4Future05]  
    ,0             --[S4Future06]  
    ,0             --[S4Future07]  
```
0 --[S4Future08]
0 --[S4Future09]
0 --[S4Future10]
'' --[S4Future11]
'' --[S4Future12]
'' --[SPtemplate]
105 --[SLTypeID]
'SalesPerson' --[SLTypeDesc]
'' --[spdocLibTemplate]
'N' --[Status]
'' --[subSiteNamePrefix]
'' --[templateFile]
'' --[User1]
'' --[User10]
'' --[User2]
'' --[User3]
'' --[User4]
0 --[User5]
0 --[User6]
'' --[User7]
'' --[User8]
0 ) --[User9]
Adding a Document Type to a New Instance Type

The sample SQL statement below creates a WSPDoc record for the entity type 105. The key to creating this record is to specify the correct values in the Instance and DocumentID fields. Other important field information:

- **Instance** — Connects the document type to an entity type.
- **DocumentID** — Must be a unique number for this document type.
- **Active** — Determines if this type of document can be uploaded.

You will need to write code that enables Doc Share to use the new entity type and its document types.

**Note:** You can modify entity types and document types on the *SharePoint Site Configuration* (98.360.00) screen.

**Example**

```sql
INSERT INTO [WSPDoc]
([Active]       ,[Crtd_DateTime], [Crtd_Prog]         , [Crtd_User]
, [Lupd_DateTime],[Lupd_Prog]         , [Lupd_User]
, [S4Future01]  ,[S4Future02]        , [S4Future03]    
, [S4Future04]  ,[S4Future05]        , [S4Future06]    
, [S4Future07]  ,[S4Future08]        , [S4Future09]    
VALUES ('0'           --<Active, char(1),>
,0              --<Crtd_DateTime, smalldatetime,0>
,'9836000'      --<Crtd_Prog, char(8),''>
,''             --<Crtd_User, char(10),''>
,9              --<DocumentID, smallint,>
,'Sales Report' --<DocumentType, char(50),>
,105            --<Instance, smallint,>
,0              --<LastGroup, int,>
,0              --<Lupd_DateTime, smalldatetime,0>
,'9836000'      --<Lupd_Prog, char(8),>
,''             --<Lupd_User, char(10),''>
,'''            --<S4Future01, char(30),''>
,'''            --<S4Future02, char(30),''>
,0              --<S4Future03, float,0>
,0              --<S4Future04, float,0>
,0              --<S4Future05, float,0>
,0              --<S4Future06, float,0>
,0              --<S4Future07, float,0>
,0              --<S4Future08, float,0>
,0              --<S4Future09, float,0>
,0              --<S4Future10, float,0>
,'''            --<S4Future11, char(10),>
,'''            --<S4Future12, char(10),>
```
<table>
<thead>
<tr>
<th>User</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>User1</td>
<td>char(30)</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>User10</td>
<td>smalldatetime</td>
<td>0</td>
</tr>
<tr>
<td>User2</td>
<td>char(30)</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>User3</td>
<td>char(30)</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>User4</td>
<td>char(30)</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>User5</td>
<td>float</td>
<td>0</td>
</tr>
<tr>
<td>User6</td>
<td>float</td>
<td>0</td>
</tr>
<tr>
<td>User7</td>
<td>char(10)</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>User8</td>
<td>char(10)</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>User9</td>
<td>smalldatetime</td>
<td>0</td>
</tr>
</tbody>
</table>
Adding a Document Type to an Existing Instance Type

The sample SQL statement below creates a WSPDoc record for the entity type of 1. This value represents the Customer entity type. The standard Doc Share entity types are:

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Customer</td>
</tr>
<tr>
<td>2</td>
<td>Project</td>
</tr>
<tr>
<td>3</td>
<td>Vendor</td>
</tr>
</tbody>
</table>

The key to creating a WSPDoc record is to specify the correct values in the Instance and DocumentID fields. Other important field information:

- **Instance** — Connects the document type to an entity type.
- **DocumentID** — Must be a unique number for this document type.
- **Active** — Determines if this type of document can be uploaded.

**Note:** You can modify this record on the *SharePoint Site Configuration* (98.360.00) screen.

**Example**

```sql
INSERT INTO [WSPDoc] ([Active]       ,[Crtd_DateTime]     ,[Crtd_Prog]    ,[Crtd_User]
,[Lupd_DateTime],[Lupd_Prog]        ,[Lupd_User]
,[S4Future01]   ,[S4Future02]       ,[S4Future03]   ,[S4Future04]
,[S4Future05]   ,[S4Future06]       ,[S4Future07]   ,[S4Future08]
VALUES ('0'           --<Active, char(1),>
     ,0              --<Crtd_DateTime, smalldatetime,0>
     ,'9836000'      --<Crtd_Prog, char(8),''>
     ,''             --<Crtd_User, char(10),''>
     ,10             --<DocumentID, smallint,>
     ,'Sales Report' --<DocumentType, char(50),>
     ,1              --<Instance, smallint,>
     ,0              --<LastGroup, int,>
     ,0              --<Lupd_DateTime, smalldatetime,0>
     ,'9836000'      --<Lupd_Prog, char(8),>
     ,''             --<Lupd_User, char(10),''>
     ,''             --<S4Future01, char(30),''>
     ,''             --<S4Future02, char(30),''>
     ,0              --<S4Future03, float,0>
     ,0              --<S4Future04, float,0>
     ,0              --<S4Future05, float,0>
     ,0              --<S4Future06, float,0>
     ,0              --<S4Future07, smalldatetime,0>
     ,0              --<S4Future08, smalldatetime,0>
     ,0              --<S4Future09, int,0>
```
Modifying a SharePoint Site

Doc Share is implemented by using a .dll file to wrap the SharePoint Client Object Model. Each function is overloaded so that there is one that accepts the same arguments as the previous Web Service calls and another that accepts a Uri instead of the root site and sub site strings. The Uri versions will allow any exceptions to be passed to the caller while the String versions wrap the Uri versions and also catch any exceptions. The following functions are available to simplify the modification of a SharePoint site.

- CreateDocumentLibrary
- CreateSite
- DoesSiteExist
- DoesSPListExist
- GetLastError
- GetSPListTemplateList
- GetTemplateList
- IsSiteOrLib
- UploadSLDocument

Each of these functions is discussed in the topics that follow, together with examples of how to call them from a Visual Basic Tools application. To use these calls, you must add a reference to Microsoft.Dynamics.SL.SharepointClient to your project. This DLL can be found in the common files of a Microsoft Dynamics SL Installation.

Microsoft Dynamics SL SharePoint Client Methods

CreateDocumentLibrary

This function creates a new document library on an existing SharePoint site. Call GetLastError() if this function returns False to learn the reason for failure.

Syntax

CreateDocumentLibrary(rootSiteURL, subsiteURL, documentLibraryTitle, documentLibraryDescription, templateName) As Boolean

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required or optional</th>
<th>Other details</th>
</tr>
</thead>
<tbody>
<tr>
<td>rootSiteURL</td>
<td>Required</td>
<td>Absolute URL to the SharePoint root site. Example: <a href="http://server/">http://server/</a></td>
</tr>
<tr>
<td>subsiteURL</td>
<td>Required</td>
<td>Relative URL to the SharePoint site on the SharePoint root site.</td>
</tr>
<tr>
<td>documentLibraryTitle</td>
<td>Required</td>
<td>Unique title applied to the new document library.</td>
</tr>
<tr>
<td>documentLibraryDescription</td>
<td>Required</td>
<td>Concise information about the new document library.</td>
</tr>
<tr>
<td>templateName</td>
<td>Optional</td>
<td>Internal name of the new document library’s template. The default site template is used if TemplateName is Nothing or “”.</td>
</tr>
</tbody>
</table>
Example

Dim Result As Boolean
Dim RootSite As String = Me.RootSite.Text
Result = CreateDocumentLibrary( RootSite, “CustomerC300”, “Customer C300 Documents”, “This is a document library dedicated to Customer C300”, ”)
If Result = True then
    MsgBox( “Library “ + RootSite + “/” + “CustomerC300” + “/” + “Customer C300 Documents”+ “ was successfully created” )
Else
    MsgBox( “Error in Creating Library. “ + GetLastError () )
End If

CreateSite

This function creates a new SharePoint site on the server. The Web Service call returns True if the site was created successfully and False if the creation fails for any reason. Call GetLastError() to learn the reason for failure.

Syntax

CreateSite( RootSiteURL, SiteRelativeURL, SiteTitle, SiteDescription, TemplateName, InheritParentPermissions ) As Boolean

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required or optional</th>
<th>Other details</th>
</tr>
</thead>
<tbody>
<tr>
<td>rootSiteURL</td>
<td>Required</td>
<td>Absolute URL to the root SharePoint site. Example: <a href="http://server/">http://server/</a></td>
</tr>
<tr>
<td>siteRelativeURL</td>
<td>Required</td>
<td>Relative URL to the SharePoint site on the root SharePoint site.</td>
</tr>
<tr>
<td>siteTitle</td>
<td>Required</td>
<td>Unique title applied to the new site.</td>
</tr>
<tr>
<td>siteDescription</td>
<td>Required</td>
<td>Concise information about the new site.</td>
</tr>
<tr>
<td>templateName</td>
<td>Optional</td>
<td>Internal name of the new site’s template. The set of possible values for this field is returned by GetTemplateList. A blank site template is used if TemplateName is Nothing or “”.</td>
</tr>
<tr>
<td>inheritParentPermissions</td>
<td>Required</td>
<td>Determines whether the new site obtains its permissions from its parent site. Value is True or False.</td>
</tr>
</tbody>
</table>

Example

Result = CreateSite( “http://SPServer”, “CustomerC300”, “Customer C300 Site”, “This is a site dedicated to Customer C300”, “”, True )
If Result = True then
    MsgBox( “Site was successfully created” )
Else
    MsgBox( “Error in Creating Site. “ + GetLastError () )
End If

DoesSiteExist

This function returns true if the root SharePoint site has a subsite with the given URL, or false if it does not.
To check for a root site, set subsiteURL to Nothing or "/". The call returns true if the SharePoint root site exists, it has a subsite with the given URL, and the user has permission to access the subsite. Otherwise, it returns false.

**Syntax**

```vba
DoesSiteExist(rootSiteURL, subsiteURL) As Boolean
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required or optional</th>
<th>Other details</th>
</tr>
</thead>
<tbody>
<tr>
<td>rootSiteURL</td>
<td>Required</td>
<td>Absolute URL to the SharePoint root site. Example: <a href="http://server/">http://server/</a></td>
</tr>
<tr>
<td>subsiteURL</td>
<td>Required</td>
<td>Relative URL to the SharePoint site on the root SharePoint site.</td>
</tr>
</tbody>
</table>

**Example**

```vba
Dim Results As Boolean
Dim RootSite As String = "Http://SPServer"
Results = DoesSiteExist(RootSite, "Site1")
If Results Then
    MsgBox("Site Exists")
Else
    MsgBox("Site Does Not Exist. Error “ + GetLastError() )
End If
```

**DoesSPListExist**

This function returns true if the SharePoint site has a list that has the given URL and the user has permission to access the site; otherwise, the call returns false.

To check for a list on the root site, set subsiteURL to Nothing or "/". The call returns true if the SharePoint site has a list that has the given URL. Otherwise, the call returns false.

**Syntax**

```vba
DoesSPListExist(rootSiteURL, subsiteURL, listName) As Boolean
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required or optional</th>
<th>Other details</th>
</tr>
</thead>
<tbody>
<tr>
<td>rootSiteURL</td>
<td>Required</td>
<td>Absolute URL to the root SharePoint site. Example: <a href="http://server/">http://server/</a></td>
</tr>
<tr>
<td>subsiteURL</td>
<td>Required</td>
<td>Relative URL to the SharePoint site on the root SharePoint site.</td>
</tr>
<tr>
<td>listName</td>
<td>Required</td>
<td>Name of the list on the SharePoint site.</td>
</tr>
</tbody>
</table>

**Example**

```vba
Dim Results As Boolean
Dim RootSite As String = "Http://SPServer"
Results = DoesSPListExist(RootSite, "Site1", "lib1")
If Results Then
    MsgBox("Document Library Exists")
Else
    MsgBox("Document Library Does Not Exist" + GetLastError() )
```
End If

**GetLastError**
Retrieves the message text from the most recent exception that prevented a call from succeeding.

**Syntax**
```
GetLastError() As String
```

**Example**
```
Result = CreateSite( "http://SPServer", "CustomerC300", "Customer C300 Site", "This is a site dedicated to Customer C300", ",", True )
If Result = False then
    MsgBox( "Error in Creating Site. " + GetLastError() )
End If
```

**GetSPListTemplateList**
This function returns a two-dimensional array of strings describing SharePoint site list templates stored on the server. Each element in the array has three parts:
- **Name** — Display name of the template.
- **Description** — Concise details about the template.
- **ID** — Internal identifier of the template. This ID should be used in the CreateDocumentLibrary Web service call.

For example, an array containing the strings ""Document Library"", "Create a document library."", and ""doclib"" could comprise one element of the array.

Any errors that occur will cause this function to return a zero-length array. Call GetLastError() to get the reason for failure.

**Syntax**
```
GetSPListTemplateList(RootSiteURL, SubSiteURL) As String()()
```

**Example**
```
This call retrieves a list of document library templates from a server and places it in a list box.
Dim Result As String()()
Result = GetSPListTemplateList("http://SPServer", ")
For Each res as String() In Result
    Listbox1.Items.Add("" + res(0) + ") + vbTab + _
    "" + res(1) + ") + vbTab + "" + res(2) + ")"
Next
```

**GetTemplateList**
This function returns a two-dimensional array of strings describing SharePoint site templates stored on the server. Each element in the array has three parts:
- **Name** — Display name of the template.
- **Description** — Concise details about the template.
- **ID** — Internal identifier of the template. This ID should be used in the CreateSite Web service call.

For example, an array containing the strings ""Wiki"", ""A site for a community to brainstorm and share ideas."", and ""WIKI#0"" could comprise one element of the array.

Any errors that occur will cause this function to return a zero-length array. Call GetLastError() to get the reason for failure.
Syntax
GetTemplateList(RootSiteURL) As String()()

Example
This call retrieves a list of site templates from a server and places it in a list box.
Dim Result As String()()
Result = GetTemplateList(“http://SPServer”)
For Each res as String() In Result
    Listbox1.Items.Add(“[“ + res(0) + “]” + vbTab + “
    “[“ + res(1) + “]” + vbTab + “[“ + res(2) + “]”
Next

IsSiteOrLib
This function identifies the URL pointing to an actual SharePoint site or document library. This call returns:
- “C” if the URL points to a SharePoint site
- “D” if the URL points to a document library
- “” if an error occurred in checking
- Char.MinValue if otherwise

Syntax
IsSiteOrLib( SiteUrl ) As Char
(Where SiteUrl is the absolute URL to the SharePoint site or document library (for example, http://server/). SiteUrl is required)

Example
Dim Result As Char
Dim RootSite As String = “http://SPServer”
Result = IsSiteOrLib(“Http://SPServer/Site1”)
If Result.ToString = “C” Then
    MsgBox(“Http://SPServer/Site1” + “ is a Site”)
ElseIf Result.ToString = “D” Then
    MsgBox(“Http://SPServer/Site1” + “ is a Document Library”)
Else
    MsgBox(“Error in call GetLastError() ”)
End If

UploadSLDocument (Web service)
This more complex Web service call uploads a file to a Sharepoint site, and when finished, returns file creation time and title (for example, “Friday, September 21, 2007::FileName”).
The fileContents parameter is a binary array of the information that resides in the file. Use the wrapper function to create the fileContents parameter for you.
System.Web.Services.Protocols.SoapException is thrown if:
- A site with the given root site URL, subsite, and folder does not exist
- A file with the given name already exists in the folder
- The file name is blank
System.UriFormatException is thrown if the root site URL is not valid.
ArgumentException is thrown if fileContents is null, the subsite does not exist, or the user does not have permission to access the subsite.

Syntax

UploadSLDocument ( filename, fileContents, subSite, DocLibraryName, rootSiteURL )

Parameter | Required or optional | Other details
---|---|---
fileName | Required | Name of the new file
fileContents | Required | Information that is contained in the file. This parameter is only needed for the Web service call.
subSite | Required | Relative URL of the site that will contain the file.
DocLibraryName | Required | Name of the Document library on the subsite that will contain the file.
rootSiteURL | Required | Absolute URL to the SharePoint root site. Example: http://server/.

UploadSLDocument

This less complex version of the Web service call returns a Boolean value. You supply the document’s file name and the function converts the document for uploading to the document Library. Call GetLastError() if this function returns False to get the reason for failure.

Syntax

UploadSLDocument ( rootSiteUrl, subSite, DocLibraryName, filename ) As Boolean

Parameter | Required or optional | Other details
---|---|---
rootSiteURL | Required | Absolute URL to the SharePoint root site. Example: http://server/.
subSite | Required | Relative URL of the site that will contain the file.
DocLibraryName | Required | Name of the Document library on the subsite that will contain the file.
fileName | Required | Name of the new file.

Example

Dim Result As Boolean
Dim RootSite As String = “http://SPServer"

Result = UploadSLDocument(RootSite, “site1”, “Lib1”, “C:\Test.Doc”)

If Result = True Then
    MsgBox( “Document was uploaded”)
Else
    MsgBox( “Error in uploading”)
End If
Implementation Examples

Creating a Site
The basic procedure for employing a Doc Share solution for a new entity type is:

1. Add custom code to create the SharePoint site if it does not already exist.
2. Create a document library if needed.
3. Add custom code to upload documents for the entity.

The code example below uses a maintenance application to create a site. Before you use this code, be sure to add a reference to Microsoft.Dynamics.SharepointClient.dll in your project. This file is included in the Microsoft Dynamics SL Common Files.

Example
Dim Result As Boolean
If ( (DoesSiteExist("http://SPServer", "Manager" + _
    bManager.ID ) = False )) Then
    Result = CreateSite("http://SPServer", _
        "Manager" + bManager.ID, _
        "Manager Site", "This is a site dedicated to Manager", "", True)
    If Result = True Then
        MsgBox("Site was successfully created")
    Else
        MsgBox("Error in Creating Site. " + GetLastError())
    End If
End If
Uploading a Document

The code example below employs document uploading for the new entity. Be sure to add a reference to Microsoft.Dynamics.SL.SharepointClient.dll in your project. This file is included in the Microsoft Dynamics SL Common Files.

Example 1

Dim Result As Boolean
Dim ResultDoc As Short
' Check for Site Existance
If ( (DoesSiteExist("http://SPServer", "Manager" + _
    bManager.ID ) = True )) Then
    If (DoesSPListExist("http://SPServer", _
        "Manager" + bManager.ID, "Documents") = False) Then
        ' If the Document Library Does NOT exist, create it
        Result = CreateDocumentLibrary("http://SPServer", _
            "Manager" + bManager.ID, "Documents", "This is a Library", "")
        If Result Then
            MsgBox("Library " was successfully created")
        End If
    Else
        Result = True
    End If
    ' If the site now exists, then upload the Document
    If Result = True Then
        ResultDoc = UploadSLDocument("http://SPServer", _
            "Manager" + bManager.ID, "Documents", "C:\Document.Docx")
        If ResultDoc = True Then
            MsgBox("Document was uploaded")
        Else
            MsgBox("Error in uploading")
        End If
    Else
        MsgBox("Error in Creating Library. " + GetLastError () )
    End If
End If

Example 2

Dim ResultDoc As Short
ResultDoc = UploadSLDocument("http://SPServer", _
    "Manager" + bManager.ID, "Documents", "C:\Document.Docx")
If ResultDoc = True Then
    MsgBox("Document was uploaded")
Else
    MsgBox("Error in uploading")
End If
Appendix D: Microsoft Dynamics SL SDK Application Upgrade Utility

Use the Microsoft Dynamics SL SDK Application Upgrade Utility to upgrade your Microsoft Dynamics SL SDK applications for compatibility with the current release. This utility converts applications that were written for Microsoft Dynamics SL 7.0 or later versions.

If your Microsoft Dynamics SL SDK application was created with the Microsoft Dynamics SL 6.5 SDK or earlier versions, you must first convert the application to the Microsoft Dynamics SL 7.0 SDK. To do this, use the Visual Basic .NET Conversion toolkit that was delivered with Microsoft Dynamics 7.x, 2011, or 2011 FP1. Once the application is converted, you can then use this utility.

The Microsoft Dynamics SL SDK Application Upgrade Utility performs the following tasks:

- Converts all the SAF.OCX control entries to Microsoft.Dynamics.SL.Controls.DLL entries.
- Replaces all the Microsoft.VisualBasic.Compatibility entries possible. Some API cannot be directly replaced and require refactoring by the developer (for example, replacing Drive and Path prompts).

**Note:** The use of this tool requires the .NET Framework 4 or 4.5.

Before running the utility, review the following information:

- Available Upgrade Options
- Steps to Upgrade Applications
- Post Upgrade Issues and How to Fix Them
Available Upgrade Options

Visual Basic 6.0 Compatibility Namespace

The Microsoft Dynamics SL SDK Application Upgrade Utility provides options to help you remove Visual Basic 6.0 (VB6) compatibility classes and methods. These are obsolete with a compiler warning in .NET Framework 4 and .NET Framework 4.5. VB6 Compatibility Namespace options are optional.

VB6 Compatibility Namespace options include the following:


  There are issues with this naïve conversion that must be understood and considered before you run this part of the conversion.

  - In VB6, and with the replacement VB6.Format, the expression parameter is converted to the type that the format string is expecting. In the .NET Framework, the .ToString is a method of the type being formatted. Because of this, strings cannot be formatted like numbers or dates, numbers cannot be formatted to dates, and dates cannot be formatted as numbers by using .ToString. Because the compiler does not know the type that the format string is expecting, it will not display a warning or error in the case of numbers to dates and dates to numbers, but with the case of strings to either you will receive a warning that the arguments to String.ToString are incorrect.

  - VB6.Format also allows for padding/aligning strings by using the at sign (@) and the ampersand (&). The closest equivalent is the .PadLeft method and .PadRight method of the String class.

  - .ToString methods are case-sensitive in the .NET Framework with their format string. Each format string must be evaluated to make sure that it works as intended in the .NET Framework.

• **Safe Remove VB6.Format** – Only performs the .ToString() part of the conversion for calls without a second parameter.

• **Remove TwipsToPixels** – Removes all calls VB6.TwipsToPixels and VB6.PixelsToTwips. “VB6.TwipsToPixels(value)” will come out as just “value”. Any hard-coded twips values should be manually converted before conversion. There are 15 twips per pixel at 96 DPI (The Windows default for a desktop). Usage of VB6.TwipsPerPixel is unaffected by this conversion and must be removed manually.

• **Remove FixedLengthStrings** – Replaces VB6.FixedLengthStrings with Strings initialized to the same size as the original. Usage of the .Value property is replaced.

  There can be issues with this conversion. If your code assigns new values to the string that are shorter than the fixed length was, and later code assumes the string is the fixed size, the code must be manually modified.

• **Remove Control Arrays** – Tries to remove all usage of VB6 control arrays. Some usage of control arrays such as referring to an element with an index variable cannot be automated and must be refactored manually. These will have the text “_REFACTOR_NEEDED” appended to the variable name and results in a compile error.

• **Remove GetItemString** - Replaces calls to VB6.GetItemString(control, index) with control.Items(index).ToString(). VB6.GetItemData is not affected by this conversion and must be corrected manually.

• **Replace TabLayout** – Replaces calls to VB6.TabLayout(...) with String.Join(vbTab, ...).

• **Remove Assembly Reference** – Removes the VB6 Compatibility library reference and namespace import from converted .Vbproj files. If the Visual Basic source files contain imports for this namespace they will not be affected by this option.
Replace SDK Controls
Replaces references to SAF controls with the new DSL controls.
This option replaces fully qualified references with the new version and removes references to Interop.SAF and Interop.SAFTypeLib in the .Vbproj file. If the project does not contain a reference to Microsoft.Dynamics.SL.Controls, it is added. The ResX file is parsed for OcxStates, properties contained in those states is inserted into the designer file, and the state is removed from the ResX File.

Example:
Interop.SAF.SAFMaskedText is replaced with Microsoft.Dynamics.SL.Controls.DSLMaskedText.

Note: This option is required for an application to be compatible with the current release.

Remove SDK Control Arrays
Performs a similar conversion to removing VB6 Control Arrays. All SDK control arrays are removed. The text "_REFACTOR_NEEDED" is appended to references to the control arrays that could not be automatically converted. If this option is not selected then existing SAF control arrays are left alone and compiler errors occur.

Clear Upgraded Control Fonts
Instructs the converter not to include the Font property when expanding the .OcxState into the designer file. If no font is specified it is inherited from the control’s parent. The converter does not know the control’s parent at conversion time. Therefore, it cannot make an informed decision whether the font should be serialized to the designer.
Steps to Upgrade Applications

To upgrade Microsoft Dynamics SL SDK applications:

1. Install Microsoft Dynamics SL SDK and Microsoft Dynamics SL Legacy Controls from the install menu.
2. Open DynamicsSLSDKConversion.exe, that is located in Program Files (x86) | Microsoft Dynamics | SL | Applications | VT | Code Converter. Microsoft Dynamics SL SDK Application Upgrade Utility appears.

3. Click Browse to locate the folder that contains the project that you want to convert. Browse for Folder appears.
4. Select the folder, and then click OK. Or, type the location of the project that you want to convert in Top Directory.
5. Click Start Processing. The Visual Basic project that is being converted appears in Project. Process will be green as it processes the projects.
6. After the process is complete, a message occurs that indicates the number of projects converted.
7. Review the results of the upgrade shown in the Log area.
8. Close the upgrade utility window.
Post Upgrade Issues and How to Fix Them

Updating the Target Framework
After you have converted the projects and upon loading the projects into Visual Studio, you may see errors that Solomon.Kernel and Microsoft.Dynamics.SL.Controls are not available. This is caused by the fact that the VB project is still targeting the 2.0 framework. You will need to modify the properties of the project to change the Target Framework to .NET Framework 4.

Unsupported Control Arrays
You must manually correct VB6 control arrays that are accessed by variable index. This work can be very extensive.

ToString does not Accept that Number of Arguments
The replacement for VB6.Format does not include parameters for FirstDayOfWeek or FirstWeekOfYear. Remove these parameters. If the application is using them, you may have to refactor the code to achieve the result that you want.

Overlapping Option Buttons and Check Boxes
The DSLOption and DSLCheck controls are based on the .NET Framework RadioButton and CheckBox controls. They are wider and have additional padding. The converter increases the width of some controls during conversion to avoid clipping text. You can change the AutoSize property to help keep the controls in line if they are taking up too much screen real estate.

Missing Ctlset_Text and Similar Methods on New Controls
Replace these calls with accessors to the equivalent property on the new control.
Appendix E: Running the .NET Framework-connected Applications from a Network

To run a .NET Framework-connected application from a network, you need to set up the correct permissions on the client machine. This can be accomplished with the Code Access Security Policy tool (Caspol.exe), which is part of the .NET Framework. For more information about this tool, see “Code Access Security Policy Tool (Caspol.exe)” on the Microsoft MSDN Library website: http://msdn2.microsoft.com/en-us/library/cb6t8dtz.aspx

The applications and assemblies delivered by Microsoft are granted the correct permissions (Full Trust) by the Microsoft Dynamics SL client installation. All of the applications and assemblies from Microsoft have strong names and are digitally signed. The permissions set by the client installation grant access to all assemblies signed with a particular key (the one Microsoft used). Your applications and assemblies will not be signed the same, so our permissions will not work for your applications.

You will need to set up Full Trust permissions for your applications. There is sufficient information on the MSDN website for you to accomplish this.

Some of the options are:

- Grant permissions to all assemblies in a particular share.
- Grant permissions to all assemblies signed with a particular key (your key).
- Grant permissions to individual assemblies.
Appendix F: Customizing Role Center Data

A select portion of Role Center items can be modified by changing the data that is used to describe each part. Parts that can be changed include the Activities and the Quick Lists. The following sections describe how to change these parts to add custom items.

Adding Activities entries

Activities are typically categorized by Module. These categories are called Activity Groups. Within each Activity Groups listing there are Activity Cues that typically represent the pending work a user or group has. Each Activity Cue can be set to link an action which includes executing a Dynamics SL Screen/Query, an external program, URL or a Quick List.

![Figure 3: Activities Role Center](image)
Adding an Activity Group

1. Define the data. To do this, you need to understand the fields in the RCActivity table.

<table>
<thead>
<tr>
<th>RCActivity Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>A unique value that identifies the Activity Group</td>
</tr>
<tr>
<td>Description</td>
<td>The displayed description for the Activity Group shown in tooltip fashion.</td>
</tr>
<tr>
<td>Title</td>
<td>The title of the Activity Group that shows in the grouping container.</td>
</tr>
<tr>
<td>Links</td>
<td>Future requirement.</td>
</tr>
<tr>
<td>ID</td>
<td>A unique value that identifies the Activity Group</td>
</tr>
</tbody>
</table>

**Example**

To add an Order Activity Group to the Activities part, define and format the data as follows, placing the data in a .csv file.

```
OM,Order Activities,Order Management, ,
```

2. Import the data into the system database by using *Database Update* (98.100.00). For more information about this screen, see “Updating Databases” and “Database Update (98.100.00)” in the System Manager Help or user’s guide.

**Note:** The RCActivity.csv file contains the standard Microsoft Dynamics SL Activity Groups. This file is a good resource to understand the required file layout.
Adding an Activity Cue

1. Define the data. To do this, you need to understand the fields in the RCCue table.

<table>
<thead>
<tr>
<th>RCCue Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActivityID</td>
<td>A value that identifies to which Activity Group the Activity Cue belongs.</td>
</tr>
<tr>
<td>Description</td>
<td>The displayed description for the Activity Cue shown in tooltip fashion.</td>
</tr>
<tr>
<td>ID</td>
<td>A unique value that identifies the Activity Cue.</td>
</tr>
<tr>
<td>Parms</td>
<td>Include specific parameters to the related SQL statement.</td>
</tr>
<tr>
<td></td>
<td>- 0 = None</td>
</tr>
<tr>
<td></td>
<td>- 1 = Microsoft Dynamics SL user</td>
</tr>
<tr>
<td></td>
<td>- 2 = Company ID</td>
</tr>
<tr>
<td></td>
<td>- 3 = Microsoft Dynamics SL user and Company ID</td>
</tr>
<tr>
<td></td>
<td>- 4+ = System Database (4, 5, 6, 7)</td>
</tr>
<tr>
<td>SQL</td>
<td>SQL stored procedure used by the Activity Cue. The SQL statement needs to return an aggregate value (count).</td>
</tr>
<tr>
<td>Status</td>
<td>Activity Cue status indicator to display.</td>
</tr>
<tr>
<td></td>
<td>- 0 = Normal (no indicator)</td>
</tr>
<tr>
<td></td>
<td>- 1 = Informational</td>
</tr>
<tr>
<td></td>
<td>- 2 = Warning</td>
</tr>
<tr>
<td></td>
<td>- 3 = Error</td>
</tr>
<tr>
<td>Title</td>
<td>The title of the Activity Cue that shows below the cue stack.</td>
</tr>
<tr>
<td>Links</td>
<td>Any linked programs relationships to the Activity Cue. The format of the link is as follows: <code>&lt;title&gt;;&lt;linktype&gt;;&lt;link&gt;</code></td>
</tr>
<tr>
<td></td>
<td>- <code>&lt;title&gt;</code> - For future use. Currently set to LT for QuickList.</td>
</tr>
<tr>
<td></td>
<td>- <code>&lt;linktype&gt;</code> - For future use. Type of link.</td>
</tr>
<tr>
<td></td>
<td>- 1 = Start Microsoft Dynamics SL program</td>
</tr>
<tr>
<td></td>
<td>- 2 = Navigate to a URL</td>
</tr>
<tr>
<td></td>
<td>- 3 = Call a related Quick List</td>
</tr>
<tr>
<td></td>
<td>- <code>&lt;link&gt;</code> - The executing link.</td>
</tr>
</tbody>
</table>

Example:

To list Sales Orders using the existing Quick Query, define and format the data as follows, placing the data in a .csv file.

```
OM, Sales Orders, SALESORD, 2, select COUNT(*) from QQ_soheader where [Company ID] = , 2, Sales Orders, XX;1; QQVIE00 QQ_soheader,
```

The following are examples of the format you should use for the RCCue.Links field:

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call Quick Query for Sales Orders</td>
<td>XX;1; QQVIE00 QQ_soheader</td>
</tr>
<tr>
<td>Open Sales Orders (40.100.00)</td>
<td>XX;1; 4010000</td>
</tr>
<tr>
<td>Open Notepad</td>
<td>YY;2; file://\C:\windows\notepad.exe</td>
</tr>
<tr>
<td>Open a Microsoft Excel file</td>
<td>ZZ;2; file://shareServer/Documents/spread1.xlsx</td>
</tr>
</tbody>
</table>

2. Import the data into the system database by using Database Update (98.100.00). For more information about this screen, see “Updating Databases” and “Database Update (98.100.00)” in the System Manager Help or user’s guide. When you use Database Update (98.100.00) to
process a .CSV file, a corresponding .Imp file that has the same name must exist for Database Update (98.100.00) to know into which table/field to import the data.

**Note:** The RCCue.csv file contains the standard Microsoft Dynamics SL Activity Cues. This file is a good resource to understand the required file layout. This file can be found in: \Microsoft Dynamics\SL\Applications\DB\common\sys.

3. After the custom Activity is added, give access rights to the users, including the System Manager, to view the new Activity.
Adding Quick List Entries

Quick Lists are associated with an Activity Cue. The Activity Cue that has focus will be what list is showing in the Quick List area (if that association exists). You can set up the Quick List to allow a line item in the Quick List to open a specific Microsoft Dynamics SL screen to show you the information from the line item. Quick Lists are intended for small set of rows. More complex levels of information may require using Quick Query.

![Quick List - Past due payments](image)

*Figure 4: Activities Role Center*
Adding a Quick List

1. Define the data. To do this, you need to understand the fields in the PVRec table.

<table>
<thead>
<tr>
<th>PVRec Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>Ignored for Quick Lists, leave empty.</td>
</tr>
<tr>
<td>PVID</td>
<td>Quick List ID</td>
</tr>
<tr>
<td>QMProg</td>
<td>Linked Microsoft Dynamics screen ID, in the same format as the Screen.ScreenID.</td>
</tr>
<tr>
<td>RecType</td>
<td>Ignored for Quick Lists, leave empty.</td>
</tr>
</tbody>
</table>
| ProcAnFieldInfo| Defines SQL stored procedure and column information to display in the Quick List. In the following format:

- SQL selection statement. Runtime replacement of special patterns exists for Company (@companyid) and Microsoft Dynamics SL user ID (@userid). The columns returned by the SQL statement should match the columns defined for the Field definitions.

- Parameter information separated by semicolons (;). If only one parameter entry exists (no semicolons) then this information is ignored and the first column only is passed. Fixed values for each parameter or a replacement for the company ID (@companyid) or Microsoft Dynamics SL user ID (@userid). The first column is always passed as the last parameter.

- Field definitions (repeats, maximum of four columns)
  - Table.Fieldname in SQL table
  - Column description to display
  - Field class value
    Note: The field class is currently ignored in Quick Lists; however, it is a required field.
  - Alternate sort display (ignored in Quick List)

Note: There are limits on the data types supported. Most string types (char, varchar, etc.) are supported. Image, text, long varchar, time, and boolean are not supported.

Example

To list Sales Orders, define and format the data as follows, placing the data in a .csv file.

```
"QL_SALESORD","4010000","","SELECT
SOHeader.OrdNbr,SOHeader.OrdDate,SOHeader.CustID,Customer.Name FROM
SOHeader with (nolock) LEFT OUTER JOIN Customer with (nolock) ON
SOHeader.CustID = Customer.CustId where SOHeader.CpnyID =
@companyid","@companyid",""SOHeader.OrdNbr;Order
Number;0;0",""SOHeader.OrdDate;Order Date;0;0",""SOHeader.CustID;Customer
ID;0;0",""Customer.Name;Customer Name;0;0"
```

2. Import the data into the system database by using Database Update (98.100.00). For more information about this screen, see “Updating Databases” and “Database Update (98.100.00)” in the System Manager Help or user’s guide.

Note: The PVRec.csv file contains the standard Microsoft Dynamics SL Quick Lists. This file is a good resource to understand the required file layout. The location of this file is \Microsoft Dynamics\SL\Applications\DB\csvs.

If the name of the file is anything other than pvrec.csv the Possible Values Import (PV.REC.00) screen, see “Importing Messages and Possible Values Lists” in the System Manager Help or user’s guide.

3. After the custom Quick List is added, give access rights to the users, including the System Manager, to view the new Quick List.
Adjusting the Display Behavior for Role Centers

You can control the height and column location of the three parts displayed for Role Centers. The RCPart table holds this information. Fields that can be changed to adjust display behaviors are:

- **ColPref** – Holds the column preference. This should be either 0 or 1.
- **Height** – The height of the part in pixels. Setting this value to 0 for the RC ACT part will produce sizing based upon the number of Cues inside the part.
- **LayoutOrder** – Used in conjunction with the column preference can control which part is displayed on top.
- **Scrollable** – Is this part fixed or scrolling capable. This should be either 0 for fixed and 1 for scrolling capable.
## Data Definitions

### RCAActivity

*Database Maintenance (98.290.00)* uses RCAActivity.csv for Microsoft Dynamics SL standard content.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Field Description</th>
<th>Data Type</th>
<th>Value Constraints</th>
<th>Required</th>
<th>Default Value</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>ID</td>
<td>String (unique)</td>
<td>Length 10</td>
<td>Yes</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Description</td>
<td>Description</td>
<td>String</td>
<td>Length 255</td>
<td>No</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Title</td>
<td>Title</td>
<td>String</td>
<td>Length 40</td>
<td>Yes</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Links</td>
<td>Links</td>
<td>Varchar</td>
<td>Length 4096</td>
<td>Future Use</td>
<td>None</td>
<td>Future Use</td>
</tr>
</tbody>
</table>

### RCCue

*Database Maintenance (98.290.00)* uses RCCcue.csv for Microsoft Dynamics SL standard content.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Field Description</th>
<th>Data Type</th>
<th>Value Constraints</th>
<th>Required</th>
<th>Default Value</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActivityID</td>
<td>Activity ID</td>
<td>String</td>
<td>Length 10</td>
<td>Yes</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Description</td>
<td>Description or Tooltip</td>
<td>String</td>
<td>Length 255</td>
<td>No</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>ID</td>
<td>ID (unique)</td>
<td>String</td>
<td>Length 10</td>
<td>Yes</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Parms</td>
<td>Include SQL Parameters for the Microsoft Dynamics SL user ID or company ID</td>
<td>Short</td>
<td>0, 1, 2, 3, (4+), {0-7}</td>
<td>No</td>
<td>None</td>
<td>0=None, 1= User ID, 2=Company, 3=Both 1&amp;2, 4+=System DB</td>
</tr>
<tr>
<td>SQL</td>
<td>SQL statement</td>
<td>String</td>
<td>Length 255</td>
<td>Yes</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Status</td>
<td>Cue Status</td>
<td>Short</td>
<td>0, 1, 2, 3, 4</td>
<td>Yes</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Title</td>
<td>Cue Title</td>
<td>String</td>
<td>Length 40</td>
<td>Yes</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Links</td>
<td>Line Data</td>
<td>Varchar</td>
<td>Length 4096</td>
<td>No</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>

### Link data for RCCue table.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Value Constraints</th>
<th>Required</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link Title</td>
<td>String</td>
<td>Length 30</td>
<td>Yes</td>
<td>Future Use</td>
</tr>
<tr>
<td>Line Type</td>
<td>String</td>
<td>Length 1</td>
<td>Yes</td>
<td>Screen Number, External Link, URL, Part Information</td>
</tr>
<tr>
<td>Link Value</td>
<td>String</td>
<td>Length 255</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
**RCPart**

*Database Maintenance (98.290.00)* uses RCPart.csv for Microsoft Dynamics SL standard content.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Field Description</th>
<th>Data Type</th>
<th>Value Constraints</th>
<th>Required</th>
<th>Default Value</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>.dll component</td>
<td>Assembly Name</td>
<td>String</td>
<td>Length 25</td>
<td>Yes</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>name</td>
<td>Part Class Name</td>
<td>String</td>
<td>Length 128</td>
<td>Yes</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>ColPref</td>
<td>Column Preference</td>
<td>Short</td>
<td>0, 1</td>
<td>Yes</td>
<td>0</td>
<td>0, 1</td>
</tr>
<tr>
<td>Description</td>
<td>Description or Tooltip</td>
<td>String</td>
<td>Length 255</td>
<td>No</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Height</td>
<td>Part default height</td>
<td>Short</td>
<td>1-32767</td>
<td>Yes</td>
<td>None</td>
<td>1-32767</td>
</tr>
<tr>
<td>ID</td>
<td>Part ID</td>
<td>String</td>
<td>Length 10</td>
<td>Yes</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Order</td>
<td>Layout Order</td>
<td>Short</td>
<td>0-10</td>
<td>Yes</td>
<td>None</td>
<td>0-10</td>
</tr>
<tr>
<td>Scroll</td>
<td>Part Scrollable</td>
<td>Short</td>
<td>0, 1</td>
<td>Yes</td>
<td>0</td>
<td>0, 1</td>
</tr>
<tr>
<td>Title</td>
<td>Part Title</td>
<td>String</td>
<td>Length 40</td>
<td>Yes</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>
Index

.NETFramework-connected application 325

C
caspol.exe 325
Code Inspector
  changing the defaults for processing 41
  log file 44
Creating
  Indexes 38
  stored procedures 39
  tables 37, 38

D
DBBUILD.ini
  [Update Scenarios] section 27
Developer guidelines
  database considerations 9
  Microsoft Dynamics SL SDK 4
  screen design 5
Development environment 4
Doc Share
  Active field 306, 308
  designing applications 303
  DocumentID value 306, 308
  entity types 308
  Instance value 306, 308
  SharePoint 310
  SharePoint Site Configuration
    (98.360.00) 304, 306, 308
    SLTypeDesc value 304
    SLTypeID value 304
  terminology 303

E
entities, Doc Share 303

F
Fieldclass
  Flex Key 70
  Non-Flex Key 70, 72
Functions
  InitLocalizationSupsystem 197
  LoadStr 205

I
InitLocalizationSupsystem function 197

L
LoadStr function 205
Localize statement 206

M
Microsoft Dynamics SL SDK
  developer guidelines 4

N
Non-Flex Key 72

R
Role Center 327

S
Statements
  Localize 206

T
Trigger property 6, 92, 98

V
Visual Basic development environment 4