Smalltalk MT™ *Visual Components*

32 Bit Development Environment for Microsoft Windows™
Version 3

Object Connect

Object Connect, 42 rue de Tauzia, 33800 Bordeaux, France

Document No. ST12010002

Contents

BEFORE YOU BEGIN XV
How to Use This Guide ..... *XV*
Target Audience ..... *XVI*
Notational Conventions ..... *XVI*
Features Overview ..... *XVII*
Smalltalk MT Editions ..... *XVIII*
System Requirements ..... *XIX*
Finding Information about Smalltalk MT ..... *XIX*

CHAPTER 1 THE DEVELOPMENT ENVIRONMENT 1

SETUP 2
GETTING STARTED 3
Starting up Smalltalk MT *3*
Workspaces *4*
Context of Evaluation *8*
Filing in Smalltalk Code *8*
Customizing the User Interface *10*
DEVELOPMENT TOOLS 13
Overview *13*
Browsing Source Code *13*
Class Hierarchy Browser *17*
Debugger *23*
Image Properties *26*
Inspectors *32*
Method Explorer *35*
Project Browser *36*
Symbol Editor *50*
Transcript Window *52*
Workspace Windows *53*
Thread Viewer *53*
IMAGE MAINTENANCE 55
Overview *55*
Saving an Image *55*
Command Line Options *56*
Modifying the Import Section *56*
Virtual Memory Settings *62*
Section Sizes *62*
Process Properties *66*
Restoring an Image *68*

TROUBLESHOOTING....72
Development Environment....*72*
Problems Generating an Executable Image....*74*
Debugging Issues....*76*
Image Corruption....*80*

CHAPTER 2 THE INTERFACE BUILDER....83

FEATURES OVERVIEW....84
VISUAL EDITING....86
Functional Overview....*86*
Application Windows....*86*
Dialog Boxes....*88*
Menus....*89*
Identifiers....*90*
WORKING WITH CHILD CONTROLS....92
Overview....*92*
Selecting Controls....*92*
Inserting Controls....*93*
Duplicating and deleting Controls....*93*
Control Placement....*93*
Control Properties....*96*
Framing Parameters....*96*
Tab Ordering....*96*
Mnemonics....*97*
USING RESOURCE SCRIPTS....98
Why use Resources?....*98*
Standard Resources....*98*
How to generate Resources....*100*
How to customize Resource Templates....*101*
How to export Resources....*101*
Example....*101*
WINDOW REFERENCE....109
Code Generation....*109*
Frame Windows....*111*
Dialog Boxes....*117*
Controls....*118*
Splitter Bars....*123*
Control Extensions....*126*
GUI BUILDER TUTORIAL....129
A Generic Application....*129*
Toolbar Sample....*138*
Slider, Progress and Spin Sample....*141*
Splitter Bar Sample....*144*
GUI BUILDER ISSUES AND LIMITATIONS....146

CHAPTER 3 BASE CLASS LIBRARIES....149

BASIC CONCEPTS....150

Objects *150*
Classes *150*
Methods *152*
Messages *153*
Variables and Namespaces *158*
Literals *162*
Control of flow Statements *169*
Source Management *174*
CORE CLASSES 178
Abstract *178*
Behavior Classes *178*
Collections *179*
Number Classes *187*
Streams *191*
String Classes *193*
System Classes *194*

CHAPTER 1 THE WINDOW FRAMEWORK 201

OVERVIEW 202
Windows Messages *202*
Events *207*
THE GUI FRAMEWORK 210
Windows Events *210*
File Handling *215*
Using Accelerators *217*
Using the Registry *219*
Implementing Online Help *221*
Process Classes *224*
Graphics Classes *230*

CHAPTER 2 WINDOW CLASSES 233

WINDOW 234
Window Class Attributes *234*
Registering a Window Class *235*
Creating a Window *237*
Getting the Window Handle *237*
Enumerating and finding Windows *237*
Window Items *238*
Window Properties *239*
Window Subclassing *240*
OVERLAPPED WINDOWS 241
Overview *241*
Initialization and Release Messages *241*
FrameWindow *242*
DialogBox *245*
Common Dialog Boxes *254*
MDI (Multiple Document Interface) *259*

SDIFrame and WinDocument *260*
Property Sheets and Wizards *261*
Tabbed Dialogs *265*
Predefined Dialogs *265*
Menus *270*
CONTROLS 272
Event Notifications *272*
Buttons *273*
List Controls *273*
Edit Controls *275*
Scroll Boxes *275*
COMMON CONTROLS LIBRARY 277
Common Control Styles *277*
Application-defined Data *278*
ComboBoxEx *278*
DateTimePicker *278*
DragListBox *279*
HeaderControl *281*
HotKey *281*
ListView *282*
MonthCal *285*
ProgressBar *285*
ReBar *286*
RichEdit *287*
StatusWindow *288*
TabControl *289*
ToolBar *290*
Tooltip *294*
TrackBar *296*
TreeView *296*
UpDown *298*
SMALLTALK CONTROLS 299
Overview *299*
ContainerWindow *299*
CustomControl *299*
OleControlContainer *300*
TrayIcon *300*
SplitPane *301*

CHAPTER 3 INTRODUCTION TO OLE PROGRAMMING 303

THE COMPONENT OBJECT MODEL (COM) 304
Abstract *304*
Smalltalk Implementation *304*
Threading Models *306*
COM Class Hierarchy *308*
OLE Allocations *311*
OLE Strings *311*

OLE CONTROLS 312
Abstract *312*
OleControlContainer *312*
InPlaceFrame and OleContainerView *314*
Dispatch Interfaces *314*
OLE DATA TRANSFER 317
Abstract *317*
IDataObject *317*
Using the Clipboard *318*
IMPLEMENTING OLE DRAG AND DROP 321
Abstract *321*
Preparing an Application for Drag and Drop *321*
Processing a dragging Operation *322*
Implementing OLE File Drag and Drop *325*
Implementing Simple File Drag-Drop *327*
ACTIVEX COMPONENT FRAMEWORK 329
ApplicationProcess Entry Point *329*
Differences between in-process and stand-alone Servers *329*
InProcessServer *330*
OleControl *331*
Testing ActiveX Components *334*
TROUBLESHOOTING OLE 336

CHAPTER 4 THE DEVELOPMENT PROCESS 339

CREATING AN APPLICATION 340
What is an Application? *340*
Creating a New Project *341*
Using Resources *341*
Hello World: a simple example *343*
SampleDialog: using an external GUI builder *344*
The Generic Sample Program *345*
GENERATING EXECUTABLES 350
Overview *350*
Application Entry Point *350*
DLL Entry Point *354*
Application Resources *354*
Using Symbols at Runtime *355*
Headless Applications *356*
The Build Process *356*
CONSOLE APPLICATIONS 359
Overview *359*
Creating a Console *359*
Console Control Handlers *360*
Processing Command Line Arguments *360*
DYNAMIC LINK LIBRARIES 364
Overview *364*
DLL Exports *364*

CHAPTER 5 ADVANCED PROGRAMMING 367

INTERFACING WITH DYNAMIC LINK LIBRARIES 368
Passing Arguments *370*
Referencing Arguments *374*
Return Types *375*
Examples *377*
ANSI and Unicode Binding *384*
Using Decorated Names *384*
Aliasing API Names *385*
Callbacks *385*
EXCEPTION HANDLING 387
Overview *387*
Common Exceptions *388*
PROGRAMMING FOR ANSI AND UNICODE 393
Abstract *393*
Implementation *393*
Further Considerations *394*
OBJECT SERIALIZATION 396
Abstract *396*
Serialization Architecture *396*
Example *397*
Using Memory-mapped Objects *397*
FINALIZATION AND WEAK REFERENCES 403
Abstract *403*
Weak Pointers *403*
Finalization *404*
Interfaces *404*
Example *405*
DEBUGGING AND DIAGNOSTIC MESSAGES 407
Debugging Messages *407*
Error Events *407*
THE COMPILER INTERFACE 408
Overview *408*
Variables *409*
Browsing *409*
Categories *409*
Classes *410*
Compiling *410*
Pool Dictionaries *411*
Files *412*
Example *412*
PERFORMANCE AND CODING ISSUES 414
Abstract *414*
Real-time Applications *414*
Steps *416*

CHAPTER 6 INLINE ASSEMBLER 421

INTRODUCTION....422
SYNTAX....423
Machine Instructions....*423*
Addressing Modes....*423*
ASSEMBLER AND SMALLTALK METHODS....426
Calling Convention....*426*
Object Types....*427*
PUBLIC ASSEMBLER ROUTINES....429
_performIntSelector....*429*
_performSelector....*429*
_performUSelector....*430*
_stalloc....*430*
_isKindOf....*431*
_overflow....*431*
EXAMPLE: IMPLEMENTING A PROXY....433

APPENDIX PSEUDO MESSAGES....435

Pseudo Messages in Object....*435*
Pseudo Messages in MemoryManager class....*437*
Function and Method Calls....*438*
Type Conversion Messages....*439*
Iteration Messages....*440*

APPENDIX GLOSSARY....443

FIGURES AND TABLES

FIGURES

Figure 1-1 Unhandled Exception Dialog 5
Figure 1-2 Unhandled Exception Dialog Details 6
Figure 1-3 Debugger Window Example 7
Figure 1-4 Unhandled System Exception 7
Figure 1-5 Class Save Dialog 9
Figure 1-6 Save Image Dialog 10
Figure 1-7 IDE Preferences Page 11
Figure 1-8 Font Dialog 12
Figure 1-9 Query Dialog 14
Figure 1-10 Method Change Browser 16
Figure 1-11 Class Hierarchy Browser 17
Figure 1-12 New Symbol Dialog 20
Figure 1-13 Method Statistics Dialog 22
Figure 1-14 Debugger Window 24
Figure 1-15 Image Properties Sheet 26
Figure 1-16 Image Properties - Optimization 27
Figure 1-17 Object Inspector 32
Figure 1-18 Byte Inspector 33
Figure 1-19 Memory-mapped Object File Inspector 34
Figure 1-20 Reference Browser 34
Figure 1-21 Method Explorer Window 36
Figure 1-22 Project Browser Window 37
Figure 1-23 Build Properties – Process Page 42
Figure 1-24 Build Properties – Resources Page 43
Figure 1-25 Build Properties – Resources Page 44
Figure 1-26 Build Properties – Options Page 44
Figure 1-27 Build Properties – Output Page 46
Figure 1-28 Symbol Editor 50
Figure 1-29 Merging Pool Dictionaries 51
Figure 1-30 Transcript Window 52
Figure 1-31 Thread Viewer 53
Figure 1-32 Image Properties – Linkage 57
Figure 1-33 DLL Properties Dialog 59
Figure 1-34 Image Properties - Section Sizes 63
Figure 1-35 Image Properties - Process 67
Figure 1-36 Source Comparisons 69
Figure 1-37 Image Properties - Debugging 77
Figure 2-1 Popup Menus with Bar Break 89
Figure 2-2 Popup Menus with Column Break 89
Figure 2-3 Left Alignment 94
Figure 2-4 Vertical Centering 95

Figure 2-5 Even Spacing 95
Figure 2-6 Push Button Alignment (Right) 95
Figure 2-7 Tab Ordering Example 97
Figure 2-8 Application Properties 112
Figure 2-9 Frame Window Properties 113
Figure 2-10 Extended Frame Window Styles 115
Figure 2-11 Event Properties 117
Figure 2-12 General Control Styles 119
Figure 2-13 Extended Control Styles 120
Figure 2-14 List Contents 121
Figure 2-15 ListView Header 122
Figure 2-16 Splitter Bars 125
Figure 2-17 Control Extension Page 127
Figure 2-18 Custom Button Sample 127
Figure 2-19 Generic Dialog Box 133
Figure 2-20 Generic Dialog Box Tab Order 134
Figure 2-21 Resource Script Dialog 139
Figure 2-22 Slider, Progress and Spin Sample Dialog 141
Figure 2-23 Splitter Sample 144
Figure 2-24 Splitter Sample - Framing Parameters 144
Figure 2-25 Splitter Sample - Sample Layout 144
Figure 2-26 Splitter Sample - Proportional Framing Parameters 145
Figure 4-1 Event Handler Chain 210
Figure 4-2 Sample Help Dialog 223
Figure 5-1 Sample Dialog Layout 250
Figure 5-2 Sample Dialog at Runtime 251
Figure 5-3 Sample Dialog Events 253
Figure 5-4 Print Preview 259
Figure 5-5 Text Prompter 266
Figure 5-6 Text Prompter with ComboBox 266
Figure 5-7 ListPrompter 267
Figure 5-8 MultiListPrompter 268
Figure 5-9 CheckListPrompter 269
Figure 5-10 DateTimePicker Samples 279
Figure 6-1 OLE Property Page Sample 313
Figure 7-1 Adding Generic 346
Figure 7-2 Build Properties (Console Application) 363
Figure 8-1 API Properties 369
Figure 8-2 Image Properties - Monitoring String Conversions 395
Figure 8-3 Object File Header 398

TABLES

Table 1-1 Typographic Conventions XVII
Table 1-1 WalkBack Dialog Buttons 6
Table 1-2 Context of Evaluation 8
Table 1-3 Compilation Warnings and Errors 19
Table 1-4 Compilation Information 19
Table 1-5 CHB Accelerator Keys 23
Table 1-6 Debugger Child Windows 24
Table 1-7 Language Options 28
Table 1-8 Optimization Options 29
Table 1-9 Inspecting Objects 32
Table 1-10 Messages in Inspector Subclasses 35
Table 1-11 Project Properties 39
Table 1-12 Project Elements 40
Table 1-13 Window Project Elements 41
Table 1-14 Command Line Switches 56
Table 1-15 Import Page Fields and Actions 57
Table 1-16 Development Image Sections 65
Table 1-17 Setup Sample Sections 66
Table 1-18 General Diagnostic Messages 78
Table 1-19 OLE Diagnostic Messages 78
Table 2-1 Application Properties 112
Table 2-2 Window Properties 113
Table 2-3 Window Frame Styles 114
Table 2-4 Win32 Class Properties 114
Table 2-5 Extended Frame Window Styles 115
Table 2-6 Application Properties 117
Table 2-7 Dialog Styles 118
Table 2-8 General Control Properties 119
Table 2-9 Extended Control Styles 120
Table 2-10 Custom Control Extension Messages 127
Table 3-1 Object Terminology 150
Table 3-2 Object Instantiation Messages 152
Table 3-3 Method Terminology 153
Table 3-4 Message Types 154
Table 3-5 Smalltalk Variables and Identifiers 158
Table 3-6 Predefined Thread Local Variables 160
Table 3-7 Types of Literals 162
Table 3-8 Sample Literal Array 162
Table 3-9 String Escape Codes 165
Table3-10 Constant Expression Limitations 168
Table3-11 Boolean Expressions and Comparisons 170
Table 3-12 System Categories 175
Table 3-13 Export Categories 176
Table 3-14 Common Pool Dictionaries 177
Table 3-15 Collection Classes Overview 179
Table 3-16 Collection Access Protocol 180

Table 3-17 Collection Enumeration Protocol 180
Table 3-18 Collection Operations Protocol 181
Table 3-19 Collection Testing Protocol 181
Table 3-20 MappingTable Access Protocol 181
Table 3-21 MappingTable Enumeration Protocol 183
Table 3-22 SequenceableCollection Access Protocol 185
Table 3-23 SequenceableCollection Copying Protocol 185
Table 3-24 SequenceableCollection Enumeration Protocol 185
Table 3-25 Arithmetic Message Equivalence 187
Table 3-26 32-bit Manipulation Messages 187
Table 3-27 Extracting 16-bit Words 188
Table 3-28 Floating Point Formats under Windows NT / 98 189
Table 3-29 Parameter Types in the API Editor 190
Table 3-30 Passing C Floating Point Parameters 190
Table 3-31 Stream Subclasses 192
Table 3-32 Semaphore States 195
Table 3-33 Struct Initialization Methods 198
Table 3-34 Retrieving Struct Fields 199
Table 3-35 Setting Struct Fields 199
Table 4-1 Event Terminology 208
Table 4-2 Event Return Values 211
Table 4-3 Event Parameters 212
Table 4-4 File Handling Methods 215
Table 4-5 File Handling Events 216
Table 4-6 Required File Handling Methods 216
Table 4-7 MRU File Handling Methods 217
Table 4-8 Help Commands 222
Table 4-9 Help Methods 223
Table 4-10 ApplicationProcess Thread Protocol 226
Table 4-11 Processor Attributes Protocol 227
Table 4-12 Processor Debug Protocol 227
Table 4-13 WinApplication Attributes 228
Table 4-14 WinApplication Initialization Methods 229
Table 4-15 Miscellaneous WinApplication Methods 229
Table 5-1 Window Class Attributes 235
Table 5-2 Window Enumeration Messages 238
Table 5-3 Window Item Messages 238
Table 5-4 Window Item States 239
Table 5-5 Window Properties 239
Table 5-6 Window Init / Release Messages 241
Table 5-7 Accessing Child Windows 244
Table 5-8 Input Data Mapping 248
Table 5-9 Property Page Class Methods 262
Table 5-10 Property Page Methods 262
Table 5-11 Property Page Events and Return Values 263
Table 5-12 Prompter Variables 266
Table 5-13 Commonly used Event Notifications 272

Table 6-1 COM Threading Models 307
Table 6-2 InProcessServer Exports 330
Table 7-1 winMain Parameters 350
Table 7-2 Runtime Files 357
Table 7-3 Command Line Parameters 361
Table 7-4 DLL Export Declaration 365
Table 8-1 Import Declarations 368
Table 8-2 Optimizing API Parameter Passing 373
Table 8-3 Export Calling Conventions 386
Table 8-4 MappedObjectStream Access Modes 399
Table 8-5 MappedObjectStream Open Methods 399
Table 8-6 MappedObjectStream Messages 400
Table 8-7 Finalization Methods in Object 405
Table 8-8 Finalization Methods in ApplicationProcess 405
Table 8-9 Compiler TLS Interface 409
Table 8-10 Compiler Global Variable Interface 409
Table 8-11 Source Code Browsing Interface 409
Table 8-12 Compiler Category Interface 410
Table 8-13 Compiler Class Interface 410
Table 8-14 Compiler Interface 411
Table 8-15 Compiler Method Interface 411
Table 8-16 Compiler Pool Interface 412
Table 8-17 Compiler File Interface 412
Table 9-1 Addressing Modes 424
Table 9-2 Addressing Smalltak Globals 425
Table 9-3 Syntactical Exceptions 425
Table 9-4 Register Usage 427
Table 0-1 Object Testing 435
Table 0-2 Object Accessing 436
Table 0-3 Miscellaneous Messages 436
Table 0-4 Pointer Access Messages 436
Table 0-5 Binary Access Messages 437
Table 0-6 Address Manipulations 437
Table 0-7 MemoryManager class Messages 439
Table 0-8 Object Messages 439
Table 0-9 Type Conversion Messages 440
Table 0-10 Iteration Messages 441

Before You Begin

Welcome to Smalltalk MT, an advanced Object Oriented Programming System for Microsoft Windows. Smalltalk MT is a high-performance implementation of Smalltalk, a proven object oriented language for application development. The design of Smalltalk MT incorporates advanced features such as native multithreading, exception handling and virtual memory management that let you exploit the capabilities of modern operating systems such as Microsoft Windows NT/2000 and Windows 9x/Me.

Smalltalk MT gives you unprecedented performance and host system integration. You can deliver ActiveX components and dynamic libraries built using Smalltalk MT.

How to Use This Guide

This manual is divided into the following parts:

Chapter 1, *The Development Environment*, provides information about setting up Smalltalk MT and getting started with the development tools. Topics include browsing source code, compiling code, and image maintenance.

Chapter 2, *The Interface Builder*, describes the visual GUI builder. This chapter also covers resource scripts and executable generation from the perspective of the Interface Builder.

Chapter 3, *Base Class Libraries*, discusses basic concepts and the core class hierarchy.

Chapter 4, *The Window* Framework, presents the GUI framework, as well as OLE container classes and interfaces. The chapter explains how to use the classes and frameworks and provides short examples.

Chapter 5, *Window Classes*, details the Window class hierarchy.

Chapter 6, *Introduction* to OLE Programming, introduces OLE and ActiveX programming.

Chapter 7, *The* Development Process, discusses application programming, building an executable for deployment, developing and delivering console applications. The chapter includes short examples that illustrate the topics.

Chapter 8, *Advanced* Programming, covers advanced topics that include linking with dynamic link libraries, exception handling, targeting ANSI and Unicode images, performance tuning, object serialization, finalization (garbage collection) and the compiler interface.

Chapter 9, *Inline* Assembler, presents the built-in inline assembler.

Target Audience

The information presented in this guide is intended for application developers already familiar with the Smalltalk language and concepts, as well as with fundamentals of Windows programming.

The document focuses on areas that differ from other Smalltalk products and provides in-depth information on implementation aspects. The primary purpose is to assist the application developer in understanding existing code.

Other sources of information include Smalltalk programming books and Win32 documentation.

Notational Conventions

The following typographic conventions are used throughout this manual:

Table 1-1 Typographic Conventions

Example	Description
`STIMAGE.SM`	All uppercase indicate filenames.
`self halt`	This font is used for code and APIs.
File Open	Commands are printed in italic text.
selector	Significant terms are bold the first time they occur.
`Processor`	This font is used for identifiers and constants.

Operating Systems: **Windows NT** means Windows **NT 4** or Windows **2000**. **Windows 9x** means Windows **95**, **98** or **Me**. Please refer to the release notes for information on supported operating systems and restrictions.

Features Overview

The following are some of the Win32 features that are specifically supported:

- Win32 exception and termination handling (`try:filter:except:` and `try:finally:`), compatible with C code
- Multithreading and callbacks from external modules
- ANSI and Unicode character set
- Inline API calls
- Portable Executable (PE) format
- COM interfaces

The class hierarchy maintains a Win32 compatible programming interface whenever appropriate, which lets you leverage on Windows programming know-how and use host operating system documentation.

The programming environment includes the following:

- **Class Hierarchy Browser** with extended browsing capabilities.
- **Project Browser** lets you group classes and/or methods into projects and sub-projects, manages projects and allows you to package a deliverable. A powerful reference resolution algorithm minimizes the code included in your application.
- **Inspectors** let you examine the contents of any object.
- **Graphical Debugger** provides a graphical, windowing debugger for user-level application debugging of a Smalltalk thread. In debug builds, a runtime debugger allows you to debug executables and DLLs.
- **COFF symbols** provide (optional) debugging information to third party debuggers and DrWatson that can be used to debug executables and DLLs built using Smalltalk MT as well as crash dumps.
- **Method Timestamp Browser, Image Comparison Tool** and more.

- **Graphical Interface Builder**.
- **Profiling Tools** for Working Set Tuning (WST) and Call Attribute Profiling (CAP).

Smalltalk MT Editions

Smalltalk MT comes in two editions that share a common code base:

- The **ANSI** edition uses ANSI as the default character set. Unless otherwise specified, all strings and calls use ANSI
- The **Unicode** edition uses Unicode as the default character set. Unless otherwise specified, all strings and calls use Unicode. This edition runs only on Windows NT, since Windows 9x does not support Unicode applications

In the ANSI model, all strings default to ANSI encoding. In the Unicode model, the default encoding is the Unicode character set. The default encoding can be overridden with the messages `asStringW` or `asStringA`.

It is possible to target the NT operating system for better performance.

Targeting Windows NT

Image Properties allow you to enable support for Windows NT, which results on better performance on that operating system but prevents the image from running on Windows 9x.

- The **Windows 9x/NT** edition runs on Windows 9x and NT. The default character encoding is ANSI. Use this edition if you target both Windows 9x and Windows NT or when developing DLLs and ActiveX components.
- **Windows NT** targets specifically Windows NT and allows you to develop both executables and dynamic link libraries (DLLs).
- **Windows NT EXE** provides better performance for processes that run on Windows NT. This setting enables an advanced memory management architecture that provides shorter garbage collection cycles and a reduced working set. We recommend using an NT edition for mission-critical or unattended operation.

Compatibility

All editions are 100% compatible, so you can file out a project from one edition and install it in another. The runtime libraries are the same for all editions.

System Requirements

Smalltalk MT will run on any Intel platform capable of running Windows NT 4.0 (SP3 or later), Windows 2000 or Windows 98/Me. Please refer to the online documentation for how to enable specific Windows 2000 support.

Finding Information about Smalltalk MT

This manual presents the basics to get you started using Smalltalk MT. Other sources are:

Source Code Documentation

The comments in the methods and classes are your primary source of information regarding the implementation.

Online Help Files

The online help files included with Smalltalk MT contain context-sensitive help as well as overviews.

The World Wide Web Site

For information on new products, upgrades and other news, see the World Wide Web sites at **http://www.objectconnect.com/** and **http://www.genify.com/**.

CHAPTER 1 The Development Environment

This chapter discusses setting up Smalltalk MT and provides an overview of the development tools. It also addresses the issue of maintaining and backing up a Smalltalk image.

Finally, a troubleshooting section presents solutions to common problems.

Setup

Smalltalk MT must be installed and de-installed with the installer program. By default, only the combined Windows 98 / Windows NT edition is installed.

To install Smalltalk MT on more than one operating system, you must run the installer on each system.

Please refer to the online documentation and readme files for more information about installation options.

Getting Started

Starting up Smalltalk MT

To start Smalltalk MT, click on the Smalltalk MT icon or start the program STMT from the explorer. In both cases, you do not launch directly the Smalltalk executable image, but a program that updates the current set of Smalltalk files if there is a newer saved image.

The development environment opens the *Transcript* window; which is a text window that remains open during the session. The window lets you edit text, evaluate code, and provides access to the Smalltalk tools.

Starting multiple images

You can work with several images in different directories. You need to copy the files `STIMAGE.EXE`, `SOURCES.BIN` and `STMT.EXE` to the target folders. When working with different releases, you must copy their respective runtime libraries to each directory.

It is also possible to start the same image more than once. This can be useful when debugging an image. Note that subsequent processes will report a sharing violation on the change log and will create a backup change log.

Image Versioning

All Smalltalk DLLs come with a version resource. The image (more specifically, the code in class **Smalltalk**) checks the version of the compiler DLL and fails if it is not the version it expects.

When you start the image (`STIMAGE.EXE`) in a directory that does not contain the source database (`SOURCES.BIN`), you are prompted for the location of the file. It is also possible to rename `STIMAGE.EXE`, therefore making it easier to test a development image under runtime conditions (i.e., in a specific directory and under a different name).

Workspaces

You can evaluate Smalltalk code in the Transcript window, in workspaces, and more generally in most edit panes of the Smalltalk development environment.

Evaluating Smalltalk code

In the Transcript window, type in a Smalltalk expression, such as `1 + 2`, select it, and choose *Display* from the Smalltalk menu. Not surprisingly, the window shows 3 at the insertion point.

You can also evaluate an expression without cluttering the text window with the result string. For instance, type:

```
MessageBox title: 'Smalltalk MT' text: 'Hello World'
```

highlight it and choose *Execute* from the Smalltalk menu. A Message Box will pop up on the screen.

Workspace Variables

Code that is evaluated in a Workspace can use workspace variables. These variables are used like global variables, but are only accessible from the workspace that defined them. When the compiler encounters an undefined identifier, you are prompted if it should be defined as a workspace variable.

Using the editor

The editor is based on the standard **RichEdit** common control. The standard behavior has been extended as follows:

- Double-clicking on the beginning of a line selects the entire line (rather than just the word under the cursor).
- *Display* and *Execute* evaluate the current line when nothing is selected.
- *Tab* and *Shift Tab* respectively indent and un-indent the selected line(s).

Compilation Errors

A compilation error is shown right next to the point where the error occurred. The compiler stops at the first error it detects.

For example, evaluating

```
processor getStartUpDirectory
```

shows:

```
processorUNDEFINED getStartUpDirectory
```

The reason is that the variable `processor` is misspelled and should be `Processor`.

```
Processor getStartupDirectory
```

shows the startup directory of the process.

Runtime Errors

An unhandled runtime exception, such as the following code fragment:

```
'abc' at: 5
```

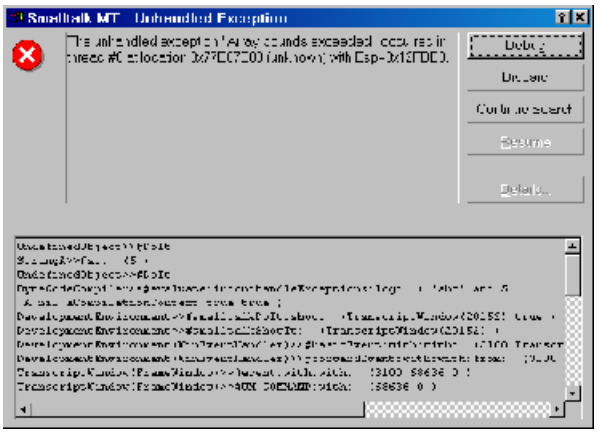
manifests itself with an error message:

Figure 1-1 Unhandled Exception Dialog

It signals that the exception „Array bounds exceeded“ occurred in thread #0 (the startup thread). Since it is a software-controlled exception, the location where it occurred is in the `RaiseException` API.

The button *More...* leads to the expanded dialog below:

Figure 1-2 Unhandled Exception Dialog Details



The text shows the call stack with the arguments that led to the exception, the exception code, description, and a stack trace. You can copy the text in the edit pane.

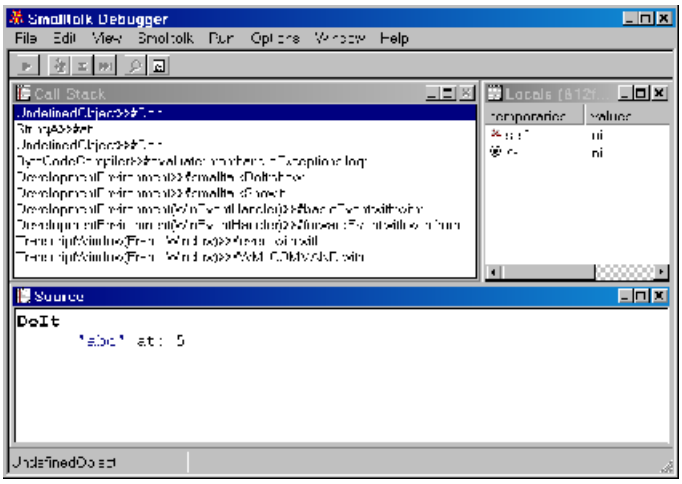
The other buttons terminate the dialog with one of the following actions:

Table 1-1 WalkBack Dialog Buttons

Button	Action
Debug	Opens the debugger on the displayed call stack.
Discard	Discards the call stack. Any non-GUI thread is exited, a GUI thread enters a new message loop.
Ignore	Passes the exception to the next handler in the handler chain. This is usually the top-level handler installed by the host system. In the case of `Execute`, the compiler dismisses the exception (there is a special evaluation handler in the compiler).

When you press *Debug*, the debugger opens:

Figure 1-3 Debugger Window Example



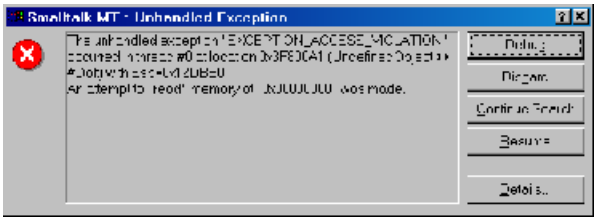
It shows that the assertion of index 5 failed in the method `at:` in **StringA**. Closing the debugger or selecting the *Go* command will discard the call stack and resume the application main loop.

Hardware and kernel exceptions such as access violations constitute another class of exceptions. If you evaluate:

```
MemoryManager atAddress: 0
```

an access violation occurs and the exception handler pops up the message below:

Figure 1-4 Unhandled System Exception



It tells you that the code in `DoIt` (the evaluation) tried to read memory at the address 0. This time, the point where the exception occurred is in Smalltalk code.

Note: The examples above assume that code optimizations are turned off. With optimizations, you may not be able to view all call frames and / or parameter values.

Context of Evaluation

An evaluation is compiled as a temporary method. The context of an evaluation defines the receiver's class, which is used to look up instance variables, pool dictionaries and other identifiers in the name space of the class.

Most Smalltalk windows let you evaluate selected code. The context of evaluation varies as shown in the table below.

Table 1-2 Context of Evaluation

Window	Receiver
Transcript or Workspace	**nil**
Class Hierarchy Browser	The currently selected class
Inspector	The inspected object
Debugger (source pane)	The receiver of the current message frame

Filing in Smalltalk Code

Smalltalk source code is stored on disk in chunk format. A source code file (not to be confused with the `SOURCES.BIN` database file) is an ASCII file that consists of chunks of code, delimited by the character (!). Original (!) characters in the source code are endoubled, so that an occurrence of (!!) is ignored by the chunk reader and interpreted as a single (!) in the source. The chunk format can also be used in a workspace; instead of *executing it* you *file it in*.

Source code can be either evaluated or installed when it is filed in, depending on the format.

Example 1:

To evaluate `Object test`, the chunk file contains:

```
" Evaluate "
Object new!
```

Note that the comment is optional.

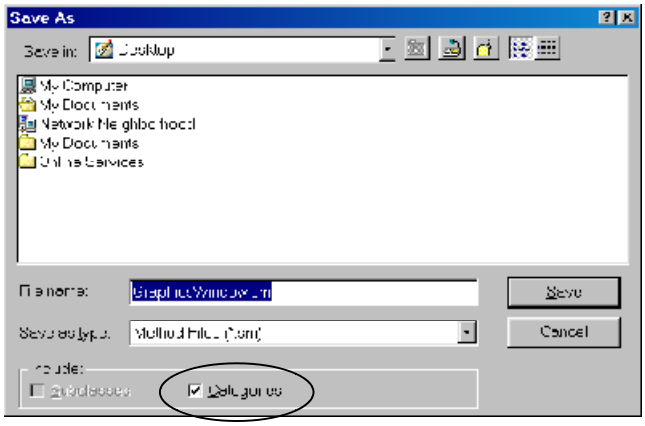
Example 2:

To install a method `Object class>>test`:

```
!Object class * methods 10:31 - 12/18/97!
test
    ^true! !
```

Smalltalk MT can read and write a common subset of this format. By default, a richer method specification that includes the method category and timestamp is used. You can turn the feature off when you file-out a class so that other Smalltalk systems can read it. Simply uncheck the *Categories* check box in the *Save As* dialog.

Figure 1-5 Class Save Dialog

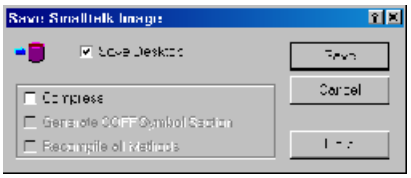


Customizing the User Interface

Saving and reloading the Desktop

Smalltalk MT lets you save and reload the placement and appearance of windows when the image is saved. Saving the desktop is enabled by default, but you can disable it from the **Save Image** dialog. You invoke this dialog by clicking on *File*|*Save Image* on the Transcript menu.

Figure 1-6 Save Image Dialog



When you check the *Save Desktop* button in the **Save Image** dialog, each window gets the opportunity to save its properties to the registry when the image is saved. When the image is reloaded, the windows restore their previous state.

A window has control over the attributes that are saved and restored. The default implementation stores the visual appearance, which includes the position and size, split pane settings (if available) and fonts. Some windows, such as the Class Hierarchy Browser, store additional information that enables them to reconstruct their former state more accurately.

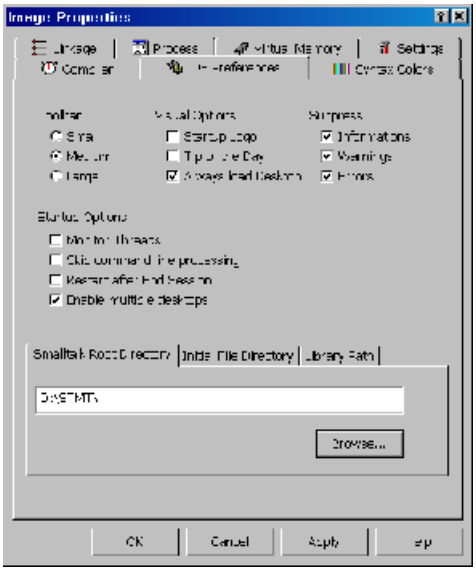
The desktop loader checks the screen resolution before it loads the desktop (otherwise, a lower resolution could place some windows off-screen). You can reuse the same implementation for your own application (it is the recommended way of storing an application's state information).

See also *Using the* Registry on page 219.

Working with Multiple Desktops

You can direct Smalltalk MT to maintain desktop settings that are specific to the development image's directory. Thus, images in different directories can work with different desktops. You enable multiple desktops by clicking on the corresponding check box in the *Image Properties* property sheet, *IDE Preferences* page.

Figure 1-7 IDE Preferences Page



Saving a Default Appearance

While the desktop defines a particular configuration, each window class can also have a registry entry that lists default values for its placement and preferences. The data is used when a new instance of the window is created.

Window placement

You can customize the placement and appearance of a window in the following ways:

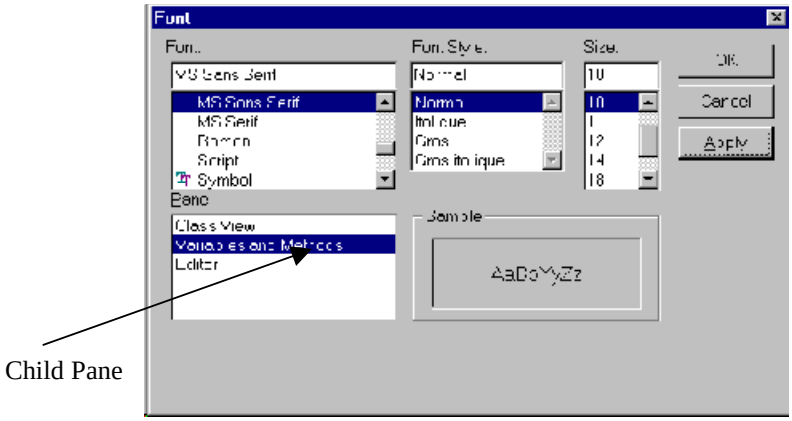
- right-click on a child window to bring up a menu that lets you set its font
- modify the size of the window to define its default size
- move the split bars to set the default split position (or ratio) on startup
- double-click on the toolbar to customize the buttons

When done, click on *Save Preferences* (usually in the menu next to the *Help* menu). The next time you open a window of the same class, it will start with the settings you saved.

Fonts

You can customize fonts by right-clicking on a pane of selecting the *Set Fonts* menu item under the *Tools* menu. If the window has several panes that support custom fonts, font selection is done via a font selection dialog that allows the user to select a font for each child pane.

Figure 1-8 Font Dialog



Development Tools

Overview

Smalltalk MT comes with a set of development windows. These windows run in the main thread (thread #0) and are managed by an instance of **DevelopmentEnvironment** (which is a subclass of **WinApplication**).

For more information on threads, see also *Processes and Threads* on page 194.

Browsing Source Code

General browse tools

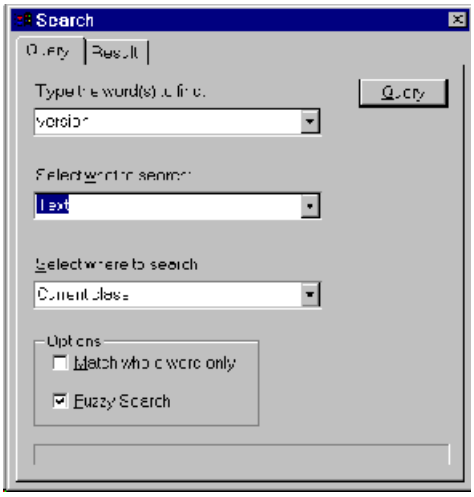
The development environment has powerful browsing capabilities. The **Query** dialog exposes most of them, but you can also use menu items. The browsing capabilities include:

- Finding text in methods (optionally whole word, case sensitive). The search text can include wildcards.
- Full text search (methods with matching text).
- References to a global variable.
- References to an API.
- Implementors of a message.
- Senders of a message.
- Sort methods by timestamp. This is useful for detecting recent modifications and additions.

- Detecting unimplemented messages (messages sent but not implemented in the image).
- Detecting unused methods (methods without senders).

It is possible to browse globally (through the entire class hierarchy) or locally (current class and subclasses). The Query dialog must be invoked from the Class Hierarchy Browser in order to browse locally.

Figure 1-9 Query Dialog



Class Hierarchy Browser

Additional browse options are available in the Class Hierarchy Browser. The browse domain is always the current class and subclasses where applicable.

Messages

This option lists the messages sent by the selected method.

Unimplemented messages

Displays unimplemented messages in the current class and subclasses. These are messages that are sent but not implemented in the image. Those messages may raise runtime errors when they are sent.

There are a couple of situations where unimplemented messages may occur:

- Callback class libraries. Most OLE interfaces send a predefined set of messages to their owner. The consequence is that OLE interface classes may send unimplemented messages when there is no code in the image that uses those interfaces.
- Messages processed via `doesNotUnderstand:`.

Classes that use unimplemented messages can implement a class method `unimplemented` that returns an array of symbols to be ignored by the browser.

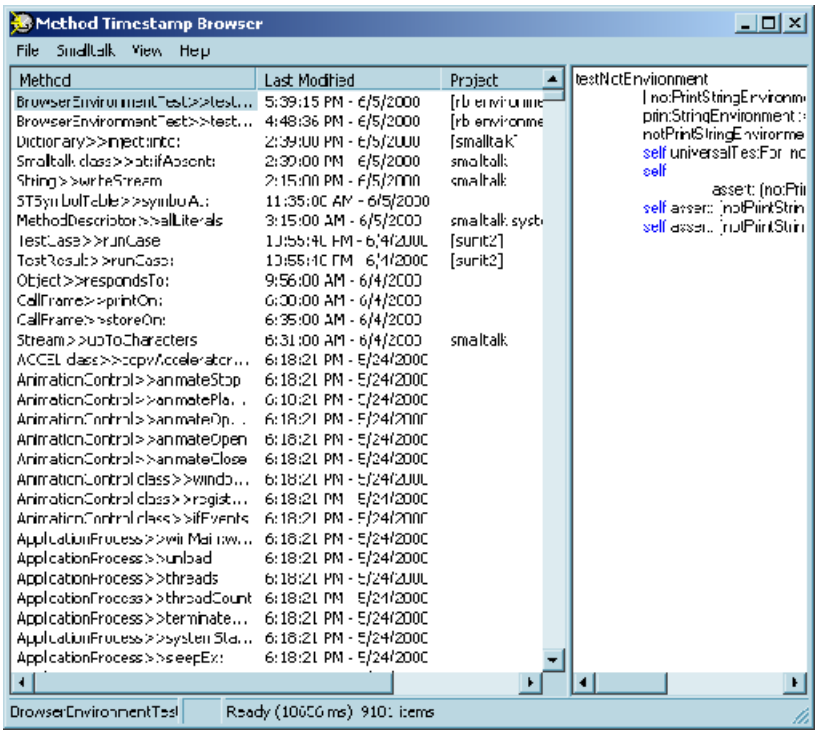
Source Comparison

This tool is invoked from the Transcript's *Tools* menu. It analyses two source images (`SOURCES.BIN` files) and reports the differences to an installable file. The generated file lists new and modified methods, classes, pool dictionaries, and methods or classes to delete.

Change Browser

The **Change Browser** displays methods (and optionally classes) according to their last-modified timestamp. This makes it easy to determine what code has been altered since a given date. The *Project* column lists the project (if any) in which a method is defined.

Figure 1-10 Method Change Browser



The **Change Browser** lets you perform the following tasks:

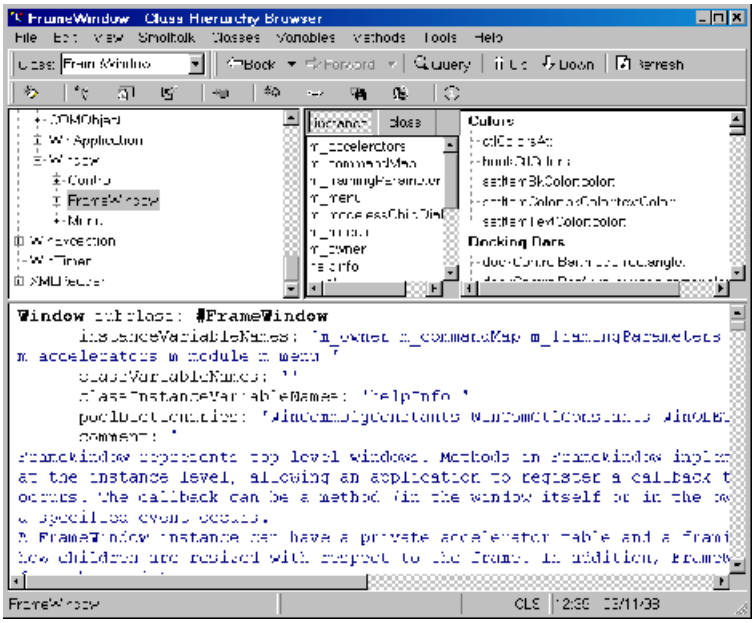
- list methods by timestamp (last modified time and date)
- change the timestamp of a method (this can also be done from the Class Hierarchy Browser)
- file-out selected methods
- edit the source code of a method and compile it

Note You can use the Change Browser to determine which methods and classes have been modified lately.

Class Hierarchy Browser

The **Class Hierarchy Browser** displays the Smalltalk class hierarchy in its upper left pane, and for each selected class the variables in the middle pane and the methods in the upper right pane. The text pane shows the selected method's source, the textual description of a selected category, or the class declaration if no item is selected in the method pane.

Figure 1-11 Class Hierarchy Browser



Class Hierarchy Pane

The classes are represented in a **TreeView** common control. The buttons *instance* and *class* let you switch between class view (class methods) and instance view (instance methods and instance variables).

A pop-up menu allows you to delete the currently selected class, invoke the **Find Class** dialog, and perform class-specific operations such as editing a window in the Interface Builder.

Variables Pane

The **variables** pane lists the instance variables (if the instance button is hilighted), the class variables and the class instance variables for the current class and each superclass. In the example depicted above, the class inherits the instance variables `m_handle`, `m_properties` and the class variables `WinMessages`, `WinMessagesEx` from **Window**, and the class instance variable `eventHandlers` from **WinEventHandler**. You must refer to the class definition to determine the actual type of a variable, though by convention, class variables begin with an uppercase letter, and class instance variables are seldom used. In addition, Smalltalk MT system code prefixes all instance variables with "`m_`" for easier recognition.

The pop-up menu lets you filter assigned methods (which are displayed in the right-most pane) and inspect class variables and class instance variables.

Methods Pane

The **methods** pane lists the methods by category. An asterisk represents the default category (*). Click on *Expand/Collapse All* to toggle between the expanded and collapsed state, which shows or hides methods under the categories.

You can change the category of a method with the *Change Category* menu item. *Rename Category* renames the category system-wide.

A category can also have a description that is displayed when the category is selected. To change the description or enter new text, select the category, edit the text in the text pane, and click on *Accept*.

Several filters can be used to display the methods:

- *Show inherited* displays also inherited methods, meaning that the full protocol of the class is displayed.
- *Show uncalled* displays methods that are implemented but not called from anywhere in the image. This option is useful for detecting obsolete methods.
- *Show private* excludes private methods when checked. A method is private when its class comment includes the word “Private”.

Note Click on *Edit|Categories* to toggle category mode **on** or **off**. When the category mode is **off**, methods are sorted alphabetically under a common node, which makes it much easier to find a method by name.

Compiling Methods

To define a new method or modify an existing one, proceed as follows:

- Click on the class in which you want to install the method.
- Click on the *class* radio button if you want to define a class method, otherwise on the *instance* button.
- To define a new method, click on *New Method* in the *Methods* menu. Otherwise, click on the method you want to modify.
- Edit the source code.
- Click on *Accept* in the *Smalltalk* menu to validate the changes.

Depending on the outcome of the compilation, you may get one of the following messages in the status bar:

Table 1-3 Compilation Warnings and Errors

Result	Icon	Description
Ok	none	The method has been compiled successfully.
Compilation error		A compilation error occurred.
Warning		The method has been compiled with warnings.
Information		The method has been compiled with information.

Warning or information messages can be any of the following:

Table 1-4 Compilation Information

Message	Icon	Description
None or any message		Check for unused local variables (highlighted in red in the source).
n new symbols defined		The code references methods that have not been defined elsewhere (check for typing errors).
Return from block		An inline block in the method returns from the method; check whether this is intended behavior.

New Symbols

You can view the set of new symbols defined right after a compilation. The right button popup menu has an item named *View New Symbols* that, when clicked, displays a list with the new symbols. This feature allows you to detect messages that are sent by the method but not implemented elsewhere (in most cases, these are typing errors or unterminated statements that compound to a message).

Example:

Define the method below and click on *View New Symbols*.

```
test
    ^self foo
```

Figure 1-12 New Symbol Dialog

The dialog may also list `test` if it is a new symbol, but only if new symbols other than the method selector have been defined as well.

Return from Block

A block return is a return statement inside a block. The return statement not only exits the block but also the method in which the block is defined. Though legal, block returns are not frequently used and come with a performance penalty. Quite often, typing errors result in unwanted blocks and block returns that this message allows you to detect.

Working with Classes

How to find a class

To find a class by name, you can either type a search string into the combobox in the toolbar or click on *Find Class* in the *Classes* menu.

Several search algorithms are available:

- If the search is a pattern search:

- If several matches were found, a list prompter with the matching classes pops up.
- If the search resulted in a single match, the corresponding class is selected.
- Otherwise, a message box is printed (if the search originated in the toolbar's combobox, only a warning sound is played).

- If a case-sensitive comparison finds a matching class, that class is selected.
- If a case-insensitive comparison finds a matching class, that class is selected.

How to delete a class

A class can be deleted by clicking on the *Remove Class* menu item in the Class Hierarchy Browser. The effect of the deletion is that the class mutates into an instance of **DeletedClass**. The class is definitely deleted when the image is saved, and the space occupied by the deleted class is recycled.

How to rename a class

First, you should detect all references to the class name (click on *References* in the *Classes* menu). Next, click on *Rename* and enter a new name for the class, press *OK* to rename it. You must now change all references to the old class name (which are listed in the previously opened browser).

How to change the Superclass of a Class

Click on *Change Superclass* in the *Classes* menu and enter the new superclass. The operation is only successful if the superclass has the same structure and size (number of class instance variables) as the class to reparent (e.g., variable bytes, indexed, pointer). Otherwise, you must file out the class, delete it, and file it in again after changing the name of the superclass in the source file. In addition, you must also recompile all references to the class.

Working with Methods

Timestamps

Each method receives a timestamp when it is compiled. When the image is saved, the timestamps of all methods that have been compiled during the session are updated to reflect the time and date the image was saved. Timestamps make it easy to track changes between sessions.

It is possible to preserve the timestamp of a method by choosing *Recompile* rather than *Accept*. This is useful when the changes to the source code are merely cosmetic, such as changing a comment.

To edit the timestamp, click on *Change Date...* under the *Methods* menu and enter the date and time into the dialog.

Browsing Changes

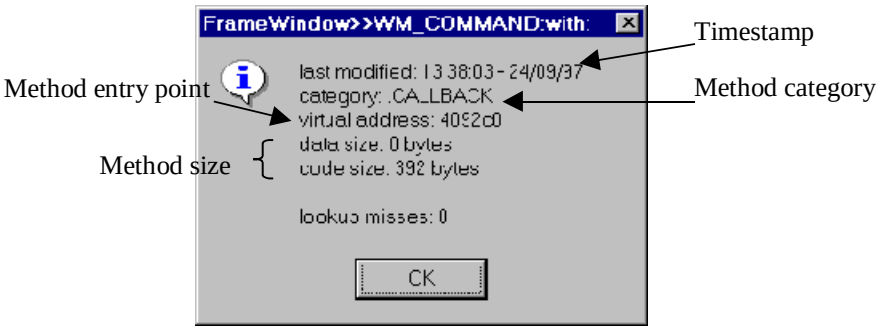
If *Log Old Sources* in the Image Properties is enabled, the system logs previous editions of a method and you can open a Change Browser on all available editions of a method's source. Click on *Methods|Browse|Changes* to open a Change Browser on the selected method.

Method Statistics

You can open a dialog with method statistics from the toolbar by clicking on the information sign. The statistics include:

- The method timestamp.
- The method category.
- The (current) virtual address of the entry point.
- The size in bytes of method literals and blocks.
- The size in bytes of the method code.
- The number of method lookup misses. A lookup miss occurs when the method's selector collides with another selector in the method dictionary, or if the method is inherited. For example, a value of zero indicates that there are no collisions. You can also display the inherited methods (*Methods|Show Inherited...*) and view the number of lookups required for the current class. The Working Set Tuner allows you to optimize the number of lookups of time-critical methods to zero, even for inherited methods.

Figure 1-13 Method Statistics Dialog



Accelerator keys

The Class Hierarchy Browser also accepts alternate accelerators, in addition to the standard accelerators displayed in the menu.

Table 1-5 CHB Accelerator Keys

Key Sequence	Action
F3	Repeat last edit operation
Ctrl + F8	Accept (source code in the edit pane)
Shift + F8	File-it-in
Shift + F9	Inspect (line or selection in the source pane)
Alt + F10	Open API Editor on highlighted API (if applicable)
Shift + F10	Open Pool Browser on highlighted constant (if applicable)
Ctrl + F11	Open a new Class Hierarchy Browser by cloning the current
Shift + F11	Open a Project Browser on the current method or class
Ctrl + D	Display (evaluate line or selection and display result)
Ctrl + E	Execute (evaluate line or selection)
Ctrl + G	Go to highlighted class
Ctrl + N	New method
Ctrl + O	File open
Ctrl + U	Uncalled methods
Alt + S	Accept
Alt + Z	Zoom (maximizes the edit pane)

Toolbar customization

You can customize the toolbar of the Class Hierarchy Browser by adding or removing toolbar buttons. To bring up the toolbar customization dialog, double-click on the toolbar or click on *Tools|Toolbar Customization.…*.

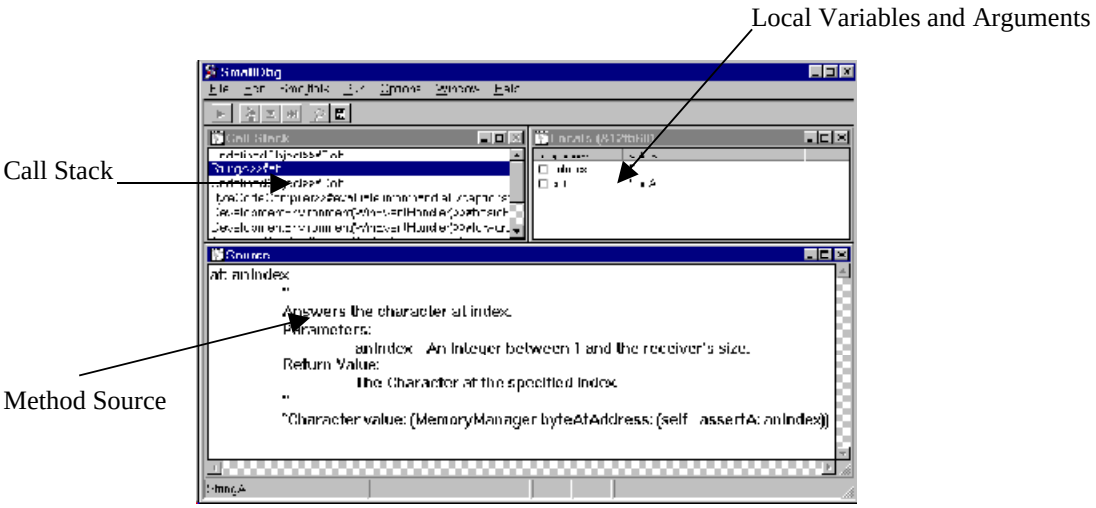
Debugger

Overview

The Smalltalk debugger is an in-process debugger that runs in one of two modes:

- After an exception, the debugger opens in the faulting thread and lets you examine the call frames that led to an exception. When you close the debugger, a GUI thread discards the current stack and starts another message loop, while a non-GUI thread is terminated.
- After executing a `self halt` statement, a debugger opens in a separate thread and freezes the debugged thread. You can skip through the code interactively, and closing the debugger resumes the execution of the thread. If a debugging thread is already running, the debugger in that thread is used. Otherwise, a new thread is created to run the debugger application.

Figure 1-14 Debugger Window



The debugger is implemented as an MDI application with at least three child windows:

Table 1-6 Debugger Child Windows

Window	Contents
Call Frames	Displays the call frames (a stack of method invocations) that lead to an exception or halt.
Locals	Once a call frame is selected, displays the local variables, the receiver and arguments of the frame. The icons on the left hand are of three types: □ an initialized variable. ! an exclamation mark signals that the variable contents have changed during a skip.

	? a question mark denotes an uninitialized variable or binary contents.
Source	Displays the method source of the currently selected frame. Evaluations are done in the context of the selected frame.
Text	Opened on demand. Allows evaluations in the current context.
Inspectors	Opened on demand. Inspect the selected local variable or result of evaluation.
Class Hierarchy Browsers	Opened on demand. Browse the class hierarchy.

Debugging Frames

After the debugger opens, you can display the source code of a method by clicking on the call frame. The source code can be modified and compiled using *Accept* (after you recompile the source, the current instruction may not be highlighted correctly because the source has changed).

Customizing Child Window Placements

You can customize the placement and font properties of child windows. Arrange the windows the way you want them to appear and click on *Save Preferences* in the *Options* menu. The settings will be remembered and used the next time the debugger opens.

The *Arrange Windows* command automatically tiles the windows in a manner that is useful for debugging. Click on *Save Preferences* to make the changes permanent.

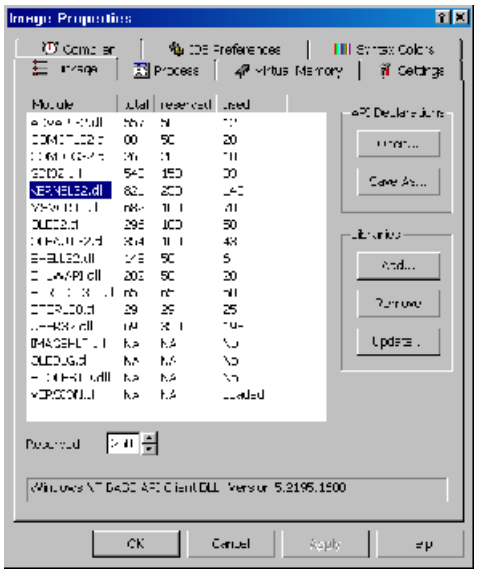
Limitations

- The Smalltalk debugger is not a debugger in the sense of Win32; in particular, it cannot display Win32 debugging events.
- If code optimizations are enabled:
  - It is possible that the debugger skips a stack frame when a method has been optimized to not create a frame. The method immediately below will not be shown.
  - In some cases, the debugger may not be able to find the calling point in the source. This is again due to optimizations and message inlining.
  - Methods that do not create a frame cannot be debugged. When you attempt to trace into such a method, the debugger emits a warning beep and does nothing.

Image Properties

You access the Image Properties sheet from the Transcript window. It contains tabs for various Image and Compiler settings that you can customize.

Figure 1-15 Image Properties Sheet



The parent sheet gives you access to several property pages:

- The **Linkage** page displays import properties and is discussed in *Modifying the Import Section* on page 56.
- **Process** and **Virtual Memory** pages are discussed in *Virtual Memory Settings* on page 62.
- **Settings** properties are discussed in *Enabling Diagnostic Messages* on page 77.
- **Optimization** is discussed hereafter.
- **Preferences** and **Syntax Colors** are detailed in the online documentation.

The Compiler Page

The Compiler page lets you adjust three types of parameters:

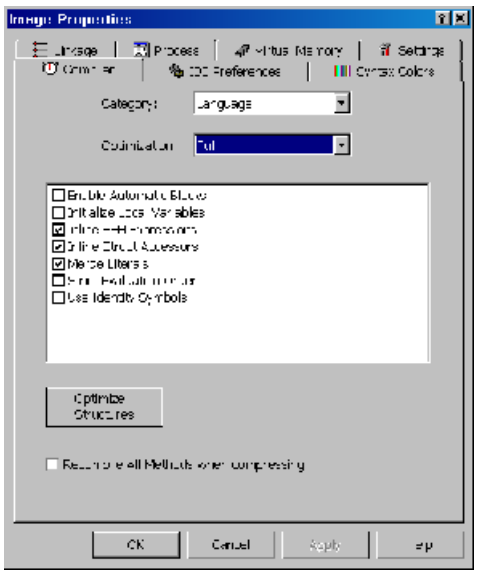
- Language options

- Optimization options
- The cache reserved for method lookup.

Each area can take one of four predefined values:

- Disable: no optimizations apply
- Compatible: conservative settings that leave assertions intact (booleans, certain loop parameters, etc.)
- Full: full optimizations
- Customize: you can adjust individual switches

Figure 1-16 Image Properties - Optimization



The compiler accepts widespread language configuration options as well as customizable optimization.

Table 1-7 Language Options

Option	Default	Action
Enable Automatic Blocks	On	Automatic Blocks allocate a stack context when invoked, which makes them reentrant.
Initialize Local Variables	Off	Initializes local variables to **nil**.
Inline Methods	On	Inlines methods under the .INLINE category. The receiver class must be well defined; **self** and immediate classes are acceptable or explicit type hints (_as:).
Inline SEH Expressions	On	Inlines Structured Exception Handlers (*try:filter:except:* and *try:finally:*). This option compiles native inline exception handlers and converts all literal blocks to inline code, resulting in highly efficient code. This improves performance but prevents you from reimplementing these messages. *Note:* The keywords *_try:_filter:_except:* and *_try:_finally:* force inline compilation.
Inline Struct Accessors	On	Structures created using *localNew* are subsequently accessed inline. This option can dramatically improve reading and writing structure fields.
Merge Literals	On	If checked, causes the compiler to merge literal objects in a method. For example, an expression like 'abc' == 'abc' returns **true** if literals are merged and **false** otherwise.
Strict Evaluation Order	Off	If enabled, all expressions are strictly evaluated from left to right. This mode is implemented for compatibility with other Smalltalk implementations and degrades performance.
Use Identity Symbols	Off	If enabled, the compiler creates and references unique symbols. This mode is implemented for compatibility and increases the size of the runtime image.

Table 1-8 Optimization Options

Option	Default	Action
Always Optimize Floating-Point	Off	Applies floating point optimizations to all methods. Usually, only methods with literal floats or with the *%FP Optimization%=ON* tag in the method comment are optimized for floating point performance. Applying floating-point optimizations to methods that do not use floats increases the code size and reduces the performance for integer arithmetic, so this option is only useful if a project *only* does floating point math.
Enable Native 32-bit Locals	On	If the result of a memory read or an API call is stored in a local variable, the native 32-bit format may be stored instead of a Smalltalk integer. This avoids unnecessary conversions when the result is reused as a native 32-bit value. The compiler counts the number of times the variable is subsequently used as a native value versus the number of times it must be converted to an object, and decides whether to convert the value before storing it in the variable or not. If this option is disabled, all local variables contain Smalltalk objects and can be viewed in the debugger.
Frame Pointer Omission	On	If checked, the optimizer tries to remove frame contexts from methods that do not require one. Note that the debugger cannot step into a method without a frame.
Inline Small Integer Operations	Off	If checked, the optimizer inlines SmallInteger operations. This improves integer performance but also increases the code size.
No Assertions	On	If checked, the optimizer removes assertions for booleans and integers in loops.
No Loop Assertions	Off	Does not verify loop parameters and receivers in the case of inlined loops. Enabling this option speeds up loops but produces unexpected results when the parameters are not of the correct type.
Optimize Boolean Expressions	On	If checked, reorders boolean expressions for maximum performance.
Optimize Floating-Point	On	Applies floating-point optimizations in methods

		with literal floats or with the *%FP Optimization%=ON* tag in the method comment.
Pentium II Optimizations	Off	If checked, generates optimized code for the Pentium II. and above
Optimize Jumps		If checked, reorders jumps for maximum performance. This option should be used in conjunction with Boolean Expression optimizations.
Optimize Tail-Recursions	On	If checked, performs tail-recursion optimizations. A tail recursion occurs when the last statement of a method calls itself recursively.
Remove Unreachable Code	On	If checked, removes unreferenced code in a method.
Re-Order Flow Control	On	If checked, re-orders flow control expressions.

Optimization Levels

For development , disable all optimizations so that the runtime recognizes common programming errors. The maximum setting will not detect certain errors.

For example, at the maximum optimization level:

```
b ifTrue: [...]
```

is equivalent to:

```
b == true ifTrue: [...]
```

meaning that there is no exception if b is not a **Boolean**.

Likewise, it is recommended to turn tail recursion optimization off while developing because it does not handle the case when you later subclass a recursive method (therefore breaking the recursion). You can turn it on again when the implementation is stable, and recompile the affected code.

Recursion Optimization

Tail recursion replaces a recursive call to the receiver method with a jump to the beginning of the method, as in the example below:

```
recursiveTest: a
    a > 1 ifTrue: [
        ^(self recursiveTest: a - 1)
    ].
    a < 0 ifTrue: [self raiseException: ST_EXCEPTION_INVALID_ARGUMENT].
    ^1
```

The compiler places certain constraints on the receiver class and the selector:

- The receiver class must *not* have **SmallInteger** as subclass.
- The method must *not* be reimplemented in subclasses.

The differences in terms of performance can be significant. If the method above is implemented in a class named **ATest** and the following statements are evaluated:

```
Time millisecondsToRun: [
    10000 timesRepeat: [ATest recursiveTest: 15]
 ]
```

An optimized call takes 19 ms with tail recursion enabled and 29 ms when it is disabled (30 ms without any form of optimization).

You can also speed up the computation by specifying that the parameter is an instance of **SmallInteger**:

```
recursiveTest: a
    a _asInteger > 1 ifTrue: [
        ^(self recursiveTest: a _asInteger - 1)
    ].

    a _asInteger >= 0 ifTrue: [^1].
    ^self raiseException: ST_EXCEPTION_INVALID_ARGUMENT
```

In this case, the call returns after just 9 ms. The effect of the pseudo-message is that all integer operations are inlined. Note also that the most often used branches come first.

Applying Changes

Optimization settings are used when code is compiled. The checkbox *Recompile all methods when compressing* invalidates the byte code of all methods in the image, which forces them to be recompiled when the image is compressed. The byte code is invalidated when the property page closes.

Inspectors

Abstract

Inspectors are windows that display graphically the contents of an object. There are specialized inspectors for byte objects, structures, dictionaries, pictures and so forth. You can also implement inspectors for your own objects.

Inspecting Objects

There are several ways to bring up an inspector, as detailed in the table below.

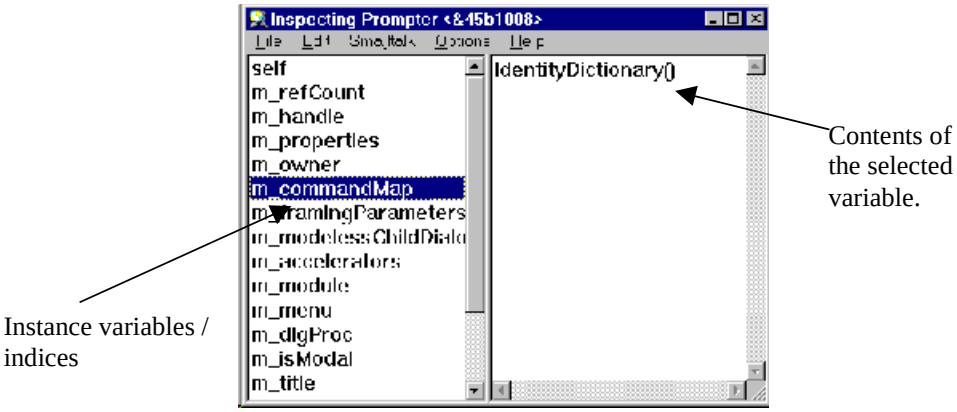
Table 1-9 Inspecting Objects

Object	How to inspect
Result of an evaluation	Highlight the expression to evaluate, and choose *inspect it*.
Class or class instance variable	Highlight the variable in the Class Hierarchy Browser, click on *Inspect* in the *Variables* menu.
Instance variable	Bring up an inspector on the object, double-click or click on *Inspect* in the *Object* menu.

Using Inspectors

The generic object inspector displays two panes. Instance variables or indices are displayed in the left pane while the right pane prints the selected variable.

Figure 1-17 Object Inspector



Evaluating code

The edit pane of the Inspector lets you evaluate code in the context of the inspected object. You can assign variables or otherwise modify the object being inspected.

Inspecting pointer objects

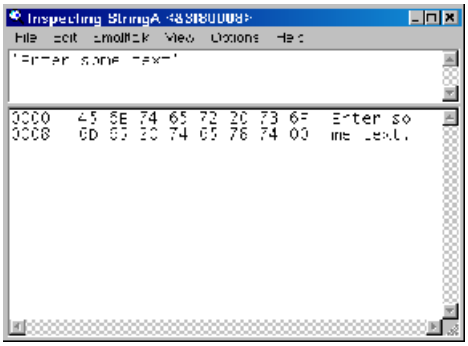
Double-clicking an item in the left-hand pane or choosing *File|Inspect* opens a new inspector on the object.

Specialized inspectors such as **MapInspector** also have an *Object Inspector* item under the *Tools* menu, which lets you open a regular object inspector on the inspected object. For example, you can inspect the instance variables of a **MappingTable**.

Inspecting byte objects

Byte objects are inspected using a byte inspector, which displays a memory dump of the object. You can evaluate code in the lower edit pane.

Figure 1-18 Byte Inspector



Inspecting memory-mapped object files

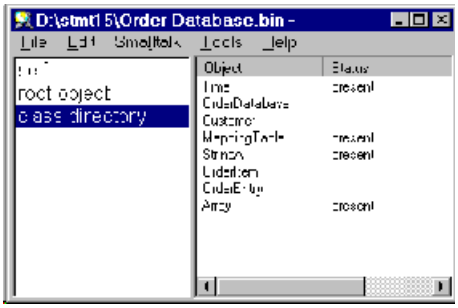
MOSInspector opens and inspects a memory-mapped object file. The inspector displays the file header information and classes required by the file, even if the file cannot be loaded into the image (because a class is missing).

The inspector lets you open and inspect a file or inspect a **MappedObjectStream** instance. The file is unmapped automatically when you close the inspector.

The left-hand pane displays three items discussed below. Clicking on an item displays information in the right pane.

- The first item, **self**, displays the file header.
- The **root object** is the root of the memory-mapped file.
- The **class directory** lists classes required by the file. A class that exists in the image is marked present. If the file references classes that are not in the image, the file cannot be mapped.

Figure 1-19 Memory-mapped Object File Inspector



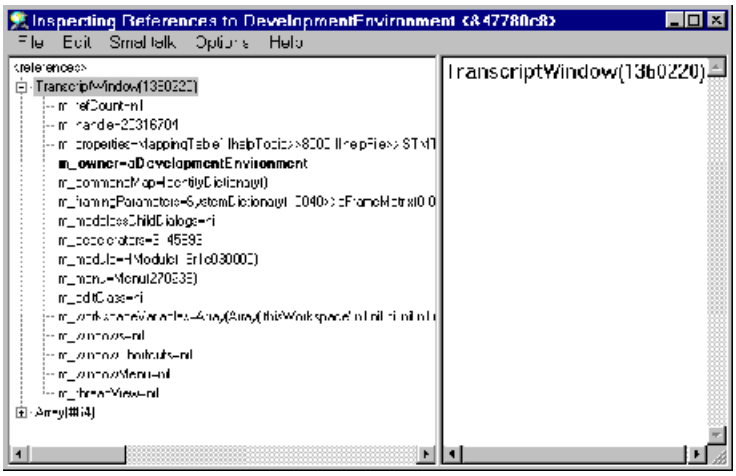
Note If you open inspectors on the root object or any other object that is in the mapped object file, you must close the secondary inspectors before you close the MOS File Inspector. Otherwise, the object references become invalid when the file is unmapped.

Inspecting object references

The reference browser is useful for tracking references to an object and resolving unwanted dangling references.

A Reference Browser is opened from an inspector window, by clicking on *File|References*. The Reference browser displays a tree of consecutive references. It is possible to expand a node to browse recursive references.

Figure 1-20 Reference Browser



Other inspectors

ImageInspector displays an image (an **Icon**, **ImageList** or **Bitmap** instance).

OleStreamInspector and **OleStorageInspector** respectively inspect OLE streams and storages.

TypeInfoInspector and **RefTypeInfoInspector** inspect OLE type information objects. This is particularly useful when exploring automation interfaces, since the inspectors list all methods, properties, argument names and types where applicable.

Implementing Custom Inspectors

You can implement customized inspectors for your data structures. The inspector window should be a subclass of **InspectorWindow** and must implement the following methods:

Table 1-10 Messages in Inspector Subclasses

Method	Description
getTargetObject	Returns the object to be inspected when the user clicks on *inspect*. The default implementation returns the inspected object.
initWindow	Creates one or several child panes. The default implementation adds the window to the Transcript's graphical menu and sets the window title.

Finally, implement the message `inspector` as an instance method in the classes to be inspected with your inspector. The method must return the inspector class rather than a window instance (remember that inspectors can also be opened in MDI mode, which requires an MDIChildWindow instance rather than a FrameWindow).

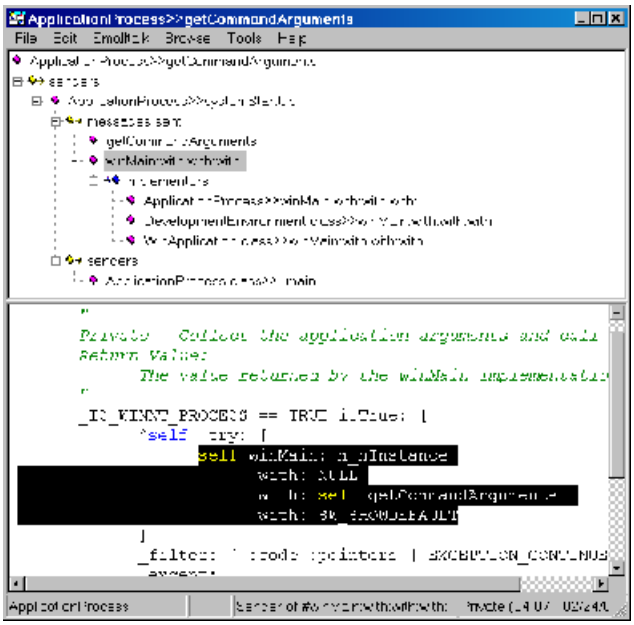
Method Explorer

Method Explorer uses a single interface for the following global and local (i.e., limited to subclasses of a class) browse operations:

- implementors
- senders
- string search
- messages sent by a method

The results are displayed in a tree. You can file-out all subnodes of a node (for example all implementors of a given message) by selecting the node and clicking on *File|Save As...* . To file-out all methods in the browser, click on *File|Save All...* .

Figure 1-21 Method Explorer Window



Project Browser

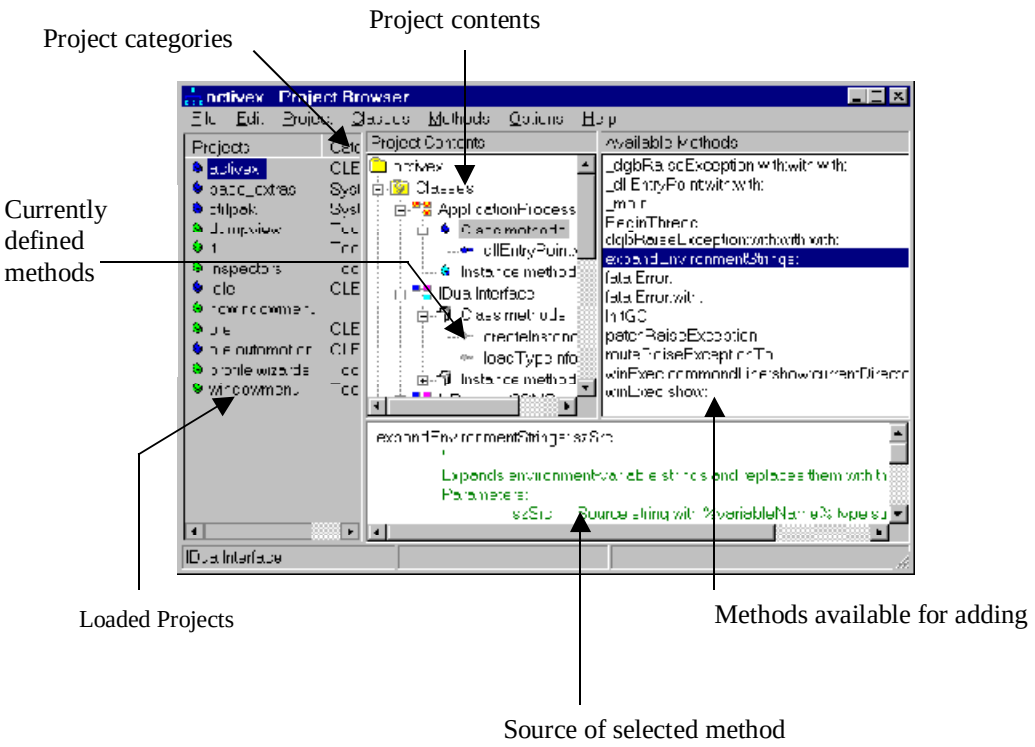
Abstract

With the **Project Browser**, you can organize your source code as projects. Each project can have an arbitrary number of classes, methods, and pool dictionaries. In addition, a project can reference other projects or external DLLs via the prerequisite folder. A project is saved as an ASCII file.

An application may require that the image be linked with external libraries. For example, *Generic* must access the library `VERSION.DLL`. A project's prerequisites list those dependencies, as well as other projects that must be installed first. **Project Browser** takes care of the installation automatically, binding with DLLs and loading dependent projects as necessary.

In addition, the **Project Browser** lets you also easily generate an executable.

Figure 1-22 Project Browser Window



Functional Overview

The **Project Browser** lets you perform the following tasks:

- **Manage** your source code: classes, individual methods, pool dictionaries.
- Define **prerequisite** projects that must be present before a project is installed.
- Define external **libraries** that must be linked before a project is installed.
- **Install** a project in the image, linking with external DLLs and loading prerequisites if necessary.
- **Remove** a project from the image.
- **Update** a project, removing project definitions that are not present in the current image.

Limitations of the Project Browser

- You cannot edit a project before it is installed, with the exception of the *remove undefined items* function.

Working with Project files

Opening a project

You can open a project file and view the items it contains. However, you cannot edit or save the project until it has been installed, unless all the items of the project are initially present in the image.

Installing a project

Installing a project entails loading any prerequisites and filing in the source code. This is done automatically by the project manager.

If the project requires DLLs that are not linked, the image is first linked with the required libraries, and the image is restarted before the remaining project elements are installed.

After a project has been installed, it is added to the list of loaded projects.

Project file format

Smalltalk MT supports two project formats: SP stores the project contents as chunks of text while SPX uses XML.

SP format

An SP project file consists of two sections; the first contains the project definitions while the second contains Smalltalk source code in chunk format. You cannot install a project file directly without using the **Project Browser**, but you can file-in the second part of the file by loading it first into a workspace.

SPX format

SPX is an XML representation of a project and its contents. This format offers the following benefits:

- Browsable by any XML viewer
- Source code can be viewed without installing the project

- Graphical representation of source code changes and conflicts with the image when the project browser displays an SPX project that has not yet been loaded
- Individual elements (e.g., class declarations, methods and scripts) can be installed selectively
- Scripts are contained in the XML file so there are no references to external files

The downside of the SPX format is that it requires an XML parser.

Sorting project methods by timestamp

This option lets you install a project and sort the methods by timestamp. This makes it easy to view changes that have been made to a project.

Managing loaded projects

Loaded project view

This view displays all currently loaded projects, the category of a project, and uses different icons to represent dependencies.

A referenced project should not be removed because it is required by other projects in the image. Attempting to remove a referenced project displays a message that asks for confirmation.

Project properties

The Project Properties dialog displays properties of the currently selected project. This includes the following information:

Table 1-11 Project Properties

Method	Description
Name	This is the name of the project, as displayed in the loaded projects view. By default, it is the same as the file name.
Category	A user-defined category name that allows you to classify projects (the category name is *not* related to method categories).
File attributes	The last modification time and date as well as the size of the file. This information is read-only.
Dependents	Click on this button to list dependent projects; i.e., projects that have the current project as a prerequisite. The project browser will load each project and recompute the dependents, so that projects files that have changed are taken into account.

Deleting a project

Deleting a project removes the project from the list of loaded projects, without removing the source code and without deleting the project file.

Typically, you would delete a project after moving its constituents to another project.

Saving a project

Click on *File|Save* or right-click on the loaded project and choose *Save* to save the project properties and source code to disk. You can also click on *File|Save As…* to save the project under a different name.

Unloading a project

This operation is the counterpart to installing a project. It removes all project source code, pool dictionaries and variables. However, it does not remove prerequisite projects.

Defining a project

The items that make up a project are listed in the table Project Elements.

Table 1-12 Project Elements

Item	Description
Class Definition	The class is owned by the project. By default, all methods are also included. When the project is removed, the class is removed as well.
Class Reference	The project adds individual methods to the class. When the project is removed, only the methods owned by the project are removed.
All Methods	All methods of the class are owned by the project. Usually, this makes only sense in conjunction with an owned class. When the project is filed-out, all methods currently defined in the class are saved along.
Method	An individual method that the project adds to the class.
Pool Dictionary	A pool dictionary that defines constants used by this project.
Prerequisite DLL	An external DLL that must be linked with the image before the project can be installed.
Prerequisite Project	Another project that must be installed before the project can be *filed in*.
Scripts	An SP project can have up to two scripts: ♦ **Pre-load**: An initialization script that is executed before the project is installed. ♦ **Post-unload**: A uninstall script that is executed after the project has been removed. If the project uses the XML format (.SPX extension), two additional scripts are possible: ♦ **Post-load**: A script that is executed after the project is installed. ♦ **Pre-unload**: A script that is executed before the project is removed.
Variables	A project can declare global or thread-local variables that are automatically added when the project is installed, and deleted when it is removed.

Note 1 Prerequisites (projects or DLLs) are not removed when the project is removed.

Note 2 If a DLL requires an API definition (.DEF) file, the file must be either in the project's home directory or in the library path used by the Project Browser (in the image properties under *IDE Preferences*). Use the Image Properties sheet to generate such a file (a .DEF file is required if the DLL uses non-default API argument types, return values, or calling conventions).

Defining a Windows Project

A Windows Project is a project that defines a stand-alone Windows application, where application encompasses both executable files and DLLs. In addition to the items above, it requires the following:

Table 1-13 Window Project Elements

Item	Required	Description
Project Directory	✕	The project is saved in a separate directory.
Project File	✕	The project file (*PROJECT*.SP) is saved in the project directory.
WINMAIN.SM		If present, defines the entry point for a stand-alone application.
RES Directory		If present, contains application resources.
Resource DLL		If present, defines the application's resources in binary format.
Resource files		Files that make up the application's resources, and that can be compiled to a DLL.

Build properties

An executable (EXE or DLL) application has an associated set of build properties that are saved along with the project.

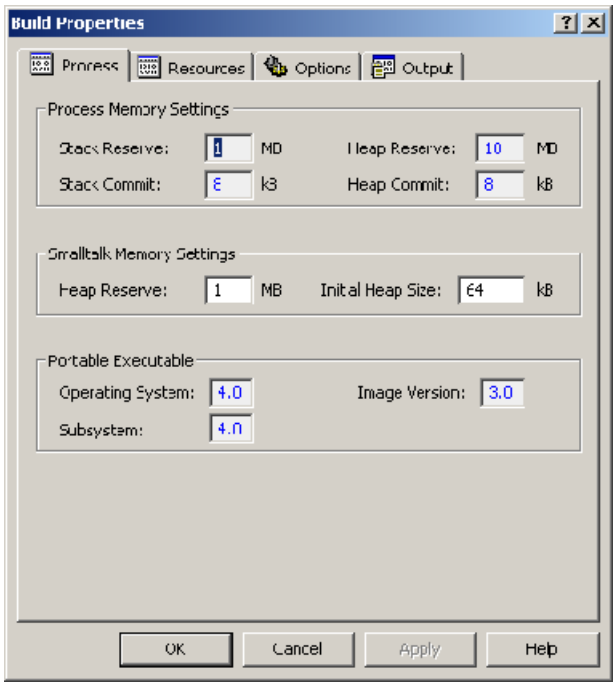
Build process properties

This page lets you define process parameters as well as the resource DLL that contains the target image resources.

The parameters are similar to the ones in the Image Properties of the development image, however they apply exclusively to the target executable. If the project does not specify its own settings, the values are inherited from the current image. For example, if the project does not define a Heap Reserve, the executable will be built with the Heap Reserve used by the development image.

The user interface represents default properties in a different color. To delete a project property, leave the field blank and click on *Apply*.

Figure 1-23 Build Properties – Process Page



Remarks

- The Smalltalk Heap Reserve can be set to a considerably lower level than the value used by the development image.
- The Initial Heap Size specifies the initial size of the private, growable heap for Smalltalk objects

Build resources

The project builder can bind an external binary resource or compile source code to a resource section. The second option assumes that a C++ compiler is installed.

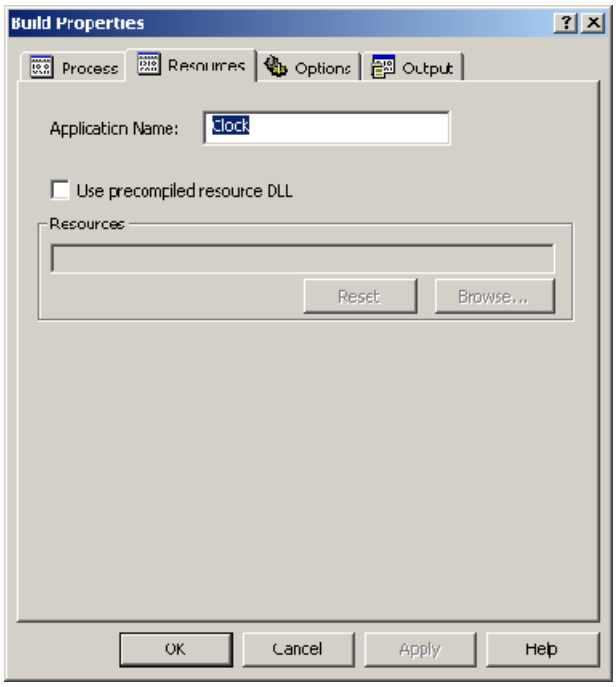
Using precompiled resources

On the Resources page, checking the box *Use precompiled resource DLL* allows you to enter a DLL with the binary resources. If you do not enter enter a file name, the builder looks for a resource DLL in the *Res* subdirectory of the project (the resource DL must have the same name as the project). If no DLL is found, the image is built without a resource section.

Compiling resources

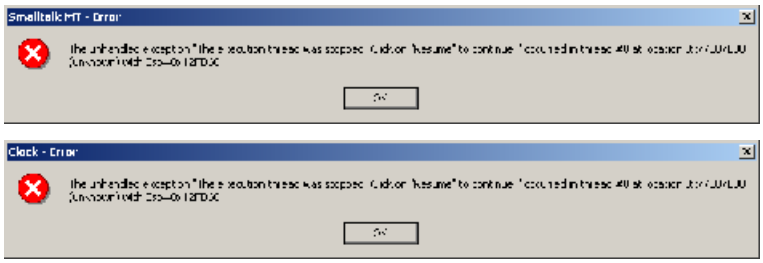
If the box *Use precompiled resource DLL* is unchecked, the project builder generates a message resource tailored to the application and builds the resource DLL automatically, if it is not already present (the *Rebuild* option in the Project menu forces a resource compilation).

Figure 1-24 Build Properties – Resources Page



The application name is used to generate messages that include the application name rather than “Smalltalk MT”. The figure below shows two runtime error message boxes, one generated with standard resources and the other with custom resources.

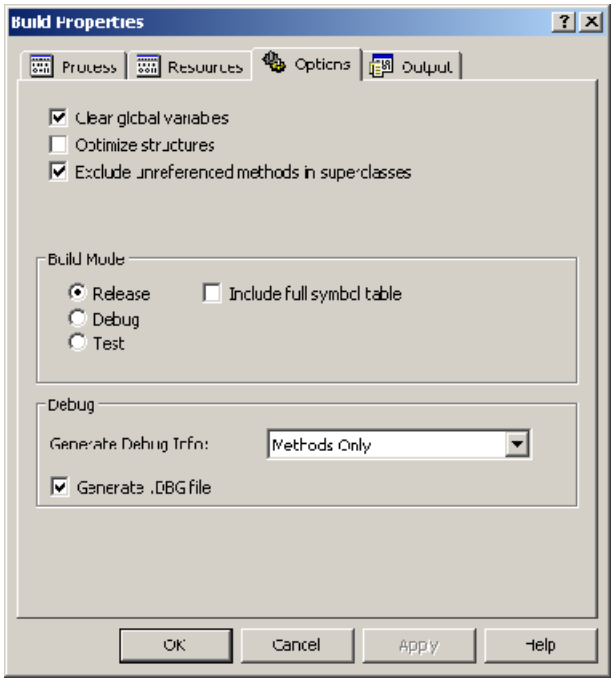
Figure 1-25 Build Properties – Resources Page



Automatic resource compilation makes it easier to maintain and build projects. In batch build mode, resources are always recompiled to ensure that the target image contains the most recent version.

Build options

Figure 1-26 Build Properties – Options Page



The Options page lets you specify the following options:

Clear global variables

When checked, global variables are set to **nil** before the image is built. Check this option if the project does not require initialized global variables.

Optimize structures

Checking this option causes the builder to optimize structure accessors, resulting in a slightly smaller image and faster structure accessing methods.

Exclude unreferenced methods in superclasses

This options lets the builder remove methods that are re-implemented in subclasses used by the target image. For example, the builder would likely remove `Collection>>includes:` because the method is re-implemented in subclasses.

Build Mode

- *Release* generates an optimized image.
- *Debug* generates an image that includes the Smalltalk runtime debugger.
- *Test* generates an image for test purposes. While the other build options install a runtime file that modifies the image's exception and error handling system, the test option does not modify the development image, so that errors that occur during the build can be debugged.

Note The debug switches in the Image Properties of the development environment also affect the output.

Debug Mode

The options that control the amount of COFF symbol information to be included are:

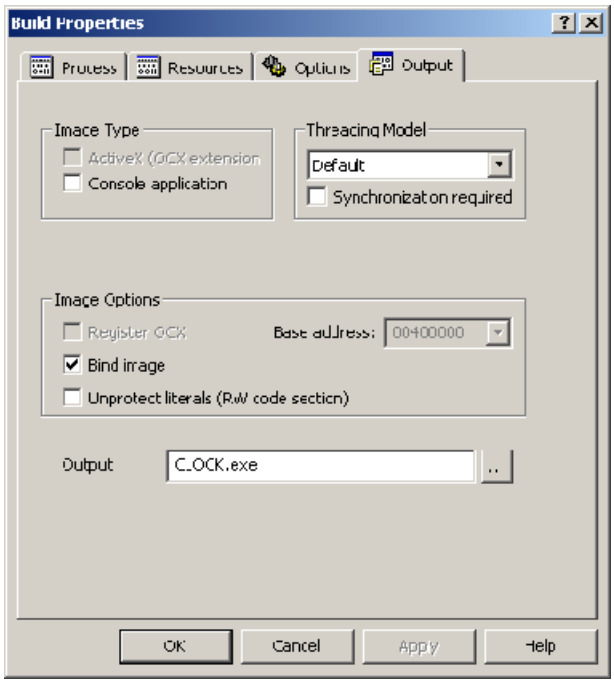
- *None*: no debug information is generated.
- *Methods*: method addresses are generated.
- *Methods + Symbols*: this option generates a COFF section and includes the used portion of the Smalltalk symbol table in the target image. It makes debugging a port-mortem dump easier.

Generate DBG info creates a separate debug file in the DBG format used by Win32 debuggers.

Note Debug Mode does not influence the format of the executable code and data. The executable portion of an image with COFF debug section is identical to the release version.

Build output

Figure 1-27 Build Properties – Output Page



This page lets you specify the type and properties of the target image.

Image Type

- *ActiveX* specifies an ActiveX component. The option is only available when the component is a DLL.
- *Console application* specifies a console application. The option is only available when the component is an EXE.

Threading Model

The threading model controls registry settings and COM initialization.

The development image supports free-threading from the ground up. It is up to a server application to decide which threading model to use. While the development image itself can be switched between apartment, free and neutral threading, each project may set the threading model in the build preferences. This means that a free-threaded image can

build an apartment or neutral threaded server image, while an apartment threaded image can build a free-threaded or neutral target image. It is also possible to specify *Default*; in this case the target image is built with the settings of the image.

Typically, an application that requires a given threading model will specify it in the build properties.

Apartment Threading (STA/MTA)

In the apartment model, a client app could call an object's methods only from the thread on which the object was created. This means that different objects could be called from different threads, but that each object would be called from only one thread. This in turn means that objects need to serialize their access to their servers' global variables and functions, but not to their own instance data. Under the apartment model, server objects are relatively easy to write, but clients are potentially tricky.

Free-Threading Model (FTM)

In the free threading model, a client app may call any object method or COM function from any thread at any time. It is up to the object to serialize access to all of its methods to whatever extent it requires to keep incoming calls from conflicting. It provides the maximum in performance and flexibility. The cost is that the objects themselves get harder to write compared to apartment model objects.

The FTM indicates that the object should straddle the apartment and context boundaries of the process and that each method call should be executed in the context and apartment that issued the call.

Both

Single-threaded or multithreaded apartment. *ThreadingModel=Both* simply means that the new object should be initialized in the activator's apartment, with all subsequent method calls being serviced there as well.

Neutral Apartment Model (NA)

A neutral apartment supports execution of its objects on any thread type and is the recommended threading model for COM components and COM+ applications.

On Windows 2000, the preferred ThreadingModel setting for non-visual components is *ThreadingModel=Neutral*.

Synchronization Attribute

Under Windows 2000, objects indicate that they need synchronized access using the Synchronization extended attribute. While the ThreadingModel setting indicates which threads in a process can dispatch calls to the object, the Synchronization attribute controls when these calls can be dispatched.

Image Options

- *Register OCX* automatically registers an ActiveX component after it has been built.
- *Bind Image* binds the target image so that it loads faster (on the current operating system).
- The *Base Address* field lets you specify a base address for a DLL or OCX. An executable always loads at the same base address, so there is no need to specify an address in this case and the option is disabled.

Finally, you can also specify the output file name and location. By default, the output directory is the development directory.

Installing a Sample Application

The installer sets up the library path so that you can easily install a project. Just open the **Project Browser** and install the project.

You are now ready to run the application. Each sample application comes with a `README.TXT` file or an HTML description that lists the additional steps required to start the sample application.

For example, the `README` file of the **Generic** application has the following code:

```
" register the window class for Generic "
Generic registerClass.
" opens Generic "
Generic new open.
```

Enter the code into the Transcript (or any other window) and evaluate it (*Execute*), or simply click on the class context menus *Register window* and *Test window* in the Class Hierarchy Browser.

Managing Projects

If possible, projects should not overlap (i.e., a given method or class should only be defined in one project at a time). If a component is used by several projects, make a separate project that becomes a prerequisite of projects that reuse the component in question. Note that prerequisite projects can have prerequisites as well.

Project Library Path

To facilitate the installation of projects with prerequisites, the Project Builder uses a **library path**. The **library path** defines the directories in which the browser looks for

prerequisites. By default, it points to the `SOURCE` subdirectory in the Smalltalk MT root, but it is possible to extend the path to include other directories as well. Click on *Library Path* in the *IDE Preferences* page of the Image Properties and enter the directories to scan, separated by semi colons. Note that the library path is not an environment variable; it is specific to each Smalltalk MT installation.

Example:

To add the subdirectory `PROJECTS` to the path, open the library prompter as described above and add the following line:

```
c:\program files\smalltalk mt\source\*; c:\program files\smalltalk mt\projects
```

You can add wildcards to let the builder search an entire directory branch. In the example above, the builder will search files in the directory branch rooted at `c:\program files\smalltalk mt\source`.

Class Initialization Methods

After a class owned by a project is installed, it is sent the message `initClass`. Implementations of this method either call a class `initialize` method or perform some initializations of the class. The default implementation of `initClass` (in **Class**) does nothing.

You can re-implement `initClass` to perform initializations once the class has been installed (but do not forget to call `super initClass`).

Using Third-Party Resource Files

You can use a third party resource editor to build the resources used by your application (menus, bitmaps, icons, dialogs, accelerators...), the only requirement being that the result is available in DLL format.

Most builders generate an include file that defines the constants used by the resources. You can file-in the constants as a pool dictionary and add it to the project (and to the classes that use those constants). To do so, open the *Symbol Editor* and load the include file in question.

Note If you use Microsoft Visual C++, the resource editor may use default constants such as IDC_CHECK1, ID_FILE_OPEN, etc. These constants are already defined in Smalltalk MT, but with differing values (MSVC generates most values on a case-by-case basis). In order to promote reusability, we recommend changing the #include statement to refer to RCST.H (in the SUPPORT\INCLUDE subdirectory).

Generating a stand-alone application

If you follow the instructions above, compiling an executable is as simple as clicking on *Build project.exe* in the *Project* menu of the **Project Browser**.

The executable must not reference the compiler, the debugger, or use development code. In particular, it cannot have `self halt` statements.

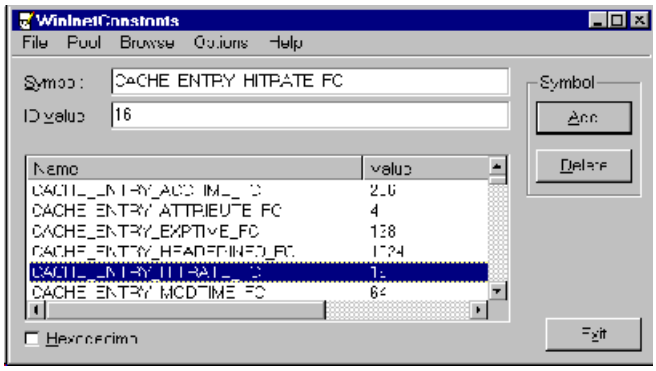
Symbol Editor

Abstract

The **Symbol Editor** lets you edit pool dictionaries. Pool dictionaries appear in the definition of a class and define constants used by the class, similar to the way include files are used in C.

After you select a pool dictionary to edit, the **Symbol Editor** displays the key and value pairs that define the pool's associations. Clicking on an entry of the list sets the key and value fields.

Figure 1-28 Symbol Editor



The symbol field must contain a valid identifier. The value field accepts any valid Smalltalk expression that evaluates to a constant.

Working with Files

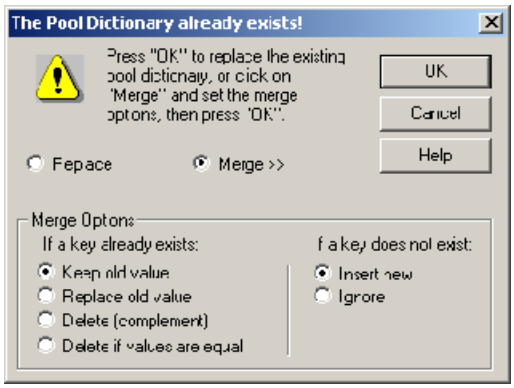
A pool dictionary can be stored on disk in two formats; the Smalltalk-specific `storeOn:` format and a C-compatible format that uses `#define` statements. You can also import a pool dictionary from disk.

When you open an include file, you are prompted for a name under which the new dictionary should be installed. Because Symbol Editor remembers the file name that corresponds to a pool dictionary, the prompter displays a default name if the pool has already been installed from a file with that name.

In the event that the pool dictionary already exists, you are given more options:

- *Replace* simply replaces the old dictionary with the new (default).
- *Merge* lets you define how to handle old key-value pairs that would be overwritten and new ones that did not exist in the old. These options are useful when you want to merge include files or update values without inserting new elements.
- *Delete* creates a complement of the existing pool and the new pool
- *Delete if values are equal* creates a pool that only contains the values that have changed.

Figure 1-29 Merging Pool Dictionaries



Example 1:

To create a pool dictionary that contains the declarations of several include files, click on the *merge* option and on *Keep old value*, *Insert new*.

Example 2:

To update the values of an existing pool dictionary, click on *merge* and *Replace old value*. Key-value pairs that are not in the *include* file are left unchanged.

Example 3:

To insert new key-value pairs without modifying existing data, click on *merge*, *Keep old value*, and *Insert new*.

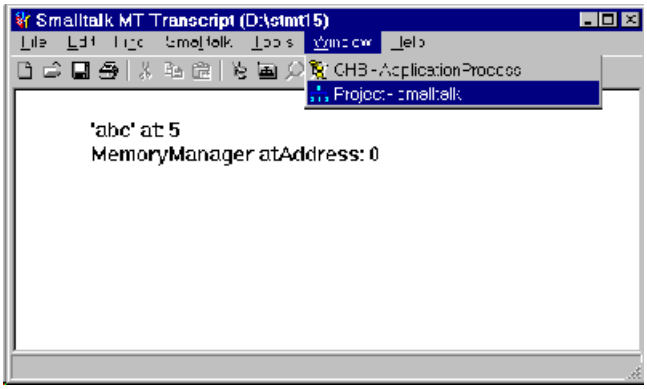
Transcript Window

The Transcript Window is a text window that is always open during a Smalltalk session. It is intended for system messages, and is also the starting point for system maintenance tools. It is referenced by the thread-local variable **Transcript**, so you can print simple messages using:

```
Transcript show: 'some info\n'.
```

The Transcript displays also graphical menu that lists all currently opened Smalltalk windows.

Figure 1-30 Transcript Window



Note The Transcript window, like workspaces, understands the RTF format. However, the contents are not saved in RTF format.

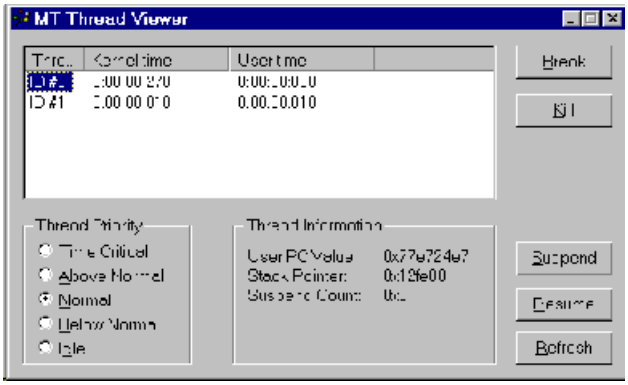
Workspace Windows

Workspace windows are text editors that let you evaluate Smalltalk code. You can set the title of a workspace by clicking on the system menu.

Thread Viewer

Thread Viewer is a utility that monitors Smalltalk threads. It is useful for stopping and debugging a thread or getting out of an infinite loop. It runs in a separate thread and lets the user set a thread's priority, suspend and resume threads, as well as break and kill threads.

Figure 1-31 Thread Viewer



Breaking a thread entails raising a resumable exception in the thread, which brings up the walkback dialog with the options of debugging, discarding, resuming or invoking a top-level handler.

ThreadViewer can be started from the Transcript's *Tools*|*Thread Viewer* menu. This installs a tray icon on the task bar. Double-click (or right-click and choose *Open*) the icon to open the actual window.

To run **ThreadViewer** automatically, check the corresponding box in the image preference sheet. See also *Image Properties* on page 26.

Breaking a thread

Select the thread you wish to stop and click on *break*. **Thread Viewer** then enters a loop that tries to stop the thread in user code (Smalltalk code). If unsuccessful, a dialog box pops up that asks whether you wish to break it in system code. Beware that breaking a thread in system code may have adverse effects, especially on Windows 9x.

Killing a thread

This option is useful if you are confronted with thread synchronization problems (a deadlocking thread for example), and lets you get rid of secondary threads. You can also kill the main thread, but this leaves you with little options except exiting the process.

Suspending and resuming a thread

This operation is straightforward, and the thread's suspend count is displayed in the window.

Problems using ThreadView

You may experience difficulties when a main thread is starving **ThreadView**, which makes the applet unresponsive. It happens mostly on NT uniprocessor systems, less on Windows 98 and never on multiprocessor machines.

Image Maintenance

Overview

The Smalltalk MT development environment comes in form of a loader, an executable image and an associated source file. The executable file contains the Smalltalk object code and data.

During the development cycle, the executable image is modified as you add your own data and code, and perhaps change existing data structures and code. The source file `SOURCES.BIN` contains the source code, while the log file `CHANGES.SL` keeps track of all modifications to the system.

The *image save* operation lets you record the changes to disk. This generates a new executable and associated source file. The loader backs up the current version and starts the new image.

Because both the source file and the executable can become fragmented, it may be useful to *compress* the image. You must also compress the image after you modify the set of statically linked dynamic link libraries.

Saving an Image

A Smalltalk session is characterized by a set of objects and data structures in the process space. Saving the session entails taking a snapshot of the objects in the system and serializing this information to disk for later retrieval.

The files with the updated contents are respectively `STIMAGE2.EXE` and `SOURCES2.BIN`. The loader `STMT.EXE` renames the current image set (`STIMAGE.EXE` and `SOURCES.BIN`) into `STIMAGE.BAK` and `SOURCES.BAK`, and the new set into `STIMAGE.EXE` and `SOURCES.BIN`. Calling the loader with the */restore* option reverts to the former settings and can be used if the new image fails to start.

The file `CHANGES.SL` logs all modifications in the system. You can use the contents of this file to recover code after an image crash. The log file can be deleted when the information it contains is no longer needed, thus recovering disk space.

`OLDSOURCES.SL` is an ASCII file that contains overwritten methods. You can browse previous versions of a method in the ClassHierarchy Browser by clicking on *Changes*.

See also *Browsing Changes* on page 22 and *Old Sources* Log on page 65.

Command Line Options

The loader program `STMT.EXE` accepts certain switches that are detailed in the table below. These switches are not passed to the executable image but ‚eaten' by the loader.

Table 1-14 Command Line Switches

Switch	Effect
/nobackup	Unconditionally overwrites STIMAGE.EXE and SOURCES.BIN with STIMAGE2.EXE and SOURCES2.BIN, and without first backing up the files.
/restore	Overwrites STIMAGE.EXE and SOURCES.BIN with respectively STIMAGE.BAK and SOURCES.BAK.

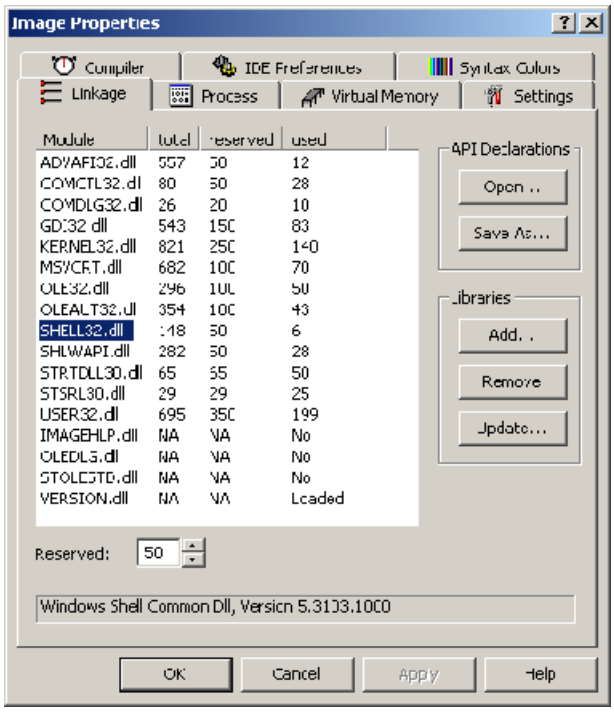
Modifying the Import Section

Smalltalk MT uses the native DLL binding mechanism for optimal performance. Before you can use an API in a module, you first link the module to the Smalltalk image and load the API specifications. You do so by opening the **Image Properties** sheet from the Transcript's *Tools* menu.

The Imports property dialog

This dialog manages the import section of the Smalltalk image. The *Linkage* page of the **Image Properties** sheet lets you add, remove and update module imports. However, the changes will not take effect until you save your image with the *compress* option.

Figure 1-32 Image Properties – Linkage



The list shows the imported modules, the number of exports in each module, the number of reserved exports, and the number of exports used in each module.

Clicking on a module displays the library name and file version in the text field below the list.

Table 1-15 Import Page Fields and Actions

Column	Meaning
Total	This column displays the number of APIs currently defined in each module.
Reserved	This column lists the number of reserved API import slots for the module. By default, this equals to the number of available APIs, unless you specify a different value in the *Reserved Imports* field below. Specifying a lower value reduces the number of APIs available simultaneously from the current module but also reduces the size of the import section.
Used	The last column shows the number of APIs currently imported, per module. The column is updated each time the image is compressed.

Add...	*Add* leads to a **File Open** dialog box that lets you specify a module (a DLL) to be imported.
Update...	*Update* prompts for the pathname of the DLL to update. By default, the pathname of the DLL in the path is displayed, but it is possible to specify an alternate path. You can use this option to update the API properties of a module that has changed. Note that this option does not modify existing API declarations; it only updates internal properties such as the ordinal and hint used when linking.
Remove	You can remove the selected module by pressing this button. You should only do this if the image does not import functions from this module (the number of used functions should show zero).
Open	*Open* lets you load an import definition list for the currently selected module (see also *Save As*).
Save as	*Save As* saves the import definition list of the selected module in a declaration (.DEF) text file in ASCII format. This is useful if you want to edit the file in a text editor (see also below).

Increasing the Number of Reserved Entries

It is often convenient to adjust the number of reserved slots (the number of functions effectively linked). Many DLLs have a large number of exports such as ANSI and Unicode versions of APIs, undocumented (private) exports and more generally functions that are not likely to be ever used.

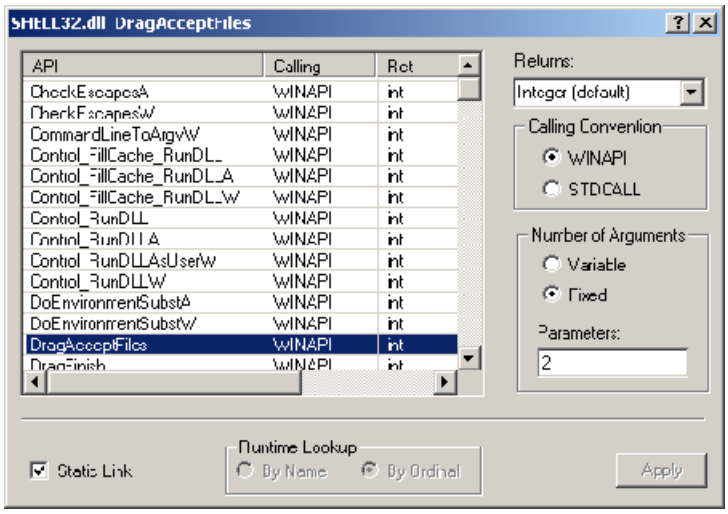
When the number of slots allocated to a particular DLL is exhausted, the compiler raises an error the next time you try to reference a new function of that DLL. In this case, you must increase the number of reserved entries in the import property page.

DLL Properties

By right-clicking on a module name, you bring up a popup menu that lets you edit the API entries of a module. You can also double-click on the module name to display this window.

The DLL Properties dialog lists all functions exported by the selected module, along with the calling convention, return type and arguments. Selecting a function displays the API properties in the same way as in the API property editor.

Figure 1-33 DLL Properties Dialog



Note	An argument count of -1 signifies that the function is currently unused. The number of arguments is updated the first time you use the function.

Working with Imports

To link with a DLL, click on *Add* and enter the name of the DLL. You just need to enter the name; the system will look in the path of the process to locate the library. However, you can also specify the absolute path name, for example if the image is currently linked with a DLL and you wish to specify a different version of that DLL.

When a DLL is no longer used, you can remove it by clicking on *Remove*. The used field should display zero before you remove the import.

Click on *Update* to reload the list of exported functions and update internal DLL properties. For example, it is a good idea to update system DLLs after a new version of the operating system has been released. This will not only give you access to new functionality but also improves the loading time of the process.

The buttons *Open* and *Save As* respectively read and write an ASCII file that lists the exported functions and API properties of the selected module. The file is saved with a .DEF extension. When the compiler binds with a library, it first looks for a corresponding DEF file before it installs the DLL. If no DEF file is found, the library is installed with default properties.

Note To move APIs from one DLL to another, update both entries, then click on the Optimizations tab and check the *recompile all methods* box.

Example LIBA.DLL currently contains a function `MyFunc`. You would like to link with a modified LIBA.DLL that does not contain this function, and with LIBB.DLL which contains MyFunc. Click on *update*, specify the pathname to the modified DLLs, check *recompile all methods*, then save, and exit the image. Copy the modified DLLs to your path and restart.

Runtime binding

Smalltalk MT also supports runtime binding. In this mode, a module is only loaded when needed and the function addresses are retrieved at runtime. For modules that are seldom needed, this has several benefits:

- The working set of the application is reduced.
- Application start-up time is improved.
- The application can test for the presence of a given DLL and degrade gracefully if the DLL is missing or is not of the expected version. With static linking, the application would not start at all.

Runtime binding can be configured for each module, and it does not affect application code. On the other hand, static binding is the best way to link required modules or to ensure that all function references can be resolved. And calling statically bound functions is faster than calling dynamically bound ones.

How to...

Enable runtime binding

You can browse the methods that reference a selected module by right-clicking on the module and selecting *Browse References*. This allows you to determine whether the module is a candidate for dynamic binding.

Click on *Change to delay-load* to load the module dynamically. The changes take effect after you save and restart the image. See also below for binding by ordinals vs. binding by function name.

Determine if runtime binding should be used

After executing your application, open the import property page. The *Used* column of each dynamically bound module displays *No* if the module has not been loaded, *Loaded* otherwise. If the module has been loaded after simply starting the application and

performing a couple of common operations, it is probably more efficient to link the module statically.

You can also build an executable of your application. The compiler warns when the target executable references dynamically bound libraries.

Note If `VERSION.DLL` is bound dynamically, the import properties page will load it when displaying version information. Therefore, `VERSION.DLL` will always appear as loaded.

Enable static binding

Proceed as above, but substitute *Change to static link* for *Change to delay-load*.

Configure Ordinal vs. Name Binding

In the import property page, double-click on a module to edit module properties. For dynamically bound modules, you can click on *By Name* to bind by name and *By Ordinal* to bind by ordinal. After modifying any of those values, you must save and restart the image.

Exported functions are identified by name and by ordinal. Binding by ordinal is faster and uses less space, while using names is more robust. If a function may not exist or if different versions of the module have conflicting ordinals, you must use name binding.

Unload a Library

An application may wish to unload a module after it has been used in order to reduce the working set of the process. Dynamic binding is implemented by **RtModuleLoader**. The method `freeLibrary:` frees a library that has been previously loaded dynamically.

Summary

An application can take advantage of runtime binding for functionality that is seldom used, therefore reducing the working set of the application. Another scenario involves libraries that must be accessed at runtime with `LoadLibrary` and `GetProcAddress`. In this case, using dynamic binding is an elegant alternative because it achieves the same result without placing a burden on the application code, which calls the functions as usual.

Finally, dynamic binding gives an application flexibility in terms of functions supported by the module. For example, an application can determine the version of the module that has been loaded and degrade its functionality gracefully, while a statically bound image would fail to load if an imported function cannot be found.

Virtual Memory Settings

The **Image Properties** sheet has tabs for specifying virtual memory (Virtual Memory page) and process memory settings (Process page).

The two most often used fields are:

- The size of the source file. The source file is a memory-mapped file and as such, it needs a pre-allocated size. The value in the field must be larger than the size of the file when it is committed to disk (e.g., after Smalltalk MT has exited). The value affects runtime performance in the Windows 9x/NT edition, however, the performance of images in Windows NT mode is not affected.
- The amount of reserved memory (Heap Reserve and Initial Heap size). This setting affects both the working set size and runtime performance. There is no optimal value, because performance depends on both the memory profile of your application (whether it produces a lot of garbage, requires a large number of objects, and other factors) and the hardware. If the allocated memory exceeds available RAM, overall performance degrades because of swapping, while a lower setting may actually yield much better performance. On the other hand, setting it too low may result in more activity in the garbage collection thread. In addition, the value represents a maximum and does not imply that the memory is actually committed. In most cases, the default value is just fine.
  - Heap Reserve: this value defines how much memory is reserved for Smalltalk object spaces. An image has 8 fixed-size spaces in which objects are allocated, plus a growable heap that is used for large objects and when the fixed-size spaces are full.
  - Initial Heap Size: this value defines how much memory is initially committed in the Smalltalk heap. If an image requires more memory that the sum of the object spaces and the heap, the heap needs to grow. Growing a heap many times is a potentially costly operation.

Section Sizes

Overview

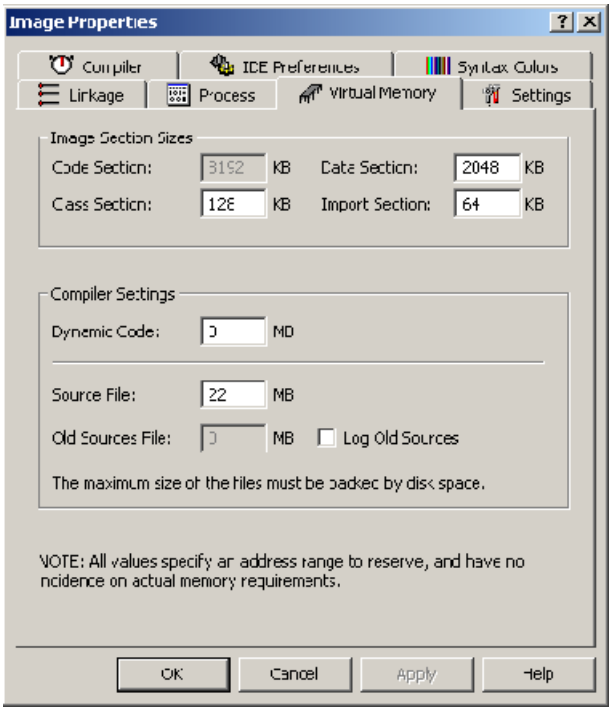
Smalltalk MT images follow the PE format and contain a collection of sections. A section is an area in an executable file that is mapped to memory by the loader of the

operating system. A section has a physical size (the size it takes up in the executable file), a virtual address and virtual size. The virtual size specifies the memory range to reserve. Obviously, the physical size is always less than or equal to the virtual size.

The Smalltalk MT development image comes with predefined virtual section sizes. The virtual section sizes must be larger than the physical sizes in order to allow the image to grow. If a section cannot hold enough data, it is necessary to increase the virtual section size. This can be done in the *Virtual Memory* tab of the **Image Properties** sheet. If there is not enough space, the image save operation fails.

Note When a runtime image is generated, all sections are optimized to minimize the virtual address space (since the executable cannot grow anyway). Therefore, changing section sizes has no incidence on a final executable.

Figure 1-34 Image Properties - Section Sizes



Code section

All referenced code is relocated to the code section when the image is saved. The code in the code section accumulates until the image is compressed. It is therefore necessary to compress the image from time to time, as this eliminates superfluous code.

Class section

The class section contains the classes that are in the image (but not the classes' contents, - see below). Since a class takes only 28 bytes plus 4 bytes per class instance variable, a small class section is sufficient.

Data section

The data section contains all static data such as objects referenced by global variables, class variables and class instance variables. It also contains the method dictionaries and other internal data.

The default value of the data section is 1MB, which may not be enough for programs that use large amounts of static data (i.e., data saved with the image). In such a case, the image save operation fails and you must first restart the image and increase the size of the data section.

Import section

The import section contains the import directory for the process. Each DLL linked with the image is associated with a reserved area in this section. The requirements increase when additional libraries are linked to the image.

The size of the import section must always be greater than the number of effectively used entries. The fill factor is displayed in a tooltip window when the mouse cursor is over the import section size field (in the *Image* tab of the **Image Properties** sheet).

Compiler settings

The compiler settings define the memory that can be allocated for compiling as well as the size of the source file.

Dynamic Code Size

In the development image, the compiler reserves a memory region to hold the compiled code. The compiled code generally accumulates until the image is saved (there is an optimization for unreferenced evaluations).

When the dynamic code section is full, the compiler warns that it cannot compile code anymore, and the image must be saved and restarted.

Source file

The source file is a memory-mapped file. It is necessary to specify a maximum size for the file (the Win32 API requires this). The source file also grows until an image is compressed.

Old Sources Log

This optional feature logs previous editions of method source code, allowing you to browse and restore previous editions of a method.

Some sample section sizes

The tables below lists some sample section sizes for a development image and an application.

Table 1-16 Development Image Sections

Section	Size in kB
.text	940
.gdata	17
.data	426
.idata	20

The runtime section sizes of a non-trivial executable such as the FileFind sample (`FILEFIND.EXE`) are listed in the following table.

Table 1-17 Setup Sample Sections

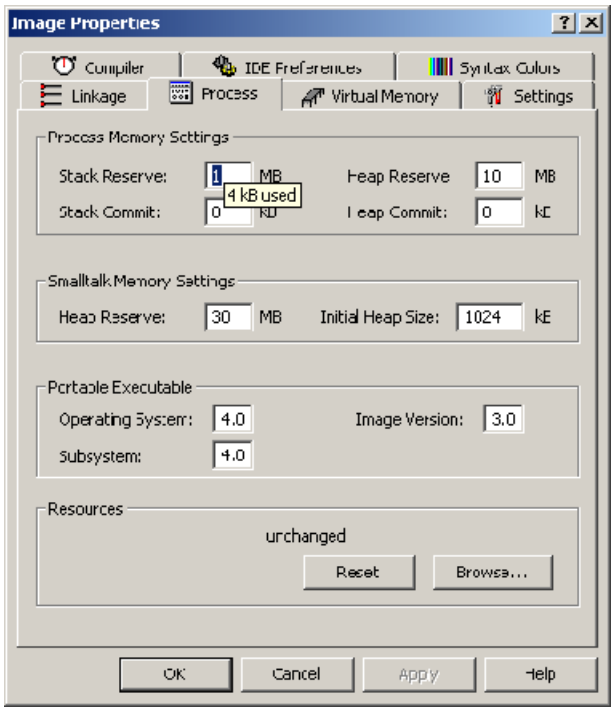
Section	Size in kB
.text (code)	87
.gdata (class)	6
.data (data)	35
.idata (import)	5
.rsrc (resources)	15

Process Properties

Overview

The process property page lets you edit properties of the executable image. The properties do also apply to generated executables. Except for the *Smalltalk heap* and *Resources* field, all values are stored in the PE (Portable Executable) header.

Figure 1-35 Image Properties - Process



Smalltalk Heap Reserve

The **Smalltalk Heap Reserve** specifies the total memory to reserve for Smalltalk. The memory actually committed is always below the total.

You can often decrease this value for executables generated with Smalltalk MT.

Stack Sizes

The reserved and committed stack size fields let you adjust the default process stack settings.

By default, space is reserved in the following manner:

- 1 megabyte (MB) reserved (total virtual address space for the stack)
- 1 page committed (total physical memory allocated when stack is created)

The operating system will grow the stack as needed by committing 1-page blocks out of the reserved stack memory. When the reserved memory region has been committed, a stack exception will occur. Stack exceptions are in general not recoverable.

Each new thread gets its own stack space of committed and reserved memory. By default, the values are taken from the process properties.

Heap Sizes

The reserved and committed heap size fields let you adjust the default process heap. The default heap is used by the C runtime and some operating system functions.

Version Fields

The version fields specify the operating system and subsystem version required by the process, as well as a private image version.

If the operating system or subsystem versions are higher than the current Windows version, the process will not start.

Resources

You can change the resources in the resource section of the image by specifying a resource DLL in this field.

Restoring an Image

Although Smalltalk MT automatically creates a backup file each time you save the image, it is desirable to keep permanently backed up copies of an image. Because the Smalltalk image evolves as you program, there is a possibility of image corruption (for example, a pointer bug might damage an internal data structure and go unnoticed for a while). That is, you may end up with both the current image and the backup (`.BAK`) files being corrupted.

There are several ways to restore an image. Before you do so, copy the current image to another directory, in case you need it later.

Reinstalling changes

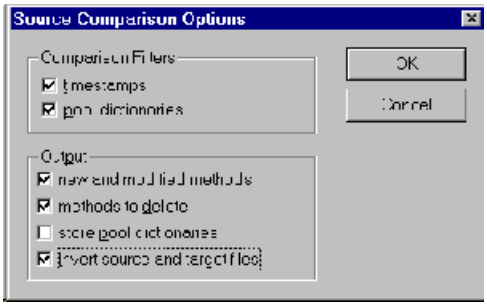
In the case of a minor problem, or if the method outlined above does not work for you, you can open the log file `CHANGES.SL` and selectively file-in the source code it contains.

Installing Image Differences

This method generates a file-in which, when applied, transforms the code of the current image to a target image.

Click on *Source Comparisons* in the Transcript. You are then prompted for a target image (a `SOURCES.BIN` file) and for the name of the file-in to be created.

Figure 1-36 Source Comparisons



Example:

Create a new class **ATest** with a method named `test`. Save and restart your image.

- Click on *Source Comparisons*.
- Enter the file `SOURCES.BAK`.
- Save the file-in under `TEST.SM`.
- Click on *Invert source and target files*. Otherwise, the file-in just deletes Atest.

The file-in contains the following code:

```
"Pool Differences"!

"Class Differences"!
Object subclass: #ATest
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''!

"Method Differences "!
"Methods to ADD "!

!ATest * methods!
test
    ^self! !
```

Restoring an image from a source file

This method synchronizes an image with a given source file by recompiling the source code. It is recommended that you only try this technique as a last resort, or at least keep backup copies and test the new image thoroughly.

1. Copy the backup `STIMAGE.EXE` file to a temporary directory. Copy the source file `SOURCES.BIN` of the corrupted image to the same directory.
2. If necessary, link with all libraries that are used by the corrupted image.
3. Start the image and file-in `UPDSRC.SM`, which can be found in the support directory.
4. Open the Image Properties sheet, click on the *Optimizations* tab and check the recompile all sources box. Click on *Ok*, then save and compress the image.
5. You now have a new image that contains the code of the corrupted image. You may have to perform additional class initializations.

UPDSRC.SM

The file-in installs the classes that are in the source file but not in the image.

```
| fContinue newDescriptors |
newDescriptors := OrderedCollection new.
[
    fContinue := false.
    Compiler classDescriptors do: [ :szClass :cd |
        (cd isMemberOf: ClassDescriptor) ifTrue: [
            (Class fromName: cd name) isNil ifTrue: [
                cd superclass notNil ifTrue: [
                    Compiler evaluate: cd storeString.
                    newDescriptors add: cd
                ]
                ifFalse: [
                    fContinue := true
                ]
            ]
        ]
    ].
    fContinue
] whileTrue: [].

newDescriptors do: [ :cd |
    Compiler classDescriptors at: cd name put: cd
].
```

Troubleshooting

Development Environment

Runtime exception while compiling code

The compiler tries to evaluate constant expressions and, if successful, stores the result rather than generating the code that performs the operation. For example, an expression such as the one below is evaluated at compile time.

```
CS_HREDRAW|CS_VREDRAW|CS_DBLCLKS
```

If the operation fails because of an exception, the compiler does not try to handle it and gives the programmer an opportunity to analyze the error. The reason is that if an expression involving constants fails at compile time, there is also a good chance that it fails at runtime as well.

For example, evaluating `1 // 0` raises a runtime exception while compiling.

In some cases, the exception occurs because an operation involves methods that are not installed when the code is compiled. Typically, this can occur when installing a project. The solution is to isolate the methods that replace or extend the base classes into a prerequisite project. This project is installed safely into the image before the main project code is filed-in.

Invalid library path

This message occurs when the **Libraries** value of the key

```
HKEY_CURRENT_USER\Software\ObjectConnect\Smalltalk MT\version\Root
```

is incorrect or missing.

To correct the situation, open the **Image Properties**, click on *IDE Preferences*, *Library Path* and enter the library path. It must at least include the `SOURCE` subdirectory of the Smalltalk installation.

Problems installing a Project

Project Browser fails to find sub-project

This error may occur when the registry entry that lists the source path is missing. In the **Image Properties**, click on *Library Path* under *IDE Preferences*, and enter the path to the source directory (e.g., `C:\PROGRAM FILES\SMALLTALK MT\SOURCE`).

If the sub-project is not in this directory, you have to either append a path entry or locate the project.

Syntax Error while filing in the project

A text window pops up with the syntax error. If correcting the error requires modifications in the project's source code, open a workspace and edit the project file (`.SP`). Do not change the first part of the file, as it contains project definitions. The second part contains the source code in chunk format. Save the file and retry.

Application fails to load resources

Most applications have an associated resource DLL that must be in your path before you can open the application. For the samples, the DLL is in the `RES` subdirectory of the project (i.e., `PROJECTS\GENERIC\RES` for the *Generic* sample). Copy the DLL (after compiling it if necessary) to your path.

Starting an application usually involves the following code sequence:

```
MyAppWindow registerClass.
MyAppWindow new open.
```

Application fails to load resources when compiled as stand-alone executable

The image builder loads the executable's resources from the DLL in the RES subdirectory (see above). The DLL must have the same name as the project (e.g., GENERIC.DLL for the Generic project). If the DLL is not present, no resources are compiled into the executable.

Opening a project file displays nothing, or generates an exception

The file in question is not a valid project file.

Problems Generating an Executable Image

Error while generating the executable

Errors that occur while generating an executable can be difficult to track because the builder installs runtime error handling. The runtime-specific code is in a set of files named RUNTIME_XXX.SM. In order to detect the error that causes a build to fail, check the *Test* build mode in the *Build Properties* of the application and rebuild the target. The *test* build mode generates a target image that includes the development debugging methods.

Note A very large executable indicates a build failure. The builder creates a mapping file and truncates it when the build completes successfully. If the build fails, the file is not truncated and is unusable.

Runtime errors

First, make sure that the diagnostic messages in the *Image Properties* sheet are enabled. They will allow you to track the problem more easily.

Many diagnostic messages are printed on an external debugger, so the second step should be to use a utility such as **DBMON** or to load the executable under **WinDbg** or another debugger.

In order to view symbolic information in a third party debugger, you must build a debugging version of the program.

Error loading a resource

If the resource (such as a menu) loaded correctly in the development image, make sure that the `RES` subdirectory contains the resource DLL as well.

Check the methods `initialize` and `registerClass` in the window class. The samples and the code generated by the application wizard define those methods under the development-only category (`.INTERNALDEV`). The runtime code uses the resources defined in the executable itself rather than the DLL (which is not needed at runtime).

You must also copy the resources for the following components:

- **Prompter**: requires dialog templates for the different prompter modes (simple, combo, list).
- **SplitPane**: requires the split cursors

The resources are in the `SUPPORT` subdirectory. See also *Application Resources* on page 354 for more details.

Window Creation fails

Make sure that the start-up file `WINMAIN.SM` performs all required window class registrations. If you use external libraries (common controls[1], OLE, etc.), you must call their initialization function at startup.

A failure to create a window raises an exception (invalid handle). If you enable debugging, a message box with the error is displayed.

Exception at runtime

Most exceptions are of the type *Message not understood*. Use DBMON or start the executable in an external debugger and watch for diagnostic messages of the type:

"An instance of class XXX did not understand Symbol(0xHH). The message was sent by an instance of YYY."

To find out the string value of the symbol, start the development image and evaluate:

```
Symbol value: 0xHH
```

Methods missing at runtime

There are several reasons why a method is not included in the runtime image:

[1] Initialization of the common control library is required on Windows 98 but not on NT

- The method is a development-time method (category `INTERNALDEV`) or is called only by development methods.
- The method is called indirectly (via `perform:`) and without using explicit symbols. Set the category of the method to `CALLBACK` (this forces the method to be included in the target image). An example of such a case is the Windows message dispatcher in Window.

Classes missing at runtime

If a class is referenced but not present, the target image crashes. A common cause is that the class is referenced by a class or global variable but not by code. The solutions are:

- Reference the class statically (i.e., by referencing it from code that gets executed)
- Add a class method named `addReferencedObjects:` to the class that references the missing classes. The method must add the classes in question the set that is passed as argument to the method. The method itself is a development-time method that gets called by the builder.

Debugging Issues

Abstract

Smalltalk MT can generate useful diagnostic messages that help you track down problems. The messages are printed on an external debugger, or, in the case of fatal errors, logged as events in the application log (on Windows NT).

Using an external Debugger

You may want to use an external debugger for any of the following reasons:

- viewing debugging messages (via `OutputDebugString`)
- tracing into external DLL code
- system-level debugging

The process to load is `STIMAGE.EXE`. Make sure you set the current directory to the Smalltalk directory that contains `STIMAGE.EXE` (in the project settings if you use MSVC). If you use the NT edition of Smalltalk MT, you must also set access violations to *notify* because the NT image internally changes the protection state of virtual memory.

For just-in-time debugging, insert the following code into your code:

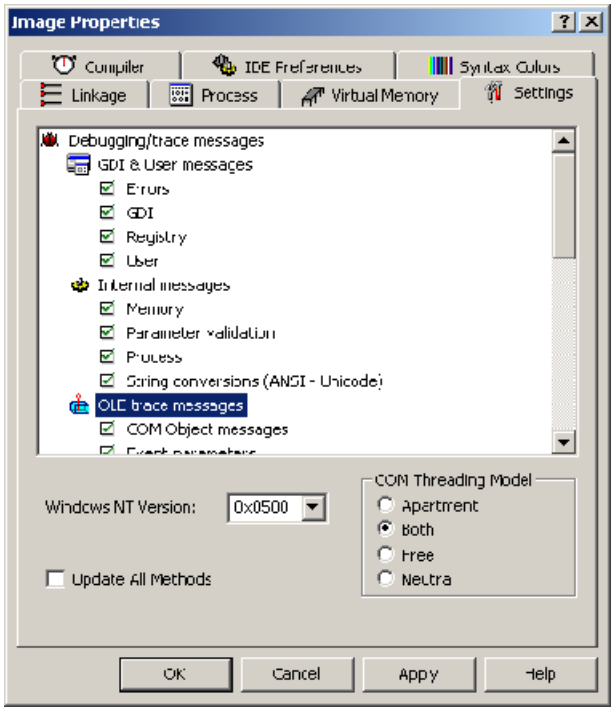
```
Self _break.
```

The debugger will breakpoint inside the method, allowing you to step out and trace into an external function.

Enabling Diagnostic Messages

You can enable specific debugging messages in the **Image Properties** sheet on the *Settings* page.

Figure 1-37 Image Properties - Debugging



The messages are printed on an external debugger via `OutputDebugString`.

Table 1-18 General Diagnostic Messages

Message Type	Description	Constants
Errors	Logs a message when an error occurs. The message can be logged in the application event (on Windows NT/2000), printed to a debugger, or displayed in a message box. Which output medium is used depends on the variable *LogMode* in MessageBox.	`_DEBUG_USER_ERRORS`
Process	Monitors certain kernel calls.	`_DEBUG_INTERNAL`
Memory	Enables memory probes and signals memory corruption. If a memory corruption occurs, a diagnostic message is printed on the debugger, if any, and logged in the application log (on Windows NT).	`_DEBUG_INTERNAL_ALLOC, _DEBUG_INTERNALDEV`
Parameter	Enables parameter validation.	`_DEBUG_PARAM`
String Conversions	Prints a message on ANSI - Unicode conversions. Useful when porting an ANSI application to Unicode.	`_DEBUG_STRING`
GDI	Monitors GDI objects. Useful in conjunction with the debug release of Windows to detect GDI problems, for example deleting a used GDI object.	`_DEBUG_GDI`
User	Enables diagnostic messages when a User API fails. Recommended during development.	`_DEBUG_USER`

Table 1-19 OLE Diagnostic Messages

OLE Message	Description	Constant
General	General OLE messages.	`_DEBUG_OLE`
Reference counting	Monitors *AddRef* and *Release*.	`_DEBUG_OLE_REF`
Interface Creation	Monitors OLE Interface creations.	`_DEBUG_OLE_CREATE`
IN	Monitors OLE callbacks.	`_DEBUG_OLE_IN`
OUT	Monitors OLE interface calls.	`_DEBUG_OLE_OUT`

QueryInterface	Monitors all calls (*Log all*) or only the ones that fail (*E_NOINTERFACE*).	`_DEBUG_OLE_QI` `_DEBUG_OLE_QI_FAIL`
Server registration	Logs registry entries as they are created or deleted.	`_DEBUG_OLE_REGISTRY`

Typically, the constants are used as follows:

```
_DEBUG_XXX == TRUE ifTrue: [
" print some diagnostic message or perform validations "
....
].
```

The compiler optimizes the statements because the value of `_DEBUG_XXX` is known at compile time. If you set the constant to `FALSE` (using **Symbol Editor** or **Image Properties**), the code is not generated. The effect is similar to an `#ifdef` preprocessor statement in C/C++.

Check *Update All Methods* if any changes you made should be global. Otherwise, only code that is compiled after you exit the dialog will be affected.

Using the build log file

The builder generates a log file (`build.log`) along with the executable. The log file contains a complete list of the methods and classes that are in the target image and serves a similar purpose as the *.map* file generated by C/C++ compilers. The log file can be used for several purposes:

Port-mortem debugging

If the target executable fails and has no debugging support (neither runtime debugging nor COFF symbols), the addresses of the log file can be used to determine the call stack at the time the exception occurs. The Post Mortem Viewer can be used to view the information in the log file. Using the log file is a poor substitute for runtime debugging but is effective and works even if the executable has no debugging support. It is therefore highly recommended to store the log file of a deliverable image.

Identifying missing methods or classes

Errors like DNU (does not understand) can sometimes be traced to missing methods. A common reason for a method not being included is that it is under a development-only category.

Errors that are due to missing classes are more difficult to trace. If there is any doubt, a quick look at the log file reveals if all required classes are included.

Identifying superfluous methods and classes

The log file lists the classes in the target image, along with the space taken. When optimizing an image, it is useful to take a look at the included classes and methods. The code can often be rearranged to eliminate dead spots.

Image Corruption

There are two sources of image corruption:

- Pointer mismatches or memory overruns when calling APIs (such as buffer overruns in `sscanf`).
- Operating system failures (system lockup or power failure) that may result in lost clusters. In most cases, these failures affect the memory-mapped source file. Typically, loading a corrupted file displays a message saying that the source file contains an unknown object.

In both cases, you may face a situation where the backup files are corrupted as well. By following the simple guidelines outlined below, you may save yourself some frustration down the line:

- Back up your image (`STIMAGE.EXE` and `SOURCES.BIN`) on a regular basis.
- Save your image before testing new code that involves API calls.
- Do not save an image that is in a dubious state, or at the very least first make a backup.

Source File is corrupt

If the image does not start because the source file is corrupted, restore the previous version (using the */restore* option when launching `STMT.EXE`) and file-in the changes. The cause of this problem is usually a shutdown problem on NT (a power failure or reset).

Error in start-up code

If the image does not start because you modified start-up code such as a `winMain` method, create a file with code that corrects the problem and start `STIMAGE.EXE` with that file as argument. The code will be compiled by the method `winmain` in **DevelopmentEnvironment**.

Unrecoverable exception or process lock-up while saving image

Saving the image involves two phases; the first serializes a copy of the executable, while the second saves the source file. The Transcript window shows the current operation in its status window.

Image crashes while the executable is saved

There are two common causes:

- A pointer corruption. The safest is to discard the binary and start all over with a backed-up version. Pointer corruptions may occur when using address manipulation routines (forcing an address with `basicAddress`, writing at memory etc.).
- The data section has insufficient space. A possible cause is that you allocated an inordinate amount of memory that is still referenced by a global. If this was intended, you must go into the *Virtual Memory Settings* page of the **Image Properties** and increase the size of the data section.

Image crashes while the source file is saved

Your source file has been corrupted. Try to restore your image from a backup by filing in changes you made. If this does not help, open the **Image Properties** sheet, click on the *Optimization* tab, check *Recompile All Methods when compressing*, and compress your image.

If the image still crashes, you have a corruption in the source code itself, probably due to a pointer corruption. Try to isolate it as outlined below:

- Check the *recompile all methods* option in the *Optimizations* tab of the **Image Properties** and recompile the image.
- Run an *image comparison* (in the *Tools* menu of the Transcript window) with a source file known to be intact. You can then overwrite the corrupt source and save the image with the *compress* option.

CHAPTER 2 The Interface Builder

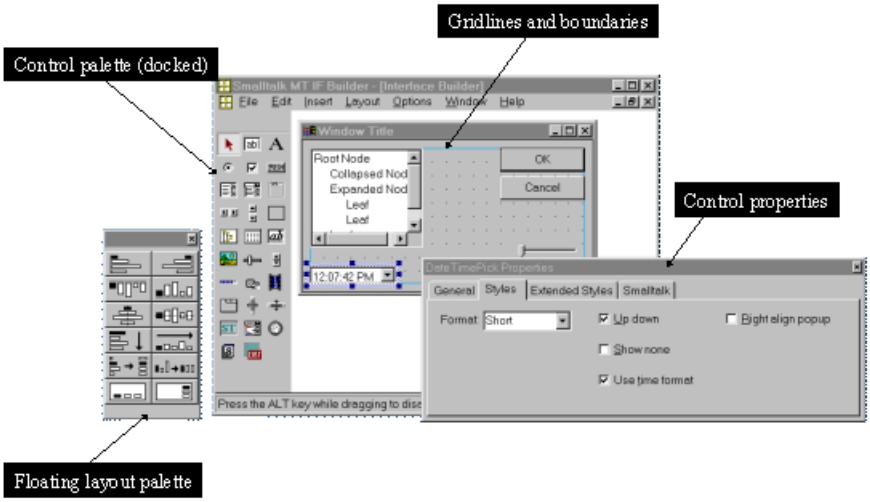
The Graphical User Interface Builder allows you to visually design the user interface elements of an application. It lets you wire events to Smalltalk methods and is fully integrated with ActiveX technology.

This chapter describes how to use the Interface Builder. It also discusses code generation and presents a builder-centric view on GUI development.

The tutorial section presents a generic Windows application, a sample that uses a toolbar, and a sample that demonstrates various controls used in a dialog box.

The last part contains technical notes on problems that you may encounter while using the Interface Builder.

Features Overview



Application Windows

In addition to regular child controls, it supports splitter windows and status bars.

Dialog Boxes

Dialog Boxes can be saved under any of the formats:

- Initialization method with dynamic in-memory construction.
- Resource template that can be included into an application's resource library.
- Dialog Boxes, as well as regular Application Windows, can host common controls, OLE components, and custom Smalltalk child windows. All dialog contents are fully compatible with the resource format used by Win32 resource compilers.

Automatic Reframing

All top-level windows (**FrameWindow** and **DialogBox** instances) support re-framing (scaling and translation) of child windows. The GUI Builder lets you easily assign framing parameters. The reframing data can be saved in resource format for dialog boxes.

ActiveX Components

Using ActiveX components is largely transparent for the developer. Events are normally routed through the stock event handling system. You can also edit the ambient OLE properties of the container.

The GUI Builder acts as a test container from which you can invoke methods and set properties. It is therefore easy to explore the capabilities of an ActiveX control, and it is possible to modify the properties of a control and save its state to persistent storage.

Extended Import Capabilities

The GUI Builder is able to import window layout, menu and event handler data from the following formats:

- A regular **FrameWindow** (which does not need to have been generated using the GUI Builder).
- A dialog resource.

Event Authoring

Connect event sources to event handlers. The GUI Builder automatically generates the event map initialization method.

No Overhead

Applications you generate using the GUI Builder do not carry any overhead and are as efficient as manually coded windows.

Extended Class Hierarchy

The window hierarchy includes subclasses of **FrameWindow** and **DialogBox** that implement lightweight ActiveX containers.

CustomControl lets you create an arbitrary Smalltalk-implemented window from a resource template.

Visual Editing

Functional Overview

The interface builder lets you design the visual components of an application. This encompasses:

- Top-level windows
- Dialog boxes
- Menus

In addition, you can edit the event handler map and connect handlers to events.

Application Windows

An application window is a **FrameWindow** other than a dialog box. An application window is normally the top-level window of an application and can have one or more child windows. The child windows are constructed at runtime in the method `initWindow`.

Note that many of the topics discussed here apply also to dialog boxes.

Creating an application window

To create an application window, click on *File|New*, select *Application Window*, and click on *Ok*. This creates an empty window. You can set properties of the window such as the class name, title and styles.

Editing an existing window

There are two ways to edit the user interface of a **FrameWindow** subclass:

- In the class hierarchy browser, left-click on the class and select the menu item *Edit GUI*.
- From the GUI builder, click on *File|Open* and select the class to open.

If the window has a menu, you can edit the menu by first opening the window in the Interface Builder or by clicking on *Edit Menu...* in the popup menu of the Class Hierarchy Browser.

Note	If the window is resizable, the GUI builder will resize it so that it fits within the window.

Using Menus

You can associate a menu with an application window. The menu will be saved together with the window, either as an inline template in a method or as a resource file.

To insert a menu, click on *Insert|Menu* or check the *Menu present* checkbox in the window properties of the frame. You can edit the menu by double clicking on it or clicking on *Edit|Menu*; which opens a menu editor where you can add, remove or modify menu items and sub-menus.

For more information, see also *Menus* on page 89.

Saving a window

Saving an application window generates methods in the window that reconstruct the user interface elements and define the event map. Optionally, if the window has a menu, the menu declaration may be saved as a separate resource file. The resource file can then be compiled using an external resource compiler.

Dialog Boxes

The main differences between a dialog box and a regular top-level window are:

- Dialog boxes always use a template that defines attributes, the layout and positions of child windows.
- Dialog boxes use the Win32 dialog procedure that provides dialog-specific behavior (handling of navigation keys etc.). For this reason, the Smalltalk implementation of a dialog differs from a regular top-level window.

Creating a dialog box

Click on *File|New* and choose *Dialog Box* to create an empty dialog. In the properties sheet, enter the name you wish to give to the dialog box subclass, and let it inherit from **OleDialogBox** if you intend to use OLE (ActiveX) features, otherwise choose **DialogBox** for a regular dialog box, or **PropertyPage** if you intend to create a property page.

The *Styles* tab in the property sheet lets you define a custom font for the dialog.

Editing an existing dialog

If the dialog responds to `openOn:`, you can edit it by right-clicking on the class in the Class Hierarchy Browser, and selecting *Edit GUI*. Alternatively, you can open the resource by specifying the name of the resource DLL and the resource identifier.

Example:

To edit the symbol (pool) editor:

1. Click on *File|Open*
2. In the class selection dialog, select **StSymbolEditor** and click on *Ok*
3. In the dialog that pops up, enter the `STDEV.DLL` for the DLL name, and `IDD_SYMBOLEDITOR` as resource identifier.

Saving a dialog

The default *File|Save* menu saves the dialog as an in-memory template. *File|Save As* gives more options, such as saving the resources as an external resource script.

Menus

Menus can be created separately or together with a frame window or dialog box. A Menu consists of a hierarchy of items. At the root of the hierarchy is the **menu bar**. A menu bar has **popup menus**, and each popup menu has **menu items**.

A menu item can carry out a command. Separators are used for visually separating two groups of menu items and have no other function. Popup menu items open a child popup menu when the user clicks on the item.

Menu styles

The figures below demonstrate menu styles. The **bar** and **column break** styles display a menu item and all following items on a new column, respectively with and without a vertical bar. The **radio check** item style displays a radio checkmark next to the item when it is in the checked state. Otherwise, an item with the **check state** displays a check mark. A menu can have one **default item**, which is displayed in bold.

Figure 2-1 Popup Menus with Bar Break

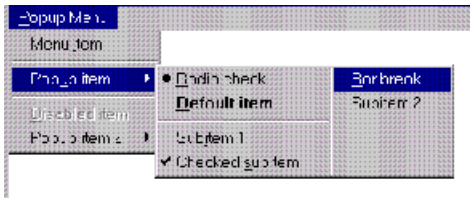
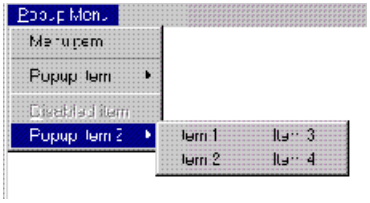


Figure 2-2 Popup Menus with Column Break



Working with Menus

Creating a menu

Click on *File|New* and choose *Menu* to open an empty menu bar. You can now add popup menus to the menu bar.

Inserting a Popup item

To add new popup items, click on *Insert|Popup*. The new popup is inserted before the currently focused item. You can then click on *Edit|Properties* or double-click the item in order to open a property sheet that lets you set attributes of the item.

Inserting a Menu item

Click on *Insert|Item* to insert a new menu item, or *Insert|Separator* to insert a separator line.

Moving items

You can drag items, separators and popup items with the mouse. An arrow indicates the target insertion point. You can also right-click to open a popup menu that lets you move the item via menu commands.

Saving a menu

You can save a menu in two formats:

- As an in-memory template in an application window method.
- As an external resource file.

Since in-memory templates are much easier to edit, it is convenient to save the menu as a method during development, and replace it with its resource counterpart when the application is ready to ship.

Edit operations on menus

You can use the *Edit* commands *Cut*, *Copy*, *Paste* and *Delete* on menu items.

Identifiers

Identifier naming conventions

The interface builder uses the following conventions when using identifiers:

- The `IDC_` prefix denotes controls.
- The `ID_` prefix denotes commands (e.g., menu ids).
- The `IDH_` prefix denotes help identifiers.

The interface builder uses these conventions to reverse-engineer an existing window. The builder may fail to recognize a symbolic name if the conventions are not observed, meaning that a numeric value is displayed instead of the identifier name.

Using Pool Dictionaries

In some cases, it is desirable to define your own identifiers. Each top-level window has an associated primary pool dictionary that receives new constants.

- Enter the name of the primary pool into the frame window properties. If the pool already exists, you can also select it in the combobox contents.
- Enter new constants into the identifier field of interface items (controls, menu items).

Working with Child Controls

Overview

The interface builder offers a variety of methods that let you arrange child windows, test mnemonics, define framing parameters and edit tab ordering. In addition, grid lines and automatic alignment lets you easily place controls onto a dialog box.

Selecting Controls

To apply an operation to one or more controls, select the controls while holding down the shift key and finish with the target. The target control defines the control placement or attributes onto which the other selections are aligned. A selected target control displays plain handles.

How to select a control


Click on the control.


The TAB key cycles between the controls placed on the parent window.

You can also use the control selection dialog for selections. This is particularly useful for controls that are outside the client area of the parent, or hidden by another control.

To select all controls at once, click on *Edit*|*Select All*.

Inserting Controls

In the control palette, click on the control symbol to insert. A tool tip appears while the mouse stays over a symbol. In the frame window being edited, click on the position where the control should be placed.

The options *Options*|*Show grid* and *Options*|*Use grid lines* respectively display a grid and align controls onto a grid boundary. You can modify the grid settings by clicking on *Options*|*Guide Settings* or temporarily disable it by pressing the Alt key.

Dropping a control near a margin automatically aligns it on the margin. The control remains pegged to the margin when the edited frame window is resized.

Duplicating and deleting Controls

Click on *Edit*|*Copy* to copy a selected control, on *Edit*|*Delete* or *Edit*|*Cut* to delete it.

Control Placement

Most alignment options work on a collection of selected controls.

▸ Moving controls

Select one or more controls to move.

Hold down the left mouse button until the cursor changes to a move cursor, then move the mouse and release the button when the control is at the desired position.

Press the position keys ← → ↑ ↓ to move the selected control.

You can move several selected controls at once by dragging any one of the controls. If the grid alignment option is on, the controls will be individually aligned on grid boundaries or margins.

To move a set of already selected controls, hold down the SHIFT key while clicking on any of the controls, then drag the set of controls with the mouse.

Resizing a Control

Select the control to resize and place the mouse cursor over one of the sizing handles. The cursor will change to an arrow. Resize the control by holding down the left mouse button and dragging the mouse.

Select the control to resize. Press the position keys ← → ↑ ↓ while holding down the SHIFT key to resize the selected control

Resizing a combo box

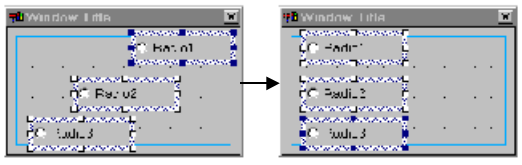
A combo box combines an edit control and a list box. A simple combo box can be resized like a regular control. Click on the down arrow icon to display or hide the frame that includes the list part of a drop-down combo box.

A drop-down combo box displays the list box only on demand, when the user clicks on the arrow icon next to the edit field. The height of a drop-down combo box is the combined height of the edit control and the list box. You cannot set the height of the edit control separately because the item height depends on the font used by the combo box at runtime. If you need to specify a height at runtime, for example because the combo box is used in a toolbar, you must set the font of the control to an appropriate value.

Aligning Controls

The alignment options *Left*, *Right*, *Top*, *Bottom* align secondary selections on the target selection.

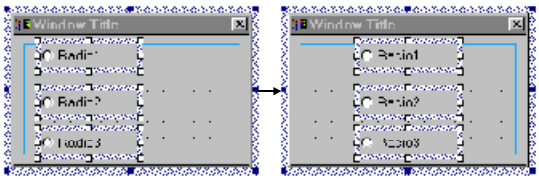
Figure 2-3 Left Alignment



Centering Controls

To center one or more controls, select the target window (a control or the parent window), select the controls to center, and click on *Vert Center* to center vertically or *Horz Center* to center horizontally. To center in the parent, select the parent window.

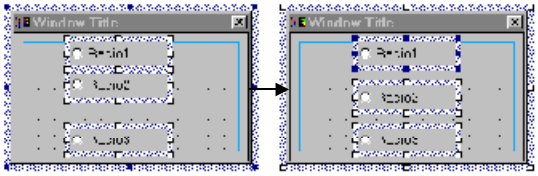
Figure 2-4 Vertical Centering



Even Spacing

This option distributes the selected controls evenly in one direction, either vertically or horizontally.

Figure 2-5 Even Spacing

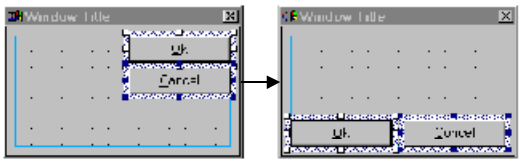


Push Button Alignment

Push buttons can be automatically centered at the bottom of the frame or aligned on the right top. The alignment parameters can be adjusted in the *Settings* dialog under *Options*|*Guide Settings*.

Select the push buttons to align and click on *Layout*|*Push Buttons*|*Right* or *Layout*|*Push Buttons*|*Bottom*.

Figure 2-6 Push Button Alignment (Right)



Resizing Controls

Controls can be made of the same size (horizontal, vertical, or both) by selecting the controls, the last selection being the target control to adjust with.

Layout|Size to Content fits the size of a selected control for the caption text of the control. This feature is automatically selected when you edit the contents of a static text field.

Control Properties

You display control properties by double-clicking on the control border or selecting *Edit|Properties*.

The property dialog displays several pages with control properties. When you modify the control's properties, most changes such as window styles are reflected immediately. Some properties, like listbox data entries, are only applied when you close the property sheet. In this case, closing the window or hitting the Enter key validates the changes, while pressing Esc closes the dialog without applying any unprocessed changes.

See also *Controls* on page 118.

Framing Parameters

The GUI builder lets you assign automatic reframing properties to individual child controls. Both regular windows and dialog boxes (including resource templates) support framing parameters for child controls. Beware that a child control must have a unique

identifier in order to be resizable; i.e., the default static identifier (`-1` or `IDC_STATIC`) will not work.

Resizing a control may also reset the framing parameters.

Tab Ordering

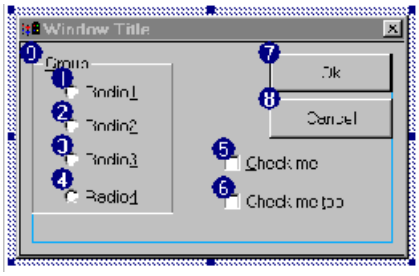
By default, Windows cycles the controls in a dialog in the order in which they have been created. The *Tab Ordering* option displays the current order of the controls. To modify the order, click successively on each control in the order in which you want them to appear on the dialog.

The `WS_GROUP` and `WS_TABSTOP` styles modify the default behavior. `WS_GROUP` marks the beginning of a group; all consecutive controls that do not have this style are part of the same group. Pressing a direction key navigates between the controls in the group. The user must use the Tab key in order to exit the group.

When the user presses the Tab key, the focus moves from one tab stop to another. A tab stop is a control that has the `WS_TABSTOP` style bit set. You must set the Tab Stop style on each control you want to be accessible through the tab key.

In summary, you define the order of the controls using the Tab Order editor and place tab stop and group styles accordingly. The figure below shows a typical layout. The first radio button and the upper check button both have the `WS_GROUP` and `WS_TABSTOP` styles.

Figure 2-7 Tab Ordering Example



Mnemonics

Mnemonics provide a quick way of moving to a given control by using the keyboard. A mnemonic is an underlined letter in the control text or in the static text field immediately preceding the control. When the user presses the corresponding key on the keyboard, the input focus moves to the control.

You can use *Layout*|*Test mnemonics* to test a window for duplicate mnemonics. If several controls have the same mnemonic, a dialog pops up that lets you select the controls with duplicate mnemonics.

When Windows encounters a duplicate mnemonic, the focus cycles between the controls that have the same mnemonic. It is therefore acceptable to allow duplicate mnemonics in a dialog.

Using Resource Scripts

The interface builder can generate resource files and the appropriate directory structure. You can choose the interface elements you wish to have in the application and the interface builder automatically creates the corresponding resource items.

Once you have created the resource folder, you can generate the resource DLL with the *nmake* utility (it comes with C compilers such as MSVC; change to a console window and type *nmake*). The resource DLL is fully functional and provides a good starting point for customizations.

Why use Resources?

It is not necessary to use resources in the early phases of development while the application is being tested in the development image. Resource generation is relatively cumbersome because it involves an external resource compiler and linker.

Once the basic application skeleton has been set up, you can use the Interface Builder once to generate the resource structure, including application icons and bitmaps. You can then compile the resources once and copy the resulting DLL to your path. This allows the application to load bitmaps and icons from the resource DLL.

It is only in the final stage that in-memory templates for menus and dialogs should be exported as resource scripts. Although there is no obligation to do so, it is good practice to separate the resources from the actual code.

Standard Resources

The interface builder can generate the following resources:

- An application icon
- A toolbar bitmap
- An About box

- A message file for error and informational messages
- Prompter templates
- A Tooltip string table for common commands (File, Edit, Help)
- An accelerator table
- A version resource

Application icon

The builder creates a default icon for your project. The identifier of the icon is `IDI_APP`, which is the default application icon used by the framework. There is nothing else to do in order to display an application icon.

Toolbar bitmap

The edit toolbar bitmap contains symbols for common edit operations. The identifier of the bitmap is `IDB_EDITTOOLBAR`.

About box

The builder generates an about box for your application. If you wish to define a default template for your applications (using, for example, the name and address of your company), you can edit the template file `ABOUT BOX.RC`. This file is located in the `SOURCE\TEMPLATE` subdirectory of your Smalltalk root folder.

Please refer to the online documentation for more information.

Message file

The message file contains error messages used by the Smalltalk MT runtime. The message sources are in the SUPPORT\MESSAGES folder.

Prompter templates

If your application uses prompters (**Prompter**, **ListPrompter**, **CheckListPrompter**, **MultiListPrompter**), you must include the corresponding dialog templates. Otherwise,

a runtime error occurs when the prompter object tries to load its resource from the executable image.

Note Since the prompters always load their template from the executable's resource section, the templates are only required in the final executable. During development, the prompters load the templates from the development image.

Tooltip string table

The string table defines strings for status text and tool tips. The framework automatically displays the text.

See also page 289 for more information on how to use tool tips.

Accelerator table

The accelerator table adds entries for demonstration purposes. You can edit the source and enter your own accelerators.

Version resource

A version resource contains version information about an executable image (`EXE` or `DLL`). Many system management tools use version resources to read the version of a file, and it is therefore highly recommended to include such a version resource.

How to generate Resources

To generate a resource script, proceed as follows:

1. In the Interface Builder, click on *File|New*, select *Resource Script* and validate.
2. Enter the location and name of the directory where you wish to store the resources. The name of the folder also becomes the name of the DLL.
3. Select the resources you wish to include from the list.
4. Click on *Ok* and run *nmake* to compile the DLL.

Note Most compilers require a DOS-compatible name for the resource file. In this case, the project name must be in DOS format.

How to customize Resource Templates

The `SOURCE\TEMPLATES` folder contains templates from which project-specific resources are derived. You can customize the templates (for example, changing the company information or localizing strings).

How to export Resources

You can export dialogs and menus authored with the Interface Builder as resource scripts (RC files). Each item is written to a separate resource script that you can include by adding the following line to the main script:

```
RCINCLUDE resource_script.rc
```

where *resource_script* is the name of the script of an interface item (dialog or menu).

Once the resource has been exported, you can remove the Smalltalk methods that create the in-memory templates:

- `initMenu` for the application menu
- `initTemplate` for a dialog template

Example

This example gives step-by-step instructions on how to build a simple application that displays a resizeable dialog box and a menu.

To generate a Resource Script

1. Click on *File|New* and *Resource Script*.
2. In the folder prompter, enter the name of a target directory (e.g., **TestWindow**). The name of the folder should be the same as the final executable name.
3. Select items from the list prompter:
   - About box
   - Application icon
   - Message file
   - Tooltip string table
   - Version resource
4. Click on *Ok* to generate the directory structure and resource files.

To create a dialog

1. Click on *File|New* and *Dialog Box*.
2. Edit the dialog box **ResourceSampleBox** by adding a couple of controls.
3. Define some framing parameters to demonstrate framing resources. In the frame properties, make sure the size box is checked to give the dialog a sizing border.
4. In the frame properties, enter `TESTWINDOW.DLL` as resource library.
5. Click on *File|Save As* and choose *Dialog and Resource template*.
6. In the *Save File* dialog box, save the resource as `DIALOG.RC` in the Res subdirectory of the `TESTWINDOW` folder.

To create a window and its menu

To create the window

1. Click on *File|New* and *Application window*.
2. Edit the frame properties
   - Set the name of the class to TestWindow.
   - Enter `TESTWINDOW.DLL` as resource DLL.

- Click on the *Window* tab, then check *Window menu* to give the window an empty menu.

To edit the menu

1. Double-click the menu or click on *Edit|Menu* to open the menu editor.
2. Add the standard *File* menu (under *Insert|Standard Popup|File*).
3. Enter a popup named *Test* and add an item *Dialog test...* with the identifier `ID_TESTDIALOG`. When prompted about a pool dictionary, enter `TestConstants`. This will be the primary pool dictionary of our main window.

Note You can also directly edit the menu by clicking on *File|New* and *Menu*. The menu can be saved as a resource template or as an in-memory window menu; in this case you must also provide the name of a **FrameWindow** subclass that contains the menu.

To save the window and menu

1. Close the menu window.
2. Click on *File|Save As* and choose *Frame window and menu template*. This option saves the frame window in the Smalltalk image and the menu as an external resource script.
3. In the File Save dialog, save the menu as `MENU.RC` in the Res subdirectory.

Note In all cases, the menu is also saved as an `initMenu` method in the frame window.

To compile the resources

Include the generated scripts into the main script

1. Load `TESTWINDOW.RC` into a text editor.
2. Add the following lines:

```
RCINCLUDE dialog.rc
RCINCLUDE menu.rc
```

Include pool constants

In our example, we use a custom pool dictionary that must also be used when compiling the resource script.

1. Open the **Pool Editor** on the pool **TestConstants**.
2. Click on *File|Save As* and save the pool as an include (.H) file under the name `TESTCONSTANTS.H`.

Compile the resource DLL

Run *nmake* and copy the resulting DLL into the path of the Smalltalk development environment.

Adding events

To finalize the sample, we need to add an event handler that opens the dialog box when the user clicks on *Dialog test...* .

1. Edit the main window **TestWindow** by right-clicking on the class in the Class Hierarchy Browser, then clicking on *Edit GUI* in the popup menu.
2. Bring up the frame properties and click on the *Events* tab.
3. Click on <*Menu*> to list the events sent by the menu.
4. Link the event `ID_TESTDIALOG` to the handler `onTestdialog`.

Implement the handler `onTestdialog` as below:

```
onTestdialog
    ResourceSampleBox new openOn: self.
    ^NULL
```

You can now test the window and its dialog.

Creating the project

1. Open the Project Browser and add the classes and pool dictionary created so far.
2. Add **TestWindow** and **ResourceSampleBox**. Make sure that the definition box is checked to include the class structure declaration.
3. Add the pool dictionary **TestConstants**.

4. Save the project under the name `TESTWINDOW.SP` in the `TESTWINDOW` folder.

Generating the executable

In order to build an executable, you just have to create a startup method `WINMAIN.SM`. The method `winMain:with:with:with:` must register window classes, create the main window and start the message loop.

The winmain method for our sample looks like:

```
!ApplicationProcess * methods!
winMain: hModule with: hPrevInstance with: cmdLineArgs with: nCmdShow
    "
    Public - Calls initialization function.
    "
    " Register Windows "
    Window registerClass.

    " Create application and run message loop "
    WinApplication new run: [TestWindow new open]
! !
```

This is also a good time to remove the method `initMenu` in **TestWindow** and add the following method:

```
windowMenu
    ^IDM_APP
```

The method `windowMenu` is called by the default implementation of `initMenu` in **FrameWindow**. It answers the identifier of a menu resource in the receiver's module.

Generated resource scripts

In this paragraph, we will examine the resource scripts that have been generated, and how you can customize the scripts.

Menu script

```
// ST/MT generated resource script
#include "windows.h"
#include "commctrl.h"
#include "rcst.h"
#include "TestConstants.h"

//////////////////////////////////////////////////////////////////////////////
//
// Menu
//
IDM_APP MENUEX DISCARDABLE
BEGIN
    POPUP"&File"
    BEGIN
        MENUITEM "E&xit",                           ID_APP_EXIT
    END
    POPUP"&Test"
    BEGIN
        MENUITEM "Dialog test...",                  ID_TESTDIALOG
        MENUITEM "",                                -1, MFT_SEPARATOR,
MFS_DISABLED
        MENUITEM "&About",                          ID_APP_ABOUT
    END
END
```

The first part defines the include files used by the script. In particular, it references the include file `TESTCONSTANTS.H`, which must define the identifiers used in the menu.

The menu is saved with a default identifier `IDM_APP`. If there is more than one menu, you assign a different identifier to each menu.

Dialog script

```
// ST/MT generated resource script
#include "windows.h"
#include "commctrl.h"
#include "rcst.h"

//////////////////////////////////////////////////////////////////////////////
//
// Dialog
//
IDD_DIALOG1 DIALOGEX 0, 0, 135, 75
STYLE WS_POPUP|WS_CAPTION|WS_SYSMENU|WS_THICKFRAME|DS_MODALFRAME|DS_SETFONT
CAPTION "Resource Sample"
FONT 8, "MS Shell Dlg"
BEGIN
    LISTBOX         IDC_LIST1, 5,5,75,42,
WS_VSCROLL|WS_TABSTOP|LBS_NOINTEGRALHEIGHT|LBS_SORT
    AUTORADIOBUTTON "Radio1",IDC_RADIO1, 89,13,39,11, NOT (WS_TABSTOP)
    AUTORADIOBUTTON "Radio2",IDC_RADIO2, 89,31,40,10, NOT (WS_TABSTOP)
    GROUPBOX        "Static",IDC_STATIC1, 83,0,48,48,
    DEFPUSHBUTTON   "Ok",IDOK, 12,56,50,14,
    PUSHBUTTON      "Cancel",IDCANCEL, 72,56,50,14,

END
//////////////////////////////////////////////////////////////////////////////
//
// Reframing Data
//

IDD_DIALOG1 RCDATA
BEGIN
    IDC_LIST1, 0,0,1000,1000,5,5,-55,-27,
    IDOK, 0,1000,0,1000,12,-19,50,14,
    IDCANCEL, 0,1000,0,1000,72,-19,50,14,
    IDC_RADIO1, 1000,0,1000,0,-46,13,39,11,
    IDC_RADIO2, 1000,0,1000,0,-46,31,40,10,
    IDC_STATIC1, 1000,0,1000,0,-52,0,48,48,
    0
END
```

As before, the first part defines the include files used by the script. After the header comes the template for the dialog. The identifier of the dialog resource is the same as the one declared in the frame properties, in the Interface Builder.

The framing parameters are stored in a raw data resource (RCDATA). Each line consists of a control identifier, followed by the scaling factors and an offset matrix in dialog units:

```
control_id, s1, s2, s3, s4, offset_x, offset_y, cx, cy
```

The offset of a scaled corner (where s1 == s2 or s3 == s4) is defined in terms of the control's extent, which makes it easier to read the parameters.

For example,

```
IDCANCEL, 0,1000,0,1000,72,-19,50,14,
```

specifies that the *Cancel* button is pegged to the bottom corner of the dialog. The reference point is (0, y), where y is the bottom of the dialog's client area (= 1000 * dialog height // 1000), to which we add (72, -19) to obtain the upper left corner. The two following words specify the control extent as (50, 14).

In the creation method, **DialogBox** attempts to locate and load an RCDATA resource that has the same identifier as the dialog box.

Window Reference

Code Generation

The GUI builder generates the code required to bring up the frame window or dialog with the properties, attributes and events authored in the GUI builder. The following paragraphs examine some commonly generated methods.

Class methods

initialize

✕	Runtime
✓	Generated if absent
✓	Overwritten

Initializes the class event handlers and is overwritten each time the frame window is saved.

ifTemplateName

✕	Runtime
✓	Generated if absent
✓	Overwritten

The GUI builder generates this method to return the symbolic resource name of the dialog. The method is only used by the interface builder to reload a dialog that has been saved to a resource file. If you remove it, the interface builder prompts for the resource identifier.

Instance methods

initWindow

✓	Runtime
✓	Generated if absent
✕	Overwritten

The `initWindow` method must call `initChildWindows` if such a method exists. The GUI builder does not overwrite an existing method.

initChildWindows

- ✔ Runtime
- ✔ Generated if absent
- ✔ Overwritten

This method attaches child controls to Smalltalk objects and performs other child initializations such as setting list data and creating headers and columns. The GUI builder overwrites this method.

initTemplate (dialogs only)

- ✔ Runtime
- ✔ Generated if absent
- ✔ Overwritten

This method creates the in-memory template for a dialog box. The GUI builder overwrites this method. It can be removed if the dialog has been saved to an external template and is subsequently loaded from a resource module.

openOn: (dialogs only)

- **?** Runtime
- ✔ Generated if absent
- ✔ Overwritten

This method initializes the template by calling `initTemplate` and creates a modal dialog box. The GUI builder does not overwrite an existing method.

initPosition

(frame windows only)

- ✔ Runtime
- ✔ Generated if absent
- ✔ Overwritten

This method returns the extent of the window. The GUI builder overwrites this method to return the window extent. Note that FrameWindow also accepts a rectangle that defines the window coordinates.

windowTitle

(frame windows only)

- ✔ Runtime
- ✔ Generated if absent
- ✔ Overwritten

This method returns the title of the window. The GUI builder overwrites this method.

windowStyle and **windowExStyle**

(frame windows only)

- ✔ Runtime
- ✔ Generated if absent
- ✔ Overwritten

These methods return window styles. The GUI builder overwrites the methods.

initMenu

(frame windows only)

- ✔ Runtime
- ✔ Generated if absent
- ✔ Overwritten

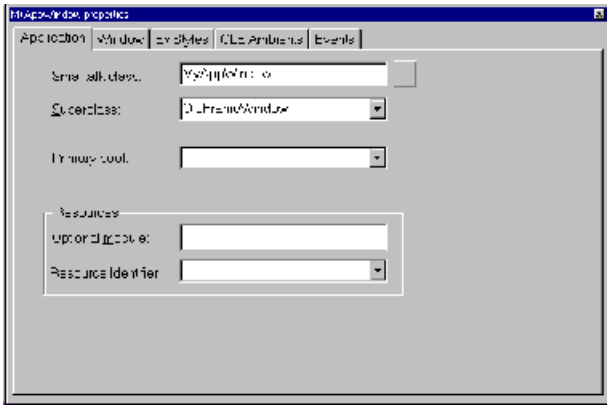
This method creates and initializes a menu. The GUI builder overwrites the method.

Frame Windows

You bring up the property sheet of a frame window by double clicking on the window's caption or by clicking on *Edit*|*Properties* while the window is selected. The sheet contains tabs for general window properties, window styles, OLE ambient properties and events.

Application properties

Figure 2-8 Application Properties



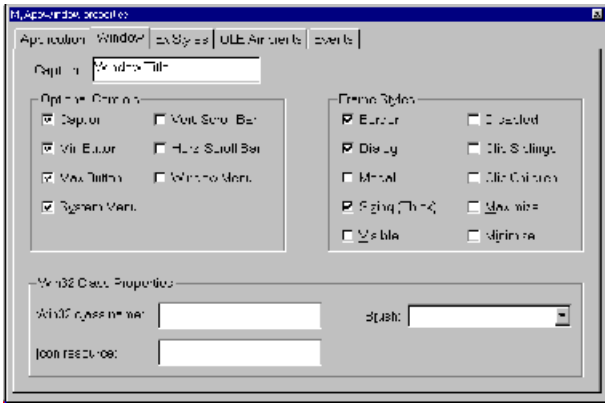
This page lets you specify the items in the table below.

Table 2-1 Application Properties

Item	Description
Smalltalk class	The Smalltalk class name of the window. If the class does not exist, it is created.
Superclass	The superclass of the window. The class must exist, and it is typically **OleFrameWindow** or **FrameWindow** for application windows, **OleDialogBox** or **DialogBox** for dialog boxes.
Primary Pool	The primary pool of the class. New constants are added to this pool.
Resources	Optional resource properties that specify the name of a resource DLL and a resource identifier. For dialog boxes, it is the name of the dialog box resource in that library. For regular top-level windows, it is the name of the menu resource.

Window properties

Figure 2-9 Frame Window Properties



The window page lets you edit visual properties of the window.

Table 2-2 Window Properties

Item	Description
Title	The caption (title) of the window. The Caption option must be checked as well.
Caption	Creates a title bar for the dialog box.
Minimize button	Creates a minimize box for the dialog box. The Caption option must be checked as well.
Maximize button	Creates a maximize box for the dialog box. The Caption option must be checked as well.
System menu	Creates a system menu for the dialog box. The Caption option must be checked as well.
Vertical scroll	Creates a vertical scroll bar.
Horizontal scroll	Creates a horizontal scroll bar.
Window menu	Whether a menu is present. You can also add and remove a menu from the *Insert* menu.

Table 2-3 Window Frame Styles

Item	Description
Border	Creates a thin border.
Dialog	Creates a dialog-box border.
Modal	Creates a modal dialog-box border.
Sizing (thick)	Creates a thick border that can be used to resize the window.
Visible	Determines whether the window is initially visible.
Disabled	Determines whether the window is initially disabled.
Clip siblings	Clips child windows relative to each other; that is, when a particular child window is repainted, this style clips all other top-level child windows out of the region of the child window to be updated. Clip siblings is for use with child windows only.
Clip children	Excludes the area occupied by child windows when drawing within the parent window. Do not use this style if the parent contains a group box.
Maximize	The window is initially maximized.
Minimize	The window is initially minimized (iconic).

Table 2-4 Win32 Class Properties

Item	Description
Class name	Name of a registered window class (a Windows operating system window class, not to be confused with a Smalltalk class).
Brush	Identifier of a brush, as used by the window class registration method.
Icon resource	Identifier of an icon, as used by the window class registration method.

Extended window styles

This page lets you set extended window styles.

Figure 2-10 Extended Frame Window Styles

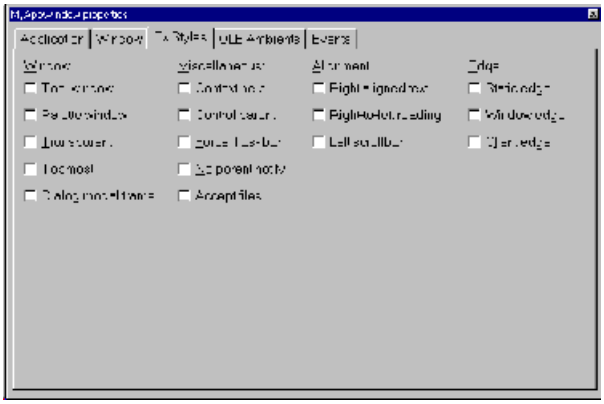


Table 2-5 Extended Frame Window Styles

Item	Description
Tool window	Creates a tool window. A tool window has a title bar that is shorter than a normal title bar, and the window title is drawn using a smaller font. It is often intended to be used as a floating toolbar. A tool window does not appear in the taskbar or in the dialog that appears when the user presses ALT+TAB.
Client edge	Creates a border with a sunken edge.
Static edge	Creates a static border.
Transparent	Makes the window transparent. Windows beneath this window are not obscured by the window.
Accept files	A window with this style accepts drag-drop files via the `WM_DROPFILES` message.
Control parent	Allows the user to navigate among the child windows of a dialog by using the TAB key.
Context help	Includes a question mark in the title bar of the window. When the user clicks the question mark, the cursor changes to a question mark with a pointer. If the user then clicks a child window, the child receives a WM_HELP message.
No parent notify	The child window does not send the `WM_PARENTNOTIFY` message to its parent window. The framework generates events for the following notifications: `WM_CREATE, WM_DESTROY, WM_LBUTTONDOWN,`

	WM_MBUTTONDOWN, WM_RBUTTONDOWN
Right-to-left reading order	The dialog box text is displayed in right-to-left reading order for languages such as Hebrew or Arabic.
Right aligned text	Specifies that text is right-aligned within a dialog box.
Left scrollbar	If present, a vertical scroll bar is to the left of the client area.
Force taskbar	Forces a top-level window onto the taskbar when the window is minimized.
Topmost	A window created with this style should be placed above all non-topmost windows and should stay above them, even when the window is deactivated.
Window edge	Creates a border with a raised edge.
Client edge	Creates a border with a sunken edge.
Static edge	Creates a three-dimensional border style intended to be used for items that do not accept user input.

Extended control styles are also discussed in *Extended* Styles on page 120.

OLE ambient values

This page lets you set ambient OLE properties. It is only available for OLE enabled windows other than dialogs.

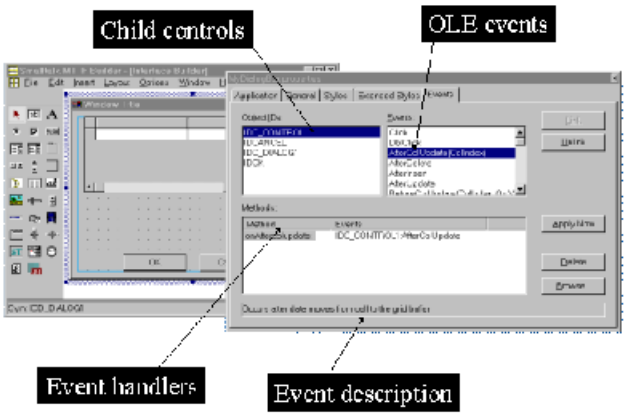
Events

The event page lists interface objects, events generated by those objects, and methods that handle those events. It lets you also define event handlers.

The left-hand list displays event sources of the current frame window. Clicking on a child control identifier or menu displays the available events in the right pane. The bottom pane displays the static event handlers that are currently defined.

See also *Events* on page 207 for more information.

Figure 2-11 Event Properties



Dialog Boxes

Dialog Box properties are similar to FrameWindow properties. The following paragraphs list only the additional fields.

Application properties

Table 2-6 Application Properties

Item	Description
Resource identifier	The name or identifier of the dialog resource in the module.

Dialog styles

The *Styles* page displays properties depicted below. For other properties, refer to frame window properties.

Table 2-7 Dialog Styles

Item	Description
Font	The font that will be used in all the controls in the dialog box. Click on the *Font* button to change the font.
Popup	Creates a pop-up window.
Child	Creates a child window.
System modal	Creates a system-modal dialog box, causing the dialog to have the `WS_EX_TOPMOST` style. The style has no effect on the dialog box or the behavior of other windows in the system when the dialog box is displayed.
No fail create	Creates the dialog box even if errors occur.
No idle messages	Suppresses `WM_ENTERIDLE` messages.
Set Foreground	Brings the dialog box to the foreground when it is created.
Center	Centers the dialog box in the desktop.
Center mouse	Centers the mouse cursor in the dialog box.
Control	Creates a dialog box that works well as a child window of another dialog box, much like a page in a property sheet. This style allows the user to tab among the control windows of a child dialog box, use its accelerator keys, and so on.

Controls

A Control is a child window - generally a system-defined window - which you place onto the client area of the frame window. The properties of a control can be viewed and edited by double-clicking on the control or selecting it and choosing *Edit|Properties*. All controls have the following properties:

- **General Properties** define the child identifier, the caption and general window styles such as the visibility, whether the control is disabled, the group and tab stop styles.
- A **Styles** page with control-specific styles.
- An **Extended Styles** page with extended (`WS_EX_XXX`) styles. These styles include 3D appearance styles (beware that Windows dialogs set some of those styles automatically).

- A **Smalltalk** page that lets you attach a control to a Smalltalk object. Dialogs do not normally create a Smalltalk object for each control; this page lets you override the default behavior and specify a Smalltalk control class that the dialog creates and attaches to the control handle.

The following paragraphs discuss some notable controls. Other controls display the general properties and behavior.

General Properties

Figure 2-12 General Control Styles

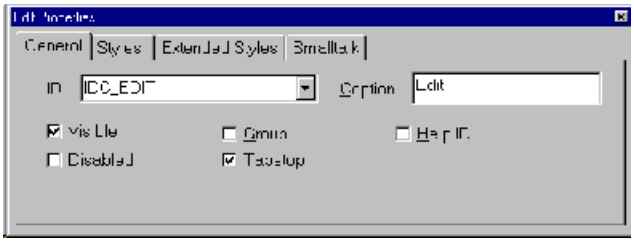


Table 2-8 General Control Properties

Item	Description
ID	The identifier of the control. This can be an existing constant from a pool dictionary defined in the class, or a new constant that will be added to the window's primary pool dictionary.
Caption	The label of the control (if applicable). To make one of the letters in the caption of a control the mnemonic key, precede it with an ampersand (&).
Visible	Determines whether the control is initially visible.
Disabled	Determines whether the control is initially disabled.
Group	A Group is a set of controls in which the user can move from one control to the next by using the arrow keys. All subsequent controls (as defined in the tab order) that do not have the Group style belong to the same group. The next control in the tab order with the Group style starts the next group.
Tabstop	The Tabstop style lets the user move to the control with the TAB key.

Help ID	Assigns a help ID to the control.

Extended Styles

Figure 2-13 Extended Control Styles

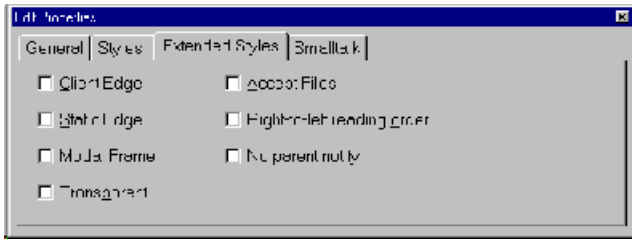


Table 2-9 Extended Control Styles

Item	Description
Client edge	Creates a border with a sunken edge.
Static edge	Creates a border with a static edge.
Modal frame	Creates a 3D frame.
Transparent	Makes the window transparent. Windows beneath this window are not obscured by the window.
Accept files	A window with this style accepts drag-drop files via the `WM_DROPFILES` message.
No parent notify	The child window does not send the `WM_PARENTNOTIFY` message to its parent window. The framework generates events for the following notifications: `WM_CREATE, WM_DESTROY, WM_LBUTTONDOWN, WM_MBUTTONDOWN, WM_RBUTTONDOWN`
Right-to-left reading order	The dialog box text is displayed in right-to-left reading order for languages such as Hebrew or Arabic.

Smalltalk Property Page

This page lets you attach a Smalltalk object to a control. You can also invoke custom property pages for a custom control.

Edit Controls

Edit controls encompass **Edit** and **RichEdit** controls. Note that the client edge style is ignored by dialogs unless you also specify the border style. Dialog boxes automatically set this style when the control has a border (this is Win32 behavior).

Custom Controls

A Custom control is a third party control that is not defined in the GUI builder. You can use a custom control directly on a dialog box by dropping the custom control symbol and specifying the Windows class name.

A custom control must be represented by a subclass of **Control** before you can use it in a regular window. The **Control** subclass must specify at least the Window class name. In general, it also provides methods to manipulate the control.

Buttons

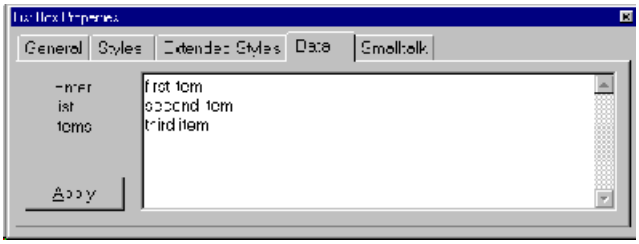
Buttons include radio buttons, check boxes, and push buttons.

Note	The standard *Ok* and *Cancel* buttons should have the IDOK and IDCANCEL identifiers.

List and Combo Boxes

List controls have an additional *Data* page that lets you specify a list of strings to display in the list or combo box.

Figure 2-14 List Contents

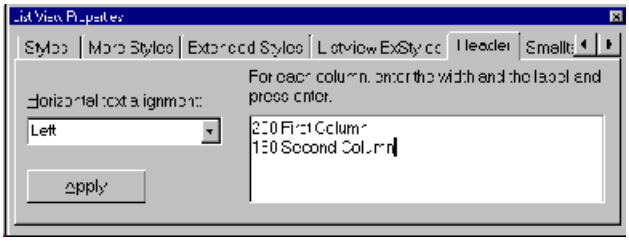


This is useful when the list contents are known at development time, so the application does not have to worry about initializing the list.

List View

A List view operates in one of three modes; **icon**, **small icon**, or **report**. In report view, the list view can feature a header. The *Header* page lets you specify the header column names and widths.

Figure 2-15 ListView Header



Picture

A picture can display an icon or a bitmap. The caption of the picture is also the resource identifier of the bitmap or icon in the resource module. You can specify a file that contains the resource to be included so that the GUI builder displays the picture at design time. Note that the picture control resizes itself automatically to the extent of the picture it displays.

Note If the dialog is not saved to a dialog resource, or if the picture is used in a regular frame window, the picture must be loaded explicitly at runtime.

Smalltalk Control

The Smalltalk control symbol inserts a Smalltalk-defined control window into the frame. The Smalltalk control is instantiated when the dialog opens, even if the dialog is based on a resource.

OLE Control (ActiveX)

OLE control encapsulates an ActiveX component. In addition to the basic window properties, you can edit the OLE properties that are specific to the component.

To insert an OLE control, click on *Insert|Insert OLE control* and select a control. You can also add an OLE control to the control palette (right-click on the palette).

Splitter Bars

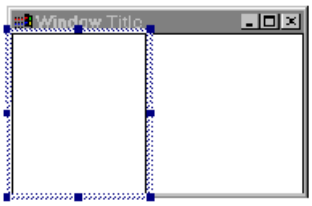
Splitter bars separate two sets of panes, either horizontally, vertically, or both, and let the user adjust their respective position at runtime. Splitters modify the framing matrices of the adjacent panes at runtime and support two modes; a fixed mode and a proportional mode. In the proportional mode, the splitter maintains a proportional position with respect to the parent. For example, a splitter positioned at 40% of the parent width moves accordingly when the parent is resized.

Inserting Splitter Bars

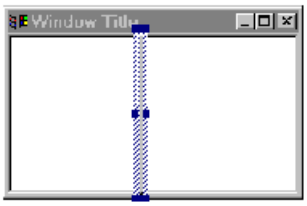
You must first create the panes to split and, if applicable, set the framing parameters they should use. Because splitter bars modify the framing parameters of adjacent panes, it is difficult to edit the framing parameters after a splitter has been inserted.

To insert a splitter, click on one of the splitter symbols and drop it between the panes you wish to split. You can insert a vertical or a horizontal splitter. When dropped, the splitter looks for adjacent panes and separates them into two sets (left/right or top/bottom, depending on the split type).

Step 1: Create the panes to split.



Step 2: Insert a (vertical) splitter between the panes.



When a splitter is inserted, adjacent panes are marked with the `WS_EX_CLIENTEDGE` style to obtain a 3D aspect.

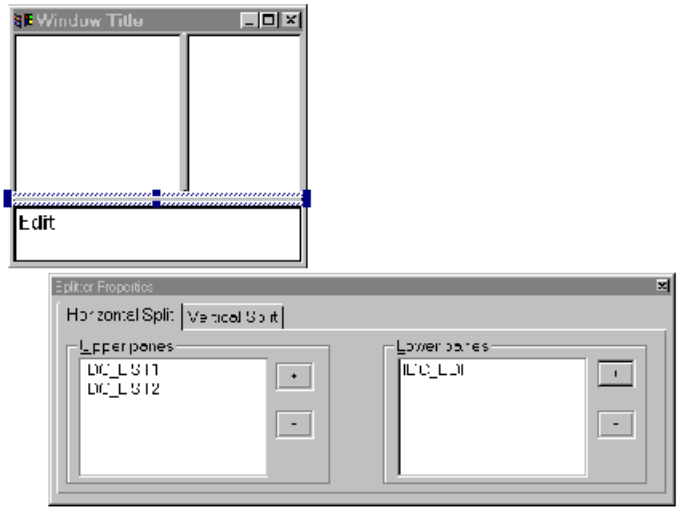
Editing Splitter Bars

The properties of a splitter include the set of panes it splits. Double-click the splitter to bring up the properties or select the splitter from *Edit|Select Control*, then click on *Edit|Properties*. It is possible to edit the collection of panes by adding new ones or deleting existing panes.

If you have more than one splitter bar, you can click on any splitter bar to edit both the horizontal and vertical panes being split.

For example, adding an edit pane and a horizontal splitter to the previous pane results in the layout below.

Figure 2-16 Splitter Bars



The properties of a splitter list the horizontal and vertical sets of panes separated by the splitters. If you wish to modify the properties or change the layout of the panes, select the splitter and delete it, then insert a new one.

Note When adding a child control manually to the list of panes separated by a splitter, you may have to set the WS_EX_CLIENTEDGE style to obtain a 3D look.

Using Splitter Bars

Internally, splitter windows are implemented by an instance of **SplitPane**, which covers the client area of the frame window. The visible parts capture the mouse input and allow the user to resize the panes. It is therefore mandatory that the child windows cover the client area completely. If the frame is resizable, the child windows must scale accordingly.

There can only be one vertical and one horizontal splitter in a **FrameWindow**. To use more than one, insert container windows as needed and split the containers.

Setting Framing Parameters

If the parent window is resizable, the child panes must be associated with framing matrices that allow them to scale with the parent.

You must reassign framing parameters each time you move a splitter bar.

Example 1: fixed split

In the simple example using two list boxes; `IDC_LIST1` and `IDC_LIST2`, set the framing parameters as following:

- For `IDC_LIST1`, click on *Use y scaling* to set the bottom at 100%.
- For `IDC_LIST2`, click on both *Use x scaling* and *Use y scaling* to set the right bottom at 100% @ 100%.

Example 2: proportional split

Click on the splitter and hold down the mouse button to reset the framing properties. Click on *Use x scaling* for both panes to enable proportional positioning.

Control Extensions

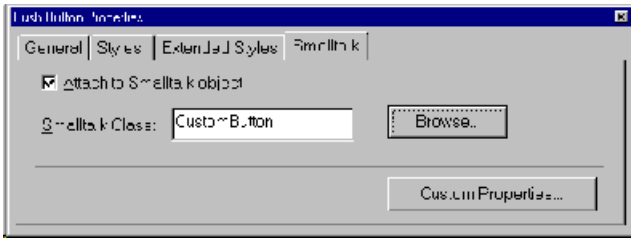
About Control Extensions

Control extensions provide a hook for the GUI builder to display custom properties of a control. The additional properties complement the window properties of the control.

Using Control Extensions

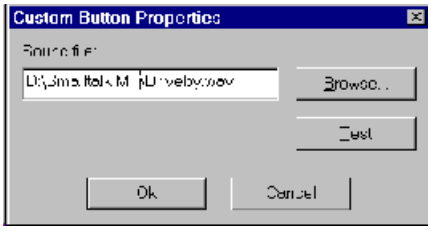
Clicking on the **Smalltalk** tab in the control's properties displays the Smalltalk properties. A control extension enables the *Custom Properties* button.

Figure 2-17 Control Extension Page



For example, the custom button sample displays the following dialog when the *Custom Properties* button is clicked:

Figure 2-18 Custom Button Sample



Implementing Control Extensions

To enable custom extensions, a control must implement the following class methods:

Table 2-10 Custom Control Extension Messages

Message	Description
ifOpenPropertiesOn:data:	Displays a custom dialog and returns an extension-specific value.
ifStoreDataOn:data:	Appends source code that initializes the control with custom data. The method takes a stream and the extension-specific value returned by `ifOpenPropertiesOn:data:`. Returns **true** if successful, **false** otherwise.

For example, the custom button sample returns the path to a sound file in the first method. The implementation of `ifStoreDataOn:data:` looks like:

```
stream nextPutAll: ' soundFile: ''' asStringA;
    nextPutAll: data asStringA;
    nextPut: $'.
^true
```

Defining events

The Interface Builder uses the class method `ifEvents` to retrieve the set of events that a control supports. A custom control can add its own events.

The return value is an array with the following information for each event:

- The event name. This must be a string. If the first character is a pound sign (#), the event is a symbol. Otherwise, it is a constant name defined in a pool dictionary of the class.
- An event description string. The description should be short enough to fit into the event description field of the Interface Builder.
- Event parameters. The format of the string is the same as the one used by C functions, for example ‘(param1, param2)’ defines two parameters. If there are no parameters, the value is zero.
- A reserved value, must be zero.

GUI Builder Tutorial

This tutorial shows you how to create a basic application that uses a main window, a menu and a dialog box. It details all steps, including generating external resources and building an executable.

A Generic Application

Creating the application window

1) Open the GUI builder, click on *File|New*, select *Application Window* and press *Ok*.
2) Double-click on the window or click on *Edit|Properties*. In the property sheet that pops up, enter **GenericWindow** in the Smalltalk class field, and select **FrameWindow** as superclass.
3) Press on the *Window* tab
    - modify the title to 'Generic Sample'.
    - check Window menu to insert a menu.
4) Close the property sheet.

Creating the Generic menu

1. Open the menu editor by clicking on *Edit|Menu*, or double-click on the menu bar of the edited window.
2. Insert a File menu (*Insert|Standard Popup|File*).
3. Insert a Help menu (*Insert|Standard Popup|Help*).
4. Close the menu editor

You can now test the window using *Layout|Test Mode*. At this point, only *File|Exit* and *Help|About* are functional. The latter displays the default about box, which is the about

dialog of the development environment. The other menu items are currently not needed, so we'll disable the most common and delete the rest. To delete an item in the menu editor, select the item, then press **DEL** or click on *Edit|Delete*. To disable it, click on *Edit|Properties*, move to the *State* tab, and click on *Disabled*.

Saving the Window

To save what we've done so far, click on *File|Save*. This is the same as clicking on *File|Save As*, then on **FrameWindow**.

Saving the window creates a new class **GenericWindow**, with methods that define attributes and create the menu. You can test it by evaluating:

```
GenericWindow new open
```

Creating the About dialog

For demonstration purposes, we will create an About box for our application. If you use a resource script, you can also generate the About box automatically, so this step is not mandatory.

1. Click on *File|New* and select *Dialog Box* to create a dialog box.
2. Click on *Edit|Properties* to open the dialog properties.
   - In the *Application* page, enter GenericAboutBox as Smalltalk class and select DialogBox as superclass.
   - In the *General* page, change the dialog title to 'About Generic'.
3. Close the property sheet.
4. Add a text label: click on the static text symbol in the palette and drop the control onto the dialog.
   - Click on *Edit|Properties* or double-click to edit the properties of the control. Enter the text to display and select *Center* for the text alignment option under the *Styles* tab. You can use special characters by pasting them from a utility such as the Character Map applet that comes with Windows. Just make sure that the font matches the dialog font (it is MS Sans Serif by default).
   - To center the control, select the control and the dialog and click on *Layout|Align Controls|Horz Center*.
5. Add the Ok button:
   - Select the pushbutton symbol on the tool palette

- Drop the button on the dialog, select *Layout*|*Push Buttons*|*Bottom* to align the push button at the lower bottom of the dialog.
- Edit the button's properties and enter IDOK as identifier, Ok for the text. Because IDOK is a predefined event, the dialog will automatically close when the user clicks on the Ok button.

6. Save the dialog (*File*|*Save*).

You can now open the about box in response to the about menu action. The event is already defined in **FrameWindow**, so the window must just re-implement the method `aboutBox` as below:

```
aboutBox
    "
    Callback that opens an about box.
    Return Value:
        This method does not return a value.
    "
    ^GenericAboutBox new openOn: self
```

Using external resources

To be fully functional, the generic application needs an icon. To that effect, we will create a resource-only DLL with an icon. We will later be able to add other resources to this file.

Add resource properties to the window

In the Class Hierarchy Browser, navigate to **GenericWindow** and right-click on the class. Select *Edit GUI* to open the GUI builder on the window.

1. Open the window properties.
2. In the *Application* page, enter `'generic.dll'` as the module name.
3. In the *Window* page, enter `Generic` as Win32 class name and `IDI_APP` as icon resource.
4. Save the window.

Entering a Win32 window class name requires that the window is registered at runtime using `registerClass`. Registering a window class allows you to use use attributes such as the icon and background brush.

Create a resource script

The easiest way to create a resource-only DLL script is to use the interface builder. Click on *File|New*, select *Resource Script*, select the options you wish to have in the DLL and validate. In this example, you would choose the following items:

- ✓ Application icon (the icon displayed by GenericWindow)
- ✓ Message file (runtime error messages for the target executable)
- ✓ Version resource (optional, the version resource is displayed in the file properties of the Windows explorer).

This creates a new directory and a subdirectory with the resources. Invoke *nmake* to compile the DLL. Copy the DLL to the current directory.

Before you can open windows of the new class, you must register the window class once by invoking:

```
GenericWindow registerClass.
```

You can then open a window using:

```
GenericWindow new open
```

The window should now use the specified class icon. You can edit the application icon file with a third-party icon editor.

Note The include environment variable must include the ...\SUPPORT\INCLUDE directory before you can compile the resources.

Saving the About dialog to a resource file

This step is optional. Typically, you will save dialogs as external resources when the design is finalized.

1. Open the previous About dialog in the interface builder.
2. Drop an icon control on the dialog and open the properties for the icon.
   - In the *General* tab, enter `IDI_APP` into the resource field.
   - If you wish to display the icon, click on *Browse* and open the icon file (in the `RES` subdirectory that has been created previously). This step is optional, since the resource identifier determines the icon to load at runtime.
3. In the Application page, edit the resource identifier and module name. Enter `GENERIC.DLL` as module and `IDD_ABOUT` as resource identifier.
4. Click on *File|Save As* and save the dialog as *Dialog and Resource template*, for example as `ABOUT.RC` in the `RES` subdirectory.

5. Close the interface builder.
6. Add the following line to the resource script (`GENERIC.RC`):

```
RCINCLUDE ABOUT.RC
```

7. Run `NMAKE` and copy the DLL to the Smalltalk directory.

Evaluate the expression below:

```
GenericWindow registerClass.
GenericWindow new open
```

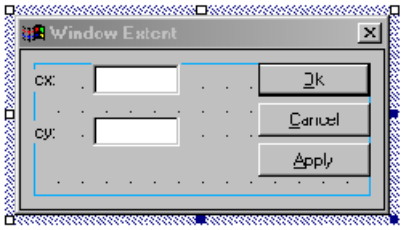
The about box should now display the application icon.

Creating a dialog

The dialog box in the following example displays a window extent and lets the user modify the extent and apply the changes.

Create a dialog as below:

Figure 2-19 Generic Dialog Box



GenericDialog adds an *Apply* button that lets the user view the changes immediately, without first existing the dialog. The processing takes place in the method `apply`, which is linked to the `IDC_BUTTON1` event in the event editor.

A dialog box reacts to the events `IDOK` (the user clicked on *Ok*) and `IDCANCEL` (the user clicked on *Cancel*, or hit the Esc key). The default action in **DialogBox** is to close the dialog, passing the command identifier to the method `closeWith:` (which, ultimately, returns the value to the caller of `openOn:`).

In this case, `onOk` calls `apply` before closing the dialog, so the implementation of `onOk` looks like:

```
onOk
    self apply.
    ^super onOk
```

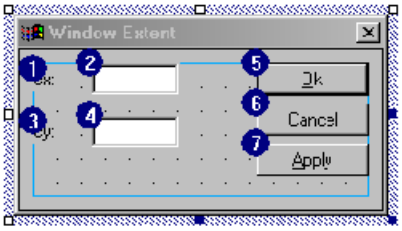
Dialog Layout Example

After placing the controls on the dialog's client area, proceed as follows:

- Align the edit fields (*Layout*|*Align Controls*|*Left*) and static text fields
- Center the static text vertically on the corresponding Edit field (*Layout*|*Align Controls*|*Vert Center* or press F9)
- Select the push buttons and click on *Layout*|*Push Button*|*Right*

Define the tab order: click on *Layout*|*Tab Order*, then click on each control in the order you want them to appear.

Figure 2-20 Generic Dialog Box Tab Order



Initializing the Dialog

The dialog holds its Windows owner in an instance variable. Modify the class declaration by adding the instance variable `m_window`. This variable is initialized in the `#openOn:` method:

```
openOn: ownerWindow
    "
    Opens the dialog box.
    Parameters:
        ownerWindow      A top-level window that becomes the owner, or NULL.
    Return Value:
        The method does not return until the dialog box is closed. The
        return value is the receiver.
    "
    m_window := ownerWindow.

    self initTemplate.
    ^self dialogBoxIndirect: nil owner: ownerWindow
```

The dialog needs to display the parent's window extent when it is first shown. This is done in the method `initWindow:`

```
initWindow
    | extent |
    m_window notNil ifTrue: [
        extent := m_window getWindowRect extent.
        self setItemInt: IDC_EDIT1 value: extent x.
        self setItemInt: IDC_EDIT2 value: extent y.
    ]
```

The message `setItemInt:value:` sets the value of an edit control to an integer. An alternative would be to use `setItemText:text:` as below:

```
    self setItemText: IDC_EDIT1 text: extent x printString.
```

A third possibility is to attach an edit object to the control and to send a message to the edit object:

```
    | edt |
    edt := self childAt: IDC_EDIT1 class: Edit.
    edt setWindowText: extent x printString.
```

The first parameter to `childAt:class:` is the control identifier, the second is the control class to instantiate. If a control object is already attached, that object is returned. Attaching a control lets you use the control's protocol.

In our simple case, using `setItemInt:value:` is the easiest and most efficient.

Retrieving Dialog Data

The implementation of the method `apply` looks like:

```
apply
    | x y |
    x := self getItemInt: IDC_EDIT1.
    y := self getItemInt: IDC_EDIT2.
    m_window setWindowExtent: x @ y.
    ^NULL
```

Implementing Validation

In this example, we will limit the extent the user can enter to the available desktop area. The apply method rejects invalid values and returns TRUE if the parent window has been resized, FALSE otherwise. Since an event handler returns an integer to step out of the event handler chain, the return values should be integers. If the method returned booleans, event lookup would continue (although in this particular example, this wouldn't make a difference).

The desktop window handle is returned by the function `GetDesktopWindow`, so the extent is obtained with:

```
hwndDesktop := Window fromHandle: WINAPI GetDesktopWindow.
maxExtent := hwndDesktop getWindowRect extent.
```

When the input data is invalid, the method sets the state of the input control to `STATE_FOCUS|STATE_HILITE`, which selects the faulty string, sets the focus to the control, and emits a warning beep. The `onOk` method must also be modified to close the dialog only when the validation succeeds. Since the `apply` method returns `FALSE` when the validation fails, the code for `onOk` looks like:

```
onOk
    self apply == TRUE ifTrue: [
        ^super onOk
    ].
    ^NULL
```

The complete implementation for `apply` is:

```
apply
    "
    Event handler for event 'BN_CLICKED' in IDC_BUTTON1.
    Return Value:
        An integer to return to the caller, nil to continue
        event handling.
    "
    | x y maxExtent hwndDesktop |
    hwndDesktop := Window fromHandle: WINAPI GetDesktopWindow.
    maxExtent := hwndDesktop getWindowRect extent.

    x := self getItemInt: IDC_EDIT1.
    y := self getItemInt: IDC_EDIT2.
    x >= maxExtent x ifTrue: [
        self setItemState: IDC_EDIT1
            state: STATE_FOCUS|STATE_HILITE
            stateMask: STATE_FOCUS|STATE_HILITE.
        ^FALSE
    ].
    y >= maxExtent y ifTrue: [
        self setItemState: IDC_EDIT2
            state: STATE_FOCUS|STATE_HILITE
            stateMask: STATE_FOCUS|STATE_HILITE.
        ^FALSE
    ].

    m_window setWindowExtent: x @ y.
    ^TRUE
```

Extending GenericWindow

We will now edit the generic menu to add an option that lets the user open the supplemental dialog.

1. Edit GenericWindow with the interface builder.
2. In the interface builder, bring up the menu editor by selecting *Edit|Menu* or by double clicking on the menu bar of the Generic window.
3. Insert a popup menu named *Options*.
4. Insert a menu item named *Extent*. Set its command identifier to `ID_OPTION_SETEXTENT`.

When you validate, the interface builder prompts you for a pool name. This is because **GenericWindow** has not yet a primary pool to which new identifiers are added. Enter `GenericConstants` into the prompter.

The next step is to edit the event map to handle the new event. Close the menu editor and open the frame window properties. Click on the *Events* tab. The left-hand pane lists the event sources.

- Click on <*Menu*> to display the events sent by the menu.
- Select `ID_OPTION_SETEXTENT` and click on *Link*.
- Enter the name of a handler method: `onSetextent`.
- Save the dialog (*File|Save*).

Implement `onSetextent` as follows (tip: in the Events page, double-click on the method to bring up the Class Hierarchy Browser):

```
onSetextent
    "
    Event handler for event 'ID_OPTION_SETEXTENT' in Menu.
    Return Value:
        An integer to return to the caller, nil to continue
        event handling.
    "
    GenericDialog new openOn: self.
    ^NULL
```

You can now test the generic window.

Creating an executable image

- Create a project with the classes **GenericAboutBox**, **GenericDialog**, **GenericWindow**.
- Add the pool dictionary `GenericConstants` to the project.
- Save the project in the `GENERIC` directory.

- Create a `WINMAIN.SM` startup file as below.
- Generate the executable (in the project browser, *Project*|*Build*|*Build EXE*)

```
!ApplicationProcess * methods!
winMain: hModule with: hPrevInstance with: cmdLineArgs with: nCmdShow
    "
    Public - Calls initialization function.
    "
    " Register Windows "
    Window registerClass.
    GenericWindow registerClass: hModule.

    " Create application and run message loop "
    WinApplication new run: [GenericWindow new open]
! !
```

Toolbar Sample

About Toolbar Sample

Toolbar Sample demonstrates how to build a simple application that uses an edit toolbar.

Creating the Application window

Follow the same steps as for the **Generic** window.

1. *File*|*New – Application Window* to create the window. This creates the window and opens the window properties.
   - Enter **SampleToolWindow** as Smalltalk class, **FrameWindow** as superclass.
   - Enter `TOOLSAMPLE.DLL` in the module field.
   - In the *Window* tab, edit the window title and click on *Menu present*. Enter the Win32 class name (`sampletoolwindow`) and specify `IDI_APP` as icon resource. The `IDI_APP` identifier is defined in the standard resource pool dictionary.
   - Close the property sheet.

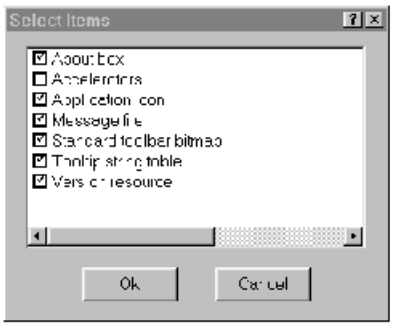
2. Insert a multiline edit control that scales with the parent. The identifier should be `IDC_EDIT` to enable automatic processing of edit events and commands.
3. Double-click on the empty menu bar (or click on *Edit|Menu*) to edit the menu.
   - Click on *Insert|Standard popup|File* to insert a default File menu. Repeat the procedure for Edit and Help.
   - Double-click on *File|Print* and disable the item in the State tab. Repeat this *for Print Preview* and *Print Setup* as well.
   - Close the Menu editor.
4. Save the window (*File|Save*).

Generating a resource script

Click on *File|New – Resource Script*. You are asked to select a destination folder. Enter TOOLSAMPLE (in a directory of your choice).

Select the items you wish to generate:

Figure 2-21 Resource Script Dialog



Go into the `TOOLSAMPLE\RES` directory and run `NMAKE` to compile the resources. Copy the resulting `TOOLSAMPLE.DLL` to the current Smalltalk directory.

To open the window with the new resources, evaluate:

```
SampleToolWindow registerClass.
SampleToolWindow new open
```

The *About* menu is now functional and the window has its own icon.

Initializing the Toolbar

The resource script generation utility inserts the following line:

```
IDB_EDITTOOLBAR BITMAP  DISCARDABLE     "edittb.BMP"
```

Add the following method to **SampleToolWindow**:

```
initToolBar
"
Private - Creates and initializes a toolbar.
"
| tb buttonArray |
buttonArray := #[
TOOLIMAGE_FILENEW ID_FILE_NEW TB_ENABLEDBUTTON 0 0
TOOLIMAGE_FILEOPEN ID_FILE_OPEN TB_ENABLEDBUTTON 0 0
TOOLIMAGE_FILESAVE ID_FILE_SAVE TB_ENABLEDBUTTON 0 0
TOOLIMAGE_FILEPRINT ID_FILE_PRINT TB_ENABLEDBUTTON 0 0
0 0 TB_SEPARATOR     -1   -1
TOOLIMAGE_EDITCUT ID_EDIT_CUT TB_ENABLEDBUTTON 0 0
TOOLIMAGE_EDITCOPY ID_EDIT_COPY TB_ENABLEDBUTTON 0 0
TOOLIMAGE_EDITPASTE ID_EDIT_PASTE TB_ENABLEDBUTTON 0 0].

(tb := ToolBar new) createToolBar: self
    hInstance: m_module
    style: WS_CHILD|WS_VISIBLE|WS_BORDER|TBSTYLE_TOOLTIPS
    id: ID_TOOLBAR
    bitmapID: IDB_EDITTOOLBAR
    bitmapCount: TOOLIMAGE_EDITCOUNT
    buttonStruct: buttonArray
    displayIndex: DPI_STD96.
```

The variable `buttonArray` is a **WordArray** that contains a `TBBUTTON` structure for each button.

Each line defines the following button attribute:

- The image list index of the button. The index is the zero-based index of the toolbar bitmap.
- The command identifier of the button.
- The type and state of the button; this can be one of the following:
- `TB_ENABLEDBUTTON, TB_ENABLEDCHECK, TB_ENABLEDCHECKEDCHECKGROUP, TB_ENABLEDCHECKGROUP, TB_ENABLEDGROUP, TB_SEPARATOR.`
- An application-defined 32 bit value, normally zero.
- The index of the button string or zero if none is defined.

Note

It is also possible to combine the type and state constants using inline expressions. For example,

```
##(TBSTATE_ENABLED | (TBSTYLE_BUTTON << 8))
```

is the same as

```
TB_ENABLEDBUTTON.
```

The edit buttons on the toolbar are functional because edit commands are maintained by the edit control.

Customizing the Toolbar

When replacing the toolbar, the following issues must be kept in mind:

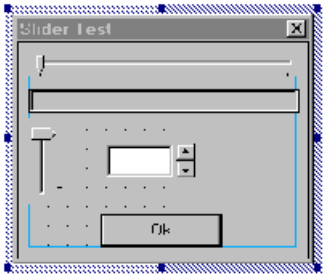
- The identifier of the toolbar must match the identifier in the resource file.
- The number of images in the image list must be updated.
- The button structure array must be updated to reflect the toolbar layout
- The display index must match the dimensions of the button bitmaps. Smalltalk MT defines three constants you can use: `DPI_STD72`, `DPI_STD96` and `DPI_STD120`. The corresponding button sizes are named `TOOLBUTTON_STD_XXWIDTH` and `TOOLBUTTON_STD_XXHEIGHT`, where XX is 72, 96 or 120. It is also possible to use a custom size; in this case the display index must specify the button extent (for example 24@22).

Slider, Progress and Spin Sample

About this sample

This sample combines an horizontal slider with a progress bar and a vertical slider with an edit field and a spin (**UpDown**) button. It demonstrates how to work with sliders, progress bars and spin buttons.

Figure 2-22 Slider, Progress and Spin Sample Dialog



Creating the Application window

The layout is straightforward if you observe the points below.

- To create the vertical slider, place a slider onto the canvas and select the vertical orientation in the slider properties.
- To associate a spin button with an edit field, select the alignment styles *Right of Buddy* and *Auto buddy*. To display integers in the edit field, check the *Set buddy integer* style. Note that the spin button window places itself onto the associated buddy when it is created.

You must attach controls that you wish to access directly. In this example, the slider controls and the progress bar need to be attached. The edit field and the spin button do not need to be attached because you can access the edit field using `getItemInt:` and `setItemInt:value:`.

Handling the events

The dialog implements the following functionality:

- When the user moves the top slider, the progress bar is updated to reflect the slider position.
- Moving the vertical slider updates the edit field, and changing the edit field value, either directly or indirectly via the spin button, updates the vertical slider.

The implementation handles the `WM_VSROLL` and `WM_HSCROLL` events to update respectively the edit field and the progress bar, and defines a handler for the `EN_CHANGED` event in the edit field. The latter updates the vertical slider each time the

edit contents change. This means that directly entering an integer value sets the slider position.

The handler code is as follows:

```
onHscrollSlider
    "
    Event handler for event 'WM_HSCROLL' in IDC_SLIDER1.
    Return Value:
        An integer to return to the caller, nil to continue
        event handling.
    "
    (self childAt: IDC_PROGRESS1) setPos: (self childAt: IDC_SLIDER1) getPos.
    ^NULL


onVscrollSlider
    "
    Event handler for event 'WM_VSCROLL' in IDC_SLIDER2.
    Return Value:
        An integer to return to the caller, nil to continue
        event handling.
    "
    self setItemInt: IDC_EDIT value: (self childAt: IDC_SLIDER2) getPos.
    ^NULL


onEditChanged
    "
    Event handler for event 'EN_CHANGED' in IDC_EDIT.
    Return Value:
        An integer to return to the caller, nil to continue
        event handling.
    "
    | slider |
    (slider := self childAt: IDC_SLIDER2) notNil ifTrue: [
        slider setPos: (self getItemInt: IDC_EDIT).
    ].

    ^NULL
```

Note that the code in `onEditChanged` first tests if the slider control has been attached. This is because the spin button generates an `EN_CHANGED` event when it is created, before the code in `initWindow` has a chance to attach the child controls.

Splitter Bar Sample

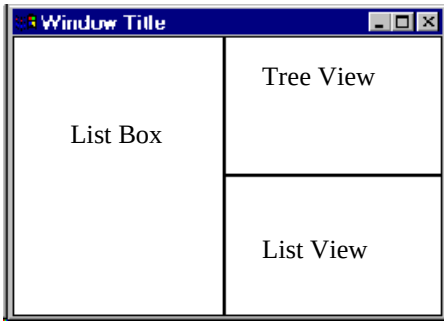
About Splitter Bars

We will see in two examples how to use splitter bars with a proportional and a relative split. A relative split separates two sets of panes at a fixed point, while a proportional split scales with the parent window, for example at 50% of the parent window width and / or height.

The Control Layout

The first step is to create the parent window and the control layout. It consists of three child panes:

Figure 2-23 Splitter Sample

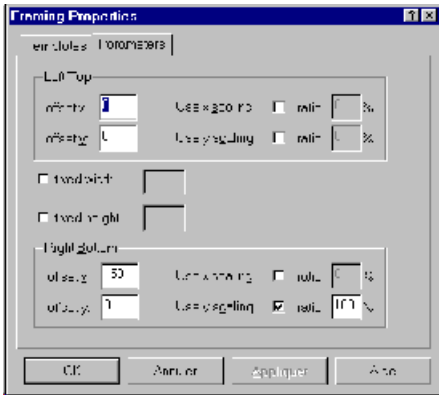


Framing Parameters

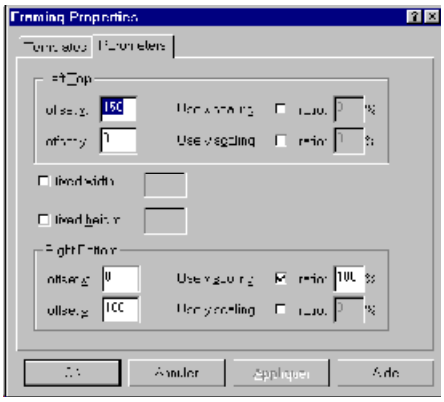
Enter the framing parameters as in the figures below.

Figure 2-24 Splitter Sample - Framing Parameters

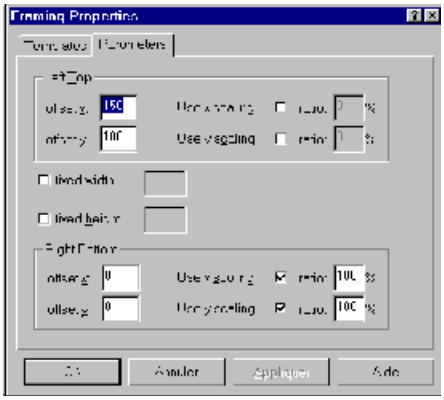
Left



Right top



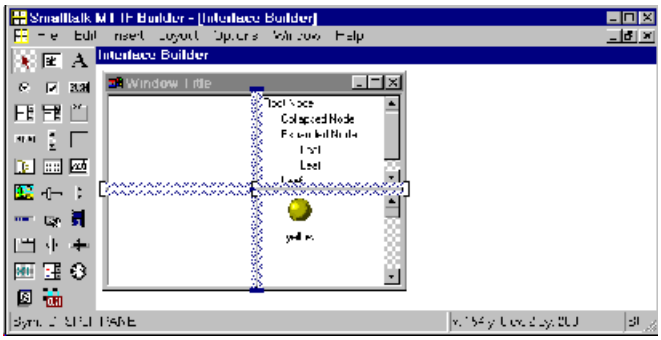
Right bottom



Insert Splitter Bars

Test the window and resize it to make sure it behaves as expected. You can now drop a vertical splitter at the right side of the listbox and a horizontal splitter between the rightmost panes.

Figure 2-25 Splitter Sample - Sample Layout



Test the window to verify that the splitter bars behave correctly.

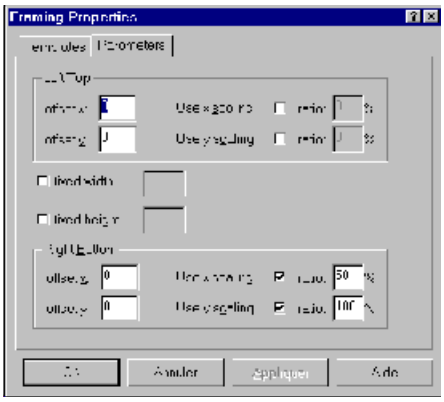
Using Scaling Parameters

Using scaling parameters is similar to the previous steps. Instead of specifying offsets in the framing dialogs, you must enter a scaling ratio such as 50% and set the offsets (previously 150) to zero.

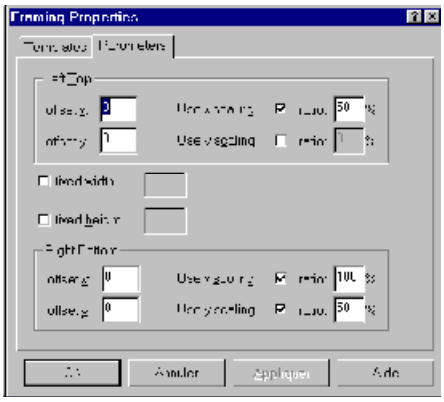
You can then drop the splitter bars and test the window.

Figure 2-26 Splitter Sample - Proportional Framing Parameters

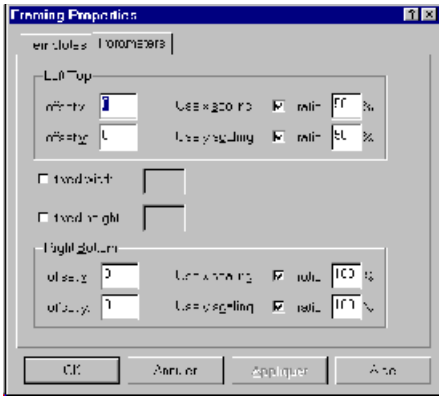
Left



Right top



Right bottom



GUI Builder Issues and Limitations

Events are not displayed as constant names

The GUI builder reverse-engineers the events from the binary event map that is stored in the window class. It compares the values against the event description map returned by the method `ifEvents` in window objects. If the event cannot be found, the builder displays its numerical value (symbols are not affected by the problem). The two main reasons for this are:

- A superclass handles notifications from controls that are not part of the layout of the edited class. The handlers appear in the event map but are ignored by the window. No workaround is necessary because the GUI builder skips events defined in superclasses when generating a new event map.
- A handler uses notifications that are not defined in the `ifEvents` method of the control in question. The correction is to add the declaration of the notification in the `ifEvents` method.

Re-editing a Dialog may modify control styles

When a dialog is created, the system dialog procedure in Win32 may change certain control styles to give the dialog a 3D effect. For example, the `WS_BORDER` style of a list box is removed and a `WS_EX_CLIENTSTYLE` added to the extended control style. When the GUI builder edits a window, it uses the actual window styles, meaning that some styles may not match the original template styles. The interface builder has special code that hides these differences in common cases.

The style differences are merely cosmetical since the final styles are identical. In some cases there can be conflicts when saving a dialog template as a resource script, resulting in incorrect styles. The workaround is to edit the resource script manually.

Editing an existing window fails

Editing a window may fail because the implementation does not expect to be opened by the GUI builder. For example, a dialog that retrieves data from a database may fail in its `initWindow` method because no connection is available. The workaround is to

temporarily disable the parts of the implementation that cause trouble. If the window is a dialog, the interface builder bypasses the normal creation after the dialog has been saved as an in-memory template.

Another reason is that a required resource DLL is not in the path, or the window has not been registered.

Editing ContainerWindow fails

The Interface Builder does not support editing **ContainerWindow** instances.

CHAPTER 3 Base Class Libraries

The first part of this chapter, *Basic Concepts*, presents the basic concepts of Smalltalk and how they are implemented in Smalltalk MT. The second part, *Core Classes*, briefly presents some of the most important classes in the system.

Basic Concepts

Objects

An object is a memory structure that stores either binary data (it is a **byte** object) or references to other objects (it is a **pointer** object), plus information that defines its behavior.

An 8-byte header that contains information about the object (its behavior and a size/flags field) precedes each object. The address of the object points to the user data, meaning that you can directly pass a Smalltalk object such as a string or a structure to an API.

Table 3-1 Object Terminology

Parameter	Description
Byte Object	An object that contains binary data (bytes).
Pointer Object	An object that contains pointers to other Smalltalk objects.
Indexed Class	A class whose instances contain a variable number of elements that can be accessed using a one-based index (`at:` and `at:put:` messages). Byte objects are always indexed.

Note — One of the salient features of Smalltalk MT is that objects reside at fixed addresses in memory, as in C. The benefit for development is that you can always use the address of an object, without fearing that it could be moved around. And at runtime, direct memory references result in better performance.

Classes

Each object in the system is an **instance** of a unique class. Each class is itself an instance of **Metaclass**. You can find out to which class an object belongs by sending the message `class`.

Example:

```
#(1 2 3) class
    → Array

12 class
    → SmallInteger

Window class
    → Window class

Window class class
    → Metaclass
```

Types of Classes

Byte Classes

A byte class creates byte objects. The class is defined using the message `variableByteClass:`.

Pointer Classes

The message `variableSubclass:` creates a class that instantiates indexed pointer objects. The message `subclass` creates a class that instantiates non-indexed pointer objects.

The superclass of a pointer class must be a pointer class (an object has either pointers or bytes, but not both). The superclass of a byte class must be a zero-size class (i.e., a pointer class without instance variables or a byte class).

You instantiate a class with one of the following messages:

Table 3-2 Object Instantiation Messages

Message	Description
*basicNew**	Instantiates an instance of the receiver, which must be a class. The message cannot be subclassed.
*basicNew:**	Same as above, and specifies the size of the instance. The argument, which must be an Integer, is interpreted as follows: ⇒ If the receiver is a byte object, the size specifies the number of bytes. ⇒ Otherwise (the receiver contains pointers), the size is the number of pointer slots. ⇒ In all cases, the actual net size is augmented by the space required for instance variables (for byte objects, this is normally zero).
New	The message can be subclassed. The default implementation in Behavior calls `basicNew`.
new:	The message can be subclassed. The default implementation in Behavior calls `basicNew:` if the receiver is indexed, otherwise raises an exception.
new:on:	The message is private and its description is only provided for completeness. It allocates an object at a specified address location and increments the pointer.
heapAlloc *heapAlloc:*	Same as `new` and `new:`, but allocates an object on the external heap. The object must be explicitly freed using `heapFree`.
localNew *localNew:*	Same as `new` and `new:` but allocates an object on the stack
new:on:	Place an object at a given pointer address

Note Only `new:` tests whether the receiver is indexed.

Methods

The behavior of an object is defined by **methods**, which are identified by **selectors**. A method is a function that is executed by sending a **message** to the object. A message consists of a selector (that identifies the method to perform within the object) and arguments that as expected by the method.

* This message is a pseudo-message

Table 3-3 Method Terminology

Term	Description
Argument	A method's argument, described by the method header.
Selector	Within an object, a method is identified by a selector. The selector is a symbol whose string representation determines the precedence and the number of arguments expected by the method.
Message	A message comprises a selector and arguments. There are several kinds of messages as described below. A message is sent to an object (the receiver of the message) in order to perform some action.
Method	A method consists of compiled code, bound to be executed when a message with the method’s selector is sent to the object for which the method has been defined.

Messages

A message is a two-way communication mechanism between objects. A message consists of a **receiver** (the object the message is sent to), a **selector**, and the appropriate number of **arguments**. When the receiver has processed the message, it returns a value to the sender.

There are three kinds of messages, listed in the following table by decreasing precedence:

Table 3-4 Message Types

Message	Description
Unary message	A message with no arguments. The selector string starts with a letter or underscore character and consists entirely of letters, underscores and numbers. Example: `1 negated`
Binary message	A message with a single argument. The selector string consists of one or two special characters (other than digits, the underscore, and letters). Example: `1 + 2` arithmetic messages: the messages `+ *` are commutative, and `a - b` is equivalent to `a + (b negated).`
Keyword message	A message with one or several arguments. Each argument is preceded by a word that starts with a letter or underscore character and consists entirely of letters, underscores and numbers, except for the last character which is a colon (:). Example: `at: index put: object`

Cascaded Messages

Cascading is a particular syntactic form that sends several messages to the same receiver. Using cascaded messages often improves readability and eliminates the need for a temporary variable. A semicolon separates each message from the next cascaded message.

Example:

```
IdentityDictionary new
    at: 16r1000 put: 'pears';
    at: 8 put: 1 @ 2;
    yourself
```

evaluates as

```
temp := IdentityDictionary new.
temp at: 16r1000 put: 'pears'.
temp at: 8 put: 1 @ 2.
temp
```

Unary Messages

A unary message is a message without arguments. The selector starts with a letter, followed by an arbitrary number of letters and digits. Unary messages have the highest precedence.

Examples:

```
#(1 2 3) size
```

```
'abc' size negated evaluates as (('abc' size) negated)
```

Binary Messages

A binary message takes a single argument. The selector is composed of one or two non-alphanumeric characters from the list below.

```
! % & * + - / < = > ? @ | ~ \ ,
```

Note	The messages `& * + = |` are commutative. A class that implements such a message must observe this rule.

Example:

```
1 + 2
```

```
1 + 2 * 3
```

evaluates as

```
(1 + 2) * 3
```

Keyword Messages

A keyword message takes at least one argument. The selector consists of keywords, separated by colons (`:`).

Each keyword starts with a letter, followed by an arbitrary number of letters and digits. The message is declared with each keyword preceding the matching argument, such as in:

```
array at: i put: object
```

Example:

```
'abc' at: 2 + 1 negated put: $b asUpperCase
```

evaluates as

```
'abc' at: (2 + (1 negated) ) put: ($b asUpperCase)
'abc' at: 1 put: $B
'Bbc'
```

Variable Argument Messages

A variable argument message is a special keyword message that generates a call to a variable-argument method. Variable-argument messages are useful when interfacing certain C functions.

Typically, you would implement variable argument calls when wrapping a function that accepts a *va_list* type parameter. Most variable-argument functions have a counterpart that accepts a *va_list*. Instead of implementing a method for every number of arguments you expect to receive, you can implement a single wrapper that works with an arbitrary number of arguments.

For example, instead of implementing the methods #`wsprintf:with:`, #`wsprintf:with:with:` and so on, String has a single method #`wsprintf:va_list:` that works with an arbitrary number of arguments.

Calling a Variable-Argument Method

The selector must be composed of one or more keywords, followed by one or more `_with:` keywords; one per variable argument.

```
keyword1:{keyword:}_with:{_with:}
```

Example:

```
raiseException: anExceptionCode
  numberOfArguments: nNumberOfArguments
  va_list: va_list
^self raiseException: ST_EXCEPTION_ERROR
  numberOfArguments: 1
  _with: szMessage asString basicAddress
```

A multi-byte parameter can be specified using `_with`*nn*`:`, where *nn* specifies the number of bytes to push onto the stack (it must be a multiple of four).

Example:

```
String wsprintf: '%f' _with8: 1.2
```

Note	Float arguments to printf functions are passed as a double precision floating point values, hence the 8 byte specification.

Implementing a Variable-Argument Method

The implementor must implement the method:

```
keyword1:{keyword:}va_list:
```

The *va_list* parameter is a the raw 32-bit address of a *va_list* structure. It is not an object, but you can access the arguments with the pseudo-message `_at:`.

Equivalent Smalltalk Code

The equivalent Smalltalk code of a variable-argument call to `wsprintf` is:

```
| arg_list |
arg_list := WordArray new: 3.
arg_list at: 1 put: 'va test'.
arg_list at: 2 put: 1234.
arg_list at: 3 put: 16rABCD.
String wsprintf: '%s %d %x' va_list: arg_list.
```

Accessing Arguments from Smalltalk

It is possible to use the variable argument convention directly in Smalltalk. The implementor must know the number of arguments (for example, they must be passed as a separate non-variable parameter).

The *va_list* parameter must be accessed using `_at:` because it is not a real object.

Example:

```
!Object class * methods!
varMethod: count va_list: va_list
  1 to: count do: [ :i |
      Transcript show: '\n',i printString,' : ',
      (va_list _at: i) printString
  ].! !
```

evaluate:

```
Object varMethod: 3
    _with: 'abc' basicAddress
    _with: 123 basicAddress
    _with: Transcript basicAddress
```

The result is:

```
1 : 'abc'
2 : 123
3 : TranscriptWindow(20246)
```

Note	An attempt to send a message to *va_list* or to read beyond the end of the list will fail with unpredictable results.

Variables and Namespaces

The Smalltalk compiler recognizes several variable and identifier types, listed below in the order of their lookup.

Table 3-5 Smalltalk Variables and Identifiers

Type	Scope	Description
Pseudo-variables	Global	**true**, **false**, **nil**, **self**, **super**.
Argument	Method	A method's argument, described by the method header.
Local variable	Method	A local variable, limited in scope to the method in which it is defined.
Block argument	Block	A block argument, limited in scope to the method in which the block is defined.
Class	Global	A class is globally accessible.
Global variable	Global	A global slot accessible from every point in the image, and in every thread.
Instance variable	Class instance methods	A pointer slot in an object, scoped to the particular instance.
Class variable	Class methods, Class instance methods	A pool of slots shared by the class in which it is defined, subclasses and instances of the classes.
Class instance variable	Class methods, Class instance methods	Same as an instance variable, but the object is a class.
Pool dictionary constant	Class	The lookup starts in the current class, in the order of declaration. It continues in superclasses until all pools have been scanned or a matching entry has been found.
Thread-local variable	Global	A variable that is visible from every point in the image, but whose value is different for each thread.

Variable names must be unique within their scope. Classes, pseudo-variables and pool constants are not variables. Classes are in the same name space as global variables, but cannot be assigned (e.g., `OrderedCollection := 1` is not allowed).

Namespaces

The compiler resolves names in the following order:

1) pseudo-variables **true**, **false**, **nil**, **self**, **super**
2) frame variables
3) classes
4) global variables
5) instance variables
6) class variables
7) class instance variables
8) pool dictionary symbols, starting in the class and continuing in superclasses
9) thread-local variables
10) aliases (a placeholder for a class)

An alias replaces the name of a class. Currently, three aliases are defined:

Dictionary maps to **MappingTable**

File maps to **FileStream**

Date maps to **Time**.

Global Variables

A global variable is globally visible to all threads in the process.

For example, after executing

```
Global := 'abc'
```

all threads in the process have access to the string `'abc'`.

Thread Local variables

Thread Local variables are used like global variables, except that their values are specific to each thread. Thread Local variables are initialized to **nil**. Thread Local storage should be used when a global scope is required in a thread.

The following table lists the predefined thread local variables.

Table 3-6 Predefined Thread Local Variables

Variable	Description
debugger	When a thread is being debugged, this variable references the debugger. Otherwise, it is **nil**.
progressIndicator	If not **nil**, references an object that has the progress bar interface. Methods that perform lengthy operations use this variable to display the progress.
RecursiveSet	Used internally to detect recursive operations, such as `printOn:` .
thisApplication	References the **WinApplication** instance for a GUI thread, otherwise set to **nil**.
Transcript	References the Transcript window in the main thread. In other threads, the variable is usually **nil**.

Class variables

Class variables can be used in instance and class methods of the defining class and subclasses. A class variable refers to a unique storage location.

Class variables are shared by the defining class and subclasses, but also by threads in the system.

Class instance variables

Class instance variables are instantiated for each subclass of the defining class. Class instance variables are only accessible from class methods in the class and subclasses.

Instance variables

Instance variables represent storage in each instance of a class. Instance variables are initialized to **nil**.

Local storage

Local variables and arguments are visible from anywhere in the defining method.

Block storage

Block variables and arguments are only visible from statements in the defining block, including nested blocks. Block arguments describe the parameters passed to the block. Block local variables improve code readability and may result in code that is more efficient.

Restrictions on using shared variables

In Smalltalk MT, certain routines of the memory management thread run concurrently with other Smalltalk threads. The benefits include better performance on multiprocessor architectures, at the expense of some restrictions when using global storage (global variables, class and class instance variables).

When a globally referenced object is de-referenced globally and re-referenced locally, there is a potential for the object being discarded because the garbage collector may not be synchronized at this point. The code below illustrates this:

```
| g |
g := Global.
Global := OrderedCollection new.
g doSomething
```

There are several ways to overcome the situation, the easiest is to use `MemoryManager>>markObject:`, which ensures that the argument is not discarded during the current cycle.

```
| g |
g := Global.
MemoryManager markObject: g.
Global := OrderedCollection new.
g doSomething
```

Literals

A literal is an object defined within a method; literals do not have to be created explicitly. It can be an instance of the following classes:

Table 3-7 Types of Literals

Literal Class	Example
Array	`#(1 2 (24.2 9) 'abc')`
WordArray	`#[1 2 WM_CLOSE]`
Character	`$c`
Float	`1.2`
Integer	`3`
String	`'abc'`
Symbol	`#at:put:`

For example:

```
#(1  mySymbol  (9.3 5 'aString') 'myString' 0.3)
```

defines an array with the following contents:

Table 3-8 Sample Literal Array

index	element	Class
1	1	Integer
2	#mySymbol	Symbol
3	#(9.3 5 'aString')	Array
4	'myString'	String
5	0.3	Float

Literal arrays can be nested at arbitrary levels. Literals have the same protection status as code (usually read-only). This prevents them from being changed inadvertently by an application. For example:

```
MyClass>>myString
    ^'myString'

MyClass new myString at: 1 put: $a
```

does not change the literal and results in a protection violation (this may not be the case in the development image).

Literal Arrays

Literal arrays can be created at compile-time. An array definition begins with the character **#**, followed by a list of elements, delimited by parentheses. Individual elements are separated by spaces. Valid elements are:

- Numbers (**Integer** or **Float** instances)
- Strings (**StringA** or **StringW** instances)
- Symbols
- Arrays
- **nil**, **true**, **false**, which are interpreted as pseudo-variables
- A precompiled expression that compiles to one of the types above (see also Constant Expressions on page 167 for more information)

Example:

```
#(1 2 3 nil #('sub' 'array' 1))
#(1 symbol #another_symbol 'string' ('an' 'array'))
#(1 ##(CLR_BLACK) ##(1 + 2))
```

In the last example, a precompiled expression is used to reference a pool constant. Entering the pool constant directly would have created a symbol instead.

Literal WordArrays

A literal word array is enclosed by square braces ([) and introduced by a pound sign (#). A **WordArray** is similar to an **Array**, except that it stores objects in their 32-bit

representation. It is also possible to use pool constants as long as they represent 32 bit values. Valid elements are:

- Numbers (32-bit **Integer** or **Float** instances; floats are compiled a single-precision values)
- A precompiled expression that compiles to one of the types above

Example:

```
#[1 2 3]
#[WM_CREATE 2 3]
```

Literal Strings

A literal String consists of a sequence of characters enclosed by quotation marks ('). To include a quote character within a string, the quote must be doubled. For instance, `'abc''def'` defines the string:

```
abc'def
```

The source character set accepts any 8-bit character (values from 0 to 255). The actual character encoding depends on the default image-encoding standard, and can be either ASCII or Unicode. You can use escape codes to specify arbitrary unicode characters beyond 255 (see below).

The messages `asStringA` and `asStringW`, when sent to a literal string, override the default character encoding. The compiler generates a literal string of the specified encoding instead (the message is not sent at runtime).

Examples:

`'Copyright\xA9 '`	inserts the copyright symbol. In ANSI, it defines a **StringA**, in the Unicode model an instance of **StringW**.
`'This is a string' asStringA`	defines an ANSI string, regardless of the current character encoding.
`'This is a string' asStringW`	defines a Unicode string, regardless of the current character encoding.

String Escape Codes

Characters inside a string can be specified as escape codes from the list below:

Table 3-9 String Escape Codes

Character Escape Code	Meaning	Hex Value
`\a`	Alert	7
`\b`	Backspace	8
`\f`	Formfeed	0c
`\n`	Newline + Return	0c0a
`\l`	Newline	0a
`\r`	Return	0c
`\t`	Horizontal Tab	9
`\v`	Vertical Tab	0b
`\\`	Backslash	5c
`\xhh`	Hex number	hh
`\Xhhhh`	Hex number (wide character)	hhhh

If the character following the backslash is not in the list above, the backslash is taken literally. For example:

```
'abc\def'  ->  abc\def
```

You can specify a character by its integer value, using `\x` or `\X`, followed be respectively 2 or four hexadecimal numbers. For example:

```
'abc\x3fdef'   ->  abc?def
```

In the Unicode model, characters whose encoding is above 255 can be coded using the extended hexadecimal notation (`\Xhhhh`).

Literal Symbols

A literal Symbol starts with a pound-sign and obeys to the rules set forth in *Messages* on page 153.

Symbols resemble Win32 Atoms because they encapsulate a handle that is used to access a string representation. The benefit of separating the symbol from its associated string is that the symbol table can be replaced at runtime by a much smaller table.

Examples

```
#at:put: asInteger
    → 33556385

Symbol value: 33556385
    → #at:put:
```

Blocks

A block is a piece of code whose execution is differed until it receives a message that requests execution. The code is executed in the context of the defining method, e.g., it has access to the local variables of the home method.

A block is delimited by square braces: **[** ... **]**. A block may accept an arbitrary number of arguments; in this case, the first declaration after the opening bracket (**[)** lists the arguments, such as in:

```
[ :arg1 :arg2 :arg3 | ... ]
```

The vertical bar **|** marks the end of the argument list. Block arguments are treated like the method's local variables.

Optionally, a list of block local variables may follow the argument declaration, if present. Block local variables are delimited by vertical bars, such as in:

```
[ :arg1 :arg2 :arg3 |  | a b |... ]
```

Types of Blocks

A **StaticBlock** is a re-entrant optimization of **ContextBlock** that stores its arguments on the stack rather than in a home context. A **StaticBlock** object is created in lieu of a **ContextBlock** if the following conditions are met:

- The block does not contain other blocks
- The block does not contain a return statement
- The block does not use any of the home method's arguments or local variables
- The block does not use the pseudo variables self and super

The compiler automatically creates **StaticBlock** instances whenever possible, and their usage is entirely transparent.

Reentrancy Issues

Due to their design, Smalltalk **Block** instances are not re-entrant (i.e., they cannot be called safely by concurrent threads). Therefore, multiple threads cannot share the same block unless synchronization is provided. Exceptions to this rule are **StaticBlock** instances, which are optimized blocks that do not require a method context. The following possibilities arise:

- The block is only used by one thread. No restrictions apply.
- The block is shared, but synchronization (i.e., a **CriticalSection**) ensures that it is only used by one thread at a time.
- The block is a **StaticBlock** and is re-entrant. However, the block statements do not have access to any of the home method's variables and arguments, including `self`. This is usually not a problem if the block arguments contain sufficient information, otherwise thread local storage can be used to store and retrieve per-thread data.

Characters

A character is defined literally by specifying the character, preceded by a **$**. For example, `$c` specifies an instance of **Character** that represents the character **c**.

A **Character** instance uses 16 bits to store the information, so it can accommodate Unicode encoding.

Constant Expressions

Abstract

Expressions that begin with the token `##(` and up to the matching closing parenthesis are evaluated at compile-time. The benefit is that an expression that evaluates to a constant is evaluated only once, when it is compiled, and not each time the code is run.

An inline expression must be relatively simple (for example, it cannot use cascaded messages or blocks). However, this is not really a limitation since the expression can call any method.

Result of a constant expression

The result of the compilation can be an instance of the following classes:

Integer, **String**, **Float**, **Character**, **Class**, (..)[2]

or a byte object such as a structure.

Objects that can not be stored as literals evaluate to **nil**. Highlight the constant expression and evaluate it to see whether it returns the expected result.

Note It is superfluous to pre-compile arithmetic expressions with integer values because the compiler takes care of it automatically.

Limitations

The resulting object must be flat, equivalent to a literal array, or a flat byte object.

Table3-10 Constant Expression Limitations

	Expression	Comments
✗	##(OrderedCollection with: 'abc' with: 'def')	Fails because the result is not an Array or a byte object.
✓	`##((OrderedCollection with: 'abc' with: 'def') asArray)`	The result is an Array.
✓	`##(LOGFONT new faceName: 'Arial' height: 12 weight: 600)`	LOGFONT copies the string contents to the structure.
✗	`##(LOGFONT new height: 12; weight: 600; yourself)`	You cannot use cascaded statements directly. The expression must be moved to a separate method, which is called by the inline expression.
✗	`##(COMBOBOXEXITEM new pszText: 'abc')`	You cannot set a pointer value inside a structure.

The resulting object is scanned by the image builder and added to the references used by the image. For example, using a precompiled `LOGFONT` structure automatically references the **LOGFONT** class.

Examples

To create an initialized `LOGFONT` structure:

```
##(LOGFONT new faceName: 'Arial' height: '12' weight: 600)
```

A ubiquitous use is passing the size of a structure, as in the code fragment below:

[2] Depends on the compiler version

```
WINAPI GetObject: hFont
    with: ##(LOGFONT sizeInBytes)
    with: lf.
```

Constant expressions can also be used in literal arrays. For example,

```
#(1 ##(WM_USER+1))
```

defines a literal array of two elements, 1 and 1024 (`= WM_USER + 1`).

The same is **true** for literal WordArrays:

```
#[1 ##(WM_USER+1)]
```

Numbers

Number literals are instances of **Float**, **Integer** or **Fraction**. The presence of a decimal point followed by one or more digits determines whether a **Float** or an **Integer** is created.

A sequence of digits defines an integer in decimal format. To use another base, the number radix can be specified (in base 10), followed by **r** and the integer in the specified base. For example, `16r41AF` is the hexadecimal representation of the integer `16815` .

Smalltalk MT accepts both uppercase and lowercase alphabetic characters for radix numbers.

Control of flow Statements

Smalltalk lets you use all major conditional and iterative constructs.

Overview

Table3-11 Boolean Expressions and Comparisons

Smalltalk expression	Equivalent C expression
A ifTrue: [B]. A ifFalse: [B]. *A must evaluate to a boolean*	if (A) { B } ;
A ifTrue: [B] ifFalse: [C]. A must evaluate to a boolean	if (A) { B } else { C };
[A] whileTrue: [B]. [A] whileFalse: [B]. *A must evaluate to a boolean*	while (A) { B };
A ifTrue: [B] ifFalse: [C]. A ifFalse: [B] ifTrue: [C]. *A must evaluate to a boolean*	if (A) { B } else { C };
[A] whileTrue: [B]. [A] whileFalse: [B]. *A must evaluate to a boolean*	while (A) { B };
A case: B1 {case: B*i*} perform: [C*i*] {...} {default: [C*n*] *B1 - Bn must be constant values*	switch(A) case: B ... break; default:
(A) or:[B]	if ((A) || (B)) ;
(A) and: [B]	if ((A) && (B)) ;
(A) || (B)	if ((A) || (B)) ;
(A) && (B)	if ((A) && (B)) ;
(A) | (B)	if ((A) | (B)) ;
(A) & (B)	if ((A) & (B)) ;

Note 1 The messages **isNil** and **notNil** are inlined.

Note 2 The **case** statements are Smalltalk MT extensions. The generated code is extremely efficient.

Loop Expressions

Smalltalk expression	Equivalent C expression
`A timesRepeat: [ B ].` `A must evaluate to an integer`	`for(i=1, i<=A, i++) B;`
`A to: B do: [ :i | C ].` `See below`	`for(i=A, i<=B, i++) B;`

Inlining Issues

to:do: and timesRepeat:

`to:do:` and `timesRepeat:` are inlined if the argument block is an immediate block. For example,

```
| block |
block := [ :i | ... ].
start to: stop do: block
```

is not inlined, meaning that the message `to:do:` is sent to `start` (this is useful if you want the loop to work on non-integers).

Example 1:

```
| b |
b := [ :r |
    Transcript show: r printString; show: '\n'.
].
PI to: 2*PI do: b.
```

works as expected.

Example 2:

```
PI to: 2*PI do: [ :r |
    Transcript show: r printString; show: '\n'.
].
```

raises a runtime exception.

to:by:do:

`to:by:do:` is inlined if the following conditions are met:

- the block is an inline block
- the increment is an integer

Example:

The block below is not inlined:

```
PI to: 2*PI by: 0.3 do: [ :r |
        Transcript show: r printString; show: '\n'.
].
```

Boolean Expressions

Logical OR and AND

Logical OR and AND expressions are part of the general control-of-flow statements. A logical expression can be expressed in terms of `ifTrue:` and `ifFalse:` statements.

The general format is:

```
receiver or: [ statements ].
```

Respectively

```
receiver and: [ statements ].
```

The block is only evaluated if the receiver evaluates to **false** (respectively **true**).

The binary operators || and *&&* can be used in lieu of *or:* and *and:*. The advantage of the binary counterpart is that multiple expressions can be combined easily, without resorting to enclosing parentheses.

Arithmetic or Boolean OR and AND

These expressions have no incidence on the control-of-flow statements because both parts of the message are always evaluated.

The general format is:

```
receiver | argument.
```

Respectively

```
receiver & argument.
```

Examples

Given the hypothetical methods `conditionX` where X=A,B,C,D:

```
conditionA
    Transcript show: 'A'.
    ^true


conditionB
    Transcript show: 'B'.
    ^true


conditionC
    Transcript show: 'C'.
    ^false


conditionD
    Transcript show: 'D'.
    ^false
```

Smalltalk expression	Transcript
self conditionA || self conditionB	A
self conditionA || self conditionC	
self conditionA | self conditionB	AB
self conditionA | self conditionC	AC
self conditionC || self conditionA	CA
self conditionC | self conditionA	CA

Source Management

Abstract

Unlike most Smalltalk implementations, Smalltalk MT distinguishes between development-time interfaces and runtime objects, in an effort to promote runtime efficiency.

All source-code related objects are completely separated from the runtime objects. A class, by itself, has no way of knowing the names of its instance variables, pool dictionaries and other source-specific items. Source code information is kept in instances of **ClassDescriptor** (for classes) and **MethodDescriptor** (for methods). These instances are stored in a memory-mapped file and loaded by the compiler.

Categories

Smalltalk methods are classified by categories. Each method descriptor has a pointer to the category to which it belongs. Because a category is unique in the system, it is possible to rename it.

Several reserved categories modify a method's properties.

Table 3-12 System Categories

Category	Description
CALLBACK	Tells the compiler that methods of this category must be present in the runtime image. Usually, methods such as WM_XXX messages, which are called indirectly, belong to this category.
DEBUG	Methods under this category are stripped from a runtime build in release mode. Typically, methods that perform diagnostic tasks and assist in debugging are placed under this category.
EXTERN	Tells the compiler that the actual code for the method is not required. Typically, one-time initialization methods and examples are stored under this category to diminish the runtime requirements. Filing in such a method compiles and installs the code. However, the code is not installed the next time the image is compressed.
INTERNALDEV	As the name implies, methods under this category are for internal development purposes only and will not appear in a final executable.

Export categories specify the calling convention that a method supports. Such methods are directly callable from external code but not as Smalltalk methods from the image. The compiler generates the appropriate entry and exit code so that the method can be called by non-Smalltalk code. The return value must have an API parameter representation, otherwise a run-time exception will occur.

Table 3-13 Export Categories

Category	Description
`CDECL`	The method implements the CDECL calling convention. The method is only included by the image builder if it is referenced by code in the build, for example by using `methodAddressAt:`.
`EXPORT_CDECL`	The method implements the CDECL calling convention and is exported. The image builder always includes methods under this category.
`WINAPI`	The method implements the WINAPI calling convention. The method is only included by the image builder if it is referenced by code in the build, for example by using `methodAddressAt:`.
`EXPORT`	The method implements the WINAPI calling convention and is exported. The image builder always includes methods under this category.

Pool Dictionaries

In Smalltalk MT, pool dictionaries define the constants used during the compilation. A pool dictionary is a **StringDictionary** that associates a string with a literal value (an integer, a float, string, character, or symbol). The compiler replaces each pool symbol it encounters with the associated value, meaning that a pool dictionary is the equivalent of an include file in C.

A class description includes a pool dictionary string that lists the pool dictionaries of the class. When an identifier cannot be resolved into a variable or class, the compiler scans the pool dictionaries in the order they appear in the class definition. If the identifier still cannot be resolved, the search continues in the superclasses. In that sense, pool dictionaries can be inherited.

With the Symbol Editor, you can edit pool dictionaries directly or by reading a C-style include file. A pool dictionary can be saved as a file in two formats: the Smalltalk file-in format (`.SM` extension) and an include file format (`.H` extension).

Common pool dictionaries are:

Table 3-14 Common Pool Dictionaries

Pool Dictionary	Description
WinBaseConstants	Corresponds to WINBASE.H and associated files.
WinCharacterConstants	Defines common characters: Cr, Lf, Esc, etc.
WinErrorConstants	Defines Win32 Error constants.
WinGDIConstants	Defines Win32 GDI constants.
WinUserConstants	Defines Win32 USER constants.
WinNetworkConstants	Defines Win32 network related constants.

Conditional Compilation

Because Smalltalk MT uses an optimizing compiler, you can achieve the equivalent of a conditional compilation with standard language constructs. If the left-hand side expression of a condition evaluates to a **Boolean**, the compiler generates only the code that is actually evaluated. For example, given a pool constant FOO:

```
FOO == 1  ifTrue: [ ^'FOO'] ifFalse: [ ^nil ].
```

If FOO is defined as 1, the code above is generated as:

```
 ^'FOO'
```

otherwise it is

```
^nil
```

The code in the blocks can be of arbitrary complexity.

Core Classes

Abstract

Smalltalk MT is designed to meet the demands for an object oriented development system capable of producing high performance, small sized executables for the Microsoft® Windows™ family of Operating Systems.

The core classes satisfy the requirements for a small, low-overhead runtime system.

All language-specific concepts follow Smalltalk standards. This includes collections, behavior classes, streams, blocks, magnitudes, and other classes that are related to neither the user interface nor the operating system.

The design of the GUI classes leverages on Win32 programming expertise. The framework preserves the overall structure of a Windows application, and the experienced Windows programmer will find many similarities with C/C++. There is always an obvious connection between a Smalltalk entity and an underlying Win32 object, and the class protocol preserves and extends the Win32 programming interface. Some of the benefits of this approach are:

- standardized API interfaces are well documented, exhibit known behavior
- training and know-how are readily available
- application code is easier to understand and debug
- performance is increased and code size reduced

Behavior Classes

Smalltalk is a class-based language, which means that every object in the system is an instance of some class that defines its behavior. Moreover, every entity known to Smalltalk is an object, which implies that classes are also instances of a class, called **Metaclass**, itself an instance of **Metaclass class**. **Behavior** defines the relationship

between instances of a class and the class itself. In particular, **Behavior** implements object creation methods (`new`, `new:`, `basicNew`, `basicNew:` , and so forth).

Class implements the methods that create Smalltalk classes. The methods based on `variableByteSubclass:` create a variable byte subclass. Only classes without instance variables can create byte subclasses (the contents of an object are either pointers or bytes, but not both).

Collections

A collection contains other arbitrary objects. All collections derive from class **Collection**, an abstract class that defines common collection methods (note that not all collections support all operations):

- collection operations
  - concatenating two collections
  - converting a collection to another collection
  - iterating over the collection
- element operations
  - adding and removing elements
  - iterating over the collection
  - accessing an element

Collections can be categorized into hashed and sequenceable collections. Hashed collections are unordered, and elements are accessed using a key. Sequenceable collections access elements using an index (an integer between 1 and the size of the array). The index defines an implicit order.

Table 3-15 Collection Classes Overview

Class	Description
Array	Fixed size array of Smalltalk objects.
ByteArray	Fixed size array of bytes, expressed as integers.
IdentityDictionary	Variable size, hashed Collection that stores key-value pairs. The keys can be arbitrary objects that are used to access the

	values. Keys are compared using identity (==).
MappingTable	Variable size, hashed Collection that stores key-value pairs. The keys can be arbitrary objects that are used to access the values. Keys are compared using equality (=).
OrderedCollection	Variable size, indexed collection of Smalltalk objects.
SequenceableCollection	Abstract superclass of collections whose elements are accessed by index.
Set	Variable size, does not store duplicates (hashed).
SortedCollection	Variable size, indexed, sorted according to a sort block.
WordArray	Fixed size array of 32 bit quantities, expressed as integers.

Collection

Collection is an abstract class that defines the access protocol of collections.

Access protocol

Table 3-16 Collection Access Protocol

Method	Description
species	The species of a collection is used to store elements of the receiver. By default, the species is an **OrderedCollection**.

Table 3-17 Collection Enumeration Protocol

Method	Description
collect:	Answers a new species of the receiver that contains the result of evaluating a block with each element.
detect:	Answers the first element for which a block evaluates to **true**, or **nil** if no such element has been found.
inject:into:	Repeatedly evaluates a block with the result of the previous evaluation and elements of the receiver.
reject:	Creates a species of the receiver that contains the elements for which a block evaluates to **false**.

select:	Creates a species of the receiver that contains the elements for which a block evaluates to **true**.

Table 3-18 Collection Operations Protocol

Method	Description
addAll:	Adds all elements of a collection to the receiver.
remove:	Removes an object from the collection and answers the removed object. If the object is not in the collection, raises an exception.
remove:ifNone:	Removes an object from the collection and answers the removed object. If the object is not in the collection, answers a the second parameter.
removeAll:	Removes the elements of a collection from the receiver.

Table 3-19 Collection Testing Protocol

Method	Description
includes:	Answers **true** if the receiver contains an element equal to a specified object.
isEmpty	Answers **true** if the receiver contains no elements.

MappingTable

A **MappingTable** stores arbitrary key - value pairs. A value can be looked up efficiently given its key. A **MappingTable** grows as required to accommodate more elements. It is possible to iterate over the values only, or over the key - value pairs, by using the standard collection protocol with either respectively a one-argument block or a two-argument block.

Table 3-20 MappingTable Access Protocol

Method	Description
at:	Answers the value stored at a specified key. Raises an exception if the key is not present.

at:ifNone:	Answers the value stored at a specified key. If the key is not present, it answers the second parameter.
	
	This allows the caller to simultaneously look up a key and test whether a key is present in an efficient manner. For example:
	
	`(aMappingTable at: aKey ifNone: nil) isNil ifTrue: [`
	`  " key is absent "`
	`  ...`
	`].`
	
	The code is functionally equivalent to the method `at:ifAbsent:`, which executes a block when the key is not found (`at:ifAbsent:` is not part of the core methods). However, `at:ifNone:` is more efficient because the test is inlined.

keyAtValue:	This method answers a key that corresponds to a specified value. The implementation performs a linear search in the collection of values. It answers **nil** if the key is not found. If there are several values that are equal to the given value, it answers the first value encountered.
keys	Answers an **Array** with all the keys contained in the receiver.
size	Answers the number of key-value pairs in the receiver.
values	Answers an **Array** with all the values in the receiver.

Table 3-21 MappingTable Enumeration Protocol

Method	Description
do:	Evaluates a specified block. If the block is a one-argument block, it is evaluated with each value in the MappingTable. If the block is a two-argument block, it is evaluated with each key and value in the MappingTable.
inject:into:	Repeatedly evaluates a block with the result of the previous evaluation. If the block is a two-argument block, it is evaluated with each value and the previous evaluation result. If the block is a three-argument block, it is evaluated with each key, value, and the previous evaluation result.
keysDo:	Evaluates a block with each value in the MappingTable.
select:	Creates a collection that contains the elements for which a specified block evaluates to **true**. If the block is a one-argument block, creates and answers a species of the receiver that contains the values for which the block evaluates to **true**. If the block is a two-argument block, creates and answers a MappingTable that contains the keys and values for which the block evaluates to **true**.

IdentityDictionary

An **IdentityDictionary** compares keys using identity rather than equality. Since an **IdentityDictionary** compares symbols using their integer value instead of identity, it can be used with symbols as well.

OrderedMappingTable

An **OrderedMappingTable** keeps track of the order in which elements are added. Iterations are performed in the order of the elements.

Example:

```
| omt |
omt := OrderedMappingTable new.
omt at: 'def' put: 456;
    at: 'abc' put: 123;
    at: 1@2 put: 789.
omt keys

    → OrderedCollection('def' 'abc' 1@2 )
```

StringDictionary

A **StringDictionary** uses string keys. Otherwise, it behaves like a regular **MappingTable**.

StringTable

A **StringTable** compares keys using case-insensitive comparisons.

Example:

```
| stable |
stable := StringTable new.
stable at: 'abc' put: 123.
stable at: 'ABC'

    → 123
```

SystemDictionary

A **SystemDictionary** uses integer or symbol keys. Internally, symbols are converted to their integer identifier.

SequenceableCollection

SequenceableCollection is the abstract superclass of collections that can be indexed.

Table 3-22 SequenceableCollection Access Protocol

Method	Description
first	Answers the first element.
indexOf:	Answers the index of an object. The implementation performs a linear search using equality. Variations of this method are `indexOf:before:` and `indexOf:startingAt:`. If the object is not found, the method answers zero.
indexOfSubcollection:	Searches the receiver for a subcollection. Answers the index of the subcollection or zero if not found.
last	Answers the last element.

Table 3-23 SequenceableCollection Copying Protocol

Method	Description
copyFrom:to:	Answers a species of the receiver that contains a range of elements.
copyWith:	Creates a species of the receiver, adds all elements and appends a specified object.

Table 3-24 SequenceableCollection Enumeration Protocol

Method	Description
reverseDo:	Evaluates a specified block with each element of the receiver in reverse order (i.e., starts with the last element and finishes with the first).
intersect:	Answers the intersection of the receiver and a specified collection (i.e., a species that contains elements that are in both collections).
replaceFrom:to:with:	Replaces a range of elements with elements of a specified collection.

Array

An **Array** is a fixed-size collection that stores objects. It is the simplest and most efficient container.

WordArray

A **WordArray** is a byte object that stores 32-bit values.

ByteArray

A **ByteArray** is a byte object that stores bytes.

String

String is the abstract superclass of **StringA**, which represents null-terminated ANSI strings, and **StringW**, which represents null-terminated Unicode strings.

Interval

Interval represents a mathematical integer interval. It is composed of a **beginning**, an **end**, and an **increment** number that characterizes the progression.

OrderedCollection

An **OrderedCollection** is an indexed collection that can grow to accommodate more elements. The elements remain in the order in which they have been added. It is possible to remove and insert elements.

SortedCollection

SortedCollection is an **OrderedColection** that sorts its elements according to a sort block. Each time an element is added, it is inserted at the position that corresponds to its sorting index.

Note It is more efficient to sort an unsorted collection of elements using `quickSort:`, rather than adding them one by one to a SortedCollection.

Set

A **Set** is a hashed collection that does not allow duplicates. The elements of a Set are unordered.

Example:

```
| set |
set := Set new.
set add: 'abc'; add: 'def'; add: 1@2; add: 'abc'.
set
    → Set(1@2 'abc' 'def' )
```

Stack

A **Stack** is an ordered, growable collection that accesses elements using stack operations such as `pop` and `push:`.

Number Classes

Magnitude classes implement double dispatching, which routes the message to the argument if it is more general. In addition, the following messages are equivalent:

Table 3-25 Arithmetic Message Equivalence

Message	Equivalent message
a + b	b + a
a - b	a + (b negated)
a * b	b * a

Integer

Integer instances are signed integer quantities up to 64 bit. A special representation is used for integers between -1073741822 and 1073741823, as these integers are represented by value rather than by pointers as usual. All other integers up to 64-bit signed quantities are represented by instances of **LargeInteger**. The system converts automatically back and forth between the two representations.

When passed as an API parameter, **Integer** instances are automatically converted to a 32-bit value.

By default, a **LargeInteger** overflow creates a floating-point value. The behavior is implemented in the message `_overflow`, and is customizable.

Some messages specifically target 32-bit integers, as depicted in the table below.

Table 3-26 32-bit Manipulation Messages

Message	Description
not	32-bit complement
extendByte	Sign-extends the low byte of an integer.
unsigned	Converts a 32-bit value into an unsigned integer.

Signed and unsigned integers

The distinction between signed and unsigned integers is a matter of interpretation. The underlying format used by the hardware encodes both signed and unsigned quantities as 32 bit words. Signed integers have the highest bit set, therefore an integer with bit 32 on can be interpreted as both a signed and an unsigned quantity. The message `unsigned`

converts a 32-bit integer to unsigned format, if applicable. This is equivalent to an (unsigned) typecast in C.

The conversion to an unsigned integer is only necessary when comparing two entities. The effect of the conversion can be that a **SmallInteger** gets converted to a **LargeInteger**, such as in:

```
-1 unsigned
```

Extracting 16-bit words

Table 3-27 Extracting 16-bit Words

Message	Description
hiword	Extracts the high 16-bit words and returns an unsigned integer.
ihiword	Extracts the high 16-bit words and returns a signed integer.
loword	Extracts the low 16-bit words and returns an unsigned integer.
iloword	Extracts the low 16-bit words and returns a signed integer.

The code fragment illustrates the difference between the signed and unsigned variants:

```
| i |
i := Integer loword: -23 hiword: -50.
i loword @ i hiword
    → 65513@32718

i iloword @ i ihiword

    → -23@-50
```

LargeInteger instances can be passed as pointers to an API but must be normalized after use because the system primitives do not expect a small integer value in a **LargeInteger** instance, as in the example below:

Example:

```
| counter |
counter := LargeInteger new.
WINAPI QueryPerformanceFrequency: counter basicAddress.
counter + 0    " answers a normalized integer "
```

Radix Numbers

Integers can be represented in bases other than 10. The format is *nrp*, where n is the number base and p the number, encoded using numbers and letters. Acceptable letters are in the range a to z or A to Z, therefore allowing bases between 2 and 36.

Float

Instances of **Float** represent a 64 bit floating point number in double precision IEEE format (Institute of Electrical and Electronics Engineers), which corresponds to the long **double** type in C.

The floating-point library uses the arithmetic coprocessor instructions, which are performed on 80 bit values. You can set the control word of the coprocessor, which gives you control over how overflows, exceptions, rounding and precision is handled.

Using floats in API calls

The API parameter descriptions support **double**, which denotes a 64-bit floating point value passed by value, **float** for 32-bit floating point values, and *&*, which forces the parameter to be passed by reference.

Table 3-28 Floating Point Formats under Windows NT / 98

C type	Parameter type	length (bits)
float	[in]	32
float	[out]	FP(0) (80 bits)
double or long double	[in]	64
double or long double	[out]	FP(0) (80 bits)

FP(0): the result is returned on the top of the floating point stack registers. Internally, all computations are done in 80-bit precision. This does also imply that all floating-point types actually refer to the same value; conversions (if any) are performed by the caller.

Smalltalk MT always uses 64 bits for **Float** instances. The default API parameter representation is a 32-bit value (i.e., a C float). If a function accepts **double** parameters,

it is necessary to define the parameters using the API editor. The format of the parameter field is:

*[*Array*]*(p_1 p_2 ... p_n)

where Array is optional and p_i defines the encoding of the parameter at position i.

Table 3-29 Parameter Types in the API Editor

Parameter encoding	Action
0 or float	Convert the parameter to its default C representation and push resulting 32-bit value.
n	Pass a multi-byte value: the parameter is a byte object from which n bytes are pushed onto the stack. n must be a multiple of 4.
double	Pass a Float by value, meaning that 8 bytes are pushed. Equivalent to n = 8.
&	Pass an object by reference. The address of the object is passed as is, without conversions taking place.

C Float Parameter types and how they are passed

Table 3-30 Passing C Floating Point Parameters

C Parameter type	Parameter passing	length (in bytes)
float	By value (default)	**0** (push immediate value) or **float**
double or long double	By value	**8** (push 8 bytes) or **double**
float*	Address	**&**

Instead of using &, you can declare the default type (0) and pass the address of the parameter (see the example below).

Example 1

Given the following function:

```
float WINAPI testFloat(float f, double fd, double *pf)
{
    return 2.3;
}
```

The declaration in the API editor is:

Item	Value
Return value	Float
Number of arguments	Fixed
Parameters	(float double &)
Calling convention	WINAPI

And you call the function using:

```
myFloat := WINAPI testFloat: fp1 with: fp2 with: fp3.
```

where fp1, fp2 and fp3 are floating point values (they can also be literals).

Note that `(float double &)` is equivalent to `(0 double &)` and `(0 8 &)`.

Example 2

For the following parameter declaration:

```
Params =  (float double 0)
```

the function is invoked using:

```
myFloat := WINAPI testFloat: fp1 with: fp2 with: fp3 basicAddress.
```

In this case, reference passing is enforced at the source code level.

Streams

Class **Stream** is the common superclass of all streams. It allows to stream over an indexable collection. An instance of **Stream** contains:

- The collection it is streaming over

- A position variable that holds the current position, an integer between 0 (the stream is set to the beginning) and the collection's size (the stream is set to the end).

The table below lists some subclasses of **Stream**.

Table 3-31 Stream Subclasses

Stream subclass	Description
TokenStreamA	Streams over a byte array and fetches token. The delimiters are adjustable. TokenStreamA can be used to parse an expression.
FileStream	Streams over a file, seen as a succession of bytes.
(File)	Use FileStream to access a file sequentially.
MemoryStream	Streams over a memory segment, seen as a byte array. Use MemoryStream to map the contents of a file to memory and access it randomly.
MappedObjectStream	Maps objects into the address space of the process. Although it is a subclass of MemoryStream for implementation reasons, it is not per se a stream.

Buffered Streams

Because **FileStream** is not buffered, it is rather slow when accessing individual elements. In addition, certain operations are not supported.

IOStream and IOWideStream

IOStream is a buffered file stream that operates in either binary or text mode. **IOWideStream** is a buffered double-byte stream that reads and writes wide character strings.

As the name implies, both streams are based on C runtime streams. A substantial benefit is that they can operate on non-file devices and be used whenever a C stream is expected (such as with printf or scanf functions). In addition, **IOStream** and **IOWideStream** implement all stream operations.

▸ When to use

FileStream implements a lightweight interface to a Win32 file object, **MemoryStream** interfaces a memory-mapped file, while **IOStream** and **IOWideStream** are buffered file streams that can be used in traditional ways, as both binary and text streams.

String Classes

Smalltalk MT has been designed to use either Unicode or ANSI as its default character-encoding format. Class **String** is the abstract superclass of **StringW** (wide character encoding) and **StringA** (single-byte encoding).

The string creation methods in **String** return **StringW** or **StringA** instances based on the default encoding of the image.

Both single-byte and double-byte characters are represented by instances of **Character**.

Internal representation

All strings in Smalltalk are zero-terminated in order to be compatible with C-based code.

Comparing Strings

The standard string comparison routine uses the Win32 API `lstrcmp`. This function uses the locale to perform a word-based comparison of two strings.

String

String implements the standard Smalltalk string interface.

StringW

A **StringW** is a Unicode string in the sense of Win32 (it also includes the terminating NULL wide character).

StringA

StringA is the class for ASCII strings. Like its Unicode pendant, it is zero-terminated and implements the standard string interface. Its elements are instances of **Character**.

Symbol

Symbol represents Smalltalk symbols. Symbols represent strings by unique identifiers, which avoids string duplication and simplifies symbol comparisons. Symbols are stored in **SymbolTable**, an instance of **STSymbolTable**.

It is strongly recommended that applications do not use the symbol-string conversion methods, as this requires the symbol table to be present in the runtime image. Shipping the runtime system without the symbol table greatly reduces the size of the executable.

System Classes

Multiprocessing Classes

Smalltalk MT provides extensive multithreading capabilities based on the Win32 API. For example, multithreading can be used to spawn several unrelated applications that respond independently to user interaction. An application can create any number of threads, and use Win32 synchronization objects to control their execution.

The internal system architecture is reentrant and multiprocessing safe. The automatic memory management runs in a separate thread and preempts garbage-collected threads to scan their memory.

Processes and Threads

Unlike traditional implementations, Smalltalk MT does not have its own processing model. Instead, a Smalltalk process (we call it thread) runs always as a native thread. Also, there are no restrictions on what can be done in a thread.

Class **ApplicationProcess** has one single instance, **Processor**, to represent the current process. It implements the startup methods and the Win32 entry point.

There are at least three running threads at any time. Thread #0 is the main thread and runs the user interface, thread #1 runs the garbage collector and thread #3 is a management thread that is usually only waiting on a set of events to occur.

You create a thread by either sending `fork` to a block or a thread creation method in **WinThread**. The latter creates an instance of **WinThread** to represent the thread object.

A thread can be exited implicitly by transferring control to the caller or explicitly with:

```
Processor exitThread: exitCode.
```

Synchronization

Synchronization objects inherit from **SyncObject**. All subclasses of **SyncObject** map to native Win32 synchronization objects. You must always close a synchronization object explicitly, except in the case of a **WinThread** object obtained through `fork`.

CriticalSection instances correspond internally to a Win32 critical section object. A **CriticalSection** ensures that the code section it protects is used only by one thread at a time. Critical sections are faster than semaphores.

Semaphore maps internally to a Win32 semaphore. A **Semaphore** contains a counter that may vary between zero and the Semaphore's maximum count. The counter defines how many threads at a time may enter the semaphore.

Table 3-32 Semaphore States

State/Function	Effect	Comments
Signaled	`count > 0`	A waiting thread is released.
Release	`count := count + 1`	
Wait	`count := count - 1`	

WinThread instances correspond internally to a Win32 thread object. A **WinThread** object is returned from one of the thread creation methods (`fork, fork:, or WinThread>>createThread:`). A thread is a waitable object that attains a signaled state after completion.

Each `fork` message releases the thread handle when the thread exits. If the application must wait for thread completion, it is more appropriate to use `createThread`.

Blocks

A **Block** object is created during the execution of a method and contains a series of statements bound to be executed when the block is evaluated or invoked by an external

API function. Block statements, which are delimited by square brackets, are executed in the context of the defining method and have access to local variables and arguments of the method. The block may contain a method return statement, which returns from the method context provided this is still possible (otherwise, it raises an exception). Blocks can be nested and have an arbitrary number of block arguments.

Blocks can be called by non-Smalltalk code (e.g., a C callback). See also *Blocks* on page 166 for more information.

Exceptions

Instances of **WinException** hold the exception information available to a filter block. It copies the information pointed to by the **exceptionPointers** parameter to safe storage, making it available for subsequent use by the handler block.

An instance of **WinException** contains two structures: a **context record** that contains the thread's context at the time the exception occurred, and an **exception record** with the exception code, flags and an array of optional information.

See also *Exception Handling* on page 387.

Memory Structures

Abstract

Any code that interfaces with the host operating system is likely to manipulate C structures. This section shows you how to define and use structures from Smalltalk.

Just as in C, you can reference a structure by pointer or by value. The advantage of a pointer is that it is more efficient in terms of storage, and there is no memory transfer to initialize a structure. Sometimes, it is necessary to use a pointer because you must work on the same memory and not on a copy.

Struct Classes

Struct is the common superclass of classes that model structures of 32 bit elements. It has two class instance variables, **readAccessors** and **writeAccessors**, which associate a symbol (the name of a field) with its position (offset) and size. Subclasses of **Struct** initialize these variables. To query and retrieve a field, an application sends a message with the symbol of the field as selector.

The advantage of this implementation is that it is easily generated from the C declaration, it can be reused by instances of **Pointer**, and it requires less code than a full implementation using individually tailored methods. The compiler has code that

manages the structures and lets you browse the field symbols (finding the senders and/or implementors of a symbol).

Initializing Accessors

The structure fields are initialized with the following messages:

Table 3-33 Struct Initialization Methods

Message	Description
addAccessor: size:	Defines the position and size of a structure field. The first argument is the field name, a symbol, the second is the size in bytes of the field. The order of the messages defines the actual offset.
addAccessor: size: bits:	Defines the integral position and size as well as the number of bits. The first argument is the field name, a symbol, the second is the size in bytes of the integral field, and the last defines the number of bits. The order of the messages defines the actual offset. Bit fields are compacted according to the same rules as in C. ***Note:*** *If the last field is a bit field, the accessors must be finalized by adding a nil accessor with a zero size.* `self addAccessor: nil size: 0`
addAccessor:type:	Defines the position and type of a structure field (see below).
addAccessor:type:size:	Defines the position, type, and size of a structure field. The type information is used in inline structure accessors. Please refer to the online documentation for more information about inline structures.

addUnion:	Defines a union of symbols that access overlapping memory areas.
	The argument is an array of elements, where each element can be an array or a symbol (that identifies a field), followed by the size of the field.
	For example, `PROPSHEETHEADER` uses:
	`addUnion: #(`
	`  hIcon 4`
	`  pszIcon 4);`
	so that `hIcon` and `pszIcon` point to the same location.

Setting and Retrieving Values

What follows applies to regular structures. Inlined structures are accessed differently, according to the field type.

When you send a field symbol to an instance of **Struct** or **Pointer**, the return value depends on the field size and can be one of the following:

Table 3-34 Retrieving Struct Fields

field size	Return Value
1, 2, 4 bytes	An Integer. To convert a signed integer (which is the default) to an unsigned, use the message `unsigned`.
N > 4 bytes	The address of the field. For example, `lfFaceName` in **LOGFONT** returns the address of the font name, so you can use `String fromAddress: lf lfFaceName` to retrieve the string.

Table 3-35 Setting Struct Fields

field size	Description
1, 2, 4 bytes	The argument to the method is converted to a raw 32-bit value with `_asCinteger` before it is stored at the field location.
N > 4 bytes	The argument to the method is converted to a raw 32-bit value with `_asCinteger`. The result is interpreted as the address from which n bytes are copied.

Pointer

Instances of **Pointer** can access a structure given its address in memory. You do not usually create a **Pointer** subclass for each structure, instead you create a pointer that uses the definitions of a **Struct** subclass. Once created, a **Pointer** instance can be used like instances of its associated **Struct** subclass, with the notable exception that it cannot respond to messages defined in the **Struct** class. If you need such functionality (for example, setting a string field and its associated size field at once), you must program it in the client, create a specialized subclass of `Pointer`, or not use a pointer at all.

You can skip to the next structure pointed to by a **Pointer** with the message *skip*. If the pointer is bounded, it will not point beyond the last structure. You create a bounded pointer by sending the message *asPointer* to a structure array (which is created by sending *new:* to the structure class).

CHAPTER 4 The Window Framework

This chapter discusses Win32 programming topics. This first part is an overview that discusses Windows messages, message loops and events.

The second part, *The GUI Framework*, presents the Windowing framework in more detail.

Overview

Smalltalk MT implements a flexible framework for processing notifications and events. Class **WinEventHandler** is the abstract superclass of classes that process and forward events. A class instance variable in **WinEventHandler** associates each handled event with a selector to invoke.

Class **Window** implements Windows message handling. Subclasses implement specialized processing for particular GUI elements, for example top-level (frame) windows, menus, and dialog boxes.

Windows Messages

Windows messaging architecture

The operating system generates Windows messages when user interface events occur, and more generally to communicate with an application.

The message pump

Most messages are not directly sent to a window, but are first placed into a queue. Each thread that uses windows must implement a message loop to retrieve and dispatch Windows messages. Class **WinApplication** implements a default message loop. It is normally not necessary to re-implement the default loop.

Method `WinApplication>>getMessage` returns the currently processed message structure. A possible use is to allow a chained event handler to retrieve the parameters of the current Windows event. It is also used in some OLE APIs such as `DoVerb`.

Note Modal dialog boxes have their own message pump (implemented by Windows). Therefore, the regular message loop is not called and accelerators not processed while a modal dialog is open.

The window procedure

The messages are processed in a window procedure, which defines the behavior of a window. The window procedure can call a default procedure implemented by Windows to handle the messages it is not interested in. To create a window with a given window procedure, an application must first **register** it with Windows. Registering entails associating a window class name (a string) with attributes such as the window procedure and optional icons, background brushes and styles. Once a window class has been registered, it can be instantiated by specifying the registered window class name in the `createWindow:` message.

Implementing a custom window procedure

In some cases, it is necessary to implement a customized window procedure. This is done by re-implementing the exported class method `WndProc:with:with:with:` in the subclass of **Window** that uses this procedure.

The different steps that make up the life of a window are outlined below:

- When a Smalltalk window is registered, it reserves a 4-byte area in the window data area.
- The window creation method passes the address of the window object as parameter to the CreateWindow function.
- The window procedure traps the `WM_NCCREATE` message and sends it to the window object. The `lparam` parameter of the `WM_NCCREATE` message points to the extra creation parameter passed to `CreateWindow`, which is in fact the address of the window object.
- The default implementation of `WM_NCCREATE` calls `registerObject` to ensure that the object is not garbage collected.
- The window procedure reads the value of the user data field.
  - If it is zero, the window has not been initialized yet and it either processes a `WM_NCCREATE` message or directly calls the default window procedure.
  - Otherwise, it checks whether the window object implements the method that corresponds to the Windows message.
    - If the method is implemented, it sends the message (`WM_XXX:with:`) to the object.
    - Otherwise, it sends `defWindowProc:with:with:` to the object.
- Finally, the default implementation of `WM_NCDESTROY:with:` releases the object with `releaseObject` and sets the user data field to zero.

A generic implementation of a window procedure is:

```
WndProc: hWnd with: msg with: wparam with: lparam
    "
    Private - The default window procedure.
    Remarks:
    We receive a bunch of WM_XXX before the WM_NCCREATE (i.e., before
    the WM_CREATE). Those are passed right away to the default window
    procedure.
    "
    | pObj wObj selectorId |
    pObj := WINAPI GetWindowLong: hWnd with: GWL_USERDATA.
    pObj ~~ NULL ifTrue: [
        " pObj is the address of the window object "
        wObj := pObj _asObject.
        selectorId := msg <= WinMessages size ifTrue: [
            WinMessages at: msg.
        ]
        ifFalse: [
            WinMessagesEx at: msg ifNone: #defWindowProc:with:with: asInteger.
        ].
        (wObj respondsTo: selectorId) ifTrue: [
            ^wObj __perform: selectorId with: wparam with: lparam
        ]
        ifFalse: [
            ^wObj defWindowProc: msg with: wparam with: lparam
        ].
    ].

    msg ~~ WM_NCCREATE ifTrue: [
        ^WINAPI DefWindowProc: hWnd with: msg with: wparam with: lparam
    ].

    " Filter the WM_NCCREATE message.
    "
    " lparam points to the object "
    wObj := (MemoryManager atAddress: lparam) _asObject.
    wObj handle: hWnd. " first set the window handle "
    ^wObj   WM_NCCREATE: wparam with: lparam.
```

Remarks

- The association between Windows messages and selectors is done in the class variables `WinMessages` and `WinMessagesEx`. The first is an **Array** that contains the Windows messages between 1 and 1000. The message lookup routine simply uses the Windows message as an index into the array. `WinMessagesEx` contains a **MappingTable** that associates the messages that are not within the range of the previous array with selectors.
- The actual implementation is optimized. Message lookup takes place in `Window>>lookupMessage:in:`. If successful, the return value is the address of the method to call. If there is no corresponding method, the return value is zero.

Using windows

A Smalltalk Window instance can either implement a window procedure or merely encapsulate a window. In the first case, it receives `WM_XXX` messages and is responsible for the window's behavior, otherwise it offers merely an interface through which the window can be manipulated. The latter is the case of all predefined Windows controls such as **ListBox**.

Processing Messages

Class **Window** implements the basic functionality for managing windows in the class method `WndProc:with:with:with:` and the instance method `defWindowProc:with:with:`. To process a given window message, simply add a method with the derived Smalltalk selector in the window, and it is called automatically. For example, implement `WM_QUERYDRAGICON:with:` to process the `WM_QUERYDRAGICON` message in your window.

User messages (i.e., messages that are not predefined by Win32) can be processed in the `defWindowProc:with:with:` method.

To process a window message named `WM_XXX`, implement a method named `WM_XXX:with:`. `WM_XXX` can be any window message below `WM_USER`.

In general, an application returns zero if it processes a message, any other value to indicate that the message has not been processed. It is often necessary to pass the window message to the default window procedure, in this case use:

```
self defWindowProc: WM_XXX with: wparam with: lparam
```

Processing WM_USER Messages

As mentioned above, only non-user messages are handled by the automatic message lookup procedure. `WM_USER` messages can be handled by re-implementing method `defWindowProc:with:with:`. Check the first parameter against the message that you wish to process, and do not forget to call the default procedure after you are done.

Message Translation

The application's message loop is implemented in `WinApplication>>run`. After retrieving a message from the queue, it sends the Smalltalk message `#preTranslateMessage:` to the active top-level window object. This gives the window an opportunity to perform window-specific accelerator translation. If the

window returns FALSE, the message is further processed by calling the Windows functions `TranslateMessage` and `DispatchMessage`.

Asynchronous and Idle Processing

Abstract

An idle event occurs when the window message queue for the current GUI thread is empty. An application can choose to perform certain low-priority tasks such as updating a toolbar during an idle event.

Idle processing

When there are no more messages in the input queue, the framework calls the method `onIdle` in WinApplication. The default implementation of `onIdle` processes asynchronous messages (see below) before it calls `onIdle` on each top-level window created by the current thread. The default implementation of `onIdle` in FrameWindow updates a toolbar, if one is present.

Asynchronous processing

It is possible to define messages (instances of **Message**) or blocks that are executed on each idle event. An applications adds a handler block to be executed asynchronously with `addIdleProc:` and removes it using `removeIdleProc:`.

The method `asyncPerform:` in **WinEventHandler** executes a handler exactly once. It can be used to defer the execution of a block or **Message** instance. Deferred execution can be useful under some circumstances to work around GUI synchronization issues.

Note that this is the only place in Smalltalk MT where asynchronous processing occurs. All other processing always occurs synchronously, in response to a Windows event or callback.

Limitations

Idle events are pure GUI events that occur very frequently, so idle handlers should not perform CPU-intensive computations because the user interface will be frozen in the meantime. An idle event runs in the GUI thread at that thread's priority, meaning that an idle handler may starve low priority threads in the system. The main advantage of idle processing is that it occurs in the GUI thread, so it is a good place for updating a toolbar or other user interface elements.

For background tasks that do not use the GUI, it is better to use a separate thread that runs at a low priority (for example idle priority).

GUI Multithreading issues

Many methods are based on the `SendMessage` API. When called from a thread that did not create the receiver window, the function synchronizes the call by waiting until the window thread pumps its message loop (i.e., calls `PeekMessage`, `GetMessage` or `WaitMessage`). In the meantime, the calling thread blocks.

This raises two issues:

- Calling `SendMessage` (directly, or indirectly by using a wrapper method) incurs a significant overhead. On the other hand, methods that are based on `SendMessage` are thread-safe.
- A deadlock occurs if the window thread is blocked.

Note that a **RichEdit** control thread must not be blocked (this is the reason why the debugger uses regular **Edit** controls).

See Also: Win32 SDK Knowledgebase, `PSS ID Number: Q95000`

Events

Events are the glue that binds together the components that make up the user interface of an application. In most cases, events are triggered by some user action.

Some Windows events require a return value, while others simply notify the application. The event handling mechanism allows an application to step out of the event handling chain and return a value to the caller.

The event routing facility leaves the developer with the choice of where to implement application functionality. This encompasses the following options:

- Create a specialized **Control** subclass to display or edit custom data. For instance, **ClassHierarchyView** displays a class hierarchy.
- Create a **FrameWindow** subclass to manage a top-level window and handle specific events. Most top-level windows in the development environment are built this way.

- Create a **WinApplication** subclass that handles application startup and exiting, and processes general events. For example, **DevelopmentEnvironment** handles general events such as `#inspect`, `ID_FILE_INSTALL`, etc.
- Use a document-based architecture.

Table 4-1 Event Terminology

Term	Meaning
Notification message	Win32 message sent through a `WM_COMMAND` or `WM_NOTIFY` message. The framework translates notifications into events.
Event	An event is identified by an integer that corresponds to the original notification code or a symbol if Smalltalk generates the event. Common events are menu commands, control notifications, and symbolic events such as `#size`, `#create`, `#close` a.s.o.
Event Handler	An event handler is invoked whenever a specified event occurs, or a specified event occurred in a given child control. The handler can be a selector invoked in the receiver, or a block in the case of instance handlers in **FrameWindow**. Handlers accept 0, 1 or 2 arguments.

Events are either sent through `WM_COMMAND` and `WM_NOTIFY` messages or created by the **FrameWindow** instance (for instance, the event `#close` is generated when a **FrameWindow** is closed). As such, a command consists of an id (such as a menu identifier) or a combination of an id and a notification message sent by a control. The callback is an object to which the message `#value:` is sent; usually a **Message** instance or a block. The consequence is that command processing is not confined to a unique object, but can be distributed over several specialized model objects.

The framework uses several predefined identifiers to provide default behavior, for example opening a file. These ids are defined in the pool dictionary `StResourceConstants`.

FrameWindow extends the event handling mechanism with a more complex dispatching algorithm that includes the ability to define instance-specific handlers. The instance handler map is created dynamically. It associates events with a block, a message, or a selector to invoke in the receiver. Because blocks and messages can route to arbitrary objects, instance handlers provide maximum flexibility.

The sequence below depicts how events are handled in **FrameWindow**.

1. An event is received (user interface notification or user event).
2. The event is dispatched to a child control according to the following algorithm:
   - If the event is a notification and if the window from which the notification originates is a child control, a new event is created from the notification code and it is dispatched to the control.
   - If a child control has the focus, the event is dispatched to the active child control.
3. If the event was dispatched and the result is an integer, the event handler chain stops and returns the 32-bit integer to the caller.
4. If an instance handler is defined for the event, the handler processes the event, otherwise event forwarding continues.
5. The event is dispatched to the class handler. If the result is an integer, the event handler chain stops and returns the 32-bit integer to the caller.
6. The event is dispatched to the owner.

This architecture has the following benefits:

- Control notifications are redirected to their Smalltalk counterparts.
- The behavior can be changed dynamically at runtime by using instance handlers. Instance handlers and override the default class handler.
- Events are routed to the owner of a frame window. This leaves it up to the application developer to decide where to implement functionality.

Note Only one child control at a time processes the event. This avoids conflicts between event handlers; for example two edit controls both trying to update a menu or toolbar.

Examples:

- **Edit** installs handlers for standard edit commands such as `ID_EDIT_COPY`, `ID_EDIT_PASTE` and so forth, as well as for the Smalltalk-generated event `updateCommandUI`.
- **CodeEdit** adds a handler for the notification `EN_MSGFILTER`.

The GUI Framework

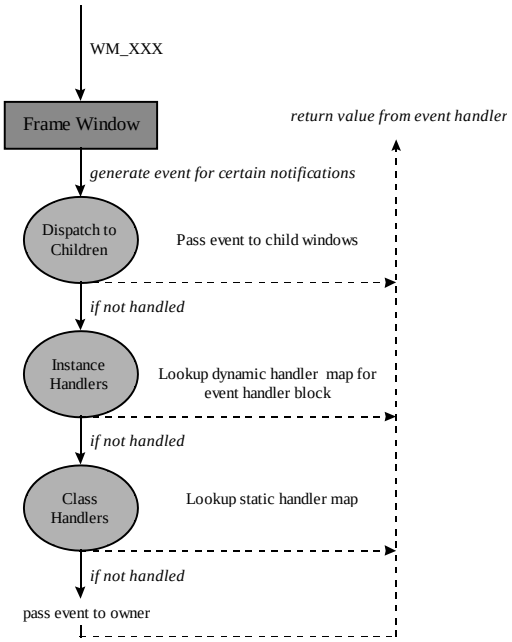
Windows Events

Overview

In the GUI event model, the primary task of an application is to respond to user actions. All user actions result in window messages (`WM_XXX`) being sent to the windows that belong to the application. The window procedure of the window calls the corresponding Smalltalk method (`WM_XXX:with:`) if is implemented, otherwise the default window procedure is invoked.

Certain messages (such as `WM_COMMAND` and `WM_NOTIFY`) are translated into events. A chain of event handlers processes the events: the frame window, child windows, and the owner of the frame and so forth. Each event handler can return a value or pass the event to the next handler.

Figure 4-1 Event Handler Chain



Static Handlers

The framework in **WinEventHandler** defines static handlers. The handler consists of a map that associates an event identifier with a selector bound to be executed in the instance. The handler is defined once when the class is initialized, hence the name static handler.

The implementation lets you define handlers naturally, using

```
when: someID perform: #someMethod;
```

or

```
when: someID in: ctrlID perform: #someMethod;
```

where `someID` defines the Windows notification, `ctrlID` designates a control identifier, and `#someMethod` is a selector that accepts zero, one or two arguments. The arguments that are passed to the method are specific to the notification. Each event has two associated arguments, but a handler can choose to process less.

Instance Handlers

Subclasses of **FrameWindow** can define instance handlers, which are defined on a per instance basis. A handler can be a message or a block (again with zero, one or two

arguments). Instance handlers are more flexible because they allow you to attach arbitrary code to an event. However, they may increase the startup time of the window.

Instance handlers modify the behavior of a window at runtime. As an extreme, you can instantiate a **FrameWindow** or **DialogBox** and define its behavior solely by installing handlers. The code following code snippet creates a window that displays 'Hello World':

```
GraphicsWindow registerClass.
GraphicsWindow new
    when: #paint perform: [ :hDC |
        hDC textOut: 'Hello World' x: 10 y: 10
    ];
    open
```

Event return values

Windows events can be menu or accelerator commands as well as child window notifications. In the case of notification events, the return value is specific to the event.

The table below lists return values for common events, please refer to the Win32 documentation for more information.

Table 4-2 Event Return Values

Notification	Return Value
WM_NOTIFY	Ignored except for notifications that specify otherwise.
WM_COMMAND	Ignored.
WM_PARENTNOTIFY	Always ignored.
#close	0 (FALSE) to close the window, any other value keeps it open.

Notifications and Commands

FrameWindow processes the messages WM_COMMAND, WM_NOTIFY and WM_PARENTNOTIFY and generates a events for the event-handling framework.

An event is composed of the following information:

Table 4-3 Event Parameters

Item	Description
Event ID	hiword = notification code (NULL for menus)
	loword = control or command ID
Param 1	`lparam` parameter (notification specific data)
Param 2	`wparam` parameter (usually the control handle)

Note The window message parameters are passed in inverse order. Because the wparam parameter is usually not of interest, a one-argument handler method is often sufficient to process the *lparam* parameter.

Child Control Forwarding

Child controls of a **FrameWindow** are first given the opportunity to handle an event. In the case of a Windows-generated notification, only the notification code is forwarded (i.e., only the **hiword** of the event code is passed, and the **loword**, which contains the control identifier, is ignored).

Menu and accelerator commands are first dispatched to the child window that has the focus. This makes it easy to implement generic event handlers in controls. For example, **Edit** installs handlers for the Edit menu as well as a handler that enables and disables menu or toolbar items, depending on the state of the edit contents. The handlers are only executed when the control has the focus, so several instances can coexist within the same frame window.

To dispatch commands to a control that doesn't have the focus, two possibilities exist. Either the frame window implements handlers that delegate to one or more child controls, or a control installs dynamic handlers in the parent (using `when:perform:`).

Event Descriptions

Each event source implements a class method named `ifEvents` that returns an array describing the events generated by the window. Each line lists the textual event constant (such as `CBN_SELCHANGE`) and a textual description. The GUI builder uses this information to build up the event connection dialog. You can also quickly view the events by browsing these methods.

Example:

```
ifEvents
    ^#(
    'BN_CLICKED'    'The user clicked the button'        0 0
    'BN_DBLCLK'     'The user double-clicked the button' 0 0
    )
```

See also *Defining* events on page 128.

Using Variable Commands

Sometimes, it is desirable to bind a range of command identifiers to an action to perform, rather than associating each command in the range individually. For example, the **Most Recently Used File** framework uses this functionality to manage a file list.

The framework reserves the interval `[ID_CHOICEFIRST - ID_CHOICELAST]` for variable commands. When a command in that range is processed, the framework raises an event whose identifier is obtained by clearing the low byte, meaning that the range is further broken down into 16 base events (`CHOICEFIRST, CHOICEFIRST + 16r10`, `CHOICEFIRST + 16r20`, ...). The argument to the event is the actual command.

Therefore, a single event handler processes up to 16 commands, allowing an application to treat these commands uniformly. A handler can translate the command identifier into an index in an array, and perform some action on an element in that array.

Example:

1. Define a handler in the class initialization method:

```
...
    addHandler: ID_CHOICE + 16r10           selector: #varCmd:;
```

2. Implement the handler:

```
varCmd: idx
    | i element  |
    i := idx - (ID_CHOICEFIRST + 16r10).
    (i >= 0) && (i <= m_array size) ifTrue: [
        element := m_array at: idx - ID_CHOICEFIRST.
        element doSomeAction.
        ^NULL  " we processed it, terminate handler chain "
    ].
    ^nil " continue handler chain "
```

Displaying a Context Menu

A Frame Window receives the `WM_CONTEXTMENU` message when the user clicks on the right mouse button while over the client area or a child window. **FrameWindow** uses the child identifier to generate a `WM_CONTEXTMENU` event, which can then be processed by an event handler. Alternatively, the frame can re-implement `WM_CONTEXTMENU` to process the message directly, which may be more appropriate if the menu to display depends on the mouse location rather than on a child identifier.

In most cases, the menu is loaded from a resource. The code below loads a popup menu identified by `IDM_POPUP_MYMENU` from the current module and displays the first popup menu.

```
Menu popupMenu: IDM_POPUP_MYMENU
    module: m_module
    index: 1
    owner: self
```

In cases where menu items must be modified before the menu is displayed, the application must load the menu, extract the popup submenu, modify the popup, then display it using `trackPopupMenu:`. The code looks like:

```
hPopup := (Menu fromResource: IDM_POPUP_MYMENU
module: m_module) getSubMenu: 0.
hPopup
    enableItem: ID_MENU1 state: state1;
    enableItem: ID_MENU2 state: state2;
    trackPopupMenu: self
```

Remarks

- Alternatively, the application can first send the popup menu to the `updateCommandUI` handler in order to update the items.
- In simple cases, it is better to define several specialized popup menus rather than a large one with many disabled items.

- The control returns to the caller when the popup menu has been closed, either because the user selected an item or after he or she pressed escape. In the meantime, the owner of the popup receives event notifications.

File Handling

An application can easily add file-handling functionality to its main window by using the file-handling framework in **FrameWindow**.

FrameWindow implements the following methods:

Table 4-4 File Handling Methods

Item	Description
fileOpen	Opens a **File Open** common dialog box that prompts the user for a file name. This method calls also `openFileString` to retrieve the file filters for this window. After the dialog box returns, the method calls `fileOpen:` with the file name and sets the title of the window to the new file name.
filePrint	Default implementation that prints the contents of the text pane, if one is present.
fileSave	This method calls `fileSaveAs` if there is no document (i.e., there is no property at `#document`). Otherwise, it calls `fileSave:` with the document name.
fileSaveAs	Opens a **File Open** common dialog box and prompts the user for a file name to save as. If successful, it calls `fileSave:` with the retrieved file name, and updates the window title.
querySave	Asks the user whether the window contents should be saved, and if so, saves the contents by calling either `fileSave:` or `fileSaveAs.` Returns 0 if the contents can be discarded, otherwise 1. This method can be called by implementations of `queryClose` and `queryDestroy`.
title	Answers the current document name of the window, or **nil** if it is undefined.
title:	Sets the document name of the window.

A subclass of FrameWindow can attach methods to events, as depicted in the table below.

Table 4-5 File Handling Events

Item	Selector
`ID_FILE_OPEN`	`fileOpen`
`ID_FILE_PRINT`	`filePrint`
`ID_FILE_SAVE`	`fileSave`
`ID_FILE_SAVE_AS`	`fileSaveAs`
`ID_FILE_MRU_FILE0`	`fileOpenMRU:` (enables MRU file handling)

The methods attached to the events are implemented in FrameWindow. The subclass of FrameWindow must merely implement methods to open and save a file. Optionally, it can also implement `queryClose` and `queryDestroy` to prompt the user when the window is about to be closed or destroyed.

Table 4-6 Required File Handling Methods

Item	Description
fileOpen:	Opens a file, given its file name. Returns true if successful, otherwise false.
fileSave:	Saves the contents to a file specified by a file name. Returns true if successful, otherwise false.
openFileString	Returns a file filter string. This is a multi-string, as discussed in *FileDialog* on page 255.
queryClose	This method is called when the contents of the window are about to change. Returns FALSE if the contents of the window can be discarded, otherwise returns TRUE.
queryDestroy	This method is called when the application should terminate. Returns 0 if the window can be destroyed, otherwise returns 1.

Most Recently Used File Framework

The **Most Recently Used File** framework manages a file list and registry entries. The most recently used file names are added to a file menu, which may include a *More files...* menu item that displays the entire list of files in alphabetic order.

A **FrameWindow** subclass that wishes to use **Most Recently Used Files** must call `fileUpdateMRU:` with the collection of files or **nil** to load the values from the registry.

Table 4-7 MRU File Handling Methods

Item	Description
fileAddMRU:	Adds a file name to the list of most recently used (MRU) files.
fileMaxMRU	Returns the maximum number of most recently used files to display.
fileMoreMRU	Opens a dialog with the complete list of (MRU) files.
fileMoreMRUString	Returns the string used to display the *More...* menu item for the MRU list. The default implementation loads the string resource `IDS_FILE_MRU_MORE` from the executable module.
fileOpenMRU:	Opens one of the most recently used (MRU) files. The parameter is the index of the file to open.
fileUpdateMRU:	Updates the list of most recently used (MRU) files. The parameter is a collection of file names to display, or **nil** to load the values from the registry.

Using Accelerators

Accelerators provide a quick way to perform commands. Typically, an accelerator is a key combination that gives direct access to a command that is also present under a menu item.

An application can support accelerators in two ways:

- Loading an accelerator table that is valid for all open windows managed by the current thread. To that effect, an application calls `WinApplication>>loadAccelerators:`, passing the name or identifier of an accelerator resource.
- Assigning specific accelerator tables to a frame window. Accelerators can be loaded by sending `loadAccelerators:` to the frame window, passing the name or identifier of an accelerator resource. Frame accelerators override the default application accelerators.

WinApplication first sends `preTranslateMessage:` to the frame window, therefore the window accelerators are translated before the application accelerators.

Example:

In the development image, DevelopmentEnvironment loads the following accelerator resource:

```
ID_ACCEL_ST ACCELERATORS
BEGIN
        VK_F3, ID_EDIT_REPEAT,             VIRTKEY
        VK_F8, ID_SMALLTALK_ACCEPT,        VIRTKEY, CONTROL
        VK_F8, ID_SMALLTALK_FILEITIN,      VIRTKEY, SHIFT
        VK_F9, ID_SMALLTALK_INSPECTIT,     VIRTKEY, SHIFT
        VK_F10, ID_EDIT_SYMBOLS,           VIRTKEY, SHIFT
        VK_F10, ID_EDIT_API,               VIRTKEY, ALT
        VK_F11, ID_CLASSES_BROWSE,         VIRTKEY, CONTROL
        VK_F11, ID_CLASSES_BROWSE_PROJECT, VIRTKEY, SHIFT

     "N", ID_FILE_NEWWORKSPACE,   VIRTKEY, CONTROL
     "O", ID_FILE_OPEN,           VIRTKEY, CONTROL
     "F", ID_EDIT_FIND,           VIRTKEY, CONTROL
     "H", ID_EDIT_REPLACE,        VIRTKEY, CONTROL

    "D", ID_SMALLTALK_SHOWIT,    VIRTKEY, CONTROL
    "E", ID_SMALLTALK_DOIT,      VIRTKEY, CONTROL
     "Q", ID_SMALLTALK_INSPECTIT, VIRTKEY, CONTROL
     "S", ID_SMALLTALK_ACCEPT,    VIRTKEY, ALT
END
```

These commands are common to all Smalltalk development windows.

CHB loads an extended accelerator table that is specific to the class hierarchy browser:

```
ID_ACCEL_CHB ACCELERATORS
BEGIN
        VK_F3, ID_EDIT_REPEAT,             VIRTKEY
        VK_F7, ID_METHODS_IMPLEMENTORS,    VIRTKEY, SHIFT
        VK_F7, ID_METHODS_SENDERS,         VIRTKEY, CONTROL
        VK_F7, ID_METHODS_GREP,            VIRTKEY, ALT
        VK_F8, ID_SMALLTALK_FILEITIN,      VIRTKEY, SHIFT
        VK_F8, ID_SMALLTALK_ACCEPT,        VIRTKEY, CONTROL
        VK_F9, ID_SMALLTALK_INSPECTIT,     VIRTKEY, SHIFT
        VK_F10, ID_EDIT_SYMBOLS,           VIRTKEY, SHIFT
        VK_F10, ID_EDIT_API,               VIRTKEY, ALT
        VK_F11, ID_CLASSES_BROWSE,         VIRTKEY, CONTROL
        VK_F11, ID_CLASSES_BROWSE_PROJECT, VIRTKEY, SHIFT
    "N", ID_FILE_NEWWORKSPACE,         VIRTKEY, CONTROL
    "O", ID_FILE_OPEN,                 VIRTKEY, CONTROL
    "F", ID_EDIT_FIND,                 VIRTKEY, CONTROL
    "H", ID_EDIT_REPLACE,              VIRTKEY, CONTROL
    "A", ID_EDIT_SELECT_ALL,           VIRTKEY, CONTROL

       "D", ID_SMALLTALK_SHOWIT,          VIRTKEY, CONTROL
       "E", ID_SMALLTALK_DOIT,            VIRTKEY, CONTROL
    "Q", ID_SMALLTALK_INSPECTIT,       VIRTKEY, CONTROL
    "G", ID_CLASSES_JUMPTO,            VIRTKEY, CONTROL
    "U", ID_METHODS_UNCALLED,          VIRTKEY, CONTROL
    "Z", ID_EDIT_ZOOM,                 VIRTKEY, ALT
    "N", ID_METHODS_NEWMETHOD,         VIRTKEY, ALT
    "S", ID_SMALLTALK_ACCEPT,          VIRTKEY, ALT
END
```

Using the Registry

Abstract

The registry is a system-defined database that applications use to store and retrieve configuration and initialization data. The registry stores data in a hierarchically structured tree. Each node in the tree is called a key. A key can contain subkeys and data entries called values.

The registry support in **FrameWindow** makes it easy to load and save registry values. It also supports saving and restoring the state of a window.

The Registry Interface

To enable the registry interface, a **FrameWindow** must have a property named `#profile` that specifies the subkey name used to store and retrieve registry values. This name, together with the root key defined in the current application, defines the key under which information is stored and retrieved.

The default profile name of a **FrameWindow** is its window class name, but an application can also assign a different value. The methods `profileName:` and `profileName` respectively set and retrieve the profile name.

Saving and restoring a Window's state

FrameWindow supports saving and restoring a window via the methods `regSaveWindow` and `regRestoreWindow:`. The first returns a **MappingTable** that associates value names to values, the second re-creates a window from state information provided in a **MappingTable**.

The methods are easy to use and subclass because the data is stored in a **MappingTable**. Subclasses can reimplement them in order to manage additional entries.

Saving User Preferences

Method `saveProfile` saves preferences of the current window to the registry. The data includes the position of split bars, fonts used by the child windows, as well as the window extent.

Using Fonts

The font registry interface allows a frame window or an application to associate customizable fonts with child controls. In addition, a frame window can let a user define the font of a window by calling the `chooseFont` method.

A window can use the message `loadProfileFont:` to load a font from the registry. The parameter to the method is the identifier of the child control for which the font should be loaded.

The message `loadProfileFonts` iterates over child controls and loads any fonts associated with a control. In both cases, the fonts are cached by the current application, so that a second access does not have to scan the registry, nor must it create a new font.

Registry Support in WinApplication

Instances of **WinApplication** store and save registry values under the subkey defined for the application, as returned by the instance method `regKey`. This makes it easy to change an application's top-level key without modifying any other code.

WinApplication also has class methods that provide low-level access methods. In addition to Win32 API wrappers, it also defines useful methods that iterate over keys and subkeys, values of a key, delete an entire branch and so forth.

Implementing Online Help

Abstract

The help interface supports tooltips, menu item description strings as well as context-sensitive help. Help in frame windows is similar to event handling: it can be defined statically in the class initialization method or dynamically at runtime.

Static help declarations define how instances of a class react to help requests. This behavior can be modified at runtime, allowing a window to be reused in a different context.

Tooltips

Tooltips are small pop-up windows such as the ones usually associated with toolbar buttons. Tooltips are discussed under *Tooltip* on page 294.

Menu Hints

Menu description strings are displayed in the frame's status bar when the user scrolls through a menu. This feature requires a status bar for functioning.

An application's resources must include a string table that associates each menu identifier with a string to be displayed. For example, the file below provides descriptions for the *File* menus:

```
STRINGTABLE
{
// Standard File commands
ID_FILE_NEW    "Creates a new document"
ID_FILE_OPEN       "Opens an existing document"
ID_FILE_SAVE       "Saves the active document"
ID_FILE_SAVE_AS    "Saves the active document under a new name"
ID_FILE_SAVE_ALL   "Saves all documents"
ID_FILE_CLOSE      "Closes the active document"
ID_FILE_PAGE_SETUP "Changes page layout settings"
ID_FILE_PRINT      "Prints all or part of the active document"
ID_APP_EXIT        "Quits the application; prompts to save documents"
ID_APP_ABOUT       "Displays program and copyright information"
}
```

More examples can be found in the `SUPPORT` directory of the Smalltalk distribution disk.

The implementation is done in `WM_MENUSELECT`. A window that wishes to override the default behavior must re-implement this message.

Calling Help

WM_HELP messages

Context-sensitive help calls the help API when the user presses the F1 key over an interface element of the application, or by some other system-defined means (using for example the mouse help cursor).

The framework tries to locate the child identifier, which is passed in the `WM_HELP` message, in the table of help identifiers. If a corresponding help identifier is found, it invokes the help system with the identifier, opening a popup help window. Otherwise, help is invoked on the default topic, which generally opens the help main window.

Help commands

The framework processes the help commands below:

Table 4-8 Help Commands

Command	Description
ID_HELP_USING	Invokes Help on Help (how to use the help system).
ID_HELP_INDEX	Opens the help index.
ID_CONTEXT_HELP	Opens help on the default help topic.

Help commands are normally located in a help menu. Dialog boxes often display a help button whose identifier is `ID_CONTEXT_HELP`. In addition, the style `DS_CONTEXTHELP` displays a help icon on a dialog's caption. When the user clicks on the icon, the mouse cursor changes to a help icon and he / she can click on any dialog item, generating a `WM_HELP` message with the identifier of the item under the cursor. Alternatively, the user can press F1, which generates the same `WM_HELP` message.

Registering Help Topics

Help handlers are declared in subclasses of **FrameWindow**, either in the class `initialize` method or in `initHelpHandlers`. An application uses the messages in the table below to register help events. These methods are implemented

both as class methods and as instance methods; the class methods define help topics and the help file name statically in the class, while the instance methods override the static help data at runtime.

Table 4-9 Help Methods

Message	Description
helpAt: put:	Associates a control identifier with a help topic identifier.
helpFile: topic:	Registers a help filename and a default topic identifier to call when help is invoked. The method also finalizes the help structure, which is why it should be called as the last method.
helpFile:contextHelp:topic:	Same as above, but allows the caller to specify a different help file for context-sensitive help.

Using HTML Help and WinHelp

Smalltalk MT supports both the newer HTML Help and the traditional WinHelp format. The API actually used depends on the help file extension; it is `.CHM` for HTML Help files and `.HLP` for WinHelp files. The help identifiers used by an application are mapped to topics using a [MAP] section defined in the Help project.

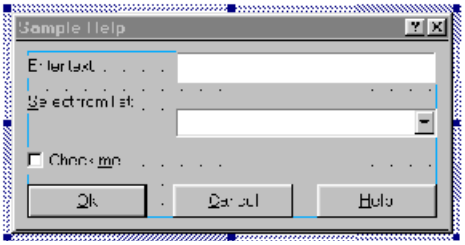
HTML Help uses the ActiveX control `HHCTRL.OCX` and runs in the same process as the caller, unlike WinHelp, which runs in a separate process. The framework uses HTML Help constants to identify the help actions to carry out. When using WinHelp, the constants are translated to their equivalent WinHelp constants.

Examples

Creating the dialog template

The sample code is based on the following dialog:

Figure 4-2 Sample Help Dialog



The dialog has the Context Help style to enable the help button in the title bar. All controls have default identifiers, except the *Help* button, which has the `ID_CONTEXT_HELP` identifier. There is no need to define a specific handler for the help button because the framework handles this command.

Creating the help file

Please refer to the Win32 SDK for information on how to create help files. You must also create an include file that associates help topic identifier names with their respective values, such as below:

```
#define IDH_INTRODUCTION  5000
#define IDH_CONTENTS      5001
#define IDH_TOPIC1        5002
#define IDH_TOPIC2        5003
#define IDH_TOPIC3        5004
```

Declaring the help handlers

You must first install the help identifiers as a pool dictionary and add it to the pools of the sample dialog. Finally, compile a class method `initHelpHandlers` as below:

```
initHelpHandlers
"
Private - Initializes help handlers.
"
self
    helpAt: IDC_EDIT  put: IDH_TOPIC1;
    helpAt: IDC_COMBO1  put: IDH_TOPIC2;
    helpAt: IDC_CHECK1  put: IDH_TOPIC3;
    " Topic in Help File "
    helpFile: 'help_sample.hlp' topic: IDH_CONTENTS
```

After initializing the dialog, you can open it and test the context-sensitive help.

Instance-based Help

We will re-use the help file and topics above to add help topics to a prompter dialog:

```
Prompter new
    helpAt: IDC_EDIT1 put: "IDH_TOPIC1" 5002;
    helpFile: 'Help_Sample.hlp' topic: "IDH_CONTENTS" 5001;
    title: 'Help Sample';
    text: 'default text';
    openOn: 0
```

Process Classes

This section examines process classes used by an application.

ApplicationProcess

Class **ApplicationProcess** implements the process startup and shutdown code.

Process entry points

A process entry point is the function called by the operating system when the image has been loaded. Each executable image has exactly one entry point.

The entry point is implemented in the class method `_main`, which initializes the Smalltalk process. In particular, it creates an instance of **ApplicationProcess** and assigns it to the global variable **Processor**. The startup code then calls the instance method `systemStartup` in **ApplicationProcess**. This method wraps a call to `winMain:with:with:with:` into an exception handler, which becomes the top-level handler for the process. The default implementation calls the operating system function UnhandledExceptionFilter:.

The method `winMain:with:with:with:` must be re-implemented when generating an executable, as explained in *Generating Executables* on page 350. In the development image, it just forwards the message to **DevelopmentEnvironment**, and it should not be modified while in the development image.

Process exit

There are several ways for an executable to exit a process.

Exiting normally

An application can exit normally by returning control to the caller.

close:

`ApplicationProcess>>close:` safely shuts down the garbage collector before evaluating a specified block. The block has the opportunity to perform cleanup operations after the garbage collector is halted. Typically, the block is used to close memory-mapped files safely.

A process that employs memory-mapped files in read-write mode must normally unload the files explicitly. This also entails removing all references to external objects. Otherwise, the garbage collector causes an access violation while it tries to scan an object that is no longer mapped into memory (notwithstanding the fact that some object pointers point to the data nirvana). Using `close:` bypasses this operation by stopping the garbage collector first. The code in the block can then safely unmap and close the memory-mapped files.

exitProcess:

This method ends the process and all its threads. It returns a specified 32-bit value to the parent process. If you need the functionality of the method #close: described above, you can call `close:` with a block that calls `exitProcess:`. Alternatively, you can also call the API `ExitProcess` directly from the block.

exitThread:

This method exits the current thread safely. If the thread is also the main thread, the whole process is also terminated, regardless of whether other threads are running.

DLL Exit and Unloading

The process that attaches to a Smalltalk DLL unloads it as well. The class method `dllEntryPoint:with:with:` automatically calls `unload` on **Processor** to safely unload the library. Unloading entails safely terminating internal Smalltalk threads and releasing resources.

Thread protocol

The following methods make up the thread protocol in **ApplicationProcess**.

Table 4-10 ApplicationProcess Thread Protocol

Message	Description
currentThread	Returns the current thread object. This is a subclass of WinThread.
sleep:	Suspends the execution of the current thread for the specified amount of time.
terminateThread:exitCode:	Terminates a thread identified by a thread handle safely. This method can be used to terminate another thread.

Note The method terminateThread:exitCode: ultimately calls the function TerminateThread. This function does not free a thread's stack, so it should not be used on a regular basis.

Process attributes

Processor is the sole instance of **ApplicationProcess**. The table below lists process-specific attributes.

Table 4-11 Processor Attributes Protocol

Message	Description
getCommandArguments	Returns an Array of command line arguments.
getMessages	Returns the module that contains the error messages. The module is loaded on demand.
getStartupDirectory	Returns the startup directory. It is extracted from the command line arguments.
getStartupPath	Returns the startup path. This is also the first command line argument.
hInstance	Returns the instance handle of the process. This is an HModule instance whose value is the loading address of the image.

Debug protocol

Debug protocol methods print to an externally attached debugger or a program that traces `OutputDebugString` calls, for example Dbmon on Windows NT. Each line is preceded by a prefix that includes the module name and the thread id for further identification.

Table 4-12 Processor Debug Protocol

Message	Description
outputDebugLine:[_with:]	Prints the prefix, a specified text, and a carriage return. This is a variable argument method that accepts an arbitrary number of parameters and formats them according to a format string.
outputDebugPrefix	Prints a prefix string that identifies the process and the thread.
outputDebugString:	Prints a specified text.

WinApplication

WinApplication manages the message loop. There can only be one instance of WinApplication per thread. However, a process can spawn multiple threads and create a WinApplication instance for each GUI thread.

The WinApplication instance for a GUI thread is referenced by the thread-local variable `thisApplication`. By default, this instance is also the owner of all top-level windows and receives all unhandled events.

For example, the development environment creates a separate thread when debugging and creates an instance of **Debugger** to manage the debugging thread.

Using WinApplication

WinApplication implements methods to load an accelerator table, specify a resource module, and set the main window and so forth. In simple cases, it is therefore sufficient to instantiate the existing class and set its attributes in the `winMain` method.

If your program uses complex initialization methods, you may consider subclassing **WinApplication**. You must create a subclass in the following cases:

- WinApplication has event handlers.
- WinApplication implements a custom exception handler.
- WinApplication uses a custom message loop.
- WinApplication uses a registration key.

The application class for the development environment is **DevelopmentEnvironment**.

WinApplication Attrinutes

The following table lists common attributes of **WinApplication**.

Table 4-13 WinApplication Attributes

Variable	Description
m_hAccelTable	An accelerator table shared by all frame windows created by the current thread.
m_hInstance	Default resource module.
m_mainWindow	Main window. See also ***Running a GUI Application*** and ***Running a Dialog Application*** on page 353.

WinApplication Methods

The following tables list methods of **WinApplication** that you may wish to reimplement in subclasses.

Table 4-14 WinApplication Initialization Methods

Message	Description
exitApplication	Exits this application. This method is reimplemented by subclasses to perform cleanup tasks, such as unregistering window classes.
exitInstance	Processes a request to terminate the application. The message loop is exited by posting a `WM_QUIT` Windows message.
initApplication	Performs one-time initializations, such as registering window classes.
initInstance:	Initializes the WinApplication instance. Subclasses usually reimplement this method to open the main window. The parameter to this method is the initial visibility, a SW_XXX constant (see also the Win32 SDK documentation).

Table 4-15 Miscellaneous WinApplication Methods

Message	Description
unhandledException:	Default exception handler.
regKey	Answers the registry subkey under which data for the application is stored.
splashWindow	Displays a copyright logo at startup.

Testing a WinApplication

You must test the application class in a separate thread. The code below creates an application in a separate thread and opens a **TextWindow** as main window.

The argument to the `run:` method is a block that evaluates to the main window. WinApplication installs a handler for the `#destroy` event of this window and calls `exitInstance` when the event occurs. The default implementation of exitInstance posts the `WM_QUIT` message, which terminates the message loop and returns from `run:`. The block returns normally and thus terminates the thread.

```
[
    | app |
    app := WinApplication new.
    app setModule: (HModule loadLibrary: 'stdev.dll').
    app initApplication initInstance: SW_SHOW.
    app run: [
        TextWindow new open
    ]
] fork
```

Note The run: block must evaluate to a valid window, otherwise the #destroy event never occurs and the loop run forever. The window creation methods raise an exception if the window could not be created.

Graphics Classes

Overview

Graphics use the GDI interface. Graphic output goes to a **DeviceContext** and uses GDI objects that must first be selected into the device context. When the drawing is finished, the selected objects must be de-selected.

Device contexts

There are device contexts for various output devices. The most common are the screen and printers. To obtain a device context for drawing onto a window client area, an application calls `getDC` on the window. For a regular window, the return value is a device context handle, which can be converted to a **DeviceContext** instance using `DeviceContext>>fromHandle:`.

Specialized windows such as **GraphicsWindow** and **GraphicView** directly return a **DeviceContext** object.

Creating and selecting GDI objects

An application must first select graphic objects into the device context, using `selectObject:`. Graphic objects encompass bitmaps, brushes, pens, fonts as well as regions. **Bitmap**, **Brush** and **Font** have class methods for creating instances. **DeviceContext** has also methods for creating common objects on the fly. For example, `selectSolidBrush:` creates and selects a solid brush with a specified color. It is

also possible to select system objects such as a system color brush (in `selectSysColorBrush`).

Drawing operations

DeviceContext has methods for text output, drawing lines and curves as well as filled shapes. It is also possible to call graphic APIs directly with a **DeviceContext** instance.

Deleting GDI objects

An import aspect of drawing is to deselect objects and delete them when they are no longer in use. Failure to do so usually leads to memory errors down the line.

It is relatively easy to spot GDI leaks. The code must be compiled with the GDI diagnostic messages turned on (in the image properties) and the code be run under a debugging version of the operating system.

CHAPTER 5 Window Classes

This chapter examines the different window classes in the system. This includes **Window**, overlapped windows (**FrameWindow** and **DialogBox**), Menus, Controls, and more specialized windows such as MDI, Property Sheets, document classes, as well as predefined dialogs. The common controls are also reviewed in detail.

Window

Class **Window** implements the window procedure that processes Windows messages. In addition, Window exposes methods that return or alter the state of the window and methods that work on child windows of the receiver window.

An application can use **Window** or one of the derived classes to manipulate windows. Class **Window** and its subclasses shield the application from the window procedure and provide default behavior.

A Smalltalk **Window** instance stores the window handle of the associated window. Incoming messages to the window procedure are routed to the appropriate method if such a method exists, otherwise they are passed to a default handler. A subclass of **Window** that wishes to process a `WM_XXX` Windows message must implement the method `WM_XXX:with:`.

An application uses **FrameWindow** or one of its subclasses to register callbacks that are invoked when a certain command is sent to the window.

Window Class Attributes

Window class attributes are used by the window registration methods. The attributes are used by the default window procedure (implemented by the operating system).

Table 5-1 Window Class Attributes

Message	Description
windowClassBrush	Returns the brush used to paint the background.
windowClassCursor	Returns the cursor displayed when the mouse is over the window.
windowClassIcon	Returns the window icon identifier in the resource file. The registration method also tries to load the small icon that corresponds to the identifier.
windowClassName	The Window class name that identifies the class for Windows. The name must be different each time you redefine any of the attributes.
windowClassStyle	A combination of window class styles. Please refer to the Win32 documentation for more information about class styles.

Example

To display a window with a dialog background color, implement the following methods:

```
windowClassName
    ^'AppWindow'


windowClassBrush
    "
    Answers the class brush for this window class.
    Return Value:
        A system window color constant.
    "
    ^COLOR_BTNFACE + 1
```

The returned brush is actually an index into the Windows system palette, and `COLOR_BTNFACE` is the index of the button background.

Registering a Window Class

You must register a window class if you wish to use an icon, a background brush, a cursor or a particular class style for a window. For a detailed discussion of window attributes, see the previous paragraph.

To register a window in Smalltalk, proceed as follows:

- Create a subclass of **Window**
- You must implement a class method `windowClassName` to return a name for registering the class. Implement a method for each attribute you wish to define (see below).
- Call the class method `registerClass` or any of the methods below before you use the window.

Window class registration methods

registerClass

This method registers the window class with the attributes returned by the `windowClassXXX` methods. The icon and cursor identifiers are ignored.

registerClass:

This method loads the icon and cursor resources from a specified module and registers the window class using those attributes.

registerClass:style:icon:cursor:

This method registers the window class using a specified class name, style, icon and cursor.

registerClass:style:icon:iconSm:cursor:hbrBackground:

This method performs the actual registration, using the specified parameters. An application can use this method to override all class attribute methods.

unregisterClass and unregisterClass:

These methods unregister the receiver's window class. Window class registration only succeeds when the window class is not registered, you must therefore use `unregisterClass` to unregister the class before you can register it again with new attributes.

Note When writing Dlls or ActiveX components, it is important to unregister all window classes that have been registered when the module is unloaded. A Dll may be loaded and unloaded several timedduring the lifetime of the process, and there is no guarantee that the Dll is loaded at the same base address. Because Window registration uses the callback address of the window procedure, failure to unregister Window classes may cause unexpected behavior or access

violations when a Dll is reloaded at a different base address.

Creating a Window

Window creation is done in the method `createWindow:[...]`. The method takes many parameters and is used indirectly through creation methods in **FrameWindow**, **Control** or **Menu**.

It is often necessary to create a Smalltalk **Window** object that encapsulates a given window. The class method `fromHandle:` returns a new instance for this purpose.

Getting the Window Handle

You can retrieve the Smalltalk window that is associated with a given handle. The method `Window>>getWindowFromHandle:` returns the Smalltalk window object, or **nil** if there is no such window.

The Smalltalk object is associated with the window property `'STOBJ'`, which makes it also accessible to non-Smalltalk code.

Enumerating and finding Windows

It is sometimes necessary to enumerate all windows that have been created by the current thread, child windows of a given window, or MDI child windows. All enumeration messages are based on Win32 functions, which ensure that the windows that are passed to the enumerator are in a consistent state. For this reason, it is generally better to use enumeration methods rather than storing references in instance variables.

Each enumeration block takes a single argument, a temporary window that encapsulates the window being enumerated. To retrieve the Smalltalk object that is associated with the window, the enumerator may use `Window>>getWindowFromHandle:`.

Table 5-2 Window Enumeration Messages

Message	Description
enumThreadWindows:	Evaluates a block with each top-level window created by the current thread.
enumChildWindows:	Evaluates a block with each child window of the receiver window.
enumMDIChildWindows	Evaluates a block with each MDI child window of the receiver window.

Note There is no guarantee that a Smalltalk object exists for the window, so the caller should check the return value against **nil**.

Window Items

Windows respond to a set of polymorphic messages that act upon items that belong to the receiver window. The items can be child controls, menu items or toolbar buttons.

Table 5-3 Window Item Messages

Message	Description
checkItem: first: last:	Checks one item in a range of consecutive items and unchecks the others.
getItemFont, setItemFont	Retrieves / sets the font of a child control.
getItemInt, setItemInt	Retrieves / sets an integer value.
setItemState, getItemState	Retrieves / sets the state of a child control.
setItemText, getItemText	Retrieves / sets the text displayed by a child control.
showItem	Sets the visibility state of a child control.

An item’s state is defined by a combination of state flags.

Table 5-4 Window Item States

Message	Description
STATE_CHECKED	Checks an item.
STATE_ENABLED	Enables an item.
STATE_FOCUS	Sets the focus to an item.
STATE_HILITE	Selects an item, or the text displayed by an item.

For regular windows, `STATE_HILITE` selects the text in an edit control and emits a warning sound. This state is often used when the user enters text that fails a validation check.

Note Refer to the implementation for additional messages that can be used with child controls.

Window Properties

Each **Window** instance can have an associated property dictionary. The dictionary can store arbitrary objects and keys.

Table 5-5 Window Properties

Notification	Return Value
properties	Returns the properties, a **MappingTable**.
propertyAt:	Answers the object stored at a given key, if any, otherwise returns **nil**.
propertyAt: put:	Stores an object at a specified key.
removeProperty:	Removes a property.

Note The properties are Smalltalk properties, and have nothing to do with Win32 properties. Win32 offers functions to store key – value pairs under a window handle. The functionality is similar, except that the keys must be strings and the values 32-bit values.

Window Subclassing

Window subclassing in the sense of Win32 consists of replacing the window procedure of a given window with another procedure. The new procedure generally passes any messages it does not handle to the old procedure. This allows a program to filter messages it wishes to process, therefore altering the behavior of the original (subclassed) window.

An application uses `subclassWindow` to route the window procedure through the receiver. For example, subclassing a **ListBox** control has the effect of sending all Windows messages to the object.

Overlapped Windows

Overview

This section discusses top-level windows. There are two basic types of top-level windows: dialog boxes (**DialogBox** instances) and regular frame windows (**FrameWindow** instances). **DialogBox** inherits from **FrameWindow**, meaning that dialog boxes have the same event interface as **FrameWindow** instances. The differences are mostly technical, since it is possible to program a regular window so that it behaves like a dialog. However, the amount of code required to mimic the behavior of dialogs is quite large.

The following paragraphs discuss overlapped (frame) windows. Note that many messages can be used with non-overlapped windows as well.

Initialization and Release Messages

FrameWindow instances may re-implement the following messages:

Table 5-6 Window Init / Release Messages

Message	Description
initWindow	Called after the window has been created but before it becomes visible.
closeWindow	Called before the window is closed. A value of `NULL` (`FALSE`) closes the window, `TRUE` prevents it from closing.
destroyWindow	Called during the processing of the `WM_DESTROY` message to perform cleanup tasks.

initPosition	Answers the initial position and / or extent of the window. The return value is **nil** (to use default positioning), a point that specifies the extent, or a rectangle. The value can also be set using `initPosition:`.

FrameWindow

Class **FrameWindow** manages a top-level window. It implements an instance-based event handler map, and handles various notification messages from the system, the menu and child controls. Furthermore, it optionally handles reframing of child windows by the means of a **FrameMatrix** object. Supplemental windows such as status bars and tool bars are supported in a transparent manner.

Although **FrameWindows** are generally overlapped windows (i.e., the `WS_OVERLAPPED` style is set), it is also possible to create a **FrameWindow** that has the `WS_CHILD` style. **ContainerWindow** is an example of a **FrameWindow** that is also a child window.

Creating a FrameWindow

This paragraph discusses creating a **FrameWindow** that is not a dialog. Please refer to *Creating a Dialog* on page 245 for information on how to create dialog boxes.

create

Method `create` creates a **FrameWindow** with default attributes. It first calls `initMenu` to give the window an opportunity to load a menu. See also `open` for more information.

createWindow:title:exStyle:style:parent:menu:

This method uses `initPosition` to retrieve the window's position and dimension. The return value from `initPosition` can be either a point that specifies the window extent or a rectangle with the exact position on the screen. Unspecified values are replaced with `CW_USEDEFAULT`, which lets Windows assign a default value based on the desktop size and the position of other windows.

createWindow:title:exStyle:style:x:y:cx:cy:parent:menu:

This is the generic window creation method inherited from **Window**. It gives the caller full control over all window attributes such as position, title, menu, and styles.

open

Use `open` to create and show a window using default attributes. The difference between the methods `open` and `create` is that the former calls `showWindow`: with the `SW_SHOW` parameter to make the window visible, followed by `updateWindow` to update the client area of the window. Therefore, calling `open` always results in a visible, updated window when the method returns.

In cases where the window is modified right after it opens, calling `create` avoids the flicker that occurs when the window is updated twice; once in `open` and the second time when it is modified. The code below demonstrates the difference:

```
CHB new open
    selectClass: Menu selectMethod: 'getItemCount'
```

and

```
CHB new create
    selectClass: Menu selectMethod: 'getItemCount'
```

In the first example, the CHB first opens on **Object** (the default), updates the screen, and then immediately selects `Menu>>getItemCount`.

The second example directly displays `Menu>>getItemCount`.

Accessing child controls

FrameWindow also implements methods to set and retrieve the contents of common controls. For example, to retrieve the child listbox using the id `IDC_LIST1`, an application sends the following message to the **FrameWindow** instance:

```
listboxAt: IDC_LIST1
```

This creates and returns a listbox object on demand that encapsulates the listbox window.

There are a number of messages that manipulate child controls of a window. These are defined in **Window**. **Window** also implements two specialized messages that set the contents of a combo box or list box to an array of strings. These messages are marginally faster but more limited than `setContents:` in **ComboBox** and **ListBox**.

In some cases, it is necessary to access a child control directly. A **FrameWindow** can use `childAt:`, which returns the Smalltalk object that encapsulates a given control. This object has been specified at creation time (in an `initWindow` method).

A dialog box does not create child controls explicitly, so using `childAt:` is not appropriate. There are messages that create a control of a specified type on the fly, given a control identifier. Once created, the control remains referenced and the same object is returned on the next call.

Table 5-7 Accessing Child Windows

Message	Description
childAt:	Returns the Smalltalk object that encapsulates a control, or **nil** if none is defined.
childAt:class:	If no Smalltalk object exists for the control, creates a control of the specified class.
comboboxAt:	Same as above; instantiates a ComboBox.
listboxAt:	Same as above; instantiates a ListBox.

Owner

A frame window forwards unprocessed events to its owner. The owner is strictly a Smalltalk entity, though it may overlap with the *owner* of a dialog box (by default, the dialog box owner is *not* assigned to the owner instance variable). The owner of a dialog box is, in Win32 terminology, the window that becomes disabled when the dialog box is opened.

Win32 Parent or Owner

The Windows parent or owner is the *parent* argument to `createWindow:[...]` methods (which calls `CreateWindow`). There are two cases:

- If the window being created is a child window (i.e., the `WS_CHILD` style is set), the argument must specify the parent window. The child window is confined to the client area of the parent.
- If the window is an overlapped window, the argument specifies the optional Win32 owner. The window always stays on top of its owner, and closing the owner also closes the owned window.

Example:

Evaluate the code below to create a **TextWindow** owned by Transcript.

```
| textWindow |
textWindow := TextWindow new.
textWindow createWindow: TextWindow windowClassName
    title: 'Owned text window'
    exStyle: textWindow windowExStyle
    style: textWindow windowStyle
    parent: Transcript
    menu: NULL
```

DialogBox

Class **DialogBox** is a subclass of **FrameWindow** that encapsulates a Win32 Dialog Box. Although an application can implement a subclass of **DialogBox** to manage a dialog, simple dialogs can often be created on the fly by specifying the appropriate handlers.

Dialog Owners

The owner of a dialog box is, in Win32 terminology, the window that becomes disabled when the dialog box is opened. In some cases, it may be identical to the Smalltalk owner, to which events are forwarded. However, in the general case, a dialog has no Smalltalk owner but a Win32 owner, which is specified in the dialog creation messages.

Creating a Dialog

A dialog box can operate in one of two modes; **modal** and **modeless**. A modal dialog prevents the owner of the dialog, normally a **FrameWindow**, from processing user interactions. The user is therefore forced to close the modal dialog before continuing his or her work. A modeless dialog allows the user to return to the previous window without closing the dialog.

A dialog often implements `openOn:`, which takes an owner window and opens the dialog using one of the methods below. The parameter can also be `NULL` if the dialog has no owner window.

Example:

```
DatePrompter new openOn: NULL
```

opens a (modal) date prompter dialog. Since the owner window is `NULL`, no window is disabled while the dialog is open.

```
DatePrompter new openOn: Transcript
```

opens the same dialog and disables the Transcript window while it is open.

Since the dialog owner parameter can be a window handle, it is possible to specify an arbitrary owner window, and in particular a non-Smalltalk window such as a common dialog.

Modal dialog box creation

A modal dialog is created using `dialogBox:template:owner:`, specifying a module handle, a resource identifier and the owner window handle. The message `dialogBoxIndirect:owner:` creates the dialog box from an in-memory template.

Modeless dialog box creation

The messages `createDialog:template:owner:` and `createDialogIndirect:owner:` open a modeless dialog box.

Dialog box template

A dialog box always requires a dialog box template. The template can be constructed using an in-memory template or it can be loaded from an external resource.

The interface builder can generate both formats. An in-memory template is more flexible since it does not require building a resource DLL. A dialog template requires fewer resources and is easily localizable.

Processing dialog box messages

The message processing of dialog boxes differs from the one of regular windows because each dialog has its own dialog box procedure. The dialog box procedure is called by the default window procedure for dialog boxes, which is implemented by the Windows subsystem. This means that a Windows message is first sent to the window procedure for dialogs, which in turn calls the dialog procedure.

The Smalltalk implementation hides these details and dispatches Windows messages as usual. However, there are two differences to keep in mind:

- The default procedure `DialogBox>>defWindowProc:with:with:` does nothing.

- A message handler for a Windows message (`WM_XXX`) returns nonzero if it processes the message and zero if it does not. This value is returned to the dialog procedure implemented by the operating system; it is not possible to return a value directly to the sender of the message. If this is required, the dialog must either implement its own dialog procedure or be subclassed.

The Dialog Box procedure

The dialog procedure is similar to the `WndProc` procedure in **Window** class, except that it is instance based and re-implementing it does not require Windows registration. A callback block that is passed to the dialog creation function implements the window procedure:

```
m_dlgProc := [ :hWnd :msg :wparam :lparam |     " the dialog procedure "
    self wndProc: hWnd with: msg with: wparam with: lparam
].
```

Implementing Dialog procedures

The default dialog procedure is implemented in `wndProc:with:with:with:`. A subclass can re-implement this method to process messages before or after they are processed by the dialog procedure.

The method `defWindowProc:with:with:` overrides the default window procedure, which normally calls the `DefWindowProc` API. In a dialog box, this method does nothing and is only called by Windows message handlers (`WM_XXX` methods) that call the default procedure.

Implementing validation

A typical validation routine retrieves the contents of a form and sets the focus to the (first) item that fails the validation test. Validation can be implemented in three ways:

- Not allowing invalid entries in the first place. For example, a numerical edit field only accepts numerical values, a date / time picker selects only valid date and time values, a spin button restricts the entry to a range and so forth.
- Performing on-the-fly validation, for example on each `EN_CHANGE` notification message from an edit field or on a kill focus event.
- Validating in the `closeWindow` method. This method called when the frame window receives a `WM_CLOSE` message, and the implementor returns zero to close the window and a non-zero value to prevent it from being closed.

An application may apply a combination of the validation techniques above. The validation routine should also give a visual indication and highlight the item that failed

the test, in addition to a message box for non-trivial errors. The message `setItemState:state:stateMask:` allows the caller to set the focus and, in the case of edit fields, highlight the text. For example, to set the focus and select the edit contents:

```
self setItemState: IDC_EDIT1
    state: STATE_FOCUS|STATE_HILITE
    stateMask: STATE_FOCUS|STATE_HILITE
```

Closing a dialog box

The method `closeWith:` closes and destroys both modal and modeless dialog boxes. In the case of a modal dialog box, the argument to this method is returned to the caller of `dialogBox:`.

DialogBox installs handlers for the `IDOK` and `IDCANCEL` events, which are processed in the *onOk* and *onCancel* methods. The default implementations close the dialog box.

Automatic Data Mapping

DialogBox has methods that transfer the data displayed by the dialog box to and from a **MappingTable** that associates each child identifier with the value

For dialogs that do not require data validation or complex user interaction, using data mapping is a very efficient and easy to use method to present and retrieve data.

The methods `setDlgData:` and `getDlgData:` respectively set and retrieve dialog data all at once with the following conventions:

Table 5-8 Input Data Mapping

Control	Data	Action
Edit	String	Sets the edit text.
Check box	Boolean	**true** checks, **false** unchecks.
Radio button	Boolean	**true** checks, **false** unchecks.
Listbox	Collection of strings.	Sets the list items.
Combobox	Collection of strings (see action)	Sets the list items. If the first element is an integer, it is interpreted as the zero-based index to select, and the item strings follow this element.

Group of Radio buttons.	Integer	Interprets the data as the zero-based index of the button to check. Other radio buttons in this group are unchecked. See also the remarks below.

Remarks

The implementation handles groups of radio buttons. The first button in the group is associated with the zero-based index of the checked button.

To take advantage of this feature, the radio buttons in the group must have consecutive identifiers. For example, given a group of four radio buttons that start at `IDC_RADIO1`, the base identifier `IDC_RADIO1` is associated with the zero-based index of the checked button. To check the third button, you would associate `IDC_RADIO1` with 2.

Child dialog boxes

Child dialog boxes are dialogs that do not have the `WS_POPUP` style bit set. Because the framework sends `preTranslateMessage:` only to top-level windows, child dialogs do not automatically receive this message.

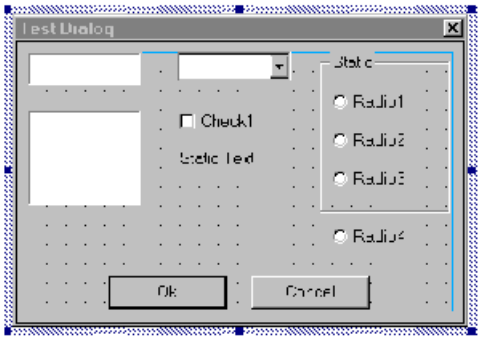
In order to enable dialog behavior (tab key processing and so forth), you must register the child dialog with the parent, using `addModelessDialog:`. **FrameWindow** delegates the `preTranslateMessage:` to any registered child dialogs. Use `removeModelessDialog:` to remove the child dialog when it closes (you do not need to do this if the parent window closes also).

Example 1: Transferring data

The following example creates a dialog box that displays some simple controls and populates them with data.

Creating the dialog

Figure 5-1 Sample Dialog Layout



In the interface builder, create a new dialog as above.

Note By using default child identifiers, you avoid creating a new pool dictionary for the dialog. The first radio button of each group must have the group style, so `Radio1` and `Radio4` must both have this style. Use the identifiers `IDOK` and `IDCANCEL` for the Ok and Cancel pushbuttons.

Testing the dialog

Save the dialog as **DemoDialog** and test it using:

```
DemoDialog new openOn: NULL
```

Passing `NULL` instead of an owner window allows you to test a modal dialog without blocking an owner window.

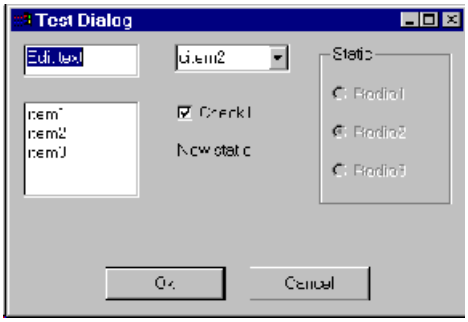
Displaying data

Evaluate the code below to set the dialog data:

```
| input |
input := MappingTable new.
input
    at: IDC_EDIT1 put: 'Edit text';
    at: IDC_LIST1 put: #('item1' 'item2' 'item3');
    at: IDC_COMBO1 put: #(1 'citem1' 'citem2' 'citem3');
    at: IDC_CHECK1 put: true;
    at: IDC_STATIC1 put: 'New static text';
    at: IDC_RADIO1 put: 1;
    at: IDC_RADIO4 put: true.
DemoDialog new
    setDlgData: input;
    openOn: 0
```

The dialog displays the data as in the figure below:

Figure 5-2 Sample Dialog at Runtime



Retrieving data

Retrieving the data is slightly more complicated because it must be retrieved when the user presses the *Ok* button, and before the dialog is destroyed. The code below installs a handler for the `IDOK` event.

```
| input output dlg  |
input := MappingTable new.
input
at: IDC_EDIT1 put: 'Edit text';
    at: IDC_LIST1 put: #('item1' 'item2' 'item3');
    at: IDC_COMBO1 put: #(1 'citem1' 'citem2' 'citem3');
    at: IDC_CHECK1 put: true;
    at: IDC_STATIC1 put: 'New static text';
    at: IDC_RADIO1 put: 1;
    at: IDC_RADIO4 put: true.
output := nil.
(dlg := DemoDialog new)
    setDlgData: input;
    when: IDOK perform: [
    output := MappingTable new.
    dlg getDlgData: output
];
    openOn: 0.
output
```

Remarks:

Given that the `IDOK` handler must also close the dialog, there are several ways to write the handler block. The code given here evaluates to the output **MappingTable**, meaning that the event framework will continue the event handler chain and evaluate the static handler defined for this event, which is the `onOk` method in **DialogBox**.

An alternative is to close the dialog in the handler:

```
    when: IDOK perform: [
        output := MappingTable new.
        dlg getDlgData: output.
        dlg close.
        0
    ];
```

An integer return value stops event processing and the control of flow returns. In this particular case, however, the return value is ignored because the dialog is closed in the preceding line.

Creating a resource DLL

1. Open the GUI builder, select *File|New...* and click on *Resource script*. Enter a new folder name (for example dialog) in the Folder dialog box and click on Ok.
2. Open the previously defined dialog in the interface builder and save the dialog as a resource (*File|Save As...* and click on Resource only, enter for example `DEMODIALOG.RC` as the file name under which to save).
3. Edit the resource script (`DIALOG.RC`) and add the line:

```
RCINCLUDE demodialog.rc
```

You can now compile the resource-only DLL by invoking the *nmake* utility from the `DIALOG/RES` subdirectory.

The previous code becomes:

```
| input lib |
input := MappingTable new.
input
    at: IDC_EDIT1 put: 'Edit text';
    at: IDC_LIST1 put: #('item1' 'item2' 'item3');
    at: IDC_COMBO1 put: #(1 'citem1' 'citem2' 'citem3');
    at: IDC_CHECK1 put: true;
    at: IDC_STATIC1 put: 'New static text';
    at: IDC_RADIO1 put: 1;
    at: IDC_RADIO4 put: true.
lib := HModule loadLibrary: 'dialog.dll'.
DialogBox new
    setDlgData: input;
    dialogBox: lib template: IDD_DIALOG1 owner: 0.
lib close.
```

Example 2: Validation and event handling

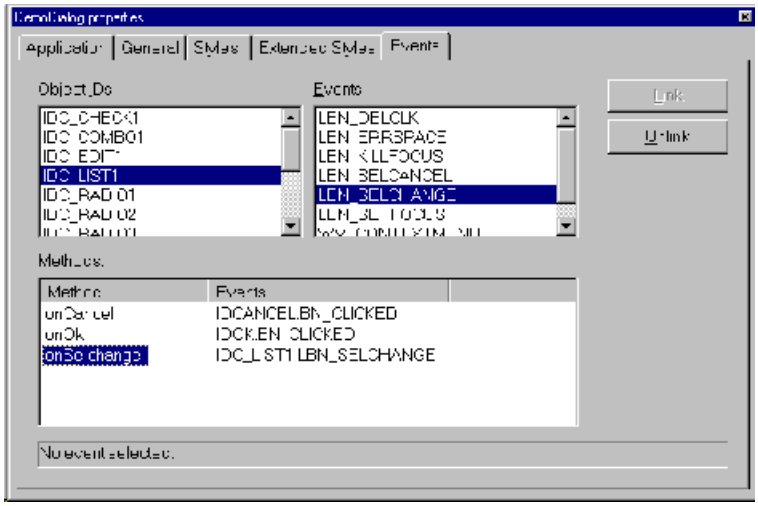
In this example, we will modify the dialog from above to implement simple validation and event handling according to the following criteria:

- The text field `IDC_EDIT1` must contain exactly 7 characters.
- The radio button group is only enabled when the list box (`IDC_LIST1`) selection is `'item2'`.
- The radio button Radio4 is hidden.

Defining event handlers

We have to define an event handler for the `LBN_SELCHANGE` in `IDC_LIST1`. The handler has to enable or disable the group of radio buttons, depending on whether the selected item is `'item2'`.

Figure 5-3 Sample Dialog Events



The implementation of `onSelchange` looks as follows:

```
onSelchange
"
Event handler for event 'LBN_SELCHANGE' in IDC_LIST1.
Return Value:
    An integer to return to the caller, nil to continue
    event handling.
"
| fEnabled |
fEnabled := (self listboxAt: IDC_LIST1) getSelItem == 1.
self enableItem: IDC_RADIO1 state: fEnabled.
self enableItem: IDC_RADIO2 state: fEnabled.
self enableItem: IDC_RADIO3 state: fEnabled.
^NULL
```

Modifying the initWindow method

The `initWindow` method is called when the dialog is created but not yet displayed. The code below limits the amount of text the user can enter into the edit field, hides the fourth radio button and calls `onSelchange` to disable the group of radio button.

```
initWindow
"
Private - Initializes the window.
"
self limitItemText: IDC_EDIT1 value: 7.
self showItem: IDC_RADIO4 state: false.
self onSelchange.
```

Implementing validation

Validation is performed in the predefined `onOk` handler. If validation fails, the method sets the focus to the edit field, highlights the text and beeps. This is done by specifying `STATE_FOCUS` and `STATE_HILITE` in the `setItemState` message.

```
onOk
"
Private - Performs validation.
"
(self getItemText: IDC_EDIT1) size ~~ 7 ifTrue: [
    " validation failed "
    self setItemState: IDC_EDIT1
        state: STATE_FOCUS|STATE_HILITE
        stateMask: STATE_FOCUS|STATE_HILITE.
    ^NULL
].
self close
```

Common Dialog Boxes

Common Dialog Boxes are system-defined dialogs for performing common tasks, such as opening a file or choosing a printer. Common Dialog Boxes are subclasses of **CommonDialog**, which is itself a subclass of **DialogBox**.

CommonDialog

CommonDialog is the abstract superclass of all common dialog boxes. Common dialogs use a structure that defines attributes of the dialog and returns data when the dialog terminates. The structure is allocated automatically, and the caller can set and retrieve the flags field (the values are specific to the common dialog). It is also possible to set and retrieve the whole structure at once using `getStruct` and `setStruct:`.

If a common dialog is opened in response to a user action or if the frame window should be disabled, the application must also set the parent with the `parent:` message. The effect is that the dialog is modal (i.e., the parent is disabled) and centered on the parent window.

ColorDialog

ColorDialog returns an RGB color value selected by the user. Using **ColorDialog** is straightforward:

```
ColorDialog new open
```

The return value is the color selected by the user or **nil** if the dialog was cancelled.

FileDialog

FileDialog is the most often used common dialog. It can be used to open or save files.

The code fragment below opens a **FileDialog**:

```
szFile := FileDialog new
    filters: 'Method Files (*.sm)\0*.sm\0All Files (*.*)\0*.*\0';
    title: 'File in';
    getOpenFileName.
```

defExt:

Sets the default extension of the dialog, for example '.txt'. The default extension is automatically appended to the filename if the user does not specify an extension.

fileName:

Sets the default file name, for example `'test.txt'`.

filters:

Sets a multi string that specifies the filters. A multi string is as String that contains substrings, which are delimited by null characters. The entire string is terminated by a double null. For example, the following is a literal multi string:

```
'Method Files (*.sm)\0*.sm\0All Files (*.*)\0*.*\0';
```

Note that a literal string is always null terminated, so specifying a null with the escape sequence ensures that the string is terminated with a double zero.

An alternate way is to create a **Stream** and append substrings using `nextPutAll:`, separated with `nextPut: 0`.

initialDir:

Sets the initial directory of the file dialog.

title:

Sets the title (caption) of the dialog box.

getOpenFileName

Opens the *Open* file dialog.

getSaveFileName

Opens the *Save* file dialog.

FindReplaceDialog

FindReplaceDialog opens a find/replace dialog box. **Edit** calls this dialog automatically when the standard edit events (`ID_EDIT_FIND` or `ID_EDIT_REPLACE`) occur. If this is not desired, an application must use alternate command identifiers for these operations.

FontDialog

An application uses `setFont:` to set the initially selected font. The argument can be either a **LOGFONT** or a **Font** object.

Example:

```
FontDialog new
    setFont: Font defaultGUIFont;
    open
```

PageSetupDialog

PageSetupDialog lets the user define attributes (paper size, margins) of a printed page. The attributes are returned in a PAGESETUPDLG structure. The caller can specify flags that control the measurement units to use and other dialog options before the dialog is opened.

By calling the class method `getDefault`, it is also possible to retrieve the default attributes without opening the dialog.

PrintDialog

PrintDialog displays a print dialog. If successful, it returns a PRINTDLG structure. The structure defines print attributes such as the device context, the range of pages to print and so forth.

By calling the class method `getDefault`, it is also possible to retrieve the default attributes without opening the dialog.

Printing with PageSetupDialog and PrintDialog

The first step is to gather the page and printing information from the user, or using default values currently defined for the printer. The structure **PRINTDLG** contains the printer device context and **PAGESETUPDLG** contains the margins. The bounding rectangle can be calculated from the page dimensions and the margins.

Given the bounding rectangle, the printing code looks as follows:

```
hdc := PrinterDeviceContext value: printdlg hdc.

(hdc startDoc: szDocTitle) ifFalse: [^false].
rcPage := pagesetupdlg rc.
rcPage bottom: rcPage bottom negated.
rcPage top: rcPage top negated.

printdlg nFromPage to: printdlg nToPage do: [ :i |
    printdlg nCopies timesRepeat: [
    hdc startPage ifFalse: [
            " an error occurred. break out of outer loop "
            i := LOOPBREAK.
        ].
    « drawing code here «
    hdc endPage.
    ]
].
hdc endDoc.
```

The variable `rcPage` contains the boundaries in device units. The y-axis of printer devices is oriented differently, hence the negation of the top and bottom coordinates.

The loop prints as many copies and pages as specified in the **PRINTDLG** structure by the user.

A functional code fragment that uses default attributes for page and printing attributes looks as follows:

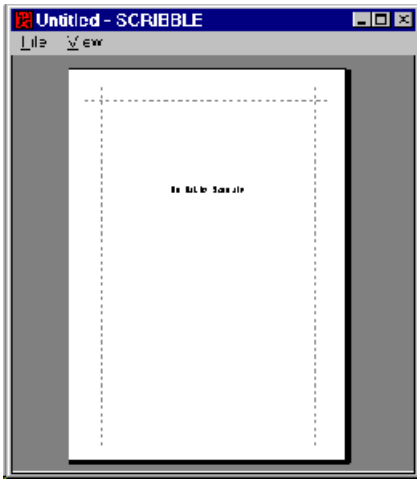
```
| hdc print_struct pagesetup_struct rcPrintPage  |
pagesetup_struct := PageSetupDialog getDefault.
print_struct := PrintDialog getDefault.
hdc := PrinterDeviceContext value: print_struct hDC.

(hdc startDoc: 'Printing Sample') ifFalse: [^false].
rcPrintPage := WinRectangle new left: pagesetup_struct rtMargin_left;
    top: pagesetup_struct rtMargin_top negated ;
    right: pagesetup_struct ptPaperSize_x - pagesetup_struct rtMargin_right;

    bottom: pagesetup_struct rtMargin_bottom - pagesetup_struct ptPaperSize_y .
hdc startPage ifTrue: [
    " drawing code here "
    hdc setMapMode: MM_HIMETRIC.
    hdc selectStockObject: BLACK_PEN.
    hdc selectStockObject: LTGRAY_BRUSH.
    hdc ellipse: rcPrintPage.
    hdc endPage.
].
hdc endDoc.
```

Using PrintPreview

PrintPreview is a Smalltalk control that displays a print preview and is able to print metafiles. The control covers the client area of the parent window. After the owner initializes the control with printing and page attributes, the control prepares a metafile and asks the owner to print on the metafile device context. The owner implements printing in a method `formatPage:on:rcFormat:`. The first parameter is the index of the page to print, the second specifies the device context, and the third is the formatting rectangle.

Finally, printing consists of playing the metafile on the printer device. A metafile is device-independant, so the printer output matches the screen display. The owner calls `filePrint:` on the **PrintPreview**, the parameter being the document title as it should appear in the print spooler.

Figure 5-4 Print Preview



MDI (Multiple Document Interface)

The MDI framework consists of two classes; **MDIFrameWindow** for the parent (frame) and **MDIChildWindow** for the child windows. An MDI child window looks almost like a regular frame window with a caption and system menu, except that it is confined to the client area of the parent frame (more precisely, the MDI client window, which is a special window created by the parent frame).

MDI windows (frame and children) have a special window procedure. You must register **MDIFrameWindow** and **MDIChildWindow** before you use these windows.

Creating MDI windows

MDI frame and child windows are created like regular frame windows. However, the internal implementation differs substantially from regular frame windows, a point to consider when re-implementing MDI creation messages.

Note	A newly created MDI child window does not necessarily receive the focus.

Using menus

The frame menu should display a standard Window menu that allows the user to cascade, tile and arrange the MDI child windows. **MDIFrameWindow** supplies handlers for these operations.

An application uses `setMenu:windowMenu:` to change the frame menu. The message replaces also the window menu, if not NULL.

Using per-child menus

It is sometimes desirable to display different menus for MDI child windows and the frame. Class **MDIChildWindowEx** is a specialized **MDIChildWindow** that manages a child menu, which replaces the frame menu whenever the child window is active.

SDIFrame and WinDocument

Overview

SDIFrame and **WinDocument** implement a document-view architecture. A **SDIFrame** window is a view on a document, usually a **WinDocument** instance. Operations such as opening a file, saving, printing and displaying are delegated to **WinDocument**.

WinDocument

WinDocument should be used whenever a document-centric architecture is required. In particular, this is true for OLE documents. Class **WinOleDocument** implements OLE specific behavior.

WinDocument is a subclass of **WinEventHandler**, meaning that it can also handle events.

SDIFrame

SDIFrame is a subclass of **FrameWindow** that delegates document management functionality to its owner, usually a subclass of **WinDocument**. **SDIFrame** also implements print preview.

InPlaceFrame adds OLE in-place editing functionality, meaning that an instance of **InPlaceFrame** can host an OLE server.

Property Sheets and Wizards

Overview

Property sheets are used to allow the user to modify properties of an object. A property sheets displays one or more property pages. A property sheet is a system-defined dialog that displays one or more property pages. The user can activate individual pages by clicking on a tab control. A page is a subclass of **PropertyPage** that is based on a dialog template, which can be defined dynamically or loaded from a resource file.

Wizards are similar to property sheets, except that the user cannot switch to an arbitrary page and is forced to navigate using *back* and *forward* buttons.

Using property sheets

A property page closely resembles a regular dialog. The main differences are that the page processes the `apply` message and may (optionally) enable the *Apply Now* button on the containing sheet. Technically, the property page is a child of the property sheet.

The following restrictions apply to the resource template:

- The page must have the `WS_CHILD` style
- The template must not be of the `DIALOGEX` type on certain versions of Windows 95.

Failure to observe these restrictions may generate exceptions at runtime.

When implementing a property page, you must provide the methods listed below:

Table 5-9 Property Page Class Methods

Message	Description
windowClassName	Answers the resource identifier of the dialog template.
windowClassIcon	Optional - Answers the resource identifier of the icon to display on the associated tab.

Table 5-10 Property Page Methods

Message	Description
windowTitle	Answers the text to display in the tab.
apply	Same as *onOk* in regular dialogs.

Creating page templates

The title of the property page defines the name of the tab.

Migrating a dialog to a property page

When a dialog becomes too large or requires multiple dialogs, it is often desirable to use a property sheet. Transforming a dialog is straightforward and involves the following steps:

- Changing the superclass to **PropertyPage**.
- Redesigning the resource template, removing action buttons such as *Ok* and *Cancel*, changing the style to `WS_CHILD` and, if applicable, the dialog type to a simple `DIALOG` (instead of `DIALOGEX`).
- Changing the `onOk` (or equivalent) validation method to `apply`.

Creating a page

Once the template has been defined, a page just needs to implement the `apply` method. This method is similar to the `onOk` method in regular dialog boxes; its task is to validate the changes and store data for further retrieval.

Creating a property sheet

In most cases, the property sheet itself does not implement much processing. In most cases, it is sufficient to use the generic **PropertySheet** class.

A property sheet can be created on the fly, as in the code fragment below:

```
| sheet |
sheet := PropertySheet new.
sheet createPage: SamplePropertyPage1.
sheet createPage: SamplePropertyPage2.
sheet hInstance: Processor hInstance
    owner: 0
    icon: 0
    title: 'Sample Property Sheet'
    flags: 0.
```

Enabling the Apply button

The `changed` method in **PropertyPage** notifies the property sheet that the user has made changes. In response to the notification, the property sheet may enable the *apply* button.

In the preceding example, it is convenient to link the `BN_CLICKED` and `EN_CHANGE` events in buttons and edit fields, respectively, to the `changed` method. Therefore, clicking on a button or entering some text enables the *apply* button on the property sheet.

Returning event values

Certain property page events require the page to return a value that defines success or failure of a given operation. **PropertyPage** automatically sets the return value as returned by an event handler.

Table 5-11 Property Page Events and Return Values

Notification	Return Values
PSN_APPLY	PSNRET_INVALID_NOCHANGEPAGE
	PSNRET_NOERROR
PSN_KILLACTIVE	TRUE (allow)
	FALSE (disable)

PSN_QUERYCANCEL	TRUE (allow)
	FALSE (disable)
PSN_RESET	void
PSN_SETACTIVE	0 (accept)
	-1 (activate next or previous)
	a property page identifier
PSN_WIZBACK	-1 (disable)
PSN_WIZFINISH	0 (accept)
PSN_WIZNEXT	-1 (disable)

Please refer to the Win32 documentation for additional information regarding these notifications.

Using external resources

In this paragraph, we will save the resources previously defined as RC (resource script) files. RC scripts can be compiled to an external resource DLL.

1. Open the interface builder and click on *File|New* and choose *Resource Script*. Enter a folder name (for example *Property_pages*). This creates a new folder and a subfolder names `RES` with an empty resource script and makefile.
2. Edit a property page in the interface builder. Open the properties for the dialog and enter the name of the resource DLL (the same name as for the folder above).
3. Click on *File|Save As...* and choose *Dialog and Resource template* as the output format. Save the resources of the two pages under for example `PAGE1.RC` and `PAGE2.RC`.

The next step is to include the scripts for each page in the main script. Open the main script (`PROPERTY_PAGES.RC`) and add the lines below:

```
RCINCLUDE page1.rc
RCINCLUDE page2.rc
```

Run *nmake* and copy the new DLL to your Smalltalk working directory.

Note	Once the DLL version works as expected, you can remove the instance method `initTemplate`.

Tabbed Dialogs

TabbedDialog and **TabbedDialogPage** implement tabbed dialogs such as the ones used by the interface builder. **TabbedDialog** is a top-level window that must at least have one tab control. **TabbedDialog** operates on a collection of child pages. Each child page is a dialog that has the `WS_CHILD` style, and the parent dialog shows the page that corresponds to the active tab and hides the other pages. On creation, the parent dialog automatically computes the size required to display all the pages it displays.

The default implementation in **TabbedDialog** does not assume that the dialog has an *Apply* or *Ok* button. All changes that the user makes are generally applied immediately. The pages are created when the parent dialog opens, unlike property sheets where pages are created on demand. These are the main differences between **TabbedDialog** and **PropertySheet**.

Individual pages can be accessed using `pageAt:`, which takes the one-based index of a page. The method `getItemPage:` returns the page that hosts a given child control. This method is more flexible in that it allows the order and contents of the pages to change without breaking code that works on a particular control or group of controls. To activate a page, use `selectPage:`.

Predefined Dialogs

Smalltalk MT comes with a set of general-purpose dialogs. The dialogs require resources whose templates are in the `SUPPORT` subdirectory.

Prompters

Prompters prompt the user for text or selections among a list of items. Several specialized Prompters provide list processing.

Prompter Variables

Table 5-12 Prompter Variables

Item	Description
list	Contains nil or a list of items to display in the combo-box.
option1, option2	Contains nil if the dialog has no checkboxes, otherwise the initial state of the checkbox items upon entry and the user selections when the dialog closes. This can be a combination of STATE_XXX constants, including STATE_DISABLED to disable a checkbox.
text	Contains the text to display in the text field when the dialog opens, and the user entry when the user presses the Ok button, a value of nil signifies that the user pressed Cancel.
title	The title to display.
selections	Contains the checked items for a CheckListPrompter.

Text Prompters (Prompter class)

A **Prompter** dialog is a simple text prompter that lets the user enter text. It can also display up to two check boxes that let the user specify options, and initialize a combo-box with predefined choices.

Figure 5-5 Text Prompter

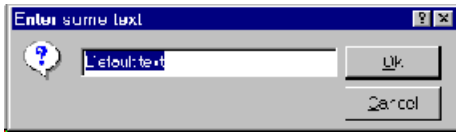
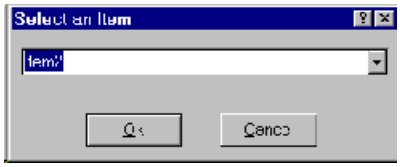


Figure 5-6 Text Prompter with ComboBox



Example:

The following code fragment opens a simple text prompter:

```
Prompter owner: Transcript title: 'Enter some text' text: 'Default text'
```

The return value is the text entered by the user or **nil** if he or she pressed *Cancel*.

To display a list of items:

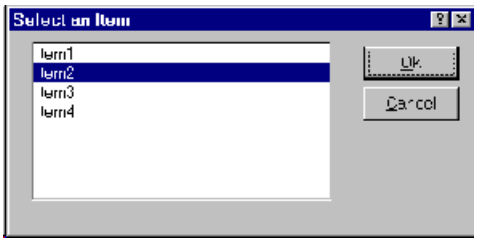
```
Prompter new list: #('item1' 'item2' 'item3' 'item4')
    text: 'item2';
    openListOn: Transcript
```

Prompter also handles up to two checkboxes. To use this feature, you must define a prompter template with two checkboxes that have the identifiers `IDC_CHECK1` and `IDC_CHECK2`. The prompter must be created using `dialogBox:template:owner:` and specifying a module handle and a template resource identifier.

ListPrompter

ListPrompter displays a list of items and lets the user select an item. There are two templates; one with a combo-box and the other with a list box. To select a particular item, set the text variable to the string to select.

Figure 5-7 ListPrompter



Example:

The protocol is the same as for a regular Prompter.

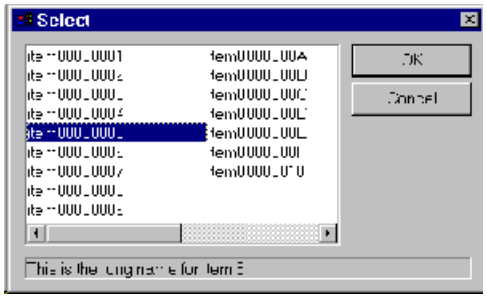
```
ListPrompter new list: #('item1' 'item2' 'item3' 'item4')
    text: 'item2';
    openOn: Transcript
```

MultiListPrompter

MultiListPrompter displays a multi-column listbox instead of a regular list, and displays a long name in an additional text field when the user selects an item. The

contents are set via `contents`, which takes a **MappingTable** that associates display names with long names. The return value of `openOn:` is the text of the selected item, or **nil** if the dialog was cancelled.

Figure 5-8 MultiListPrompter



Example:

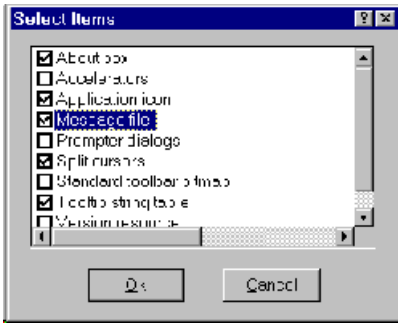
```
| contents |
contents := MappingTable new.
1 to: 16 do: [ :i |
    contents at: 'item',i printHexString
    put: 'This is the long name for item ',i printString
].
(MultiListPrompter new contents: contents) openOn: Transcript.
```

CheckListPrompter

CheckListPrompter displays a list of items, with a check box left to each item. The caller can set the initial check states using `selections:`. The `openOn:` method returns an array with the check status of each item. This is currently 0 (not checked) or 1 (checked). Future extensions may define additional values.

The return value is an array with the selection state of each item (either 0 or 1).

Figure 5-9 CheckListPrompter



Example:

```
| rgTemplates options|
rgTemplates := #( 'About box'
    'Accelerators'
    'Application icon'
    'Message file'
    'Standard toolbar bitmap'
    'Tooltip string table'
    'Version resource').
options := CheckListPrompter new
    contents: rgTemplates;
    selections: #(1 0 1 1);
    openOn: Transcript.
```

Opening a Prompter

If you use a predefined template, simply call `openOn:` and pass the owner window as the first parameter. Otherwise, call `dialogBox:template:owner:` to use a custom template, passing the module, the template name and the owner window.

The opening method is normally modal, so the call returns when the user closed the dialog. Use `text` to retrieve the user entry. A value of **nil** indicates that the user pressed *Cancel*.

Adding Online Help

All prompters come with a help icon on the caption and support online help. Please refer to *Instance-based Help* on page 224 for an example on how to do this.

Menus

About menus

Each top-level window can have one menu. A menu sends commands to the frame window to which it belongs. A menu consists of a collection of popup menus, which themselves contain menu items, separators or other popup menus.

A menu can be created from an in-memory template or it can be loaded from a resource. While it is possible to create popup menus from an in-memory template, menu resources always consist of a top-level menu.

Creating a menu

The instance method `windowMenu` returns a **Menu** instance or a menu identifier (a string or integer). If the method returns an identifier, the framework loads the menu with the matching identifier from the receiver's resource.

A menu can also be created from a template. To create a menu with popup menus, an application uses code as below:

```
Menu new insertItem: (Menu createPopupMenu
    insertItem: ID_FILE_NEW text: '&New' fType: 0 fState: 0 before: -1;
    insertItem: NULL text: NULL fType: MFT_SEPARATOR fState: MFS_DISABLED before: -1;
    insertItem: ID_APP_EXIT text: 'E&xit' fType: 0 fState: 0 before: -1;
    yourself) text: '&File' fType: 0 fState: 0 before: -1;
insertItem: (Menu createPopupMenu
    insertItem: ID_HELP_INDEX text: '&Help Topics...' fType: 0 fState: 0 before: -1;
    insertItem: NULL text: NULL fType: MFT_SEPARATOR fState: MFS_DISABLED before: -1;
    insertItem: ID_APP_ABOUT text: '&About' fType: 0 fState: 0 before: -1;
    yourself) text: '&Help' fType: 0 fState: 0 before: -1;
yourself
```

Creating a floating popup menu

A floating popup menu can be loaded from external resources and displayed using `popupMenu:module:index:owner:` in class **Menu**. The parameters to this method are:

- The identifier of the top-level menu.
- The module from which to load the menu.
- The zero-based index of the popup menu. For example, it would be 0 for a File popup menu, 1 for Edit and so forth.

- The owner of the popup. The owner is the window that receives commands from the menu.

If the popup menu must be constructed at runtime, it is displayed using `trackPopupMenu:` and must be destroyed when the menu is no longer used. The method `trackPopupMenu:` displays the popup and returns when the user either selected an item or closed the menu.

Modifying a menu

An application uses `insertMenu:` to insert a new menu item, `deleteMenu:` to delete an item, and `setItemState:` to modify the state of an item.

Controls

Windows controls are predefined windows implemented by the operating system or extensions. A control implements its own window procedure.

The appearance and functionality of a control is often defined by styles that extend the standard window styles such as `WS_CHILD` and `WS_VISIBLE`. Each control uses its own set of styles.

An application can send messages to a control to perform some action or retrieve data. Subclasses of **Control** encapsulate the functionality. Internally, a **Control** subclass communicates with the window through predefined window messages above `WM_USER`.

A control relays user-related events to the parent window. These take the form of notification messages such as `CBN_SELCHANGE`.

Event Notifications

The table below lists commonly used event notifications. Most controls have a large number of additional notifications. Note that the notifications are sent after a user action occurred, programmatic changes to a control do normally not produce notifications.

Table 5-13 Commonly used Event Notifications

Notification	Applies to	**Description**
`BN_CLICKED`	Button	The user clicked the control.
`CBN_SELCHANGE`	Combobox	The selection changed.
`CBN_EDITCHANGE`	Combobox	The edit contents changed.
`EN_CHANGE`	Edit	The edit contents changed.
`LBN_SELCHANGE`	Listbox	The selection changed.
`LBN_DBLCLK`	Listbox	The user double-clicked the control.
`STN_CLICKED`	Static	The user clicked the control.

`STN_DBLCLK`	Static	The user double-clicked the control.
`LVN_ITEMCHANGED`	List view	An item changed. Use this to detect selection changes.
`TCN_SELCHANGE`	Tab control	The selection changed.
`TVN_SELCHANGED`	Tree view	The selection changed.

Buttons

Buttons are generally used in dialogs. In most cases, it is not necessary to create a Smalltalk object for the button, and the state of an individual button can be retrieved and set using the Window item messages (`setItemState`, `getItemState`, etc.).

It is often necessary to check one button (radio button or check box) in a group of buttons and uncheck the remaining buttons. This can be accomplished with `checkItem:first:last:` as in the example below:

```
self checkItem: IDC_RADIO2 first: IDC_RADIO1 last: IDC_RADIO2.
```

Likewise, the identifier of the button that is checked in a group of radio buttons can be retrieved with `getCheckedItem:`.

List Controls

List controls encompass the **Listbox**, **Combobox**, **ComboboxEx**, **ListView** and **TreeView** controls. All these controls display a collection of items and let the user select one or several items.

Adding Items

The methods that add items are specific to the control. You would use `insertItem:` with listboxes and comboboxes, while other controls such as Listview require more information.

When adding a large number of items, it is useful to call `setRedraw:` to inhibit redrawing while the items are added, and you have of course to reset the redraw state

when the list is filled. For example, this cuts down the time it takes to fill a ListView by half.

Associating items with objects

In the case of sequential listboxes, items are identified by indices, which can be used to access the elements of an associated sequenceable collection.

Otherwise, you can use the parameter that can be associated with each item to store the address of the associated Smalltalk object. The method `getItemData:` returns the Smalltalk object, given the index or handle to an item. Remember, however, that the data parameter is not garbage collected. Therefore, the Smalltalk object must be referenced from the Smalltalk image.

In most cases, working with a list control involves performing some action when the user selects one or several items or when he or she double-clicks an item.

You must first define a handler for the event in the class `initialize` method of your frame window. For example:

```
addHandler: LVN_ITEMCHANGED in: IDC_LIST1 selector: #onSelChange;
```

Replace `LVN_ITEMCHANGED` with the appropriate event id for the control you're working with.

Retrieving the selection

In the case of a single-selection control, use `getSelItem`. The return value is the zero-based index of the selected item or, depending on the control, a handle to the item, or **nil** if nothing is selected.

For multiple selections, use `getSelItems` to obtain an array of item indices or handles.

Setting the selection

The message `selectItem:` selects the specified item. To ensure that the item is visible, use `ensureVisible:`.

Edit Controls

The edit controls an application can use are **Edit** and **RichEdit**. An edit control can be single-line or multi-line. **RichEdit** is a more advanced control that is able to display RTF text.

An application uses `limitText:` to limit the amount of text a user can enter. Since edit controls have a default limit, it is often necessary to increase the limit. To display large amounts of text, an application can pass `16r0FFFFFFF` as the maximum value (passing a zero value may not work as documented in the Win32 RichEdit specifications).

Conflicts with Accelerator Keys

Some keys produce the same ASCII values as CTRL+key combinations. These keys conflict with edit controls if one of the CTRL+key combinations is used as a keyboard accelerator.

Scroll Boxes

ScrollBar encapsulates a scroll bar. A vertical scroll bar sends a `WM_VSCROLL` message to the parent window, and a horizontal scroll bar sends a `WM_HSCROLL` message. The default implementation in **FrameWindow** generates `WM_VSCROLL` and `WM_HSCROLL` events when receiving these messages. The parameter to each of these notifications contains the scroll code and the position. Note that for frame scrollbars, the control identifier is zero.

The parent window can also implement the `WM_VSCROLL` and / or `WM_HSCROLL` messages, which are sent when the user clicks on the scroll bar. It may delegate to the scroll bar object by calling the `updateScrollPos:` method, which processes the input and updates the scroll bar accordingly.

Using Scroll Boxes

An application initially calls `setScrollRange:to:redraw:` to set the scroll range and `setScrollPage:` to set the size of a page (when the user clicks on the bar). It must also implement `WM_VSCROLL` and `WM_HSCROLL` handlers such as in the code fragment below:

```
onVscroll: wparam
| iPos |
iPos := (self childAt: IDC_SCROLL1) updateScrollPos: wparam.
iPos notNil ifTrue: [
    self updatePosition: iPos
].
^NULL
```

The message `updateScrollPos:` updates the scroll bar and returns the new position if it changed, otherwise it returns **nil**.

Using scroll bar styles

A window that has the `WS_VSCROLL` or `WS_HSCROLL` styles has an associated scroll bar that sends `WM_VSCROLL` or `WM_HSCROLL` messages. The parent window can use `fromHandle:` to create a temporary **ScrollBar** object for the scroll bar, given the window handle in *lparam*.

Common Controls Library

The common controls library is a supplemental DLL that ships with the operating system. While standard controls were available on Windows 3.x, common controls first appeared with Windows 95. The library is regularly extended with additional controls or new functionality.

To function properly, the library must first be loaded and initialized. This can be done with a call like the one below:

```
Control initCommonControlsEx: ICC_WIN95_CLASSES|ICC_DATE_CLASSES|
ICC_USEREX_CLASSES|ICC_COOL_CLASSES.
```

Initialization is normally performed in `WinApplication>>initInstance`.

Common Control Styles

This paragraph discusses styles that apply to header controls, toolbars, status bars and rebars (sometimes also called CoolBars). These controls are able to resize and align themselves on a specified size of the frame.

`CCS_LEFT`, `CCS_TOP`, `CCS_BOTTOM` and `CCS_RIGHT` position the control at the left, top, bottom or right side of the parent window, and resize the control so that it has the same width or height as the parent. The style bits `CCS_NOPARENTALIGN` prevents the control from moving (but it is resized automatically), while the style `CCS_NORESIZE` prevents it from resizing.

You will generally specify the automatic alignment styles for a stand-alone control, and have to turn them off if the control (usually a tool bar) is used in a **ReBar** (CoolBar), or if both horizontal and vertical controls are used. The resizing code in **FrameWindow** and **Control** takes care of the placement and aligns first the horizontal bars, then the vertical bars.

Application-defined Data

Most common controls that display multiple items can associate an application-defined 32-bit value with an item. It is also called *data* or *lparam* value. The Smalltalk implementation of these controls can store and retrieve an arbitrary Smalltalk object. The conversion to 32 bits is done automatically.

Associating a real object with a list item is often convenient and allows an application to use the control in an object-oriented way. However, an application needs to reference the objects it passes to the control by some means because the control does not keep a strong reference to the data objects.

ComboBoxEx

A **ComboBoxEx** control is an extension of the standard combo box control. ComboBoxEx provides support for item images and item indentation, making it easy to display a hierarchical structure. For more information about using and initializing image lists, see also *List View* on page 122.

To insert items into the control, you use the message `#insertItem:text:iImage:indent:data:`.

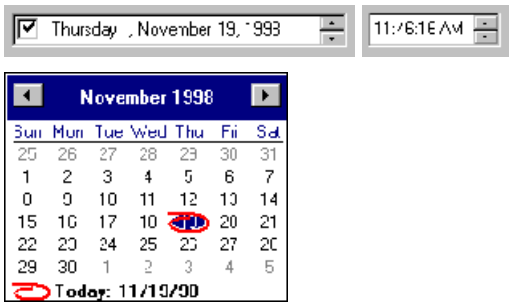
Using ComboBoxEx

Because ComboBoxEx is not a real ComboBox, its behavior differs. ComboBoxEx works in two modes: drop-down and combo edit / list control. The Smalltalk implementation tries to hide the differences whenever possible.

DateTimePicker

A **DateTimePicker** lets the user select or enter a date or time. The control accepts and returns instances of Time.

Figure 5-10 DateTimePicker Samples



DragListBox

A **DragListBox** enables the user to drag items from one position to another. An application can use a drag listbox to let the user define a particular sequence in a collection of items.

Creating a DragListBox

A **DragListBox** is created like a normal **ListBox**. It is initialized with the message `makeDragListBox`. The message `subclassParentWindow:` subclasses a specified parent window and sends **DragListBox** notifications to the parent.

The code sample below initializes a **DragListBox**:

```
(lb := self childAt: IDC_LIST1)
makeDragList subclassParentWindow: self.
```

Using a DragListBox

The parent of the listbox must handle the notifications `DL_BEGINDRAG`, `DL_DRAGGING`, `DL_CANCELDRAG`, and `DL_DROPPED`. The parameter to each of these messages is a **Pointer** object to a `DRAGLISTINFO` structure (remember that the event is sent directly by the **DragListBox** Smalltalk object without going through a `SendMessage`).

Note that the **DragListBox** draws the insertion cursor onto the client area of its parent. Therefore, the parent must update its client area when the dragging operation completes.

Processing the DL_BEGINDRAG notification

The parent needs to retrieve the source item and store it for later usage. The return value is TRUE to enable the operation and FALSE to disallow it.

Example:

```
onBegindrag: pdlinfo
    | dlb |
    dlb := self childAt: IDC_LIST1.
    m_draggedItem := dlb LBItemFromPt: pdlinfo ptCursor
        autoScroll: TRUE.
    m_draggedItem ~~ -1 ifTrue: [
        ^TRUE
    ].
    ^FALSE
```

Processing the DL_DRAGGING notification

The parent retrieves the item under the cursor using `LBItemFromPt:autoScroll:` and draws the insertion cursor using `drawInsert:`. The return value specifies the cursor to display.

Example:

```
onDragging: pdlinfo
    | nItem dlb |
    dlb := self childAt: IDC_LIST1.
    nItem := dlb LBItemFromPt: pdlinfo ptCursor
        autoScroll: TRUE.
    nItem ~~ -1 ifTrue: [
        dlb drawInsert: nItem.
        ^DL_COPYCURSOR
    ].
    ^DL_STOPCURSOR
```

Processing the DL_CANCELDRAG notification

This notification signals that the user cancelled the operation by pressing the escape key or clicking the right mouse button. The parent window needs to clear the dragged item and update the client area.

Example:

```
onCanceldrag: pdlinfo
    m_draggedItem := nil.
    self updateClientRect.
    ^FALSE " no return value "
```

Processing the DL_DROPPED notification

The parent window typically processes this notification message by inserting or copying the dragged list item before the list item under the cursor.

Example:

```
onDropped: pdlinfo
    | dlb dropIndex|
    dlb := self childAt: IDC_LIST1.
    dropIndex := dlb LBItemFromPt: pdlinfo ptCursor
        autoScroll: FALSE.
    m_draggedItem ~~ dropIndex ifTrue: [
        dlb selectItem: (dlb insertItem: (dlb getItemText: m_draggedItem)
            at: dropIndex).
        dlb deleteItem: (m_draggedItem > dropIndex ifTrue: [
            m_draggedItem + 1.
        ]
        ifFalse: [
            m_draggedItem.
        ]).
        self updateClientRect.
    ].
    m_draggedItem := nil.
    ^FALSE " no return value "
```

Implementation Issues

DragListBox is implemented in a peculiar way because the Windows control uses a registered message to send notifications to the parent. In addition, some of the notifications require a return value, which is difficult to implement in a dialog.

The message `subclassParentWindow:` subclasses the parent window and hides these details. The parent window simply uses the control like any other Windows control, and receives notifications through the event handler system.

HeaderControl

A **HeaderControl** displays column headers. The **HeaderControl** is seldom used alone; it is often used in conjunction with a **ListView** control in report view. The user can drag the dividers and click or double-click on columns.

HotKey

A hot key is a key combination that activates a given application window. Each top-level window of an application can have one associated hot key. When the user presses

the hot key from any window on the desktop, the corresponding top-level window is activated.

A Hot-Key control enables the user to enter a key combination that can be used as a hot key. An application retrieves the key combination with `getHotKey`.

An application sends `WM_GETHOTKEY` to a window to retrieve the current hot key for this window, and `WM_SETHOTKEY` to set a new value.

ListView

List views display a list of items. There are two basic view types, *icon view* and *report view*. Icon views consist of three subtypes: *small icon view*, *large icon view*, and *list view*.

The most widely used view type is *report view*, which displays columns of data. An optional header control displays the headers of the columns, and the user can click on a column header to sort the data (sorting is still the application's responsibility). An application can switch between the different views by changing the list view style with the `setStyle:` method.

Creating a ListView

A **ListView** object must first be attached to the list view control. This is normally done in the `initWindow` method, and it is a good time to set the column widths and headers for a report view. The code to perform this typically looks as follows:

```
| lv |
lv := ListView new attachTo: self id: IDC_LIST1.
lv
    insertColumn: 0 fmt: LVCFMT_LEFT cx: 280 text: 'Name';
    insertColumn: 1 fmt: LVCFMT_LEFT cx: 140 text: 'Value'
```

You can also use the GUI builder to generate this code.

Using Image Lists

You can specify an image list to be used with the list view. An image list is usually created from a bitmap that resides in a resource DLL.

The code fragment below loads an image list for icon view:

```
    imlLarge := ImageList loadImage: m_module
        resourceName: IDB_LARGEICONS
        cx: 32
        cGrow: 0
        crMask: CLR_NONE
        type: IMAGE_BITMAP
        flags: LR_LOADTRANSPARENT|LR_SHARED|LR_CREATEDIBSECTION.
```

In this example, the image list is created from a DIB resource. The `LR_LOADTRANSPARENT` flag makes the images transparent (based on the color of the first pixel in the upper left corner), and the `LR_SHARED` flag specifies that the resource can be shared. The caller must also specify the width of each image in the bitmap.

It is also possible to use the class method `loadDIBImage:resourceName:cx:` in **ImageList**, which defaults to the parameters above. The complete initialization code that also inserts items into the list view (referenced by the variable `lv`) is:

```
| imlSmall imlLarge |
imlSmall := ImageList loadImage: m_module
    resourceName: IDB_SMALLICONS
    cx: 16.
imlLarge := ImageList loadImage: m_module
    resourceName: IDB_LARGEICONS
    cx: 32.
lv setImageList: imlSmall type: LVSIL_SMALL;
    setImageList: imlLarge type: LVSIL_NORMAL;
    insertItemRow: 0 labels: #('yellow') iImage: 0 state: 0 data: 0;
    insertItemRow: 1 labels: #('pink') iImage: 1 state: 0 data: 0;
    insertItemRow: 2 labels: #('red') iImage: 2 state: 0 data: 0;
    insertItemRow: 3 labels: #('orange') iImage: 3 state: 0 data: 0;
    insertItemRow: 4 labels: #('cyan') iImage: 4 state: 0 data: 0;
    insertItemRow: 5 labels: #('green') iImage: 5 state: 0 data: 0;
    insertItemRow: 6 labels: #('blue') iImage: 6 state: 0 data: 0.
```

Inserting Items

To populate a **ListView** with data, an application uses either `insertItemRow:labels:data:` or an extended method that also sets an image index and the state of the row. The parameters are the row index (or –1 to append at the end of the list), an array of strings that specifies the text to display in each column, as well as an application-defined object.

The application-defined value should identify the object being displayed in the row. The application can retrieve it with `getItemData:`, passing the row index to this method. The methods in **ListView** convert the object to and from a 32-bit value.

Sorting a ListView

The user can sort the columns by clicking on a column header. You must first implement an event handler for the `LVN_COLUMNCLICK` event that is triggered when the user clicks on a column header.

An application uses the method `sortItems:with:` to sort the list view control. The first parameter is a sorting block that compares two 32-bit values previously associated with an item and returns the result of the comparison. The resulting value can be negative, zero, or positive. The second parameter is passed to the sorting block as third argument.

The 32-bit values are the addresses of the associated objects, so a typical block looks like:

```
lv sortItems: [ :pa :pb :lparam |
    pa _asObject. compareWith: pb _asObject.
].
```

The method `compareWith:` is implemented in **String**, **Number** and **Time**. Alternatively, you can also use code such as the one below:

```
lv sortItems: [ :pa :pb :lparam |  | a b |
    a := pa _asObject. b := b _asObject.
    a  < b ifTrue: [-1]
    ifFalse: [
        a == b ifTrue: [0]
        ifFalse: [1]
    ]
].
```

For performance-sensitive scenarios, you can implement a class method and export it so that it is called directly. You can take advantage of the third parameter to pass any data that is required to compare the items.

Processing Notifications

A **ListView** is a list presenter, so the most important notification is `LVN_ITEMCHANGED`. The parameter to this notification message is an **NMLISTVIEW** structure with further information about the change event. In general, list presenters ignore the parameter and retrieve the selected item with `getSelItem`. However, the **ListView** tends to send many `LVN_ITEMCHANGED` notifications when the user selects a new item, so it is good practice to filter out the messages of interest.

Code that processes the structure looks like:

```
selChanged: pnmhdr
    | pnm_listview |
    pnm_listview := NMLISTVIEW fromAddress: pnmhdr.
    pnm_listview uNewState ~~ (LVIS_SELECTED|LVIS_FOCUSED) ifTrue: [^NULL].
    " Further processing here ... "
```

The code above is somewhat inefficient because it creates a new NMLISTVIEW object on each invocation, only to find out that most of them are not needed. The version below uses stack-local allocations to avoid creating garbage:

```
selChanged: pnmhdr
    | pnm_listview |
    NMLISTVIEW fromAddress: pnmhdr pointer: (pnm_listview := Pointer localNew).
    pnm_listview uNewState ~~ (LVIS_SELECTED|LVIS_FOCUSED) ifTrue: [^NULL].
```

The first line creates a local **Pointer** to a `NMLISTVIEW` structure. The message `fromAddress:pointer:` initializes the **Pointer** instance with the field declarations of the `NMLISTVIEW` structure.

MonthCal

A month calendar control implements a calendar-like user interface. This provides the user with a very intuitive and recognizable method of entering or selecting a date. The control use Time instances to set and retrieve dates.

ProgressBar

A **ProgressBar** displays the progress of an operation in a rectangle that is gradually filled. A progress bar is often used to indicate the progress of a lengthy operation.

ReBar

A **ReBar** control acts as a container for other child controls. It displays one or more bands of controls, where each band can have a gripper bar, a text label, a bitmap and a child window.

Creating a ReBar

A **ReBar** control is created with the message `createReBar:style:align:id:`. The alignment style defines the orientation of the control (horizontal or vertical).

The code below creates a **ReBar**:

```
rebar := ReBar new createReBar: self
    style: WS_VISIBLE|WS_BORDER|WS_CHILD|WS_CLIPCHILDREN|WS_CLIPSIBLINGS|
        RBS_TOOLTIPS|RBS_VARHEIGHT|RBS_BANDBORDERS|
        CCS_NODIVIDER|CCS_NOPARENTALIGN
    align: CCS_TOP
    id: ID_TOOLBAR.
```

Inserting Child Controls

The first step is to create the control to insert, the second is to insert it into the ReBar..

How to insert a toolbar

The code that creates the toolbar can specify the current window as parent. The message `insertControl:[...]` inserts the control into the **ReBar**, which also entails reparenting it.

The parameters `cxMinChild` and `cyMinChild` specify the minimum width and height of the bar. The **ReBar** displays the band on a new line as needed.

```
rebar insertControl: self initToolBar
    index: 0
    cxMinChild: 100
    cyMinChild: 22
    cx: 60
    text: nil
    barStyle: RBBS_CHILDEDGE|RBBS_BREAK
    bitmap: 0.
```

How to insert a combo box

1. Creating the combobox

Once the combo box has been created, the message `subclassEdit` subclasses the edit field of the combobox to notify the parent about key events. The parent may process the `ESC`, `TAB` and `ENTER` keys.

The message `setFont:` sets the font of the combobox to the default GUI font, which is the font used to draw menus and the labels on the **ReBar**.

```
cbx := ComboBox new createWindow: NULL
    exStyle: 0
    style: WS_CHILD|WS_CLIPSIBLINGS|WS_BORDER|WS_VSCROLL|CBS_SORT|
        CBS_DROPDOWN|CBS_AUTOHSCROLL|WS_VISIBLE
    x: 0
    y: 0
    cx: 200
    cy: 200
    parent: rebar
    id: IDC_CONTROL1.
cbx subclassEdit setFont: Font defaultGUIFont.
cbx setContents: #('Item1' 'Item2' 'Item3'); selectItem: 0.
```

2. Inserting the combobox

The minimum width in `cxMinWidth` specifies the minimum size of the control. It is also possible to specify a minimum height; the value **nil** lets the **ReBar** use the current height of the control as minimum value.

Finally, the flag `RBBS_FIXEDSIZE` specifies that the combobox has a fixed size. Without this flag, the width of the combobox is variable.

```
rebar insertControl: cbx
    index: 0
    cxMinChild: 100
    cyMinChild: nil
    cx: 200
    text: 'Item:'
    barStyle: RBBS_CHILDEDGE|RBBS_FIXEDSIZE|RBBS_GRIPPERALWAYS
    bitmap: nil.
```

The last parameter can also specify a bitmap to use as background. In this case, the parent must delete the bitmap when it is destroyed to free the bitmap resource.

RichEdit

A **RichEdit** control accepts formatted text and embedded OLE documents. The control also understands the RTF format.

RichEdit20 is a more advanced version of **RichEdit** that understands more RTF formatting. Beware that its CrLf character translation is not compatible with **RichEdit**.

StatusWindow

A **StatusWindow** (also called Status Bar) is a horizontal window at the bottom of a frame window. It has parts in which information can be displayed.

Creating a StatusWindow

A **StatusWindow** is opened using `openWindowIn:`. To set the parts, the parent window can use `setParts:`, which takes an array of part widths. Another useful message is `setCharParts:`, which also sets the part widths but uses the number of characters as a measurement unit.

The code below opens a status window and defines 3 parts; the first part uses 20 character units, the second is able to display 40 characters, and the third extends to the right edge of the parent.

```
(StatusWindow new openWindowIn: self)
    setCharParts: #(20 40 -1).
```

Using a StatusWindow

The framework automatically displays menu help text in the status bar. This is done in simple mode, where the control only displays one part.

The message `setItemText:text:` sets the text in a specified part. A **FrameWindow** can also use the message `setStatusText` and `setStatusText:in:` to set the text in a status bar part.

Using a Progress Bar inside a StatusWindow

The message `openProgressBarIn:` opens a progress bar in a given part. The messages `setPos:`, `setRange:`, `setStep:` and `stepIt` act on the progress bar. The message `close` closes the progress bar, if there is one (otherwise it closes the status bar). This makes it easy to pass a status bar to code that expects a progress bar.

The message `forkProgress:` opens a progress bar, executes a given block in a separate thread, and closes the progress bar on exit. It also initializes the thread-local variable **progressIndicator** with the status bar. Code executed in the thread typically calls `setRange:` to set the progress range and `stepIt` to indicate the progress.

TabControl

A **TabControl** displays a row of tabs the user can click onto. It is often used to display pages of information. In general, you will not create a **TabControl** directly but use it through a **PropertySheet** or a **TabbedDialog**.

Creating a TabControl

Creating a **TabControl** is analogous to creating a regular control. You can try out the different styles with the GUI builder.

An interesting feature is the button style, when combined with the multi-select style. The user can press several buttons by holding down the control key.

Initializing a TabControl

The method `insertItem:text:iImage:param:` adds rows to a **TabControl**. The first parameter specifies the zero-based row index to insert, the second the tab label, followed by an index into an image list and an application-specific parameter.

The code fragment below creates three tab buttons:

```
(self childAt: IDC_TAB1)
    insertItem: 0 text: 'list 1' iImage: 0 param: 0;
    insertItem: 1 text: 'list 2' iImage: 0 param: 0;
    insertItem: 2 text: 'options' iImage: 0 param: 0.
```

Using a TabControl

The event `TCN_SELCHANGE` notifies the owner of a **TabControl** that the user clicked on a tab. The message `getSelItem` returns the selected tab index, if any.

Example

The following example simulates pages by showing and hiding different controls when the user selects a tab.

The code that handles the **TabControl** selection notification is as follows:

```
| iSel rgPageIds|
rgPageIds := #( (##(IDC_LIST1)) (##(IDC_LIST2)) (##(IDC_RADIO1) ##(IDC_RADIO2))
).

" get zero-based selection index "
iSel := (self childAt: IDC_TAB1) getSelItem.
iSel notNil ifTrue: [
    iSel := iSel + 1.
    rgPageIds basicDo: [ :i :rgIds |
        " hide / show those items "
        rgIds basicDo: [ :j :id |
            self showItem: id state: i == iSel
        ]
    ]
].
^NULL
```

Note It is easier to use a **TabbedDialog** in order to duplicate the functionality above.

ToolBar

A Tool Bar lets you arrange buttons and more generally controls such as combo boxes. Class **ToolBar** encapsulates the ToolBar control. Toolbars use image lists to represent button images. Optionally, a toolbar can display a label next to each button.

Creating a ToolBar

You can create a **ToolBar** like any other window. Once created, you can add buttons to the tool bar, set the properties such as the image lists used to display buttons, and add the labels that appear under or next to each button.

However, there is a more straightforward method that creates and initializes the tool bar in one step: `createToolBar:hInstance:style:id:bitmapID:bitmapCount:buttonStruct:displayIndex`.

Notable parameters to this method are:

- The bitmap identifier (`bitmapID`) that identifies the bitmap resource for the image list. Pass `NULL` if there is no bitmap.
- The number of images in the image list.

- The button structure. This is an array of `TBBUTTON` structures, one for each button. You will find it convenient to define the structure in a **WordArray** as follows:

  ```
  #[ <index of tool image> <command id> <button style & state>  0 <index of
  string>]
  ```

  - **Index of tool image** the zero-based index of the image you wish to display in the button.
  - **Command id** the command identifier (an event id) that the button sends when the user clicks it. It usually also corresponds to a menu command.
  - **Button style & state** is a compound value that specifies the style and state of the button. Common combinations are defined in the pool dictionary `StResourceConstants`.
  - **Index of label string** A label may be associated with each button. The value you specify here is the zero-based index in a string list to be passed to the tool bar.

- The display index. This can be either a point that specifies the button extent or one of the following values:

  `DPI_STD72, DPI_STD96, DPI_STD120`.

Initializing the ToolBar

You can use `loadStrings:module:` to set the label list of the tool bar. The first parameter is an identifier of the string resource, the second specifies the module that contains the resource. The label list must be a multi-string (null-terminated strings with the last string terminated with a double zero).

For example, the line below defines the browse labels of the browse toolbar in the Class Hierarchy Browser.

```
IDS_STBROWSE "Back\0Forward\0Stop\0Up\0Down\0Refresh\0Home\0Font\0Query\0"
```

If the toolbar uses labels, the string index of each button structure must be initialized as well. The code fragment below initializes the button structure array for the browse toolbar:

```
buttonArray := #[
TOOLIMAGE_BROWSEBACK     ID_BACK      TB_ENABLEDBUTTON 0    TOOLIMAGE_BROWSEBACK
TOOLIMAGE_BROWSEFWD      ID_FWD       TB_ENABLEDBUTTON 0    TOOLIMAGE_BROWSEFWD
0                        0            TB_SEPARATOR     -1   -1
TOOLIMAGE_BROWSEQUERY    ID_SEARCH    TB_ENABLEDBUTTON 0    TOOLIMAGE_BROWSEQUERY
0                        0            TB_SEPARATOR     -1   -1
TOOLIMAGE_BROWSEUP       ID_UP        TB_ENABLEDBUTTON 0    TOOLIMAGE_BROWSEUP
TOOLIMAGE_BROWSEDOWN     ID_DOWN      TB_ENABLEDBUTTON 0    TOOLIMAGE_BROWSEDOWN
0                        0            TB_SEPARATOR     -1   -1
TOOLIMAGE_BROWSEREFRESH  ID_REFRESH   TB_ENABLEDBUTTON 0
TOOLIMAGE_BROWSEREFRESH
].
```

Special Considerations

If you use the `TBSTYLE_LIST` style and display labels, you may have to enlarge the buttons using the message `setButtonWidth:`. Otherwise, labels may be cut off.

Updating the Toolbar

In most cases, a toolbar is updated during idle processing. When there are no more messages in the message queue, the message loop sends the message `onIdle` to each top-level window. The default code in **FrameWindow** looks for a control whose identifier is `ID_TOOLBAR` and raises the event `#updateCommandUI` with the control. This event must be handled be the window or child controls to update button or menu states.

To benefit from this behavior, the toolbar must be created with the identifier `ID_TOOLBAR`. If this is not the case, or if more than one toolbar are attached to the window, it is necessary to implement `onIdle` in order to dispatch the update event to the toolbar(s).

The code to update two toolbars typically looks as follows:

```
onIdle
    self event: #updateCommandUI
        with: (m_properties at: ID_TOOLBAR+1)
        with: NULL.
    self event: #updateCommandUI
        with: (m_properties at: ID_TOOLBAR+2)
        with: NULL.
    ^TRUE
```

Toolbar customization

Toolbar customization allows a user to customize the buttons displayed by the toolbar. To enable the built-in customization feature, the toolbar must have the `CCS_ADJUSTABLE` style.

The user can customize the toolbar by dragging buttons while holding down the shift key, or by double-clicking on the toolbar. In the latter case, a dialog allows the user to add or remove buttons. The events that the parent window must handle are:

`TBN_QUERYINSERT`, `TBN_QUERYDELETE`, and `TBN_GETBUTTONINFO`.

FrameWindow provides an implementation that allows the user to drag, delete, and add buttons. To enable the default processing, the parent window must link the events to the default handlers in the event initialization method:

```
self
    " toolbar customization handlers "
    addHandler: TBN_QUERYINSERT   in: ID_TOOLBAR selector: #tbQuery;
    addHandler: TBN_QUERYDELETE   in: ID_TOOLBAR selector: #tbQuery;
    addHandler: TBN_GETBUTTONINFO in: ID_TOOLBAR selector: #tbGetInfo:;
```

The window must also implement a `tbButtons` method that returns a **WordArray** with all possible buttons. The array has the same structure as the one used in the toolbar initialization method; it just defines all buttons that the user may add. The handler `tbGetInfo:` initializes the customization dialog from this structure.

It is also important that the toolbar has associated tooltip strings in the resource module. These strings are loaded by the customization method (otherwise, the user just sees the button bitmaps but has no clue about what the button does). Of course, this does not mean that the toolbar must also display tooltips, it just requires that the corresponding string table is in the module. For example, there is little value in displaying tooltips when the buttons already have a label.

Tooltip

The **Tooltip** control displays a small window with descriptive text when the mouse hovers over a tool. The control can manage multiple tools, and a tool can be a child window or a rectangular area.

The text can be predefined or returned at runtime when the control sends the `TTN_NEEDTEXT` notification. Smalltalk MT offers two mechanisms for this case:

- If the identifier of the tooltip control is `ID_TOOLTIP`, an event is generated and the application must handle the event `TTN_NEEDTEXT::ID_TOOLTIP`. The first parameter is a pointer to a `TOOLTIPTEXT` structure to be filled.
- Otherwise, the framework assumes that the parent's window module contains a resource string under the tool's identifier `+ TOOLTIPOFFSET`.

Using Tool tips

The easiest way is to attach a predefined string to each tool. The example below attaches tooltips to dialog controls:

```
(Tooltip new createTooltip: self)
    addToolWindow: (self getDlgItem: IDC_RADIO1) text: 'first option';
    addToolWindow: (self getDlgItem: IDC_RADIO2) text: 'second option';
    addToolWindow: (self getDlgItem: IDC_COMBO1) text: 'select item'
```

If you use a resource file, you must define a string table as below:

```
STRINGTABLE
{
IDC_RADIO1 + TOOLTIPOFFSET          "first option"
IDC_RADIO2 + TOOLTIPOFFSET          "second option"
IDC_COMBO1 + TOOLTIPOFFSET          "select item"
}
```

The creation code becomes:

```
 (Tooltip new createTooltip: self)
    addToolWindow: (self getDlgItem: IDC_RADIO1;
    addToolWindow: (self getDlgItem: IDC_RADIO2;
    addToolWindow: (self getDlgItem: IDC_COMBO1)
```

The framework handles the tooltip notification automatically (it is the same code that returns the tooltip text for tool bars).

Alternatively, you can define the resource string at creation time, using the message `addToolWindow:text:module:` and passing a resource identifier as well as a module handle. This method is slightly more efficient at runtime.

Dynamic Tooltips

You may use this method if the text to be displayed varies at runtime, and if using `#updateTipText` to change the text is not desirable (for example, because the text depends on the mouse position). Add the following line to the event initialization method in the application window:

```
addHandler: TTN_NEEDTEXT in: ID_TOOLTIP selector: #onNeedText:
```

The handler method o`nNeedText`: must analyze the `NMTTDISPINFO` structure, which is passed via the notification message, and set the text field to the appropriate value. The `idFrom` field of this structure identifies the tool, either by its window handle if it is a window or by an identifier provided at tool creation. The `uFlags` field contains the `TTF_IDISHWND` flag if the identifier is a window handle.

Therefore, the code for a handler method looks as follows:

```
onNeedText: lparam
    "
    Private - Handles the tooltip text callback notification.
    Parameters:
        lparam    A pointer to a NMTTDISPINFO structure.
    Return Value:
        Always NULL.
    "
    | nmhdr szText id|
    nmhdr := NMTTDISPINFO fromAddress: lparam.
    id := nmhdr idFrom.
    nmhdr uFlags & TTF_IDISHWND ~~ 0 ifTrue: [
        " retrieve the tool identifier "
        id := WINAPI GetWindowLong: id with: GWL_ID
    ].

    szText := id
    case: IDC_RADIO1 perform: ['currently option 1']
    case: IDC_RADIO2 perform: ['currently option 2']
    case: IDC_COMBO1 perform: ['select now']
    default: [id printString].
    nmhdr szText: szText.
    ^NULL
```

Using Tooltips in Property pages

Using statically defined tooltips in a property page is the same as using tooltips in a regular dialog. Dynamic tooltips require a hook procedure for the property sheet.

TrackBar

A **TrackBar**, also called slider, is functionally equivalent to a scroll bar. The slider can be vertical or horizontal, and generates the corresponding `WM_XSCROLL` notifications when the user changes the position.

TreeView

A **TreeView** displays a hierarchy of items. The user can expand and collapse nodes.

Creating a TreeView

Creating a **TreeView** is straightforward. If the control is to use an image list, this is also the right time to load and set the image list. The code fragment below loads an image list from a resource module:

```
imlSmall := ImageList loadDIBImage: resourceModule
    resourceName: IDB_SMALLICONS
    cx: 16.
tv setImageList: imlSmall.
```

Initializing a TreeView

TreeView maintains a **TVINSERTSTRUCT** structure that is used to insert or modify an item. The application sets default attributes to use with the message `mask:state:stateMask:iImage:iSelectedImage:`. The message `addItem:to:param:` adds a new item to a specified node. The parameters to this method are the item text, the handle to the parent node, and an application-defined value that identifies the node. The root node is defined by the constant `TVI_ROOT`.

Using a TreeView

A **TreeView** is a list presenter, so the most important notification is `TVN_SELCHANGED`. The parameter to this notification message is an `NMTREEVIEW` structure with further information about the change event. A simple handler can ignore the parameter and retrieve the selected item with `getSelItem`.

Implementing Drag-drop

This paragraph discusses how to implement drag-drop in **TreeView** controls. The technique for other list controls such as **ListView** is the same.

Note that OLE drag-drop is not discussed here. Therefore, the dragging technique introduced here is limited to a single window or a set of windows managed by the same application.

Initiating a dragging operation

The event `TVN_BEGINDRAG` signals the owner that the user started dragging an item. The parent window handles the event by retrieving the image list associated with the control (or loading an image list if the control has no associated image list). It then calls `beginDrag:dxHotspot:dyHotspot:` in the image list to create a temporary image list to drag. The method `ImageList>>dragEnter:with:with:` locks the window and begins drawing the image.

The parent window must also maintain an instance variable that receives the item being dragged, and is set to **nil** when no drag operation is in progress. Finally, it captures the mouse input and hides the cursor.

Processing mouse move messages

The parent window implements `WM_MOUSEMOVE:with:with:` to process mouse messages during the dragging. `ImageList>>dragMove:with:` draws the drag image list at the specified position.

The method `hitTest:with:` in **TreeView** answers the item under the cursor. To highlight a drop target, the parent sends `selectDropTarget:` to the tree view.

Operations that modify the screen must be wrapped into `ImageList>>dragLeave` and `ImageList>>dragEnter:with:with:` to unlock the window and let it update the screen.

Ending the dragging operation

The parent window implements `WM_LBUTTONUP:with:with:` to finalize a dragging operation. This entails calling `ImageList>>dragLeave` and `ImageList>>endDrag`. The parent window retrieves the drag target with `getDropHilite` and performs the operation.

UpDown

An **UpDown** button, also called a spin button, consists of two scroll bar buttons. It is equivalent to a scroll bar and generates scroll events.

A spin button is often attached to a **buddy** window, which displays the spin position.

Smalltalk Controls

Overview

Smalltalk MT implements a set of windows that you can reuse, such as **SplitPane**, **ContainerWindow**, **CustomControl** and **ViewControl**. Finally, **TrayIcon** displays an icon on the taskbar and manages user interactions.

ContainerWindow

A **ContainerWindow** groups a collection of controls so that they can be moved or resized as one unit. A **ContainerWindow** can have a **SplitPane**, which acts on the child windows of the container.

Using **ContainerWindow** enables an application to display more than one split pane.

CustomControl

CustomControl implements the code needed to create an arbitrary Smalltalk child window from a dialog template. An application does not directly use **CustomControl**, and the class is only used in conjunction with dialog resources. **CustomControl** implements the window procedure that loads the creation data and instantiates a target control.

A dialog template that uses a Smalltalk control must specify the **CustomControl**'s class (`'STCONTROL'`) and a byte array of creation data that contains at least the name of the Smalltalk window to instantiate. It may also contain optional data that is passed to the new instance via the message `createWindow:size:`. The first parameter is a pointer to the remaining data (after the class name has been read) and the second

specifies the size of this data. The creation data is specific to the control (and can be void).

Currently, only **OleControlContainer** is implemented as a custom control.

Example:

The line below defines an **OleControlContainer** in a dialog script (RC file):

```
CONTROL         "",IDC_CONTROL1, "STCONTROL", WS_CHILD|WS_VISIBLE, 10,5,190,91,
    BEGIN
    0x6c4f0000L,0x6e6f4365L,0x6c6f7274L,0x746e6f43L,0x656e6961L,0x72L
    END
```

OleControlContainer

An **OleControlContainer** encapsulates an OLE control. The OLE server is created as a child of the **OleControlContainer**, meaning that an application can interact with the container as with any other control, namely moving, sizing and hiding / showing the window. In particular, the application does not need to implement any OLE specific code.

The data that pertains to the OLE server objects is serialized to a byte array, which is stored as a separate resource, or, in the case of in-memory templates, as a property of the dialog. There can be more than one OLE server objects (i.e., instances of **OleControlContainer**) in a dialog. When the dialog is created, it iterates over the binary data stream and initializes each control with its data, by sending it a window message. The message to send is defined in the binary resource. The architecture is therefore very general and not limited to OLE containers.

TrayIcon

A **TrayIcon** manages a tray icon, which is an icon that appears on the Windows taskbar. TrayIcon presents a programmer-friendly interface to tray icons.

TrayIcon enforces the correct user interface behavior for tray icons, as per the Windows Interface Guidelines for Software:

- Tray icons should have tooltips.

- Right-click should display a popup menu with commands that bring up property sheets or other windows related to the icon.
- Left-double-click should execute the default command in this menu.
- Left-click should display further information or controls for the object represented by the tray icon.

To use TrayIcon, implement a subclass and override the instance methods `#defaultAction` – which handles the user's double-click on the icon - and `#windowMenu`. TrayIcon runs in its own thread and message loop and is therefore completely independent from the main application.

Note TrayIcon is implemented as a dialog in order to use the message loop implemented by the operating system for modal dialog boxes. Otherwise, it would have to provide its own message loop.

SplitPane

A **SplitPane** separates a window region vertically, horizontally, or both, and allows the user to drag the boundaries.

Technically, a **SplitPane** is at the bottom of the Z order of the panes it manages, and adjusts the framing ratios of the windows according to the split position.

CHAPTER 6 Introduction to OLE Programming

This chapter presents OLE programming in Smalltalk MT. OLE is a vast topic; so we will focus on the Smalltalk implementation and some of the most often used programming topics.

The first section introduces some OLE programming fundamentals. It assumes that you are already familiar with DCOM technology. If you are interested in more detailed information concerning DCOM, refer to the DCOM documentation in the distribution kits for Windows NT and Windows 98 or on Microsoft's Internet site. Useful books on the subject are "Inside OLE, Second Edition" by Kraig Brockschmidt, "Understanding ActiveX and OLE" by David Chappell, and "Inside COM" by Dale Rogerson, all published by Microsoft Press.

The following sections discuss using OLE controls, implementing OLE data transfer and OLE drag - drop.

Finally, a section on ActiveX components explains how to implement ActiveX controls in Smalltalk, and a troubleshooting section discusses OLE debugging and diagnosis tools.

The Component Object Model (COM)

Abstract

COM (Component Object Model) is the foundation of OLE. The component object model is a binary communication format that closely resembles a C++ object. All OLE components communicate through COM interfaces.

A COM object exposes an interface to the outside world. A globally unique identifier, the GUID, which is assigned once when an interface is designed, identifies interfaces. An interface consists of a set of functions that can be invoked. It is generally possible to obtain (query for) other interfaces, so it is easy to extend the functionality of an object.

The salient property of an interface is that the implementor must support all functions in the interface. Therefore, it is not possible to only implement a subset of the interface functions.

Smalltalk Implementation

The implementation in Smalltalk mimics the C++ model. A COM object maintains a table of function pointers in an instance variable. The function pointers point to exported instance methods that are implemented as C++ member functions, meaning that the first parameter is the receiver (*this* in C++ and *self* in Smalltalk). The table is built when the class is initialized, meaning that creating a COM instance at runtime is very fast.

COM initialization

Every COM class requires an `initIID` method that sets the IID of the class, and an `initVTBL` method that defines the names of the exported functions. For example, **IAdviseSink** has the following class initialization methods:

```
initIID
    "
    Private - Generated method. Initialize the interface's IID
    "
    IID := GUID IIDFromString: '{0000010F-0000-0000-C000-000000000046}'
```

The IID of a COM interface can usually be found in the IDL file that describes the interface.

```
initVTBL
    ^super initVTBL,#(
        OnDataChange:with:with:
        OnViewChange:with:with:
        OnRename:with:
        OnSave:
        OnClose:
    )
```

The virtual table of **IAdviseSink** contains the methods inherited from **IUnknown** and adds five additional methods. The declaration lists the exports, so the first parameter to each method is always the **this** pointer (in C++ terminology).

COM creation

A COM object can either encapsulate an external interface or expose Smalltalk functionality to external code. In the first case, it is created from a pointer (`fromPointer:`) or from an address (`fromAddress:`). Otherwise, it is simply allocated using `new`. The creation method determines whether the virtual table being used is from an external interface or the class table. Once it has been created, the COM object can be used in both ways (inside out or outside in) without the caller worrying about the nature of the object.

Reference counting

COM uses reference counting. A COM object is de-referenced with the message `Release` and referenced with `AddRef`. Smalltalk MT extends the reference counting mechanism to the owner of a COM object. The implementations in **WinEventHandler**, **OleControl**, and **OleControlContainer** encapsulate reference handling, so it is generally transparent to applications that do not deal directly with COM objects.

Each COM instance maintains its own reference count. When the reference count reaches zero, the owner is also released. This does not necessarily mean that the de-referenced COM object can be discarded. In most cases, the owner maintains a strong Smalltalk reference to the COM objects it exposes, and it is only when the reference count of the owner reaches zero that everything can be garbage collected. As a

consequence, Smalltalk referencing and COM reference counting are distinct topics, the only thing that can be said is that a COM object cannot be garbage collected as long as its reference count is not zero.

COM method implementation

When implementing a COM object, three possibilities arise:

- The COM object is used as a callback interface that exposes Smalltalk functionality.
- The COM object encapsulates an interface implemented by external code
- A combination of the two possibilities above.

In the first case, it is only necessary to implement the exported methods (.EXPORT). The second case only requires the methods that call the external object (OUT category), while the third case requires both.

The COM classes under **IUnknown** are mostly shallow implementations that delegate messages to the owner, if such an owner exists. The owner must implement a subset of the interface, and does not have to implement the methods that are handled in a generic way by the COM object.

Threading Models

COM and DCOM (distributed COM) use RPC to communicate across process boundaries. DCE RPC use a pool of threads to manage incoming network calls, meaning that a COM object instance can be called simultaneously in arbitrary threads.

COM also provides simpler threading models that provide automatic synchronization of COM calls. The table below outlines the supported threading models:

Table 6-1 COM Threading Models

Threading Model	Description
Single-threaded main (STA Main)	Main thread for all instances.
Single-threaded apartment (STA)	One thread per instance
Free-threaded (MTA)	Multiple threads per instance
Both (Single-threaded or multithreaded apartment)	The new object should be initialized in the activator's apartment, with all subsequent method calls being serviced there as well.
Neutral	A neutral apartment supports execution of its objects on any thread type and is the recommended threading model for COM components and COM+ applications. On Windows 2000, the preferred Threading Model setting for non-visual components is Neutral.

The single-threaded model is the most restrictive. All COM object instances are created and accessed from the main thread. The single-threaded apartment model allows the application to create multiple threads, however all calls to an instance are made in the same thread. Finally, the free-threaded model allows an object to be created and accessed by arbitrary threads, including threads that have not been created by the application.

The threading model of a server is specified by the registry entry for objects of that class:

```
HKEY_CLASSES_ROOT\{clsid}\InprocServer32
       “ThreadingModel” = “Apartment”
```

Image Threading Suport

Smalltalk MT supports the full extent of free threading using COM (or any other technology, for that matter). In most cases, free threading allows a cleaner implementation and provides optimal performance and scalability.

As discussed above, free threading allows an arbitrary thread to call a COM object. This raises two issues:

- The Smalltalk MT garbage collector needs to be notified when a new thread allocates memory.
- Smalltalk code should not use thread-local (TLS) storage in the context of COM calls.

The base image implements global, thread-safe object references (used by `registerObject` and `releaseObject`).

The Apartment Model

By default, Smalltalk MT uses the single-threaded apartment model. One advantage of this model is that it avoids synchronization issues and allows each thread to use thread-local variables (TLS). Smalltalk MT supports both the single-threaded main model and the apartment model, i.e., it is possible to create additional threads that interact with COM interfaces.

The main drawback of the single-threaded model lies in the implementation used to synchronize threads. COM creates a hidden window in each thread and uses the function `PostThreadMessage` to communicate between threads. Therefore, a thread is forced to call `PeekMessage` and `DispatchMessage` on a regular basis to ensure that messages are dispatched and that no deadlock occurs. This is normally the case for window applications that use COM functionality as clients, but may require some work in the case of non-GUI server applications.

Non-GUI applications need to wrap synchronization calls (such as `Event>>wait`) into a call to `WinApplication>>msgWaitForMultipleObjects:`. The parameter to this method is a **WordArray** that contains the synchronization objects to wait upon and the call returns when any of the objects is signaled.

The free-threading Model

While free threading is generally the best choice for non-GUI server applications, it can be difficult to integrate with GUI applications because of limitations of the windowing subsystem. On the other hand, GUI applications already provide a message pump, so it is generally more convenient to use the apartment model for graphical applications.

COM Class Hierarchy

COM classes (also called interfaces) inherit from **IUnknown**. The compiler has special support for managing the virtual function table of a COM object. The virtual function table always reflects the Smalltalk hierarchy (COM itself is a binary protocol that does not use inheritance *per se*).

A COM interface can be used in one of two modes: as an object that wraps an external interface and as an object that exports methods. Both modes can also be combined. Beware that an **IUnknown** instance that wraps an external interface is always initialized as a weak pointer object, while an exported interface remains a regular pointer object.

How to...

Create an IUnknown instance

There are two ways to create an interface:

- Given a pointer to an interface, you use the class methods `fromAddress:` or `fromPointer:` to create an object that wraps the interface.
- The methods `new` and `owner:` create an interface that exports Smalltalk methods. In the second case, the interface object has an owner. The interface delegates certain OLE callbacks to the owner, which can otherwise be an arbitrary object.

For example, the code below (taken from `IDataObject class>>oleGetClipBoard`) creates an **IDataObject** from the clipboard contents:

```
oleGetClipBoard
    | ppDataObj hresult|
    ppDataObj := LONG localNew.
    (hresult := WINAPI OleGetClipboard: ppDataObj basicAddress) == S_OK ifTrue: [
        ^self fromPointer: ppDataObj.
    ].
    ^self setLastError: hresult
```

The following statement creates an **IDataObject** on a string:

```
    IDataObject new data: 'abc'.
```

Note When an IUnknown object wraps an external interface, it is created as a weak pointer. This means that its contents are not garbage collected. If you create a COM interface with additional instance variables, you must either make sure that the contents are referenced by some other means, or prevent them from being collected by calling `registerObject` on each element. In the second case, you must also call `releaseObject` on each element when the interface goes out of scope. This is best done in the `releaseObject` method of the interface, which is called automatically when the reference count reaches zero.

Create a new COM interface class

First, locate the COM interface to inherit from and create a subclass. There are two cases:

- A type library exists for the interface. Right-click on the new class and choose *Generate Interface from Type Library*. This opens a file dialog where you select the library (a .TLB file) that corresponds to the interface.
- Otherwise, you must declare the COM functions with their number of arguments in an `initVTBL` class method. The exported methods must appear in the VTABLE order as it is specified in the IDL or include file. Since Smalltalk COM interface inherit the VTABLE contents, the declaration must not include methods defined in superclasses.

This generates the GUID and VTABLE initialization methods, and optionally the exports and outbound methods.

Generate interface methods

A dialog lets you specify which methods the compiler should generate. In most cases, you'll need either outbound or inbound methods, but seldom both at the same time. You can generate the methods from the type library or from the `initVTBL` class method, the difference is that in the first case the method description (if available) is included.

When an interface class is initialized, a warning is printed on the transcript window for each method that is referenced by the virtual table but not implemented. You can ignore the warning for outbound interfaces, since the VTABLE contents are not used in this case. In the case of inbound (exported) interfaces, a warning indicates that a method call may fail at runtime because the implementor of an interface must provide all methods that are part of the interface.

Use GUIDs

In Smalltalk, globally unique identifiers are instances of GUID. GUID has class methods that create a GUID from a string such as a registry value or a C declaration.

Creating a GUID

A GUID is a 16-byte structure that uniquely identifies an interface or some other object. You can generate new GUID values with the method `createGuid` in GUID class. The GUID class method `fromBytes:` creates a copy of an existing GUID or GUID address, and `fromProgID:` answers the GUID that corresponds to a program name used by OLE automation.

Using Interface GUIDs

You can retrieve the GUID of an interface class by sending the message `iid`. It is also possible to pass an interface class directly to an API; the effect of this is to pass the GUID of the interface by reference.

Given a GUID, it is possible to retrieve the name of the interface it corresponds to with the method `name`. The method looks up the registry and returns the interface name if one is defined; otherwise it just prints the GUID string.

Comparing GUIDs

The method `isEqual:` compares two GUIDs, or a GUID against the address of a GUID and returns **true** or **false**.

OLE Allocations

General allocations use **IMalloc** to allocate and free supplemental OLE objects. Interfaces use reference counting and do not have to be freed explicitly.

OLE Strings

OLE uses two types of strings; BSTR and regular C strings. In both cases, Unicode is used. BSTR instances have a different encoding (the length field precedes the string data) and must be allocated and freed using `SysAllocString` and `SysFreeString`.

Interface implementations and Variants normally create BSTR, so there is usually no need to create a BSTR explicitly. `String>>fromBSTR:` converts a BSTR to a Smalltalk string and frees the BSTR.

OLE Controls

Abstract

An OLE control is a component that supports a set of interfaces that enable it to be inserted in and interact with OLE container applications. Since an OLE control has to support a large number of interfaces to qualify, Microsoft introduced the ActiveX specification, which reduces the number of mandatory interfaces.

OleControlContainer

Class **OleControlContainer** acts as an ActiveX control container. An instance of **OleControlContainer** encapsulates a single OLE control and exposes OLE automation on that control.

Like **OleControl**, **OleControlContainer** involves a large number of interfaces and methods, so we'll just discuss events, getting and setting properties, and invoking OLE methods on that object.

Events

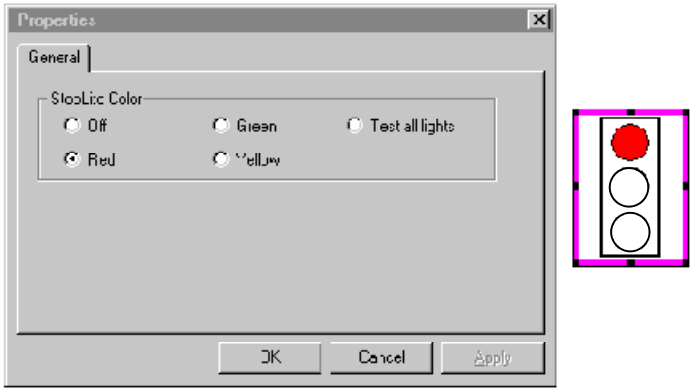
Events are received through the `m_IDispatchEvents` member and transit through the `oleEvent:with:` method. The arguments to the message are a dispatch id and a pointer to a `DISPPARAMS` structure. In most cases, the event has no parameters or the parameters are not of interest, which is the reason why the event handler just passes the raw pointer.

The `oleEvent:with:` implementation in **OleControlContainer** creates a regular event, using the event id as a notification code and the container's child identifier.

OLE properties

The methods `getProperty:` and `putProperty:value:` respectively get and set a property. It is always preferable to use the property get and set methods rather than method invocations because some implementations refuse to access a property through a method call.

Figure 6-1 OLE Property Page Sample



Note OLE properties, window properties and Smalltalk properties are different items.

OLE methods

OleControlContainer exposes a large number of `invoke:` methods. Each of the methods accepts a method name or dispatch id. The `invoke:[with:]` family of methods automatically create an array of variants to hold the parameters.

The return value of an invocation is either **nil** if the method did not return a value, or a **Variant**. If the method failed, an error **Variant** with the error code is returned. Therefore, the caller must check whether the return value is an error variant (VT_ERROR) or of some other type.

Using OleControlContainer

An application uses **OleControlContainer** like it uses a regular control. The control can be sized, shown and hidden; in short, it functions like a normal control.

Creating an OleControlContainer

The easiest way to create and initialize an **OleControlContainer** is to generate the code with the GUI builder. The interface builder takes a snapshot of the ActiveX component and serializes the control to a byte array. The creation method uses `loadFromBytes:` to re-instantiate the OLE control, including the properties as authored in the interface builder.

The method `insertObject:` can be used to insert an object at runtime. This is useful if the type of control is not known in advance.

Processing OLE events

Events fired by an ActiveX control are funneled through the standard event framework. For example, the code below installs a handler for the "AfterUpdate" event of the Calendar control:

```
addHandler: 1 "AfterUpdate" in: IDC_CONTROL1 selector: #onAfterupdate
```

The handler is implemented as:

```
onAfterupdate
    | time |
    time := (self childAt: IDC_CONTROL1) getProperty: 'Value'.
    (self childAt:IDC_DATETIMEPICK2) setSystemTime: time.
    ^NULL
```

Unloading an ActiveX control

The ActiveX control is automatically unloaded when the container closes. If an application wants to unload the control without closing the container, it can send `unload` to the container.

InPlaceFrame and OleContainerView

InPlaceFrame supports the document view architecture and acts as a frame container for embedded documents. **OleContainerView** manages an embedded document, which can be an OLE control. This architecture is appropriate for document based applications

that support embedded menus and frame negotiation (i.e., status bars and toolbars inserted by the active document object).

Dispatch Interfaces

An OLE control exposes methods and properties through an **IDispatch** interface.

IDispatch

IDispatch is the generic dispatch interface. In addition to the **IDispatch** interface functions, it implements the methods depicted below.

IDispatch server methods

`typeInfo`
: The **typeInfo** attribute is initialized with the dispatch type information, an **ITypeInfo** interface. The type information is loaded from an external type library (TLB file) or from an in-memory resource. The type library describes the properties and methods that an object supports and allows the dispatch interface to marshal arguments and return values across process boundaries.

IDispatch client methods

`GetIDsOfNames:with:`
: This method maps a collection of names to an array of DISPIDs. Method invocation uses numeric values to identify member functions and properties. GetIDsOfNames:with: uses the information in the type library to map names to identifiers.

`GetTypeInfo:`
: Returns an interface on the type information (an **ITypeInfo**).

`GetTypeInfoCount`
: Returns the number of type information interfaces supported by the object.

`GetIDsOfNames:with:`

`Invoke`
: `Invoke:[...]` invokes methods and sets or retrieves properties. **IDispatch** provides several Invoke layers with increasing level of complexity. It also implements `putProperty:value:` and `setProperty` to access properties.

IDispatchAmbients

A container uses **IDispatchAmbients** to expose its ambient properties. Class **IDispatchAmbients** implements an **IDispatch** interface on a mapping of ambient properties. The mapping associates ambient property identifiers (`DISPID_AMBIENT_XXX` constants) with property values.

IDispatchEvents

A container uses **IDispatchEvents** to manage connection point containers that receive notifications from an OLE control. When the control fires an event, the **IDispatchEvents** interface sends `oleEvent:with:` to its owner, which in turn generates a Smalltalk event that is routed through the event handling framework.

The **IDispatchEvents** object needs to be initialized with the dispatch identifiers of the event interface. This information is obtained from the type library.

IDispatchMessage

This interface is used by Smalltalk ActiveX components to expose instance methods. Upon invocation, the **IDispatchMessage** object calls the owner with the method that corresponds to the specified dispatch id.

OLE Data Transfer

Abstract

OLE data transfer uses the **IDataObject** interface to transfer arbitrary data objects. Common mechanisms for transferring data are the clipboard and OLE Drag and Drop.

IDataObject

OLE data is transferred through **IDataObject**. The provider creates an instance of **IDataObject** and initializes it with as many data formats as it wishes to support. The consumer can enumerate the available formats with `EnumFormatEtc` and retrieve one with `GetData`.

IDataObject encapsulates an external `IDataObject` interface and also provides a convenient way to expose data objects to the outside world. **IDataObject** is self-contained and does not require the owner to implement callback functions.

Data formats

Each data object can be transferred in a given format. A data format is identified by a `CF_XXX` constant; for example, `CF_TEXT` designates unformatted text. OLE defines many data formats, and it is possible to define private formats.

Setting data and formats

An application sets the data, its format and the type of supported medium with `setData:format:tymed:`. The type of medium can be one of the `TYMED_XXX` enumeration constants; the most widely used are `TYMED_HGLOBAL` for binary data and `TYMED_GDI` for handles.

An **IDataObject** can contain data objects in different formats. The application first sets the most accurate format, so that the enumeration returns the most complete formats first.

Private data formats

Specifying a format upwards of `CF_CF_PRIVATEFIRST` and a medium type of `TYMED_HGLOBAL` best supports private formats. The system automatically frees the global pointer when the data is discarded (for example when the clipboard is emptied), so the application does not have to implement extra code for that purpose.

Enumerating Data Formats

The method `EnumFormatEtc:` automatically enumerates all data formats supported by the **IDataObject** instance and returns a collection of **FORMATETC** structures. For example, the code below retrieves the collection of cfFormat fields:

```
(clipData EnumFormatEtc:TRUE) collect: [ :e | e cfFormat ].
```

IDataObjectEx

Class **IDataObjectEx** is an extension of **IDataObject** that delegates most callbacks to the owner of the interface. In particular, it allows the owner to provide data on demand and supports advisory connections. Advise sinks are notified of changes to the object.

IDataObjectEx requires the owner to implement methods that process the callback functions.

Using the Clipboard

This section discusses using the OLE clipboard functions. It is also possible to use non-OLE clipboard functions to place data onto the clipboard. However, OLE interfaces provide a uniform mechanism that can also be used with OLE Drag and Drop.

Setting clipboard data

An application that wishes to place data onto the clipboard creates an **IDataObject** to hold the data in one or several formats. To make the data available, the application calls `IDataObject>>oleSetClipboard`.

Example:

The code below makes a bitmap object available in three formats:

- A private format. For the sake of simplicity, this example stores a string that contains the module name and resource name of the bitmap.
- A GDI bitmap.
- The text 'Bitmap'. Again, this is just to demonstrate the functionality.

```
| ifData bmp |

bmp := Bitmap loadResource: 101

       module: (HModule loadLibrary: 'resedt.dll').

ifData := IDataObject new.

ifData setData: ' 101 "resedt.dll"'  format: CF_PRIVATEFIRST tymed:
TYMED_HGLOBAL.

ifData setData: bmp format: CF_BITMAP tymed: TYMED_GDI.

ifData setData: 'Bitmap' format: CF_TEXT tymed: TYMED_HGLOBAL.

ifData oleSetClipBoard
```

To test the clipboard contents, you can for example open WordPad and click on *Edit|Paste Special*. The first format is a bitmap, the second unformatted text.

Retrieving clipboard data

An **IDataObject** interface on the clipboard data can be obtained with `IDataObject class>> oleGetClipBoard`. The storage medium that contains the data is retrieved via `GetData:medium:`.

In the case of a standard format such as `CF_TEXT` or `CF_HDROP` (a list of file names), `GetData:` automatically retrieves the data, so the code for retrieving unformatted text looks like:

```
| clipData data|
clipData := IDataObject oleGetClipBoard.
data := clipData GetData: CF_TEXT.
clipData Release.
data
```

Data types not supported by `GetData:` must be retrieved through the `STGMEDIUM` structure. Calling `GetData:medium:` with a `STGMEDIUM` structure initializes the structure and returns the format constant.

The following code retrieves a private data format:

```
| clipData data medium|
clipData := IDataObject oleGetClipBoard.
data := nil.
medium := STGMEDIUM new.
(clipData GetData: CF_PRIVATEFIRST medium: medium) notNil ifTrue: [
    clipData Release.
    data := String fromAddress: medium lockGlobalData.
    medium unlockGlobalData.
    medium close.
].
data
```

The caller must lock the global handle using `lockGlobalData`. This returns a pointer to the data. When the data is copied to safe storage, the global memory handle is unlocked using `unlockGlobalData` and the medium is closed.

Testing clipboard data

An application often needs to determine whether a data transfer can be successful. The method `QueryGetData:` in **IDataObject** returns a **Boolean** that indicates the probable outcome of a subsequent `GetData` call. The method is often used to enable or disable an *Edit*|*Paste* menu item.

The code below checks the clipboard for the `CF_TEXT` format:

```
| clipData  isOk|
clipData := IDataObject oleGetClipBoard.
isOk := clipData QueryGetData: CF_TEXT.
clipData Release.
isOk
```

Implementing OLE Drag and Drop

Abstract

OLE Drag and Drop is a uniform data transfer mechanism that works across processes. Data can be transferred via drag and drop as well as through the clipboard. Implementing OLE Drag and Drop is straightforward and in fact easier to support than non-OLE Drag and Drop because OLE already performs low-level work such as capturing the mouse.

Preparing an Application for Drag and Drop

Before a window can act as a drop target, it must be registered with the OLE Drag and Drop handler. This is typically done in the `initWindow` method of the parent window.

The frame window creates an **IDropTarget** interface for each child window that may act as a drag target. The interface can be tagged with an identifier so that the parent can identify the drag target later. The message `registerWindow:` assigns a window to the **IDropTarget** interface and registers it with OLE Drag and Drop.

The code fragment below illustrates this:

```
m_dropTarget := IDropTarget owner: self id: ID_IDROPTARGET0.
m_dropTarget registerWindow: (self childAt: IDC_LIST1).
```

Processing a dragging Operation

Initiating a dragging operation

The parent window needs to handle dragging events such as the ones sent by the common controls. For example, a **TreeView** sends a `TVN_BEGINDRAG` notification when the user starts dragging an item.

The parent window or the application prepares for dragging as follows:

1. It creates an **IDataObject** on the data to transfer.
2. It creates an **IDropSource** interface and initializes the cursors for the different drag effects (*none*, *copy*, *move*, *link* and *scroll*). Setting the cursors is optional; the **IDropSource** object uses a default OLE cursor when no drag effect cursor has been defined.
3. Finally, it invokes `doDragDrop` on the **IDropSource** source, specifying the data object and the allowed drag effects. The method returns with the final effect when the dragging operation is complete. If the drag effect is a move, the application deletes the source or performs any other actions related to the drag effect.

Example:

```
onBegindrag: lpnmtv
    | pnmtv iml tv ptvitem itemData iData iSource itemImage|
    NMTREEVIEW fromAddress: lpnmtv pointer: (pnmtv := Pointer localNew).
    TVITEM fromAddress: pnmtv itemNew pointer: (ptvitem := Pointer localNew).
    tv := self childAt: IDC_LIST1.

    " Retrieve the image list used by the tree view "
    iml := tv getImageList: TVSIL_NORMAL.

    " get the item text and image index "
    itemData := tv getItemText: ptvitem hItem.
    itemImage := (tv getItem: ptvitem hItem) iImage.

    " create a data object on the item's data "
    (iData := IDataObject new) data: itemData.

    " create a drop source interface "
    iSource := IDropSource new.

    " set the drag cursors to display while dragging "
    iSource cursorCopy: (iml getIcon: itemImage flags: ILD_TRANSPARENT).
    iSource cursorMove: (iml getIcon: itemImage flags: ILD_SELECTED).
    iSource doDragDrop: iData effect: DROPEFFECT_COPY|DROPEFFECT_MOVE
```

Drop target notifications

- The drop target sends `dragEnter:[...]` to its owner when the dragging enters the drop target window to which the **IDropTarget** interface belongs. The parameters comprise the drop target, the drop source, keyboard flags as well as mouse cursor coordinates. The parent window returns a drop effect that indicates the result that a drop would have.
- It then sends `dragOver:[...]` while the object is dragged over the window. The owner of the drop target interface processes this message and returns a drop effect. It highlights the drop target to give the user a visual clue about where the drop would occur. For example, a **TreeView** can set the drop target item with `selectDropTarget:`.
- When the dragged object leaves the window, the owner receives a `dragLeave:` message. The application must invalidate all data pertaining to the dragging operation and reset its appearance to clear any drop target highlights.
- If the user releases the mouse while a drop is enabled, the owner receives a `drop:[...]` message and performs the dragging operation. The data is retrieved with `GetData:`, passing an appropriate format to the function. The return value indicates the type of operation (*none*, *copy*, *move*, *scroll* or *link*).

dragEnter

The simplest form for `dragEnter:[...]` is to accept any source:

```
dragEnter: anIDropTarget
    with: anlDataSource
    with: grfKeyState
    with: pt_x
    with: pt_y
    "
    Indicates whether a drop can be accepted, and, if so,
    the effect of the drop.
    "
    ^DROPEFFECT_COPY
```

dragOver

```
dragOver: anIDropTarget
    with: grfKeyState
    with: pt_x
    with: pt_y
    | effect tv hItem|
    tv := anIDropTarget window.
    (hItem := tv hitTest: pt_x with: pt_y) == NULL ifTrue: [
        tv selectDropTarget: NULL.
        ^DROPEFFECT_NONE
    ].
    effect := grfKeyState & MK_CONTROL ~~ 0 ifTrue: [DROPEFFECT_MOVE] ifFalse:
[DROPEFFECT_COPY].

    tv selectDropTarget: hItem.
    ^effect
```

dragLeave

```
dragLeave: anIDropTarget
    "
    Removes target feedback and releases the data object.
    Return Value:
        S_OK or an error value.
    "
    anIDropTarget window selectDropTarget: NULL.
    ^S_OK
```

drop

```
drop: anIDropTarget
    with: anlDataSource
    with: grfKeyState
    with: pt_x
    with: pt_y
    | tv hItem dropTarget szItemText iImage|
    tv := anIDropTarget window.
    dropTarget := tv getDropHilite.
    tv selectDropTarget: NULL.
    (hItem := tv hitTest: pt_x with: pt_y) == NULL ifTrue: [^nil].

    " get the item text "
    szItemText := anlDataSource GetData: CF_TEXT.
    iImage := #('yellow' 'pink' 'red' 'orange' 'cyan' 'green' 'blue')
        indexOf: szItemText.
    iImage ~~ 0 ifTrue: [
        iImage := iImage - 1.
        tv mask: TVIF_TEXT|TVIF_PARAM|TVIF_IMAGE|TVIF_SELECTEDIMAGE
            state: TVIS_EXPANDED
            stateMask: TVIS_EXPANDED
            iImage: iImage
            iSelectedImage: iImage.
        tv addItem: szItemText to: dropTarget param: 0.
        tv expandItem: dropTarget flag: TVE_EXPAND.
    ].
    ^DROPEFFECT_COPY
```

Revoking Drag and Drop

When the window is closed OLE drag and drop is revoked using `revokeDragDrop` on the **IDropTarget** interface. The code typically looks like:

```
destroyWindow
    m_dropTarget notNil ifTrue: [
        m_dropTarget revokeDragDrop.
        m_dropTarget := nil.
    ].
```

Implementing OLE File Drag and Drop

Accepting File drag-drop with OLE is like implementing general-purpose drag and drop. The data format for dragging files is `CF_HDROP`.

Initializing the parent window

The parent needs to create an **IDropTarget** instance for every drop target. An application that accepts files usually enables the frame, so the code looks like:

```
    m_dropTarget := IDropTarget owner: self id: ID_IDROPTARGET0.
    m_dropTarget registerWindow: self.
```

dragEnter

The `dragEnter:[...]` method enumerates the formats of the data object, looking for `CF_HDROP`. It may further constrain the drop to a single file.

The example code below retrieves the collection of files, rejects multiple file drops, and stores the files in a property of the parent window. The data is later needed to respond to the `dragOver:[...]` message.

```
dragEnter: anIDropTarget
    with: anIDataObject
    with: grfKeyState
    with: pt_x
    with: pt_y
    | formats hdrop files |
    formats := anIDataObject EnumFormatEtc: DATADIR_GET.
    formats isNil ifTrue: [ ^DROPEFFECT_NONE ].
    hdrop := formats detect: [ :e | e cfFormat == CF_HDROP ].
    hdrop isNil ifTrue: [^DROPEFFECT_NONE].

    files := anIDataObject GetData: CF_HDROP.
    files size ~~ 1 ifTrue: [ ^DROPEFFECT_NONE ].
    self propertyAt: #IDataObject put: files.
    ^DROPEFFECT_COPY
```

dragOver

As explained above, the `dragOver:[...]` implementation looks like:

```
dragOver: anIDropTarget
    with: grfKeyState
    with: pt_x
    with: pt_y
    (self propertyAt: #IDataObject) notNil ifTrue: [^DROPEFFECT_COPY].
    ^DROPEFFECT_NONE
```

dragLeave

`dragLeave:` removes the property that stores the data object and returns `S_OK`.

```
dragLeave: anIDropTarget
   self removeProperty: #IDataObject.
   ^S_OK
```

drop

The method `drop:[...]` finally opens the dropped file.

```
drop: anIDropTarget
    with: anIDataObject
    with: grfKeyState
    with: pt_x
    with: pt_y
    | files |
    files := self removeProperty: #IDataObject.
    files size ~~ 1 ifTrue: [ ^E_UNEXPECTED ].

    self fileOpen: files first.
    ^S_OK
```

Implementing Simple File Drag-Drop

A frame window can also choose to implement non-OLE file drag and drop. This requires the following:

- The extended style (returned by the instance method `windowExStyle`) must include `WS_EX_ACCEPTFILES`.
- The window must implement `WM_DROPFILES:with:`

Accepting file drops using non-OLE file drag-drop requires less code than OLE drag-drop. Applications that do not wish to support OLE may want to use simple drag-drop.

Processing WM_DROPFILES

The first parameter of this message is a data structure with the file names that have been dropped onto the window's caption. The structure is opaque and must be accessed using Win32 functions.

The implementor calls `dragQueryFile:at:` to retrieve the file name at a specified position and `dragFinish:` to release the drag-drop data structure.

Example:

```
WM_DROPFILES: hDrop with: lParam
    "
    The WM_DROPFILES message is sent when the user releases the left mouse button
    while the cursor is in the window.
    "
    | szFile |
    self queryClose ~~ NULL ifTrue: [ ^nil ].

    szFile := self dragQueryFile: hDrop at: 0.
    self title: szFile.
    self fileOpen: szFile.
    self dragFinish: hDrop.
    ^NULL
```

Differences between Simple and OLE Drag and Drop

Unlike the more elaborate OLE protocol, the simple file manager type drag-drop is an all-or-nothing operation; either the recipient accepts files or not. In particular, the user does not get visual feedback about whether a file drop would be successful or not.

On the other hand, OLE drag drop does not recognize file drops from the old style File Manager, so the user must use the Windows Explorer.

ActiveX Component Framework

ApplicationProcess Entry Point

The class method `dllEntryPoint:with:with:` in ApplicationProcess is called by the operating system when the DLL is loaded, about to be unloaded, or when threads are created and destroyed.

The DLL handler processes `DLL_PROCESS_ATTACH` and `DLL_PROCESS_DETACH` notifications to respectively initialize and clean up the Smalltalk environment, including a concrete subclass of **InProcessServer**. It also processes `DLL_THREAD_ATTACH` so that the client can create multiple threads in the apartment-threading model. For example, Microsoft Internet Explorer creates a new thread for each window.

Differences between in-process and stand-alone Servers

ActiveX components are always in-process (DLL) servers. However, it may be desirable to implement a component as both a stand-alone and an in-process server. This paragraph depicts the considerations between both implementations.

In the case of a stand-alone server, the component registers an instance of **IClassFactory**. A client creates an instance of **IClassFactory** and calls the exported method `CreateInstance`. `CreateInstance` first calls `registerThread` in **Processor** to make sure the thread is registered in the Smalltalk system. Any other entry point that may be called from a non-registered thread must also call `registerThread` before any further processing takes place. Calling `registerThread` more than once has no effect.

An in-process server handles the `DLL_THREAD_ATTACH` notifications and automatically registers any thread that is created in the process after the DLL is loaded. Therefore, it is not necessary to call `registerThread` explicitly.

InProcessServer

InProcessServer implements a COM in-process server. The subclass **InProcessControlServer** implements an OCX control server (which may implement more than one ActiveX control). InProcessServer implements the mandatory exports for a COM in-process server. On startup (i.e., `initializeLibrary`), it creates an instance of itself that will implement the server functionality. To extend the functionality, you can create subclasses of InProcessServer.

The entry points `initializeLibrary` and `uninitializeLibrary` are called when the DLL is respectively loaded and unloaded. Class **InProcessServer** maintains a thread-safe lock count that determines when the DLL can be unloaded. An instantiated ActiveX control is responsible for increasing the lock count upon creation and releasing it when all references to the object are gone.

InProcessServer maintains an object table that contains the IIDs and server classes the library exposes, as well as a set of registry entries that allow the library to register and de-register at runtime. The contents are automatically generated when the class method `initialize` of an **OleControl** is called. Typically, an OLE control would be initialized when loaded into the development image, therefore initializing the registry data maintained by the InProcessServer subclass. When the target DLL is built, the registration data becomes static data in the DLL.

Table 6-2 InProcessServer Exports

Function	Description
DllCanUnloadNow	Answers whether the DLL from which it is exported is still in use.
DllGetClassObject	Retrieves a class object from the DLL.
DllRegisterServer	Supports self-registration for ActiveX controls.
DllUnregisterServer	Supports self-deregistration for in-process servers.

Using InProcessServer

The ActiveX framework takes care of registration, de-registration, and maintaining the lock count. To extend InProcessServer, create a subclass and reimplement the instance methods as needed. The methods that you are most likely to extend are `initializeLibrary` and `uninitializeLibrary`. You can add code that registers window classes here.

OleControl

OleControl is the abstract superclass of ActiveX controls. It implements all the code required to get a working ActiveX component.

Creating an ActiveX Control

Implementing an OleControl subclass

To implement an ActiveX component, you must first create a subclass of **OleControl**.

The minimal set of methods that you must implement is:

initialize

If you expose OLE methods, you must register them with the primary dispatch object. The message `exposeMethod:dispID:` takes a selector and a dispatch ID (which is defined in the IDL file). The code for the **StopLite** sample is:

```
initialize
    super initialize.
    m_size x: 75 y: 105.

    m_lights := 1.
    m_IDispatchPrimary
        exposeMethod: #aboutBox dispID: DISPID_ABOUTBOX;
        exposeMethod: #nextLight dispID: 2;
        exposeMethod: #lightColor dispID: 0;
        exposeMethod: #lightColor dispID: 1;
        exposeMethod: #invalidateControl dispID: DISPID_REFRESH.
```

You must, of course, also implement the corresponding methods.

onDraw:with:with:

This method must implement drawing. The first parameter is the device context to draw upon, the second defines the bounding rectangle, and the third is the rendering device context (such as a printer DC).

getCustomVerbs (optional)

Answers a **WordArray** of custom verbs. Each verb is defined by the following structure:

```
typedef struct tagVERBINFO {
    LONG    lVerb;                // verb id
    ULONG   idVerbName;           // resource ID of verb name
    DWORD   fuFlags;              // verb flags
    DWORD   grfAttribs;           // Specifies an OLEVERBATTRIB enumeration.
} VERBINFO;
```

It is almost the same as the `OLEVERB` structure, only that the verb text has been replaced with a string resource identifier, which allows better localization. It is convenient to define verbs as a **WordArray**. For example, the custom verbs for **StopLite** are:

```
getCustomVerbs
    "
    Answers an Array of custom verbs.
    "
    ^Array with: (#[CTRLIVERB_ABOUTBOX IDS_ABOUTBOXVERB 0
OLEVERBATTRIB_ONCONTAINERMENU])
```

Firing events

To fire an event, you call `fireEvent:`, which takes an event id as parameter. The event id must be defined in the IDL file.

initServer

This method initializes static data that is used for registration and at runtime. You must generate GUIDs for the class identifier, type library identifier, event interface, and primary dispatch interface. In addition, the control's name and version attributes must also be set.

The code below is taken from **StopLite**:

```
| regMap |
clsID := GUID IIDFromString: '{20048BB3-DB68-11CF-9CAF-00AA006CB425}'.
typeID := GUID IIDFromString: '{20048BB0-DB68-11CF-9CAF-00AA006CB425}'.
eventID := GUID IIDFromString: '{20048BB2-DB68-11CF-9CAF-00AA006CB425}'.
primaryDispatchID := GUID IIDFromString: '{20048BB1-DB68-11CF-9CAF-
00AA006CB425}'.
progIDName := 'Smalltalk.StopLiteControl'.
lVersion := Integer loword: 0 hiword: 1.
```

Once the attributes (class instance variables) have been set, you call `initRegMap` to create a collection that defines the registry entries for the control. The map is then used by the property page registration, if applicable, and the final registration, which looks like:

```
InProcessServer
    addObject: clsID class: self;
    registryEntries: regMap
```

See also *Implementing Property Pages* below for how to register property pages for the control.

Implementing Property Pages

In reasonably simple cases, you don't have to implement a property page yourself. Given that the control's properties are known, **OlePropertyPage** manages setting and getting of exposed properties automatically. You just have to define a dialog template, which can contain radio buttons, check-boxes, combo-boxes and edit fields to display and modify the properties. If this is not sufficient (for example, because the property page uses other controls or performs custom validations), you may subclass **OlePropertyPage**.

A property page is created at runtime by the container and must be registered in the `OleControl>>initServer` method. The method to add a property page is:

```
addPropertyPage: szGUID
    title: szTitle
    docString: docString
    helpFile: szHelpFile
    helpContext: dwHelpContext
    dialogClass: dlgPageClass
    dialogTemplate: idTemplate
    dispIDMappings: dispIDMappings
    regName: szRegName
    registryEntries: regMap
```

You must generate the GUID for the property page. The `docString` parameter is currently not used. The `helpFile` and `helpContext` parameters define context sensitive help for the property page. The next parameter, `dlgPageClass`, is usually **OlePropertyPage** or a custom subclass of **OlePropertyPage**. Follows the template id of the property page and the property mappings, a **WordArray** that consists of child control identifiers and property identifiers. For example, **StopLite** uses:

```
#[IDC_OFF 1]
```

The control `IDC_OFF` is associated with the property index 1. All subsequent radio buttons in the group are automatically associated with consecutive property indices.

The implementation of the method `addPropertyPage:` creates an **OlePropertyPageInfo** object that creates an instance of the associated **OlePropertyPage** class at runtime.

XFactory

XFactory is a specialized **IClassFactory** that works in conjunction with **InProcessServer** to instantiate an object from the **InProcessServer**'s object table.

XFactory sends the message `createInstance` to the object being created and there is normally no need to modify or subclass XFactory.

Creating an OCX

DEF exports file

In most cases, you can use the standard DEF file for ActiveX controls.

An OCX requires the exports below:

```
LIBRARY   StopLiteControl
EXPORTS
    InProcessControlServer>>DllCanUnloadNow :: DllCanUnloadNow @1
    InProcessControlServer>>DllGetClassObject:with:with: :: DllGetClassObject @2
    InProcessControlServer>>DllRegisterServer :: DllRegisterServer @3
    InProcessControlServer>>DllUnregisterServer :: DllUnregisterServer @4
```

Replace **StopLiteControl** with the name of your OCX and save it under the name of your library, using the DEF extension.

Creating distribution files

You can distribute your OCX as a cabinet file that uses the standard ActiveX setup. Please refer to the ActiveX SDK for up-to-date information.

Every Smalltalk executable, DLL or EXE, requires the runtime file `STRTDLL`*`XX`*`.DLL`. Given the size of this file, adding it always to the cabinet file won't make a significant difference, so you can avoid the extra step required to download shared files separately.

Testing ActiveX Components

Overview

Testing components is critical because an ActiveX control:

- Is distributed in binary form.
- Must work with a variety of containers.
- Involves a large set of interfaces.

Steps

The first step is to develop a container application that tests all aspects of the control. This allows you to test both the container and the control in the Smalltalk environment.

You can use the GUI builder to generate a skeleton that loads an OLE control. In order to load the control from the image, replace the `loadFromBytes:` message with `loadOleControl:`, and pass an instance of the control as parameter. For example, a method that loads the **StopLite** control from the current image reads like:

```
initChildWindows
| ctrl |
ctrl := StopLiteControl createInstance.
ctrl AddRef.
(OleControlContainer openWindow: IDC_CONTROL1
    title: NULL
    exStyle: 0
    style: WS_CHILD|WS_VISIBLE
    parent: self
    framing: (FrameMatrix scale: #(0 0 0 0) offset: #(35 24 242 133)))
    loadOleControl: ctrl.
ctrl Release.
    "loadFromBytes: #[
      16r20048bb3 16r11cfdb68 16raa00af9c 16r25b46c00
      16r3c4571aa 16r00000000 16r00001565 16r00000b44
      ] size: 32."
```

This code skips the DLL loading and **IClassFactory** creation methods, and instead uses an object from the image. From that point on, the code interfaces are exactly the same as in the general case. In particular, the flow of control transits through all OLE interfaces and exports.

Make sure that you test with all OLE debugging messages turned on, and check that the reference counting is working properly (i.e., the component is properly unloaded when the container closes).

Once the component is running as expected, it is time to test it with other containers such as the test container application TstCon32 from MSVC or Visual Basic. The test steps above should have eliminated obvious defects. The debug messages are your primary source of information, and you may look for failed `QueryInterface` invocations or interface calls that are not in the expected order.

Troubleshooting OLE

The OLE implementation in Smalltalk MT comes with a large number of diagnostic and debugging messages. You can enable some or all of the messages from the Smalltalk *image properties* sheet. Note that the debug traces include the name of the executable and the thread identifier of the current thread.

Note	A common error cause is that a control cannot be inserted into window, or fails when saved to or restored from storage. Re-registering the control often helps.

CHAPTER 7 The Development Process

We will see in this chapter how to build executable applications. The first part examines the requirements and gives step-by-step examples. The examples detail every step and do not use the Interface Builder in order to give a more general view.

The second part discusses runtime-specific topics, including how to add resources to the executable. Some of the topics overlap with Chapter 2, *The Interface Builder*.

Creating an Application

In this chapter, we will examine what makes up an executable application and the steps required generating an executable image.

The last part, *Console Applications*, discusses Console (non-GUI) programming. This is useful for programming small command-line utilities, or more generally applications that do not require a graphical interface.

What is an Application?

We use the term application to designate a set of classes, methods and external resources that cooperate to perform a certain task. The target may, but does not have to, be an executable program.

An application consists of:

- Smalltalk source code
- External resources such as icons, menus, dialog templates ...

In Smalltalk MT, external resources are usually stored in a DLL that can be loaded by the development image. When the final runtime is generated, the resources are incorporated into the executable.

A GUI application generally involves the following classes:

- A **FrameWindow** subclass that displays the top-level window.
- One or more child windows of the frame.
- Dialog boxes.
- Possibly an owner of the frame. The owner handles events not processed by the frame.

Creating a New Project

You use the Project Browser to create a new project that will hold your source code. You can then save your code to disk, so you are not tied to a particular development image.

Using Resources

Resource items

Although external resources are not mandatory, most applications will use external resources at some point prior to shipment. External resources must be compiled to a resource-only DLL, which is then loaded by the application. The resources that are commonly used by an application include:

- Menus
- Icons
- Bitmaps
- Dialog boxes
- Version resources
- String tables

The built-in interface builder can generate in-memory menus and dialog resources, and the other resources are rather static and have little or no incidence on the application code. This means that you can start with a simple DLL that only contains the static resources and keep often changing items in the Smalltalk image.

Using third-party resource editors

You can use commercial tools such as MSVC or simply the SDK tools that come with the Win32 SDK. In both cases, the output consists of a resource description file and an associated *include* file. The next step is to compile the resources to a DLL that can be loaded into the development image, for example, using a *makefile* generated by the interface builder in Smalltalk MT.

After installing the resource include file as a pool dictionary in your GUI classes, you can load resources from the DLL. The Symbol Editor lets you import include files and convert them to pool dictionaries.

Loading resources

Every **FrameWindow** and **WinApplication** subclass has an instance variable that points to the module of the application. This module is used when loading resources. By default, a **FrameWindow** uses the module of the current application (as referenced by the thread-local variable `thisApplication`).

However, you can also override it in your window initialization method and load a specific resource module.

```
initialize
    "
    Private - Sets the resource file for this instance.
    "
    super initialize.
    m_module := HModule loadLibrary: 'myresource.dll'.
```

If you register your window class, you must also reference this DLL when you register the class:

```
registerClass
    ^self registerClass:
        (HModule loadLibrary: 'myresource.dll')
```

When the target executable is generated, the resources that are in the DLL are integrated into the EXE file, so that you do not have to ship the DLL along with your application. However, this means that the previously defined initialization methods should do nothing at runtime. You achieve this by placing them under the category `.INTERNALDEV`. The image builder ignores methods that belong to this category, so they do not appear in the target executable.

Registering the window class

`RegisterClass` defines attributes of the window; the name under which it is registered, the window class style, the background brush and the cursor to display in the client area. In most cases, you will at least associate a custom icon with the class, so registering your class is required.

Once a class is registered, you can create window instances (using `CreateWindow`).

Hello World: a simple example

The purpose of this simple program is to demonstrate the various ways to handle events. The Hello World program opens a window and prints `'Hello World'`.

Sample 1: the C Approach

This solution implements the sample the way one would program it in C or C++.

First, we need a frame window. So, let's make a subclass of **FrameWindow** and name it **HelloWorldWindow**.

```
FrameWindow subclass: #HelloWorldWindow
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

You can open it with:

```
    HelloWorldWindow1 new open
```

Secondly, we have to implement `WM_PAINT` in **HelloWorldWindow** to paint a `"Hello World"` in the client area.

```
WM_PAINT: wparam with: lparam
    "
    Private - Process the paint message.
    "
    | hDC ps |
    ps := PAINTSTRUCT new.
    hDC := DeviceContext value:
        (WINAPI BeginPaint: m_handle with: ps).
    hDC textOut: 'Hello World' x: 10 y: 10.
    WINAPI EndPaint: m_handle with: ps.
    ^NULL
```

To open the window, evaluate:

```
       HelloWorldWindow new open
```

Sample 2

Locate **GraphicsWindow**. This window does just what we want; it handles the `WM_PAINT` message and invokes its `drawOn:` method.

Instead of making the frame window a direct subclass of **FrameWindow**, we make it inherit from **GraphicsWindow** and re-implement `drawOn:` to handle the drawing.

```
drawOn: hDC
    "
    Private - Process the paint message.
    "
    hDC textOut: 'Hello World' x: 10 y: 10.
```

Sample 3

There is an even easier way: since **GraphicsWindow** raises the event `#paint`, all we have to do is to install an event handler that draws the text:

```
GraphicsWindow new
    when: #paint perform: [ :hDC |
        hDC textOut: 'Hello World' x: 10 y: 10
    ];
    open
```

This code creates an instance of **GraphicsWindow** and installs the `#paint` handler. The source in **GraphicsWindow** shows that the event takes a device context as its first parameter.

SampleDialog: using an external GUI builder

You can easily incorporate resources generated by a GUI builder such as MSVC (note that other dialog editors would do the job as well). Once you have defined the script, you compile it to a resource-only DLL that you can load from the Smalltalk development image. You must also create a pool dictionary from the associated include file in order to be able to access the resources by their symbolic names.

The example below demonstrates how to create a dialog box:

1. Build your resource script with MSVC and save it to disk. This creates an resource (.RC) and an include (.H) file (if you use default values, it creates `SCRIPT1.RC` and `RESOURCE.H`).
2. Compile it to a resource-only DLL. To do so, copy `DUMMY.DEF` and `MAKEFILE` from the `RES` subdirectory of a sample, and edit the line "DLL=" in the `MAKEFILE` to make it refer to the resource file name (in our example, the line reads "`DLL=SCRIPT1`").
3. Copy the DLL to your working directory.
4. Create a subclass of **DialogBox** (named for example **SampleDialog**).

5. Load the constants defined in `RESOURCE.H` to a pool dictionary: open the Symbol Editor, click on *File Open*, locate *RESOURCE.H* and click on *OK*. You have to enter a name for the new pool dictionary (for example `SampleConstants`), then click *on Add to Class SampleDialog* under the *Pool* menu.
6. Implement the `initialize` and `openOn:` methods in the dialog.

The `initialize` method loads the library.

```
initialize
    "
    Private - Sets the resource file for this instance in
    the     development image.
    "
    super initialize.
    m_module := HModule loadLibrary: 'script1.dll'.
```

The `openOn:` method loads the dialog.

```
openOn: aWindow
    "
    Opens the receiver window with the specified owner.
    Parameters:
        aWindow    Owner of the dialog box.
    Return Value:
        This method does not return a value.
    "
    self dialogBox: IDD_DIALOG1 owner: aWindow
```

The `initialize` method is usually only intended for the development image, so you should set its category to `INTERNALDEV`. When you generate an executable, the resources are normally loaded from the executable image.

The Generic Sample Program

The **Generic** sample is derived from the Generic sample application in the Win32 SDK samples. It illustrates the basic steps needed in order to create a functional Windows application. We will reuse the resources and help file of the C sample.

The **Generic** sample illustrates the following:

- Basic steps needed to create a window
- Creating a pool dictionary from an include file
- Linking with a DLL

See also *A Generic Application* on page 129 for how to build the Generic example with the interface builder.

Step 1: Analysis

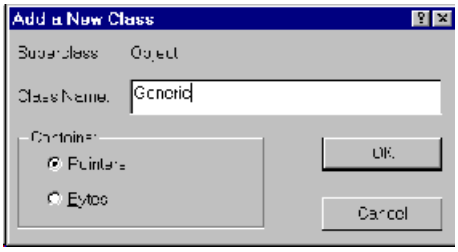
The application reacts to the following events:

- *File exit* closes the application.
- *Help* about brings up an about box.
- *Help topics* starts help for this application.
- Right mouse click brings up help menu.
- `WM_DISPLAYCHANGE` displays a message box saying that the display changed on a plug-and-play system.

Step 2: Create the Generic class

You must identify the appropriate superclass for the application. Here, it is **FrameWindow** because the generic window is a single document frame window. Select **FrameWindow**, choose the *Add Subclass* menu, and enter **Generic** as the name of the new class:

Figure 7-1 Adding Generic



Press *OK* to create the **Generic** class. At this point, you have an empty frame window. To open the window, evaluate:

```
Generic new open
```

Step 3: Load the resources

You can use any resource builder to build the resources. For example, the dialog editor that comes with the Windows SDK will do the job.

Create a resource DLL with the Generic resources.

Create a new pool dictionary with the resource constants as follows:

- Open the Symbol Editor (*Methods*|*Edit Symbols*).
- Open `GENERIC.H` from the File menu and create the **GenericConstants** pool dictionary.
- Finally, add the pool dictionary to the pool dictionary declaration string. You can now use the symbolic names in **Generic** methods.

Each window has a reference to the library from which it loads its resources. By default, the library of the current **WinApplication** instance is used. In our example, we must load the module `GENERIC.DLL` when the code is executed in the development image. In the runtime image, this code is unnecessary because the resources are integrated into the executable.

initialize

The initialize method is called automatically upon object creation. We subclass it in order to load the library.

We want to initialize the library handle:

```
hInstance := WINAPI LoadLibrary: ‘generic.dll’.
^super initialize
```

The last message invokes the initialize method in the superclass.

windowClassMenu

Method `windowClassMenu` answers the menu for the window:

```
^Menu fromHandle: (WINAPI LoadMenu: hInstance with: 'GENERIC')
```

windowClassTitle

Method `windowClassTitle` answers the title for the window:

```
    ^'Generic'
```

Step 4: Install the event handlers

The event handler map associates user interface events with handlers in the class. It is initialized in a class method:

```
initialize
    super initialize.
    self
        addHandler: IDM_EXIT         selector: #close;
        addHandler: IDM_ABOUT        selector: #about;
        addHandler: IDM_HELPTOPICS   selector: #helpTopics.
```

The initialize method must be called once to initialize the class.

```
close
    WINAPI FreeLibrary: hInstance.
    ^super close

about
    | aboutDlg  |
    (aboutDlg := DialogBox new)
        when: #create perform: [
            self initDialog: aboutDlg
        ];
        when: IDOK  perform: [ aboutDlg close ].
    aboutDlg dialogBox: hInstance
        template: 'ABOUTBOX'
        owner: self
```

The first *about* prototype simply brings up the About dialog box, so you add an empty `initDlg:` method in **Generic**. You can later fill the dialog with information.

Step 5: Linking with an external DLL

The *about* method uses APIs from `VERSION.DLL`, which must be linked with the executable image.

To install the API definitions, open the image properties dialog from the Transcript menu, and press the *Add* button. This leads to a *File Open* dialog box that lets you locate the DLL. In our case, it is in the Windows NT `SYSTEM32` subdirectory. You can view the APIs by double clicking on the module name. The button *Save* in the property page writes the imported API list to an ASCII file.

The next step is to save and rebuild the image by checking the *compress* option in the image save box. This rebuilds the import table and makes the imported APIs effectively available. You can now implement and test the `initDlg:` method.

Additional Steps

You should use the Project Browser to manage your source code. It lets you create a package that contains the code you implemented, as well as prerequisites that are required for your code to run. In this case, the project requires the library `VERSION.DLL`.

The Project Browser lets you also remove the code from the image. Alas, it does not unlink the library, so you must remove VERSION.DLL manually if you wish to minimize the size of the import section.

When the project is re-installed, the library is automatically loaded.

Note Certain versions of the development image may already be inked with `VERSION.DLL`.

Generating Executables

Overview

In most cases, generating a stand-alone executable from a Smalltalk project is straightforward.

The first step is to identify the components that make up the executable application. It is also necessary to perform initializations and bindings that are required by the application but are done automatically by the development image, such as initializing system libraries.

Application Entry Point

The application entry point is the top-level node from which references are computed. For an executable, this is the method `winMain:with:with:with:`.

The parameters to this method are:

Table 7-1 winMain Parameters

Parameter	Description
HModule	An HModule with the module handle to the current process. It can for example be used to load resources from the executable file.
HPrevInstance	Always zero, implemented for compatibility reasons.
CmdLineArgs	An Array of command line arguments.
NCmdShow	An integer that specifies the initial show state of the application’s main window.

An application must perform the following actions (bold marks indicate mandatory initializations):

Application Type / Action	GUI	OLE	Console
OleInitialize		X	
initCommonControlsEx	X		
Registering window classes	X	X	
AllocConsole			X

OleInitialize

The function `OleInitialize` initializes the OLE library. You must initialize the library before you can call OLE functions. The library is closed with `OleUninitialize`.

Example :

```
!ApplicationProcess * methods!
winMain: hModule with: hPrevInstance with: cmdLineArgs with: nCmdShow
    "
    Public - Calls initialization function.
    "
    " Initialize OLE "
    WINAPI OleInitialize: NULL.

    " Register Windows "
    Window registerClass.
    OleContainerView registerClass.
    SimpleContainerFrame registerClass: hModule.

    " Create application and run message loop "
    WinApplication new run: [SimpleContainerDoc new open].

    " Uninitialize OLE "
    WINAPI OleUninitialize.
! !
```

Note	There can be multiple `OleInitialize` calls, each balanced by an `OleUninitialize` call.

initCommonControlsEx

The method `initCommonControlsEx:` initializes the common controls library. You must initialize the library before you can use common controls.

The parameter to `initCommonControlsEx:` is a combination of flags that specify the Windows classes you wish to use.

Example:

```
Control initCommonControlsEx: ICC_WIN95_CLASSES|ICC_INTERNET_CLASSES|
ICC_DATE_CLASSES|ICC_USEREX_CLASSES|ICC_COOL_CLASSES.
```

Note 1	Do not register Win32 controls and common controls, as these are maintained by the operating system.

Note 2	Use the function InitCommonControls (`WINAPI InitCommonControls`) to initialize an older version (pre 4.70) of the common controls library.

Registering window classes

Each application-defined window class that you use must be registered before you can create a window of that class. This is done by sending the messages `registerClass` or `registerClass:` to the Smalltalk class that encapsulates the window. The optional parameter specifies a module from which resources such as icons are to be loaded. If the resources have been compiled into the executable, the initialization method passes simply the `hModule` parameter.

Note	Dialog boxes are pre-registered by Windows and must only be registered if they use for example a custom class icon.

AllocConsole

A console (non-GUI) process that is not explicitly marked as a console application must allocate a console for output. This topic is covered in more detail in *Console Applications* on page 359.

Running a GUI Application

After the external libraries have been initialized and the window classes registered, the entry point routine creates the main window and starts a message loop. The code that performs this typically looks like:

```
WinApplication new run: [SimpleContainerDoc new open].
```

The method `run` in **WinApplication** takes a block argument that evaluates to the main window. The **WinApplication** instance registers a handler for the `#destroy` event, so it knows when the main window is destroyed. It then enters the message loop. When the main window closes, the `#destroy` handler is invoked, which in turn exits the message loop by posting a `WM_QUIT` message to the thread's message queue.

The next statement is only evaluated after all windows in the main thread have been destroyed.

Running a Dialog Application

An application that only displays modal dialog boxes does not need to run a message loop because the Windows system manages the message loop while the dialog is running.

The `WINMAIN.SM` file of such an application typically looks like below:

```
winMain: hModule with: hPrevInstance with: cmdLineArgs with: nCmdShow
    "
    Public - Calls initialization function.
    "
    " init common controls if we use them "
    WINAPI InitCommonControls.

    " open a modal dialog "
    DialogApp new openOn: 0.
```

The first statement initializes the common controls library, which is necessary if the dialog uses any of the common controls. The next statement opens the modal dialog box, specifying a `NULL` owner. When the dialog box terminates, control-of-flow returns to the caller and the process terminates.

If the dialog is modeless (for example to enable accelerators), it is necessary to create an instance of **WinApplication** to run the message loop. The last statement must therefore be replaced with:

```
WinApplication new run: [ DialogApp new openModelessOn: NULL]
```

The block evaluates to the main window (in this case, an instance of **DialogApp**). When the main window is destroyed, the application loop exits and the process terminates.

DLL Entry Point

The DLL entry point is the method `dllEntryPoint:with:with:` in ApplicationProcess class. It is called when the module is loaded and unloaded as well as when a thread is created and destroyed. The second parameter, `fdwReason`, specifies the reason for which the method is called.

If a DLL needs to perform initializations or cleanup tasks, you must call your initialization / unitialization method under `DLL_PROCESS_ATTACH` respectively `DLL_PROCESS_DETACH`. After modifying the DLL entry point method, save it under the name `DLLMAIN.SM` in you project directory. It is the counterpart of the `WINMAIN.SM` file for executables, with the difference that it is optional for DLLs.

Note that it is important for a DLL to cleanup properly, since it may be loaded and unloaded dynamically during the lifetime of the process. Otherwise, straggling memory blocks and operating system resources may accumulate and, ultimately, prevent the process from functioning properly. It is also necessary to unregister any window classes that the component registered on startup.

Application Resources

It is important to include implicitly used resources from the development image, in addition to resources that are specific to the application. The following paragraphs discuss the most important standard resources that an application uses. The source of the resources is provided, so it is easy to build custom solutions.

The easiest way to set up directories is to use the resource generation option under *File|New|Resource Script*. This opens a dialog that lets you choose the resource items you wish to include.

Message file

The message file contains standard runtime messages that are used for error and warning messages.

To include the message binaries, add the following line to your resource script (RC file):

```
LANGUAGE 0x9,0x1
1 11 "stmsg.bin"
```

You generate the source for a message file as follows:

1. In the Class Hierarchy Browser, select WinException
2. Right-click on the class and select Build Message File..., click on No when prompted whether you want to include development messages

Prompter dialogs

If the application uses any form of **Prompter**, you must include prompter dialog templates. The template scripts are located in the `SUPPORT` subdirectory.

String tables

The standard string tables provide short tips for standard toolbar buttons and menu item descriptions.

Tooltip strings are used by the standard toolbar for *File* and *Edit* commands. When the mouse hovers over a toolbar button, the corresponding tooltip message is displayed.

Menu item descriptions are displayed in a status bar when the user scrolls through the menu. The framework automatically loads and displays the corresponding string if one is defined.

Using Symbols at Runtime

By default, and in order to reduce the size of the executable, a runtime image includes a simplified version of the symbol table that only contains strings for the class symbols. In this case, symbols cannot be added at runtime, so sending `asSymbol` to a string returns **nil** when the string is not the name of a class. Of course, the symbols themselves still exist, it is just not possible to obtain the string that corresponds to a symbol and vice versa, unless the symbol corresponds to a class.

It is also possible to generate an image that contains the full symbol table and implements all the symbol functions available in the development image. The cost is a larger executable, so you should only use this option if the application specifically requires the symbol functionality.

Headless Applications

A headless application is an application that does not display a user interface, or does so only optionally. Typically, these are server processes that run unattended and must not display messages to the user.

Process Types

A Smalltalk process can run as a regular Win32 application or as a console process. A console process displays a console window unless it has been started as a detached process.

A Win32 application has no obligation to create a message loop or a window. Therefore, server processes can implement the `winMain:with:with:` method in **ApplicationProcess** to start up the server environment.

Error Handling

All standard error handling is done in `MessageBox class>>logMessage:id:` and `MessageBox class>>printLastError:`. The latter is only called when debugging flags are turned on.

To suppress standard error messages (such as `doesNotUnderstand:`), you must re-implement `logMessage:id:`. For example, you can use `reportErrorEvent:` in **ApplicationProcess** to report an error to the application log (on Windows NT).

The same can be applied to debug messages.

The Build Process

The image generator first loads the application startup code (in `WINMAIN.SM`) and one or more runtime files with runtime-specific code. The builder first marks all the code and classes that can be reached from that entry point. The next step is to recompile the code for the target image, and the last step optimizes the executable by compacting the sections. Therefore, development settings for the section sizes do not apply to runtime images.

Runtime Code

Several runtime files come with the product. The purpose of the runtime code is to replace all references to the development image before the pruning process begins. For example, exception handling invokes the debugger at runtime, which itself references the compiler, so this code needs to be replaced.

There are files for release and debugging builds of executables and DLL images. The project build properties determine which files are loaded, so the information below is only useful if you wish to look at the contents of a runtime file or need to modify runtime code.

Table 7-2 Runtime Files

File	Description
`RUNTIME.SM`	Release runtime code that replaces the exception handling system, installs simplified symbol handling and replaces some error and diagnostic code.
`RUNTIME_DBG.SM`	Runtime code that includes the Smalltalk runtime debugger.
`RUNTIME_NULL.SM`	Minimum runtime code that just eliminates the reference to the debugger. Because it does not replace the exception handler, it is convenient to use this file when tracking errors that occur during the build.
`RUNTIME_DLL.SM`	DLL release runtime file. The Project Browser automatically loads this file when compiling a DLL.
`RUNTIME_DBG_DLL.SM`	DLL runtime code that includes the Smalltalk runtime debugger.
`RUNTIME_SYM.SM` `RUNTIME_SYM_DLL.SM`	Release runtime code for EXE and DLL images that use a symbol table at runtime.

Several other files provide a common code base and are loaded indirectly via the code above.

The Project Browser looks for the runtime file in the following directories:

1) The project directory
2) The current directory
3) The project library path (generally, this includes the `SOURCE` subdirectory)

This means that a file in the project directory or in the image directory overrides the default runtime file.

Console Applications

Overview

A console application is a Win32 executable that runs in console mode. Because you can create a console for a regular Win32 GUI application, developing and debugging a console application is straightforward.

This section explains how to create a console window and the differences between development and runtime code, and addresses command line processing issues.

A console consists of an input buffer and one or more screen output buffers. High-level I/O streams let an application read and write to the buffers sequentially.

Class **ConsoleStream** manages input and output for character-mode applications. A character application reads input from a standard input stream and writes to a standard output stream. In addition to these basic tasks, an application can also access the screen buffer and trap mouse events.

Creating a Console

A process that is not tagged as a console application allocates a console using:

```
ConsoleStream allocConsole.
```

And when it is done,

```
ConsoleStream freeConsole.
```

de-allocates the console. Note that there can only be one console per process.

If the process is already marked as a console application, the steps above are not necessary.

Accessing the Standard I/O Streams

As mentioned previously, a Console has associated standard input and output streams. To retrieve a standard stream , you use `getStdHandle:` with one of the constants `STD_INPUT_HANDLE`, `STD_OUTPUT_HANDLE`, or `STD_ERROR_HANDLE`. It is then possible to read and write using standard stream methods (`next`, `next:`, `nextPut:`, `nextPutAll:`, etc...) as well as **FileStream** methods.

Alternatively, an application can use `createFile:` with `'CON'` as the file name and `GENERIC_READ` for the input stream, `GENERIC_WRITE` for output. However, unlike the standard streams obtained with `getStdHandle:`, streams returned by `createFile:` cannot be redirected.

Console Control Handlers

Console control handlers handle the following keyboard events:

CTRL+C, CTRL+BREAK, or CTRL+CLOSE.

The default system handler calls `ExitProcess` on any of these events. An application can define a replacement handler that handles these and / or additional input events.

Method `setConsoleCtrlHandler` in class **ConsoleStream** installs a handler in `HandlerRoutine:`, The default handler in **ConsoleStream** discards the events so that the process is not terminated (which is quite useful when the console is attached to a GUI application).

To install the default handler:

```
ConsoleStream setConsoleCtrlHandler.
```

Processing Command Line Arguments

Console applications often have to implement command line parsing (what follows applies to any type of application, not only console processses). The initialization methods in **ApplicationProcess** do most of the work, including wildcard expansion, so all that remains is to analyze an array of command line parameters.

The **cmdLineArgs** parameter that is passed via `ApplicationProcess>> winMain:with:with:with:` is an array with the following data:

Table 7-3 Command Line Parameters

Parameter	Description
1	Name of executable.
2	First command line parameter.
i	Command line parameter at position i - 1.

Wildcard expansion expands expressions such as `*.XLS` into a collection of file names that populate the array above. To retrieve the unexpanded parameters, use

```
ApplicationProcess>>getRawCommandArguments.
```

You can also retrieve the unprocessed command line with the API `GetCommandLine` at any time. However, beware of complications if you have to parse file names manually because there are different formats (with and without quotation marks) and differences between Windows 98 and Windows NT.

Console Skeleton

The code below is a skeleton for console applications. It assumes that the executable is compiled as a console application. Therefore, there is no need to allocate a console explicitly at runtime. However, it is necessary to allocate a console window in the development environment.

```
winMain: hModule with: hPrevInstance with: cmdLineArgs with: nCmdShow
    | stdin stdout szInputFile szOutputFile fileInput fileOutput sz step idx |
    _IS_DEVELOPMENT == TRUE ifTrue: [
        " disable if IMAGE_SUBSYSTEM_WINDOWS_CUI "
        ConsoleStream allocConsole.
        ConsoleStream setConsoleCtrlHandler.
    ].
    stdin := ConsoleStream new getStdHandle: STD_INPUT_HANDLE.
    stdout := ConsoleStream new getStdHandle: STD_OUTPUT_HANDLE.

    stdout nextPutAll: 'Display application name\n'.

        " implement command line parsing "

        " implement your processing here... "

    _IS_DEVELOPMENT == TRUE ifTrue: [
        " <- disable if IMAGE_SUBSYSTEM_WINDOWS_CUI "
        ConsoleStream freeConsole.
    ].
```

Sample command line parsing:

```
" command line argument parsing "
cmdLineArgs size
case: 1 perform: [
    " no arguments "
]
case: 2 perform: [
    (cmdLineArgs at: 2) = '?' ifTrue: [
        stdout nextPutAll: '\nDisplay Usage and Copyright Information\n'
    ].
]
default: [
    " multiple arguments "
].
```

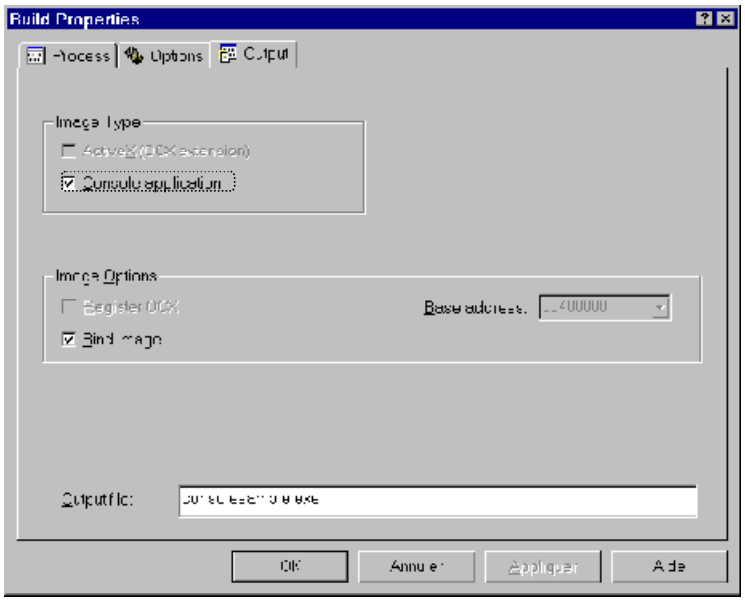
Testing a Console Application

From the development image, evaluate the code above. If there is an exception, do not forget to call `ConsoleStream>>freeConsole` before you restart.

Generating a Console Process

Project Browser has an option for generating console processes. The system automatically allocates a console for the process, unless it has been started with the `PROCESS_DETACH` flag.

Figure 7-2 Build Properties (Console Application)



Dynamic Link Libraries

Overview

Smalltalk MT lets you build general-purpose DLLs that can be used by any program capable of loading DLLs, either statically or at runtime. From an application's point of view, a DLL is a software component that exports a given set of functions. A DLL is bound to the process in which it is executed, but nevertheless has the capability to allocate private memory and run threads. In particular, a DLL written in Smalltalk runs its own garbage collector.

DLL Exports

You can export any class method that has been compiled under the category `.EXPORT` or `.EXPORT_CDECL`. The methods must be listed in an export declaration file (.DEF) that has the following format:

LIBRARY *name_of_library*

EXPORTS

class>>exported_method :: *exported_name* *@ordinal*

Each line under EXPORTS specifies an export.

Table 7-4 DLL Export Declaration

Parameter	Description
name_of_library	Name of the library.
class	Class that implements the exported method.
exported_method	Smalltalk selector of the exported method.
exported_name	Name to export the method as.
ordinal	Ordinal to export the method as.

Limitations

A DLL export in Smalltalk is exactly the same as a C function, with all the limitations that a non-object mechanism imposes. In particular, the Smalltalk receiver is not defined. For this reason, the stub code generated by the compiler assigns the class in which the method is defined to `self`. The consequences are:

- It is not possible to inherit an exported class method. The method must be implemented each time it is exported.
- It is not possible to export instance methods (because the receiver is not known at compile time).

If any of the functionality listed above is required, you must use a binary object standard such as COM. COM objects can expose instance methods and also support inheritance.

CHAPTER 8 Advanced Programming

The first part of this chapter discusses the DLL interface. It examines how to call an API, return values and arguments. A series of examples demonstrate common function calls. We will also address ANSI and Unicode API bindings, using decorated names, and implementing exports.

The following sections discuss exception handling and supporting native ANSI and Unicode applications.

Object serialization is an interface that allows serializing a complex object to a stream or a memory-mapped file. The section on finalization and weak references is of interest if you want to provide finalization methods that are called when an object gets out of scope. Debugging messages and error events assist you in tracking down bugs.

The Smalltalk Compiler interface allows you to implement development tools.

The last section discusses performance issues and interacting with the garbage collector.

Interfacing with Dynamic Link Libraries

There are two ways of interfacing with a DLL. One is runtime binding and is not discussed here (it involves loading the module, querying an API address and calling the procedure), the other consists of statically linking the library so that the references are fixed up by the loader. The latter method is the most economical in terms of resources and performance.

Before you can link statically with a library, the API definitions must be loaded as described in Image Maintenance.

For each API, Smalltalk needs to know the information in the table below:

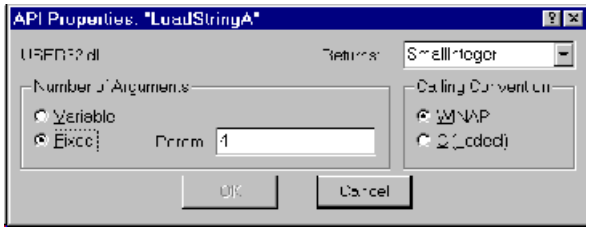
Table 8-1 Import Declarations

Declaration	Description
API name	The name of the API must be unique within the system for load-time binding.
DLL name	The name of the DLL.
Calling Convention	**WINAPI** This is the standard calling convention for Windows. The callee cleans up the stack. **C** The standard C calling convention, also known as **_stdcall**. It allows passing a variable number of arguments, and the stack is cleaned up by the caller (which is less efficient on Intel® architectures). Of the Windows base APIs, only *wsprintf* functions use it.
Argument Count	The API can accept either a variable or a fixed number of arguments. All WINAPI functions require a fixed number of arguments, but C type functions may accept a variable number of arguments. For a fixed argument count, the entry can consist of: the number of arguments (i.e., 1 for `MessageBeep`) or an array with the size of each argument (i.e., #(8) for `WindowFromPoint`)

Return Type	**Integer (LONG)** (Default) The return value is converted to an integer. **ULONG** The return value is an unsigned 32-bit integer. **SmallInteger** The return value is a 32 bit long, but its value does not exceed 31 bits. Reduces the code somewhat. **USHORT** The return value is a 16 bit unsigned integer that is converted to an Integer. Unusual for 32 bit code. **SHORT** The return value is a 16 bit signed integer that is converted to an Integer. Unusual for 32 bit code. **Float (FLOAT)** The return value is a float (a float is passed on the top of the floating-point stack). A Smalltalk Float instance is created. **A Smalltalk 4 byte object class** An instance of the 4-byte object class is created and the 32 bit return value copied to the object. This can be used for `Handle`, `LONG`, `Semaphore` etc.
Hint	The hint entry is an ordinal generated automatically by the import editor.

You can edit the API information by selecting an API name in the class hierarchy browser and choosing the *edit API* menu:

Figure 8-1 API Properties



The initial default entries are:

- *WINAPI calling convention*
- *unknown fixed argument count*
- *Integer return value*

The number of arguments is updated when the first call is compiled. If a subsequent call uses a different number of arguments, the API Properties box pops up, so that the developer may correct the entries. This kind of lazy initialization turns out to be far less tedious and error prone than updating all API information blocks at once. However, it is mandatory to set the correct entries for APIs that follow the C calling convention.

See also *Programming for ANSI and Unicode* on page 393 for how to maintain different character encoding.

Passing Arguments

Overview

By default, the compiler generates code that sends the message `_asCinteger` to each argument before it is pushed onto the stack. You can override the default by using the pseudo-message `basicAddress`.

API arguments can be passed by value or by reference. In the first case, the value of the parameter is pushed onto the stack. The value is always widened to 32 bits, or, in the case of multi-byte parameters, the total number of bytes transferred is a multiple of 32 bits. Passing by value is generally used for integers, large integers (`LARGE_INTEGER` or `int64` type in C) and double precision floats.

Passing by reference means that the address of a parameter, rather than the parameter itself, is passed.

The pseudo-message _asCinteger

The message `_asCinteger` requests the C compatible format of an object. The return value is a raw 32-bit value, unless the receiver raises an exception because it cannot be passed to an API.

32-bit values

A 32-bit value is the default parameter passing convention. The object itself responds to the message `_asCinteger` and returns a 32-bit value. Integers and Handles respond with their value. Byte objects such as WordArrays and Strings return the address of the object. Floats respond with a 4-byte floating-point value (equivalent to a simple precision C *float* type).

Passing by Reference

You can force an object to pass its address with the pseudo-message `basicAddress`. This is equivalent to the prefix & in C. Small integers do not have an address, it is therefore necessary to first create a LONG with the integer as its value.

Passing multi-byte structures by Value

If the API takes a multi-byte structure, the result of `_asCinteger` is interpreted as the address of a byte object whose contents are pushed onto the stack.

Passing multi-byte objects by value rather than by reference is unusual, except for small objects such as floats and sometimes points.

Examples

Example 1

This example retrieves the computer name of the current machine. The API is defined as:

```
BOOL GetComputerName(
    LPTSTR lpBuffer,      // address of name buffer
    LPDWORD nSize  // address of size of name buffer
);
```

The Smalltalk code looks like:

```
| buffer length |
buffer := String new: 256.
length := LONG value: buffer size.
(WINAPI GetComputerName: buffer with: length basicAddress) == TRUE ifTrue: [
    ^buffer copy
].
^nil
```

It first creates a **LONG** that holds the maximum number of characters the function can copy. On return, the **LONG** contains the number of characters copied. Since the string as also null-terminated, the code simply uses

```
buffer copy
```

to obtain the exact string. Another possibility is to use:

```
buffer copyFrom: 1 to: length asInteger
```

This code works with the default properties of the API. It is also possible to modify the return type to BOOL in the API editor; in this case the API call above would become:

```
(WINAPI GetComputerName: buffer with: length basicAddress) ifTrue: [
```

Note that the previous code still works *after* the return type is changed to BOOL because the compiler uses lazy type conversions and the comparison against a 32-bit value (TRUE) has higher precedence.

Example 2

This example calls the function `IsProcessorFeaturePresent`. The function is defined as:

```
BOOL IsProcessorFeaturePresent(
    DWORD ProcessorFeature       // specifies the processor feature
);

WINAPI IsProcessorFeaturePresent: PF_MMX_INSTRUCTIONS_AVAILABLE
```

Evaluating the expression above returns either 1 or 0. You can change the return type to BOOL to let it return a Smalltalk **Boolean**.

Example 3: Passing a multi-byte structure by value

The API `ChildWindowFromPoint` takes two arguments, the handle of a window and a point, which is passed as an 8-byte structure.

```
WINAPI ChildWindowFromPoint: handle with: aPoint
```

Example 4: implementing _asCinteger

```
Window>>_asCinteger
    ^handle _asCinteger
```

The compiler has special code that handles the message `_asCinteger`. It ensures that the result is a 32-bit value in the generated method implementation, and when you use `_asCinteger` it assigns the raw 32-bit type to the result. For example:

```
#[1 2 3] _asCinteger
```

evaluates to the address of the WordArray.

Example 5: Using LONG objects

Given the following API definition:

```
BOOLEAN WINAPI MyFunc(LONG, LONG, *LONG);
```

The code below invokes the function in C:

```
LONG myVal;
LONG val=1234;
MyFunc(val, 777, &myVal);
```

In Smalltalk, the code becomes:

```
|myVal val|
myVal := LONG new.
val := LONG value: 1234.
WINAPI MyFunc: val with: 777 with: myVal basicAddress.
```

Optimizing with pseudo-messages

It is possible to optimize parameter passing when the type of an argument is known in advance.

Table 8-2 Optimizing API Parameter Passing

Message	Applies to	Effect
_asInteger	SmallInteger	Converts a Smalltalk SmallInteger to a 32-bit value using inline code.
basicAddress	byte objects	Passes the address of the object.

Note Do not use _asInteger on immediate (literal) integers. The compiler already generates optimal code in this case.

Referencing Arguments

Smalltalk provides automatic memory management that frees the developer from the tedious and error prone task of de-allocating memory. An unreferenced object is eventually garbage collected. Because the garbage collector scans each thread's stack, API arguments are referenced at least until the function returns.

A problem may arise if arguments are still in use *after* the API returns, without being referenced from the Smalltalk image. Typically, this can be the case when a callback API copies arguments to local storage, as illustrated by the following code fragment:

```
| idThread |
idThread := LONG new.
^WINAPI CreateThread: NULL
        with: NULL
        with: [ :argument | self someMethod ]
        with: NULL                                  " argument "
        with: CREATE_SUSPENDED
        with: idThread basicAddress.
```

The method returns before the thread had any chance to run. As a result, the block

```
[ :argument | self someMethod ]
```

is de-referenced and discarded, which generates an exception once the thread is resumed. Note that if the thread runs right away, the call will probably not fail in most cases, although there is no guarantee that it executes correctly.

The code fragment shown below can also be a source for erratic errors:

```
self setDlgProc.
^WINAPI DialogBoxParamA: hinst
       with: idTemplate
       with: ownerWindow
       with: dlgProc
       with: NULL
```

If the caller does not reference the dialog box instance, its life depends on it being referenced by the stack. For a dialog procedure, this may often be the case, and results in code that works most of the time.

Smalltalk MT provides a convenient method for keeping references without resorting to global variables. An application can use `registerObject` to reference the receiver by the current thread, and `releaseObject` to de-reference it. The messages

respectively increment and decrement the number of references; and the object is subject to being discarded once the number of references reaches zero and there are no references from the Smalltalk image.

Return Types

Abstract

The intrinsic return value of a function is always a 32-bit value returned in the accumulator (EAX), except in the case of a floating-point return type where a float is returned on the top of the floating point stack.

The compiler recognizes several predefined types and compiles special code that transforms the return value into one of these types.

Integer

Integer is the default return type. When the function returns, Smalltalk converts the 32-bit value into a **SmallInteger** or **LargeInteger**. A **LargeInteger** is created when the result does not fit within 31 bits.

SmallInteger

SmallInteger simply tells the compiler that the result is always less than 31 bits wide; therefore, it is not necessary to generate code that tests for an overflow and creates a **LargeInteger** if necessary.

Another way to accomplish the same effect without modifying the API declaration is to send the pseudo-message `_asShort` to the return value, as in the example below:

```
^(WINAPI SendMessage: m_handle
    with: CB_DELETESTRING
    with: index
    with: NULL) _asShort
```

The `CB_DELETESTRING` message returns the number of remaining elements or –1 if an error occurs, so the return value clearly cannot exceed 31 bits.

BOOL

A return type of BOOL means that the function returns 0 on failure and a non-zero value on success. The return value is converted to a Smalltalk Boolean (**true** or **false**).

If the API documentation specifies any other type than `BOOL`, it is better to use direct comparisons. For example, OLE functions often return `S_OK` or `S_FALSE`, where `S_OK == 0` and `S_FALSE == 1`.

Compiler Specific

Immediately after the function returns, you can compare the return value against a 32 bit integer using identity:

```
(WINAPI MyFunc: param) ~~ 0 ifTrue: [...]
```

This is useful if you want to be compatible with existing code.

You can also compare directly using:

```
(WINAPI MyFunc: param) ifTrue: [...]
```

In all cases, the native code is optimized (it does not really generate the Smalltalk **Boolean**).

SHORT and USHORT

A SHORT or USHORT return value denotes a 16 bit return value (i.e., only the low order 16 bit word is valid). A SHORT is sign-extended (i.e., a value of 16rFFFF is converted to –1). The return value is always a **SmallInteger** instance.

CHAR and UCHAR

A CHAR or UCHAR return value denotes an 8 bit return value (i.e., only the low order 8 bit word is valid). A CHAR is sign-extended (i.e., a value of 255 is converted to –1). The return value is always a **SmallInteger** instance.

ULONG

A ULONG return type creates an unsigned integer. The result is the same as (WINAPI funcXYZ) unsigned. Using this return type is only necessary when the result must be compared against another integer.

A Smalltalk byte class

It is possible to create an instance of a Smalltalk byte class (such as a subclass of **LONG**) automatically. You must enter the name of the Smalltalk class with the API editor or edit the `DEF` file that describes the functions of the DLL.

Note It is often preferable to create the Smalltalk object using an explicit message such as `value:`, which provides more error-checking.

Float

The **Float** (or FLOAT) return type is a special case of instantiating a Smalltalk class. Please note that it is mandatory to specify **Float** (or FLOAT) for a function that returns a floating-point value. This is because the floating-point value is taken from the floating-point stack, and not from the accumulator (the EAX register).

Examples

Overview

You will find the following examples on the distribution disk. The calls are implemented in class methods of a class named **LibTest**. The C source code is also provided.

Example 1: Floats and characters

```
LONG WINAPI testFunc1(long double f, CHAR c)
{
    return (LONG)f + c;
}
Smalltalk Code:
    ^WINAPI testFunc1: f with: c

Example:
LibTest testFunc1: 22.3 char: $c
```

API Parameter Declaration	(double 0)
Return Value	Integer

The first argument, a **Float** instance, is pushed onto the stack. The second argument is first converted with the hidden message `_asCinteger`, and the resulting 32-bit value is pushed onto the stack.

The return value is a signed integer (the default return type in Smalltalk MT).

Example 2: ULONG return type

```
ULONG testFunc2(float f, UCHAR c)
{
    return (LONG)f + c;
}
Smalltalk Code:
    ^WINAPI testFunc2: f  with: c

Example:
LibTest testFunc2: 22.3 char: -1
LibTest testFunc2: -22.3 char: 0
```

API Parameter Declaration	(double 0)
Return Value	ULONG

This example is similar to the previous example. The return value has been declared as a ULONG, meaning that a negative 32-bit value is converted to a **LargeInteger**. The other difference is the type of the second parameter. For example, a negative value such as –1 is interpreted as 255 in the function.

Please note that the calling convention is the default C calling convention (**_cdecl**).

Example 3 & 4: Passing pointers

The following examples pass pointers to 1, 2, and 4 byte arrays. The API call overrides the default API representation of each parameter to ensure that an address is passed.

The are several ways to fill a byte object. The pseudo-messages `_byteAt:` and `_wordAtOffset:` directly write at the specified index (respectively offset in bytes) of the receiver object. Therefore, the caller must make sure that the parameters are of the correct type.

```
CHAR WINAPI testFunc3(ULONG *lp1, CHAR *pChar, UCHAR *pUChar)
{
    lp1[0] = pChar[0];
    lp1[1] = (LONG)pUChar[0];
    lp1[2] = (LONG)pUChar[1];
    return *pUChar;
}

Smalltalk Code:
    WINAPI testFunc3: array basicAddress
  with: pChar basicAddress
  with: pUChar basicAddress


Example:
| array pUChar |
array := WordArray new: 3.
pUChar := ByteArray new: 2 * 2.
pUChar _byteAt: 1 put: -3.
pUChar _byteAt: 2 put: 4.
LibTest testFunc3: array pChar: $a pUChar: pUChar.
array
```

API Parameter Declaration 3

Return Value Integer (default)

```
VOID testFunc4(LONG *lp1, SHORT *pShort, USHORT *pUShort)
{
    lp1[0] = pShort[0];
    lp1[1] = (LONG)pUShort[0];
    lp1[2] = (LONG)pUShort[1];
}


Smalltalk Code:
WINAPI testFunc4: array basicAddress
  with: pShort basicAddress
  with: pUShort basicAddress


Example:
| array shortParam ushortParam|
array := WordArray new: 3.
shortParam := LONG new.
shortParam _wordAtOffset: 0 put: -3.

ushortParam := ByteArray new: 2 * 2.
ushortParam _wordAtOffset: 0 put: -3.
ushortParam _wordAtOffset: 2 put: 4.
LibTest testFunc3: array pShort: shortParam pUShort: ushortParam.
array
```

API Parameter Declaration	3
Return Value	Integer (default)

Examples 5 & 6: Passing structures

```
LONG WINAPI testFunc5(POINT pt)
{
    return pt.x + pt.y;
}
Smalltalk Code:
^WINAPI testFunc5: aPoint
^WINAPI testFunc6: aPoint x with: aPoint y

Example:
LibTest testFunc5: 12@22
```

The examples 5 and 6 illustrate the different ways of passing a **WinPoint** by value. Example 5 declares the parameter as an 8-byte structure that is passed by value, while example 6 declares two arguments, x and y.

Example 5:

API Parameter Declaration	(8)
Return Value	Integer

Example 6:

API Parameter Declaration	2
Return Value	Integer

Examples 7 & 8: Floating point values

```
float WINAPI testFunc7(float f1, double f2)
{
    return f1 + f2;
}
Smalltalk Code:
    ^WINAPI testFunc7: f1 asFloat with: f2 asFloat

Example:
LibTest testFunc7: 12.2345 float: PI
```

Example 7:

API Parameter Declaration	(float double)
Return Value	FLOAT

```
double WINAPI testFunc8(double f1, double f2)
{
    return f1 + f2;
}
Smalltalk Code:
    ^WINAPI testFunc8: f1 with: f2

Example:
LibTest testFunc8: 12.2345 float: PI
```

Example 8:

API Parameter Declaration	(double double)
Return Value	FLOAT

Example 9: Floating point structures

```
double WINAPI testFunc9(double *pf1, float *pf2)
{
    return pf1[0] + pf1[1] + ((pf2[0] + pf2[1])* 2);
}
```

Smalltalk Code:

```
       ^WINAPI testFunc9: pf1 with: pf2
```

Example:

```
    | pf1 pf2 |
    pf1 := Struct fromArray: #(##(PI negated) ##(2 sqrt negated)).
    pf2 := WordArray with: PI // 2 with: 2 sqrt // 2.
    LibTest testFunc9: pf1 float: pf2
```

API Parameter Declaration	2
Return Value	FLOAT

The first argument to the function is a pointer to an array of 8-byte (double precision) floats, the second a pointer to an array of 4-byte (single precision) floats.

Example 10 & 11: More structures

```
LPPOINT WINAPI testFunc10(POINT pt1, USHORT uOffset)
{
    LPPOINT pt;

    pt = (LPPOINT)malloc(sizeof(POINT));
    pt->x = pt1.x + uOffset;
    pt->y = pt1.y + uOffset;
    return pt;
}
Smalltalk Code:
    | lpPoint |
    lpPoint := WINAPI testFunc10: aPoint with: uOffset.
    lpPoint ~~ NULL ifTrue: [
        | answer |
        answer := WinPoint new fromBytes: lpPoint.
        WINAPI free: lpPoint.
        ^answer
    ]
    ifFalse: [
        ^nil
    ].

Example:
LibTest testFunc10: 1@2 offset: 10
```

API Parameter Declaration	(8 0)
Return Value	FLOAT

This function takes a point and an integer, and returns a new point allocated using malloc. The Smalltalk code that calls the function copies the contents to a **WinPoint** and frees the memory allocated by the function (knowing that the memory has been allocated with `malloc`).

```
int WINAPI testFunc11(FORMATETC* pformatetc)
{
    pformatetc->cfFormat = CF_TEXT;
    return 0;
}

Smalltalk Code:
    | formatEtc |
    formatEtc := FORMATETC new.
    WINAPI testFunc11: formatEtc.
    ^formatEtc cfFormat

Example:
LibTest testFunc11
```

API Parameter Declaration	1
Return Value	Integer (default)

Example 12: Callback functions

This sample illustrates passing back and forth function and method addresses.

The callback function sample passes the addresses of two exported class methods to a C function. The C function calls both exports and returns the address of another function implemented in C. The calling Smalltalk code finally calls the function using the returned function address.

```
FN1* WINAPI testFuncCall(FN0* pfn0, FN2* pfn2){
    LPSTR result;

    result = pfn0();
    pfn2(result, 123);

    return testFunc0;
}

Smalltalk Code:
    | pfn |
    pfn := WINAPI testFuncCall: (self methodAddressAt: #FN0)
        with: (self methodAddressAt: #FN2:with:).
    ^MemoryManager callAPI: pfn with: 789.

Example:
LibTest testFuncCall
```

API Parameter Declaration	1
Return Value	Integer (default)

The exported class methods are simply declared as:

```
FN0
    "
    typedef LPSTR (WINAPI FN0)();
    "
    Transcript show: '\nFN0'.
    ^'"result from function FN0"'

FN2: a with: b
    "
    typedef LONG (WINAPI FN2)(LPSTR, LONG);
    "
    Transcript show: '\nFN2('.
    Transcript show: (String fromAddress: a).
    Transcript show: ', ',b printString,')'.
    ^0
```

The function whose address is returned by the DLL is defined as:

```
LONG WINAPI testFunc0(LONG l)
{
    return 0 - l;
}
```

ANSI and Unicode Binding

Many functions have ANSI and Unicode implementations. In Win32, these functions are usually post-fixed with respectively A or W.

Smalltalk MT automatically binds with the appropriate API version. For instance:

```
WINAPI GetCommandLine
```

resolves into:

`WINAPI GetCommandLineA` with ANSI encoding

`WINAPI GetCommandLineW` with Unicode encoding.

It is of course possible to override the defaults and call the API directly by specifying the final function name.

You see the real name of the API when you open the API property editor (highlight the API in the Class Hierarchy Browser, then click on *Edit API* to bring up the API dialog).

Using Decorated Names

A C++ compiler generates decorated names in the absence of an explicit export function name. For example, the function:

```
void CALLTYPE test(void)
```

can be exported as

`_test`, `?test@@ZAXXZ` or `_test@0`.

To call a decorated function such as `?test@@ZAXXZ`, use the syntax below:

```
WINAPI ##decorated_function_name: argument1 with: argument2
```

where *decorated_function_name* is the function you wish to call, for example `?test@@ZAXXZ`.

If the function has no parameters, use void as in the example below:

```
WINAPI ##?test@@ZAXXZ: void
```

Aliasing API Names

It is possible to specify an alias for a function. This is necessary if more than one library implements a given function and the function must be called in both libraries. The alias is used as the API lookup key.

For example, to create an alias for the function Foo in library LibA:

1. Link with library LibA.
2. Create an alias for Foo. There are two ways to create an alias:
   - From the image properties, open the library properties for LibA and right-click on Foo. In the popup menu, click on *Rename...* and enter a new name; for example FooB.
   - Call `ByteCodeCompiler>>apiRename:in:name:`, specifying the real function name, the library and the new aliased name. You can add the line to an initialization script.

Callbacks

Calling Smalltalk code from external modules either involves an exported method (using the categories `.EXPORT/.WINAPI` or `.EXPORT_CDECL/.CDECL`) or a block. Callback parameters are always converted to Smalltalk integers, and the result is converted back to a 32-bit value.

By design, any Smalltalk block can be called from non-Smalltalk code. Smalltalk blocks preserve the registers as required by the WINAPI calling convention, and arguments are automatically converted to **Integer** instances.

- Any Smalltalk block can be called by external code using the WINAPI (**_stdcall**) calling convention. The only requirement is that the return value can be converted to an API parameter.

- Class methods that belong to the `EXPORT_CDECL/CDECL` or `EXPORT/WINAPI` category can be called back or exported. They must return a 32-bit value.
- Instance methods under the `EXPORT` category behave like C++ member functions; the first argument must be the receiver. This makes it possible to implement OLE interfaces directly as Smalltalk objects.

Table 8-3 Export Calling Conventions

Category	Calling Convention
EXPORT WINAPI	The calling convention is WINAPI. Win32 normally uses this convention.
EXPORT_CDECL CDECL	The calling convention is STDCALL, the normal C calling convention.

Note An exported method must not be called directly by Smalltalk code. To call an exported method, you must use MemoryManager>>callAPI:[with:] or MemoryManager>>stdcallAPI:[with:] with the address of the exported method (i.e., you call the method like you would call an external function, given a pointer to the function). The address of a method is returned by the method #methodAddressAt: aSymbol.

Exception Handling

Overview

The exception handling system provides a structured and efficient mechanism for handling exceptional events. It is based on Win32 structured exception handling and offers a seamless interface to hardware faults and software exceptions alike.

An application can protect a block of code from abnormal termination by specifying a filter block that is invoked whenever an exception occurs, and a handler block to which the control is transferred for those exceptions that are accepted by the filter block. Another variation is **finalization**, where an application protects a block of code by a termination handler, which is executed when the guarded block is exited, be it through the normal control of flow, or by an exception. Finalization blocks typically contain cleanup code such as closing a file.

Exceptions are identified by an exception code. Win32 exception codes are defined in `WinBaseConstants`, Smalltalk exception codes in `ExceptionConstants`. An application may pass up to `EXCEPTION_MAXIMUM_PARAMETERS` additional parameters (32 bit values) that can be retrieved by an exception filter. In Smalltalk, instances of **WinException** and subclasses are used to represent a specific type of exception. For example, **CompilerException** represents a compilation exception and provides methods to create an exception object from of the information passed to the filter block, as well as default methods to handle the exception.

Most software exceptions defined in Smalltalk MT are non-resumable. However, nothing prevents you from raising continuable exceptions as in the following example.

Example:

```
self try: [
    MessageBox title: 'Raise Exception' text: ''.

    WINAPI RaiseException: ST_EXCEPTION_ERROR
        with: 0
        with: NULL
        with: NULL.
    MessageBox title: 'After Exception' text: ''
] filter: [ :a :b |
    ... " do something that corrects the error condition "
    EXCEPTION_CONTINUE_EXECUTION
]
```

Common Exceptions

Exceptions are listed in the class initialization methods of **Exception**. An exception is defined by its code, which is a compound of a facility code and an exception code. There are currently two facility codes defined; `ST_FACILITY_RUNTIME` and `ST_FACILITY_COMPILER`.

There are two sources of exceptions: programming errors that should be eliminated before an application is shipped, and abnormal runtime behavior that can be caught by the application. For example, an application might guard against invalid handles and missing libraries and degrade its functionality gracefully.

Note The compiler evaluates constant integer expressions at compile time. It is therefore possible that the compilation results in an exception (for example 1 // 0 raises an exception).

Runtime Exceptions

Operating-system Exceptions

Exception Code: `EXCEPTION_XXX`

Please refer to the Win32 documentation for additional information about system exceptions. Certain exceptions cannot be handled or leave the system in an unstable state.

- The breakpoint exception cannot be handled when an external debugger is attached to the process.

- The stack overflow exception occurs after all the reserved stack space has been committed. The operating system commits an additional page to let the exception handler do its work. To recover from this exception properly, the handler routine has to free the unneeded stack space and reset the stack. Otherwise, the next stack fault cannot be handled at all and the process is shut down.

Memory Allocation Error

Exception Code: `ST_EXCEPTION_MEMORY_ERROR`

You get a memory allocation exception when the amount of reserved memory is exhausted after a garbage collection cycle.

Upon initialization, the memory manager reserves spare memory that is committed when an out of memory exception occurs. The purpose of this additional reserve is to be able to handle the exception gracefully.

For example, an application might allow the user to perform arbitrarily complex calculations that could exhaust the initially reserved memory space. By wrapping the calculations in an exception handler, it is possible to handle memory exception gracefully.

General Error

Exception Code: `ST_EXCEPTION_ERROR`

This exception is user-defined. The argument is an error description string.

Aborting

Exception Code: `ST_EXCEPTION_ABORT`

A requested operation could not be carried out. For example, a library could not be loaded. The caller should handle this exception to exit gracefully.

Buffer Overflow

Exception Code: `ST_EXCEPTION_BUFFER_OVERFLOW`

An operation resulted in a buffer overflow.

Does Not Understand

Exception Code: `ST_EXCEPTION_DOESNOTUNDERSTAND`

An object is unable to respond to a message. The exception takes two arguments; the selector and the sender of the message.

Invalid Allocation

Exception Code: `ST_EXCEPTION_INVALID_OBJECTSIZE`

A requested object allocation size is invalid.

Invalid Handle

Exception Code: `ST_EXCEPTION_INVALID_HANDLE`

A handle is invalid, usually after an API failed to return a valid handle.

Floating Point Exceptions

Exception Code: `EXCEPTION_FLT_XXX`

These exceptions denote floating-point exceptions generated by the floating-point library. The constants are defined in `WinBaseConstants` and in class **Exception**.

You must set the floating-point control word using _controlfp to enable floating point exceptions, The method is implemented as a class method in float. See also *Float* on page 189.

Example:

```
| a cw result |
a := -2.3.
cw := Float _controlfp: 0 not mask: 0.
Float _controlfp: cw & (_EM_ZERODIVIDE|_EM_INVALID|
    _EM_DENORMAL|_EM_OVERFLOW|_EM_UNDERFLOW|_EM_INEXACT) not
    mask: _MCW_EM.

self try: [
    result := a // 0.0.
] filter: [:code :ptrs |
    code == EXCEPTION_FLT_DIVIDE_BY_ZERO ifTrue: [
        Float _clearfp.
        EXCEPTION_EXECUTE_HANDLER
    ]
    ifFalse: [
        EXCEPTION_CONTINUE_SEARCH
    ].
]
except: [ result := nil.].
result
```

Development Exceptions

Function Not Supported

Exception Code: `ST_EXCEPTION_NOT_SUPPORTED`

A requested operation is not supported by the receiver. For example,

```
Boolean new
```

raises this exception.

Array Bounds Violation

Exception Code: `ST_EXCEPTION_ARRAY_BOUNDS_EXCEEDED`

An attempt to access an out-of-bounds element was made, for example accessing an array element with an invalid index.

Invalid Argument

Exception Code: `ST_EXCEPTION_ARRAY_BOUNDS_EXCEEDED`

An attempt to access an out-of-bounds element was made, such as in:

```
(Array new: 4) at: 5.
```

Invalid Number of Arguments

Exception Code: `ST_EXCEPTION_INVALID_NUMBER_OF_ARGUMENTS`

The number of arguments to a **Block**, a **Message**, or a `perform` method, is invalid.

Invalid API Arguments

Exception Code: `ST_EXCEPTION_INVALID_API_ARGUMENT`

An API parameter is incorrect. The default implementation of `_asCinteger` raises this exception.

Invalid Boolean

Exception Code: `ST_EXCEPTION_INVALID_BOOLEAN`

A non-boolean was encountered where a boolean was expected. For example,

```
nil ifTrue: [...]
```

raises this exception. Note that the assertion is subject to optimizations.

Invalid Loop Parameter

Exception Code: `ST_EXCEPTION_INVALID_LOOP_PARAMETERS`

A `to:do:` or `to:by:do:` loop was entered with non-integer parameters. See also *Inlining Issues* on page 171.

Programming for ANSI and Unicode

Abstract

This section refers to the **default encoding** used by an application. The default encoding specifies the character encoding of strings and the API versions to call.

ANSI encoding means that the default character encoding is done in ANSI, and the application links with the ANSI libraries. Note that this does not prevent the application from using Unicode strings, only that the encoding needs to be specified and the appropriate methods and APIs called. The same applies to a Unicode application that uses ANSI strings.

The following paragraphs will examine how Unicode and ANSI models are implemented in Smalltalk MT and what you must consider before you plan to build a Unicode application. You may also refer to the Win32 documentation for general information on character encoding issues and how they affect the Win32 API.

Implementation

With Smalltalk MT, you can build Unicode and ANSI versions from the same source code. Smalltalk MT uses a technique similar to the one used in the C API:

- The constant `UNICODE_ENABLED` (in `SmalltalkConstants`), if set to 1, enables Unicode, otherwise enables ANSI.
- A call to an API for which both Unicode and ANSI versions exist defaults to the current encoding (example: `GetCommandLine` maps to either `GetCommandLineA` or `GetCommandLineW`).
- The C-parameter representation of strings defaults to the current encoding: if an ANSI string is passed to an API in the Unicode model, it is first converted to Unicode and vice-versa for a Unicode string in an ANSI application.

- The pool dictionaries that come with Smalltalk MT are specific to the character encoding in use. For example, `TVN_SELCHANGED` maps to -402 (`TVN_SELCHANGEDA`) in ANSI and -451 (`TVN_SELCHANGEDW`) in the Unicode model.

When porting an application from ANSI to Unicode or vice-versa, you must ensure that code that expects strings of a certain type is still called with the correct type. You can enforce the type via `asStringW` or `asStringA` if necessary. For example, all OLE functions require Unicode strings, while other functions and some compiler-related code require ANSI strings.

If your application writes to files in text mode, you must also decide whether the file should contain wide characters or ANSI. Since **FileStream** writes the binary contents of an object, you must ensure that the arguments to methods such as `nextPutAll:` are of the correct type.

Further Considerations

Updating Pool Dictionaries

Different constants may be defined for Unicode and ANSI. You can use the method `translatePool:from:to:` in **ByteCodeCompiler** to convert a pool dictionary. The algorithm looks for constants that are also declared with a `$W` and `$A` postfix, and changes the value of the generic constant. For example, if `Foo`, `FooA` and `FooW` are defined, the value of `Foo` is changed to the one of `FooW` if Unicode is enabled, otherwise to the value of `FooA`.

The algorithm works well with the Win32 include files that are already installed, but may not be appropriate in all cases and may produce unexpected results.

Updating Imported Functions

When you change the characteristics of an API for which both Unicode and ANSI implementations exist, the modifications are propagated to both encoding forms. For example, setting the return type of `CreateSemaphore` to **Handle** updates both `CreateSemaphoreA` and `CreateSemaphoreW`.

Some DLLs define three implementations for different encoding; for example `FooFunc`, `FooFuncA`, `FooFuncW`. In most cases, the generic API defaults to the ANSI encoding, and you should remove the generic version (`FooFunc`) from the list of imported APIs, using for example:

```
Compiler apiRemove: 'FooFunc' .
```

Other Issues

Some structures are defined with Unicode and ANSI implementations. The recommended Smalltalk implementation uses the value of the `UNICODE_ENABLED` constant to initialize the structure differently.

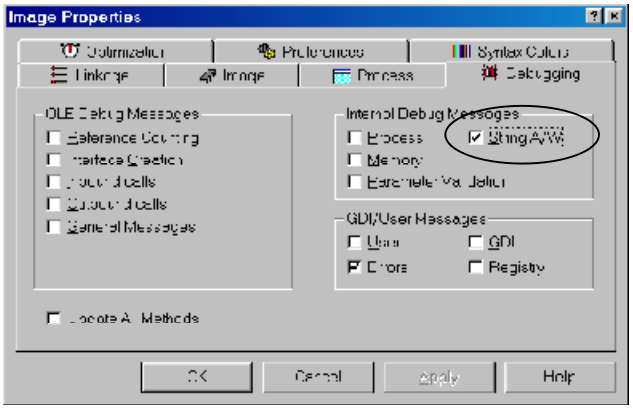
For example, the class initialization method in `WIN32_FIND_DATA` reads as below:

```
self
    addAccessor: #dwFileAttributes size: 4;
    addAccessor: #ftCreationTime size: 8;
    addAccessor: #ftLastAccessTime size: 8;
    addAccessor: #ftLastWriteTime    size: 8;
    addAccessor: #nFileSizeHigh      size: 4;
    addAccessor: #nFileSizeLow       size: 4;
    addAccessor: #reserved   size: 8.
UNICODE_ENABLED == TRUE ifTrue: [
    self
        addAccessor: #cFileName size: MAX_PATH * 2;
        addAccessor: #cAlternateFileName size: 14 * 2.
]
ifFalse: [
    self
        addAccessor: #cFileName      size: MAX_PATH;
        addAccessor: #cAlternateFileName size: 14.
].
```

Monitoring String Conversions

Once you have your Unicode application running, you can view string conversions in an external debugger by checking the String A/W conversion debugging messages in the *Debugging* page of the *Image Properties* sheet. This is useful for detecting unnecessary conversions that slow down the code. The figure below shows the corresponding page of the Image Properties:

Figure 8-2 Image Properties - Monitoring String Conversions



Object Serialization

Abstract

Serialization is a process by which the binary contents of an object are stored to a stream in such a way that the binary format of the object can be reconstructed later. In Smalltalk MT, the serialization medium is always a memory region, and the interface is exposed via **MappedObjectStream**, which is based upon a memory-mapped file. The contents can then be saved to disk or any other physical medium.

Memory-mapped object files let you map very large amounts of objects into memory without noticeable overhead. Under optimal conditions, the mapping operation is instantaneous (the process is similar to the mapping of an executable file to the address space of a process). This is the case when the same Smalltalk MT executable writes and reloads a file.

When creating a memory-mapped file, you must specify a maximum size. Memory-mapped files cannot grow (this is a limitation of the current Win32 API), so the allocated size should be sufficiently large to hold all elements, including a safe margin.

Serialization Architecture

The serialization library traverses a root object and all references that lie within a specified set of memory intervals, copying the contents to a target stream. Circular references are detected, meaning that an object that is referenced twice is not duplicated.

Serialization is generally used in conjunction with memory-mapped files. A memory-mapped object file contains a serialized root object.

A mapped object file always contains the following information:

- A magic number that identifies a Smalltalk MT memory-mapped file.

- The preferred load address.
- A class directory that lists the class names and method dictionaries used within the file.

The loader first examines the class directory and replaces all method dictionary references in the file that do not match the current image. The preferred load address is the address at which the file maps without applying fixups, otherwise the loader relocates all objects in the file.

When a memory-mapped file is saved the header and the class dictionary structures are updated. The next time the file is mapped by the same process, there will be no loading overhead.

Example

To save an object:

```
| obj |
obj := Array with: Time getLocalTime
    with: Transcript getFont.
MappedObjectStream saveObject: obj fileName: 'test.bin' size: 16r10000.
```

To re-map the object:

```
| mos obj szResult|
mos := MappedObjectStream new.
obj := mos openFile: 'test.bin'.
"..."
szResult := obj printString.
" when done, close the file "
mos close.
szResult
```

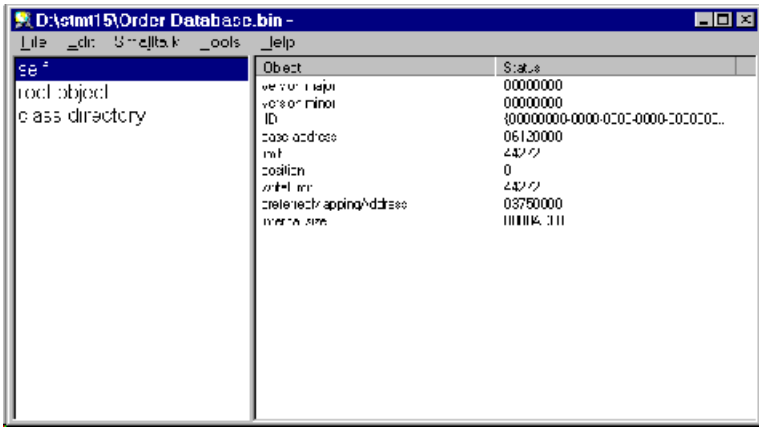
Using Memory-mapped Objects

To use memory-mapped object files, you use **MappedObjectStream**. The serialization framework characterizes a file by a header structure. The header contains version information, a GUI that identifies the file type, and a pointer to a class directory. The header enables an application to determine whether a given file is appropriate without depending on external attributes such as the name and extension.

The object file header

The object file header of a memory-mapped object stream can be viewed with the Mapped Object Stream inspector.

Figure 8-3 Object File Header



The version field

The version field consists of two 32-bit integers that represent the version. The caller uses this information to handle different versions. For example, an application can use this field to convert objects between different versions.

The IID

Mapped object files can use an IID to identify the type of the file. The IID can be generated with an OLE IID generator.

The IID provides a way to uniquely identify a file type. Typically, an application reads the file header and checks the IID and version information to make sure it can process the file. This is much better than relying on the file name extension to make assumptions about the file type.

The Class Directory

The class directory identifies the object classes that are used in the file. You can view the class directory of a memory-mapped object file by opening it in the MOS inspector.

Opening and creating a file

Memory-mapped object files use the memory-mapped file interface to implement object mapping. As the name suggests, objects that reside on disk are mapped into the address space of the process and can be accessed like regular objects. The address space is also registered with the garbage collector, so it is possible to reference newly created objects from memory-mapped objects (provided the file has been opened in read/write mode).

Table 8-4 MappedObjectStream Access Modes

Access	Description
FILE_MAP_READ	The file is opened in read-only mode. The file can be shared.
FILE_MAP_WRITE	The contents of the file can be modified, and the file region is scanned by the garbage collector. The file cannot be shared.
FILE_MAP_COPY	This flag gives copy-on-write access and registers the mapped region with the garbage collector. The file can be shared; other processes see the original disk image. The contents cannot be saved to the original file; instead, a new file must be created and the mapped region copied to the new file.

File modes

There are two ways to open, modify and save a file. Either the application opens the file in write-through mode (`FILE_MAP_WRITE`) and simply serializes and closes it when done, or it opens it in copy-on-write mode (`FILE_MAP_COPY`) and creates a new file when saving.

This does not necessarily entail a complete serialization; it is sufficient to transfer the binary contents after the serialization takes place, using a method such as `copyToFile:`. The second method is more secure, since a failure does not destroy the original file. However, some applications first make a copy of the document and work on the copy, in these cases the first technique is more appropriate since it incurs less overhead and the original stays untouched anyway.

Table 8-5 MappedObjectStream Open Methods

Method	Description
openFile:[size:]	Maps the file in copy-on-write mode and shares it both for reading and writing.
mapFile:size:	Maps the file in exclusive read/write mode.

File size

If the application intends to write to a memory-mapped file, it must allocate sufficient space when it maps the file because the Win32 API does not yet support growing memory-mapped files. The tradeoff is that too high a value wastes disk space, while insufficient space ends with a protection violation when the file is updated. To open a file in read-only mode, specify zero as the file size.

Storing data

An application uses `nextPutAll:` to store a root object. The method performs a shallow copy of the object, the actual serialization is performed when the file is saved. There can only be one root object in the file, and calling `nextPutAll:` more than once overwrites any previously stored object.

Saving a file

Saving a memory-mapped object file entails traversing the stored objects, serializing references that point into one of a specified set of memory intervals. The method `serializeIntervals:` performs the serialization. The serialization either updates an existing memory-mapped object file or creates a new one, based on a magic number that is written at the beginning of the file and identifies a Smalltalk MT memory-mapped object file.

The serialization interface consists of the following methods:

Table 8-6 MappedObjectStream Messages

Message	Description
openFile:[size:access:shareMode:]	Opens and maps a memory-mapped file into the address space of the current process.
serializeIntervals:	Serializes the current contents of the stream. The parameter specifies an array of code intervals to serialize from.
nextPutAll:	Inserts a root object to the stream.

Closing a file

Before you close a file, there must be no references that point to a location within the file, except local variables in a static context (a method context that does not use context blocks). Otherwise, an exception may occur when the garbage collector tries to access a memory location within the formerly mapped file.

For example, if `m_rootObject` and `m_file` are instance variables that reference respectively a memory-mapped object file and the root object within the file, use code such as the fragment below to close the file:

```
m_rootObject := nil.
m_file close.
m_file := nil
```

Note that the last line is not mandatory but it is generally a good idea to nil out a closed file stream.

An alternative is to use the method `unloadAndClose`, which replaces all references to objects in the file with **nil**.

Remarks

Closing a memory-mapped object file entails removing the address range from the garbage collector's list of regions to scan, unmapping the view, closing the handle and truncating the file to its actual length.

Controlling the source intervals

The default implementation serializes the following objects:

- Objects allocated on the Smalltalk heap
- Objects stored in the data section (static objects such as those referenced by class variables).
- Literal objects that are stored in the static code section.

By default, the method does not serialize:

- Global objects such as pseudo-variables (**nil**, **true**, **false**) and classes, meaning that those references still point to the image.
- Literal objects that reside in the dynamic code section, which is created by the compiler during development.

- Objects that are allocated on an external heap, another memory-mapped file, or more generally memory intervals that do not belong to the default set of intervals.

The behavior can be changed, in particular you can specify

```
Compiler codeIntervalEx
```

to include literal objects created during the current session, and

```
aMemoryMappedFile virtualAddressSpace
```

to include objects from another memory-mapped file.

Note Do not include the global data section, as this would create duplicates of pseudo-variables and / or classes, which is hardly desirable. Store class names rather than classes and use `Class fromName:` to convert a class name to a class.

Windows 98 specific issues

Windows 98 is slower when opening mapped files, and you may experience a delay when closing a mapped file.

Finalization and Weak References

Abstract

The purpose of finalization is to perform certain actions when an object is about to be reclaimed. A weak reference is a reference that is ignored by the garbage collector.

You can use finalization and weak arrays to free external system resources when an object is reclaimed, freeing the consumer of such objects from de-allocation tasks.

However, you should keep in mind that:

- There is no order in which objects are reclaimed.
- There is not even a guarantee that an object is reclaimed at all because garbage collection only runs when needed.

For example, finalization is not appropriate for device contexts and graphical objects. There is no control over when a garbage collection occurs, so you may end up manipulating a device context that belongs to a window which has been closed.

Weak Pointers

A weak pointer is a reference to an object that is ignored by the garbage collector. Therefore, the referencee may be garbage collected if there are no other (strong) references to it.

The pointer must be marked as invalid when the object it points to gets out of scope. To that effect, a weak pointer object is registered as below:

```
ApplicationProcess>>weakAdd:makeWeak:
```

This ensures that an invalid pointer is set to **nil**.

A weak pointer object can be explicitly removed using:

```
ApplicationProcess>>weakRemove:
```

or implicitly when it gets de-referenced (`Processor` keeps itself a weak reference to weak arrays).

Access to a weak array must be synchronized with the garbage collector. The method:

```
ApplicationProcess>>gcCriticalSection
```

returns the critical section of the garbage collector, which must be used to access a weak array safely. The critical section should wrap the access in order to prevent the garbage collector from reclaiming the object being retrieved.

Finalization

In some cases, it is desirable to be notified when an object goes out of scope. The framework for weak arrays can also support finalization of an arbitrary object. The object is registered with:

```
ApplicationProcess>>weakAdd:makeWeak:
```

but this time the `makeWeak` parameter is set to **false**. The memory scavenger calls the method `weakFinalize`, which must check via `weakTest` whether the object is still alive. If it is not, it may perform its custom finalization. Otherwise, it must call `weakFinalize` in the superclass to ensure that invalid pointers are set to **nil** (of course, objects without instance variables do not have to call this method).

It is also possible to keep the object alive by calling `MemoryManager>>markObject:`, passing the object as parameter.

Interfaces

Methods in Object

The table below lists the methods in **Object** that are used for finalization.

Table 8-7 Finalization Methods in Object

Method	Description
weakTest	Answers **true** if the receiver is still referenced, **false** otherwise (meaning that it is about to be discarded).
weakFinalize	The default implementation in **Object** finalizes the receiver, replacing each invalid pointer with **nil**, and returns the number of elements set to nil. The message can be subclassed.

Methods in ApplicationProcess

The table below lists the methods in **ApplicationProcess** that relate to finalization and weak pointers.

Table 8-8 Finalization Methods in ApplicationProcess

Method	Description
gcCriticalSection	Answers the critical section of the garbage collector.
weakAdd:makeWeak:	This method is used for two purposes: if the `makeWeak` parameter is **true**, it makes the first parameter a weak object and registers it. if the `makeWeak` parameter is **false**, it registers the first parameter so that it gets notified via `weakFinalize` on each cycle.
weakRemove:	Use this method to un-register an object so that it becomes a normal object.

Example

Class **FinalizedObject** demonstrates finalization.

When the object is created, it is registered so that is receives the `weakFinalize` message on each garbage cycle.

```
new
    "
    Create a finalized object.
    "
    | answer |
    answer := super new.
    Processor weakAdd: answer makeWeak: false.
    ^answer
```

The method `weakFinalize` is implemented as below:

```
weakFinalize
    self weakTest ifFalse: [
        " being discarded:
        implement your finalization here "
        ...
    ]
```

Debugging and Diagnostic Messages

Debugging Messages

ApplicationProcess implements messages that print a string to an attached debugger. You can use `DBMON.EXE` from the SDK to view the messages.

See also *Debugging Issues* on page 76 for how to use debugging settings and *ApplicationProcess* on page 224 for the debugging protocol in ApplicationProcess.

Error Events

An application can also write an error event to the application log file, using `reportErrorEvent:`. The event is also shown on the attached debugger, if applicable. See also *Error Handling* on page 356.

The Compiler Interface

Overview

Class **ByteCodeCompiler** exposes many methods for code browsing and manipulation.

Variables

TLS Interface

Table 8-9 Compiler TLS Interface

Method	Description
addTlsSymbol:	Defines a new TLS variable.
removeTlsSymbol:	Removes a TLS variable
enumTLSVariables:	Enumerates TLS variables.

Global Variables

Table 8-10 Compiler Global Variable Interface

Method	Description
addGlobalSymbol:	Defines a new global variable.
enumGlobalVariables:	Enumerates global variables.
removeGlobalSymbol:	Removes a global variable.
resetGlobalVariables	Sets all public global variables to **nil**.

Browsing

Table 8-11 Source Code Browsing Interface

Method	Description
iterateOn: with:	Repeatedly evaluates a block with class and instance methods of the current class and subclasses.

Categories

Categories are represented by unique strings. This has two implications: first, changing a category name automatically affects all methods that are under this category. Secondly, only the category manipulation in Compiler should be used to make sure that the category names remain unique.

Table 8-12 Compiler Category Interface

Method	Description
addCatDescription: text:	Adds a category and its description only if the category does not already exist.
addCategory:	Adds a new category or returns the unique category string for a category.
browseReferencesToCategory:	Opens a MethodExplorer on all methods that belong to a given category.
defaultCategory	Returns the default category (i.e., uncategorized).
purgeCategories	Purges the set of categories by removing unused categories.
referencesToCategory:	Answers a collection of method descriptors that use a specified category.
renameCategory: to:	Renames a category.
setCatDescription: to:	Changes the description of an existing category.

Classes

Most of the class manipulation methods are private. Public methods can be found in **Behavior**.

Table 8-13 Compiler Class Interface

Method	Description
allSortedClasses	Answers a sorted array of classes in the system.

Compiling

Most of the class manipulation methods are private. Public methods can be found in **Behavior**.

Table 8-14 Compiler Interface

Method	Description
compile: [...]	Compiles a string.
evaluate: [...]	Evaluates a string and returns the result.
load: [...]	Loads a string (compiles it and updates the source code without installing the executable code).

Table 8-15 Compiler Method Interface

Method	Description
removeMethod: in:	Removes a method from a class.

Pool Dictionaries

Most of the class manipulation methods are private. Public methods can be found in **Behavior**.

Table 8-16 Compiler Pool Interface

Method	Description
poolAdd: value:	Adds a new pool dictionary.
getPoolDictionaries	Answers all pool dictionaries.
GetPoolDictionary:	Answers the pool dictionary at a specified pool name.
translatePool: from: to:	Translates a pool dictionary from ANSI to Unicode or vice-versa.

Files

Most of the class manipulation methods are private. Public methods can be found in **Behavior**.

Table 8-17 Compiler File Interface

Method	Description
fileIn:	Installs (files-in) a source file.
fileLoad:	Loads (compiles without installing the executable code) a source file.
saveImage:	Saves the image. The parameter indicates whether the image should be compressed.

Example

The code fragments below finds all methods in the image that belong to a given category and opens a **MethodExplorer** on the result set. It is functionally equivalent to using `browseReferencesToCategory:`.

```
| methods |
methods := OrderedCollection new.
Compiler iterateOn: Object with: [ :md |
    md category = 'My Category' ifTrue: [
        methods add: md
    ].
    true
].
(MethodExplorer new open contents: methods searchString: '')
```

Performance and Coding Issues

Abstract

Smalltalk MT is built for performance and professional development and has many features that help you resolve your performance problems.

Most methods in the base image have been tuned for performance and reduced code size.

You must also keep in mind that generated runtime images are much smaller and may sometimes be significantly faster than the Windows 98 development image.

Real-time Applications

Real-time applications require a guaranteed response time. The most important parameter in achieving real-time response times is the thread priority. A real-time capable application often has one or more high-priority threads that wait on input events and process these events quickly. The single most important factor is the latency; this is the time it takes a thread to wake up and actually start processing.

The latency is determined by the operating system and the garbage collection activity. A thread that is being garbage collected cannot respond to an event until the collection cycle finishes. Therefore, the factors to watch are object allocations and garbage collector scheduling. Tips on how to avoid memory allocations are provided in the next section.

Garbage collector thread priority

The garbage collector runs mainly when object allocations occur. If the system does not consume much memory, the garbage collector is idle and does not use any processor time. Raising the garbage collector's priority ensures that higher priority threads in the

system do not starve the collector thread. The handle to the garbage collector thread is in the instance variable `m_hGC` in **ApplicationProcess**.

Forcing garbage collection cycles

An application can ensure that collection cycles do not interfere with critical actions by manually forcing a GC cycle at appropriate intervals. This ensures that, at a given point, all unreferenced memory has been recycled.

Starting and stopping the garbage collector

A thread can start and stop the garbage thread by sending respectively `gcStart` and `gcStop` to Processor. The thread must not allocate any objects from the Smalltalk heap while the garbage collector is stopped, otherwise a deadlock could occur. These methods are used to prevent other threads from interfering with the current thread, e.g., a secondary thread forcing a GC cycle that disrupts the execution of a critical thread.

Operating system and hardware issues

The garbage collector uses handcrafted assembler code that is able to scan tens of megabytes of memory without noticeable interruptions. For example, the standard Windows 98 image maps the whole source file into memory, therefore scanning 16 MB of source data, notwithstanding allocated memory and static data. A final executable image generally only scans a couple of hundred kilobytes.

In Windows NT EXE mode, the image uses exception handling to track dirty pages. This often reduces the data to scan dramatically.

A thread that creates lots of garbage is frequently obliged to wait for the garbage collector to complete. Because the user and GC threads may only use a fraction of their allocated time slice, the time it takes the operating system to schedule threads becomes an important factor. This accounts in part for the performance difference in computation-intensive tasks between Windows 98 and Windows NT.

Summary

The best operating system - hardware combination for real-time applications is Windows NT on a multiprocessor machine. An application that does not have heavy memory requirements such as those required by a loop creating a large number of

objects generally performs smoothly. Time-critical code that does not perform allocations can be insulated from other threads by stopping and restarting the garbage collector.

Steps

The first step is to identify the performance sensitive areas. If you do not have performance problems, there is no need to waste time optimizing your code. You may just introduce some bugs while doing so.

In most cases, the origin of a performance problem is an inadequate algorithm. The next candidate is inefficient coding, for example evaluating a constant expression repeatedly in a loop.

Once you have eliminated the causes above and still think your code does not run as fast as it should, examine the following points:

- memory consumption
- using low-level methods

Trim the Fat!

After they have passed the initial prototyping stage, most applications are loaded with unnecessary code. You can use the Class Hierarchy Browser to list methods that are never called (this code is not included in the final executable anyway, but removing it will reduce the maintenance effort).

Another point to consider is how you use instance variables, via accessor methods or directly. While the coding issue is somewhat controversial (but we leave this up to you), the performance issue is not. In Smalltalk MT, a direct variable access is done in one or two native machine instructions (which is as fast as you can get), while a method call involves at least a dozen machine cycles.

Reducing Memory Allocations

Most performance problems in Smalltalk are related to memory management. This is because the garbage collector must scan large amounts of memory during each cycle,

and the more allocations are being done, the more it cycles. Several techniques reduce memory consumption:

- Allocate objects once and reuse them. For example, you can keep them referenced by an instance variable.
- Look for blocks that are evaluated repeatedly. Each non-static block is allocated on the Smalltalk heap.
- If a method creates and returns a new object, consider passing an argument to the method in which the information is stored. For example, if a method returns a rectangle, you can rewrite it so that it takes a rectangle as argument and updates it. This allows you to call that method in a loop without incurring a garbage collection penalty.
- Use locally allocated byte objects. The pseudo-messages `localNew` and `localNew:` create an object on the stack. However, be sure you do not reference local objects by non-temporary objects and contexts.

Optimizing block contexts

Smalltalk MT distinguishes between three types of blocks:

- Context blocks require a home context, which is allocated on the Smalltalk heap.
- Static blocks are pre-allocated and do not use the context of the defining method.
- Pseudo-blocks are inlined (for example, `ifTrue:` , `ifTrue:ifFalse:` and `to:do:`).

You can turn a context block into a static block by providing the information it needs in additional arguments. The advantage of static blocks is that they do not require memory allocations.

For example, consider the block below:

```
| a |
...
something do: [ :i | i + a ].
```

The block is non-static because it uses the variable `a`. If you are unsure, highlight the block and select *Display*. A static block will print itself, otherwise you have a syntax error due to an undefined variable.

If you replace the code above with:

```
| a |
...
something inject: [ :value :i : | i + value. value ] with: a
```

the block is static and the method's context is allocated on the stack. If you highlight the block and evaluate it, it shows `aStaticBlock`.

Inlining

Code inlining is an option that avoids blocks altogether. The compiler automatically inlines a certain number of messages (`ifTrue`, `ifFalse`, `whileTrue`, `whileFalse`, `and`, `or`) when the parameter is a literal block. The code that gets generated is similar to the one generated by a C or Pascal compiler. Loops (`to:do:`) are also inlined and an exception is generated when the parameters are not integers. In the case of a `to:by:do:` loop, the code is only inlined when the increment is a literal integer (and the block must also be literal).

Using `to:do:` type loops is most useful when iterating over sequenceable collections. For example, given the code below:

```
anArray do: [ :e | ... ]
```

The inlined counterpart looks like:

```
1 to: anArray size do: [ :i |  (anArray at: i) ... ].
```

The code generated by the compiler resembles the following pseudo-code:

```
| i s|
s := anArray size.
i := 1.
[ i <= s ] whileTrue: [
    (anArray at: i) ...
    i := i + 1
].
```

The expression `whileTrue:` generates a comparison and a branch. The benefit of this code is that it avoids a context allocation and provides better locality of reference than block (context or static) based code.

If the receiver is known to be an **Array**, you can also use:

```
anArray basicDo: [ :i :element |... ].
```

The line generates inline code that directly accesses each element of a pointer object. The loop does nothing if `anArray` has no elements.

CHAPTER 9 Inline Assembler

This chapter presents the built-in inline assembler. Since Smalltalk MT is natively compiled, assembler statements fold naturally into the environment. Even if you do not intend to code assembler routines, you can read this chapter to gain better insight into the architecture.

The chapter concludes with an example that implements a proxy.

Introduction

The Smalltalk MT inline assembler lets you inline assembler expressions, or even implement whole methods in assembler. This gives developers the opportunity to fine-tune time-critical code sections and implement low-level operations in assembler.

The generated code integrates seamlessly into the Smalltalk environment. You can use assembler to implement low-level functions such as for example bit shifts, or to improve the speed of a particular method. For example, implementing the message `_doesNotUnderstand` requires assembler coding.

In order to reduce the size of the compiler, the inline assembler uses a hand-coded parser with limited syntax and error checking. The syntax follows closely the Intel reference books.

Syntax

An inline assembler statement starts with the keyword **_asm{** and continues until the closing parenthesis. To Smalltalk, the assembler block is an expression, so you can for example code:

```
| stack_pointer  |
stack_pointer := (_asm{ mov EAX, ESP }) basicAddress.
```

The expression assigns the current value of the stack pointer to `stack_pointer`. The pseudo-message `basicAddress` performs the integer translation into a Smalltalk integer instance.

Machine Instructions

All operands must be in lower case (`add`, `mov`, ...), while registers can be in uppercase (`EAX`, `EBX`, ...) or lowercase (`eax`, `ebx`, ...).

Addressing Modes

The assembler supports all addressing modes.

Table 9-1 Addressing Modes

Syntax	Example
[REG]	mov EAX, [EBX]
[REG+offset]	mov EAX, [EBX+4]
[ESP+1*NIL+offset]	Replaces [ESP+offset], which is not supported in this form by the processor.
[REG1+index*REG2+offset]	mov EAX, [EBX+4*ECX-4]
[mem]	mov EAX, [16r410000]

- *offset* is a 32 bit signed integer.
- *index* is one of the values 1, 2, 4, 8
- `REG`, `REG1`, `REG2` are registers: `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP`, `ESP`

Immediate Values

In addition to integers, you can use hexadecimal numbers in Smalltalk notation (`16rxxx`) as well as symbols.

Example:

```
mov EAX, 16r1000
mov EBX, #size
```

It is also possible to use constants defined in one of the following pool dictionaries:

`_StImageConstants`

`_StInternalConstants`

`ExceptionConstants`

`SmalltalkConstants`

The offset part of an indirect addressing mode can also be a pool constant.

Example:

```
mov EBX, ST_EXCEPTION_ABORT
mov EBX, [EAX+OFFSET_BEHAVIOR_SUPERCLASS]
```

Local Labels

A local label is defined on a separate line, and must be terminated with $:.

Example:

```
mylabel:
    dec ECX
    jne mylabel:
```

Global Labels

Assembler code can reference all global Smalltalk objects, plus specific global routines. Global assembler labels are prefixed with $, and not terminated with $:.

Example:

```
mov EAX, [Processor]
mov EBX, Object
call $_isKindOf
```

Table 9-2 Addressing Smalltak Globals

Syntax	Example
Global Variable	`mov EAX, [Processor]`
Class	`mov EAX, Object`

Notable Syntax Exceptions

Certain Intel x86 instructions require a particular coding that does not follow the general rule. Because the inline assembler is mostly table driven, it does not recognize alternate syntactical equivalents and generates either the wrong code or fails. If in doubt, refer to the microprocessor manual.

Table 9-3 Syntactical Exceptions

Expression	Correct Encoding
[ESP+offset]	`[ESP+1*NIL+offset]`
int 3	`int3`

Assembler and Smalltalk Methods

Calling Convention

Smalltalk MT internally uses a subset of the WINAPI calling convention. The receiver object is always expected in `EAX`, a return value is also in `EAX`. The method must preserve `EDI` and can change any other register (this is where the calling conventions differ).

Using inline assembler statements

There are two possible uses of the **_asm** directive:

- If the assembler statement is the first statement of a method, the compiler does not generate a context and you are responsible for returning from the method. This is useful for implementing an entire method, but does still allow you to append Smalltalk code.
- Otherwise, the compiler generates a context and returns from the method.

Example:

```
LargeInteger>>lobyte
    "
    Answers the low-order byte of the receiver.
    Return Value:
        An Integer.
    "
    _asm{
        mov EAX, [EAX]
        and EAX, 16rFF
        lea EAX, [EAX+1*EAX+1]
        ret
    }
```

Table 9-4 Register Usage

Register	Contents
EAX	Receiver or result of expression
EBX, ECX, EDX, ESI	General purpose
EDI	Must be restored after use
EBP	Base Pointer
ESP	Stack Pointer

Object Types

There are two fundamental object types in Smalltalk MT: **SmallInteger** instances and others. A **SmallInteger** is a compact representation of an integer that is obtained by shifting the value left and adding 1. That way, it can be distinguished from an object by testing the first bit (object structures are always aligned on either 32 or 64 bit boundaries).

Extracting a 32 bit value from an integer

The 32-bit value is therefore obtained by testing the first bit, and if it is set to 1, shifting right by one position, otherwise the value is encoded in a **LargeInteger** or **LONG**. The C macro below performs such a conversion:

```
#define MAKE_STDWORD(a) ( a & 1 ? a >> 1 : *(*ULONG)a )
```

Converting a 32 bit value to an integer

The opposite is a bit more complicated because a **LargeInteger** must first be allocated if the result overflows. Code that performs this conversion typically looks as follows:

```
    add EAX, EAX
    jo makeLargeInt:
    or EAX, 1
    ...

makeLargeInt:
    call $_overflow
        ...
```

The routine `$_overflow` is used to convert an already shifted value to a **LargeInteger**, and takes care of sign extending the result.

Public Assembler Routines

The assembler library is part of the runtime environment of Smalltalk MT.

_performIntSelector

Performs a selector (in EBX) on the receiver (in EAX). The receiver must be a **SmallInteger**.

Input

Register	Description
EAX.	Receiver object (a SmallInteger)
EBX	Selector (symbol id)

_performSelector

Performs a given selector on an object.

Input

Register	Description
EAX	Receiver object (not a **SmallInteger**)
EBX	Symbol id

Output

Register	Description
EAX	Result of evaluating the message

Example:

```
mov EAX, Array
mov EBX, #new
call $_performSelector
```

_performUSelector

Performs a given selector on an arbitrary receiver.

Input

Register	Description
EAX	Receiver object (accepts also **SmallInteger**)
EBX	Symbol id

Output

Register	Description
EAX	Result of evaluating the message

_stalloc

Allocates an empty object. The routine always returns a valid object or raises an exception.

Input

Register	Description
EAX	Size in bytes (including the header).

Output

Register	Description
EAX	New (empty) object

The routine allocates a block of memory. The size is in EAX, the result is returned in EAX as well. The return value points to the net data, which is preceded by an 8-byte header. The caller is responsible for setting the method dictionary. The method

dictionary is always in the first instance variable of the class, so the following code allocates an 8-byte **ByteArray**:

Example:

```
mov EAX, 16
call $_stalloc
mov EBX, [ByteArray]
mov [EAX-SIZE_OF_HEADER], EBX
```

Note An allocation either succeeds or raises an exception, so there is no need to check for an error condition.

_isKindOf

Tests if the object in EAX is kind of a given class, passed in EBX.

Input

Register	Description
EAX	Receiver (no SmallInteger).
EBX	Class to test against.

Output

Register	Description
EAX	A **Boolean** (**true** or **false**)

_overflow

If an overflow occurs when converting a 32-bit value in EAX to a **SmallInteger**, this routine converts the result to a **LargeInteger** instance.

Input

Register	Description
EAX	overflowed small integer

Output

Register	Description
EAX	a LargeInteger instance

The routine is typically used as follows:

```
; EAX contains a 32-bit value to be converted
    add EAX, EAX
    jo ov:
    or EAX, 1
proceed:
    …

ov:
    call $_overflow
    jmp proceed:
```

Example: Implementing a Proxy

When the selector lookup routine fails to find an entry, it sends the message `#_doesNotUnderstand` to the receiver. The default implementation of this method in **Object** builds a `#doesNotUnderstand:` message and sends it to the receiver. This is rather time-consuming, because the routine must first allocate a **Message** object and fill in the parameters. The method below implements a proxy (**Proxy** is a subclass of **nil** that sends all messages to a target object, which is in its first instance variable).

```
_doesNotUnderstand
    "
    Send the message to m_target or #fault if the target is nil.
    "
    _asm{
        mov ESI, EAX
        mov EAX, [EAX]  ; load m_target
        mov EBX, EDX    ; EBX=unknown selector
        cmp EAX, nil
        jne $_performSelector; perform if not nil
        push EBX        ; save unknown selector
        mov EAX, ESI    ; reload proxy
        mov EBX, #fault
        call $_performSelector ; perform #fault
        pop EBX         ; restore unknown selector
        jmp $_performSelector
    }
```

The code dispatches the message in just five machine instructions (not counting the initial unsuccessful message lookup).

Appendix Pseudo Messages

Pseudo Messages in Object

The messages listed below are compiled inline, without validation of the arguments. An incorrect use may result in corruptions. Pointer object messages can only be used with pointer objects, while byte object messages are only meaningful with byte objects.

Table 0-1 Object Testing

Method	Description
==	Answers true if the receiver and the argument are identical.
~~	Answers true if the receiver and the argument are not identical.
isNil	Answers true if the receiver is (identical to) nil.
notNil	Answers true if the receiver is not (identical to) nil.
isSmallInteger	Answer whether the receiver is a small integer.
_isKindOf:	Answer true if the receiver inherits from the argument (a Class).
isMemberOf:	Answer true if the receiver is an instance of the argument (a Class).

Table 0-2 Object Accessing

Method	Description
basicAddress	Answer the address at which the object is stored.
_class	Answer the class of the receiver. The receiver must be a true object (not a SmallInteger).
_size	Answer the number of 32 bit words in the receiver.
_sizeInBytes	Answer the number of bytes in the receiver.

Table 0-3 Miscellaneous Messages

Method	Description
case:{perform:}[default]	Tests against constants, performs the block for which the test was successful, or an optional default block.

Table 0-4 Pointer Access Messages

Method	Description
_at:	Answer the instance variable at an index.
_at:put:	Set the instance variable at an index.

Table 0-5 Binary Access Messages

Method	Description
_byteAt:	Answer the byte at a one-based index.
_byteAt:put:	Set the byte at a one-based index.
_doubleAtOffset:	Answer the double-precision float at a zero-based offset.
_doubleAtOffset:put:	Set the double-precision float at a zero-based offset.
_floatAtOffset:	Answer the single-precision float at a zero-based offset.
_floatAtOffset:put:	Set the single-precision float at a zero-based offset.
_longAt:	Answer the signed long at a one-based index.
_longAt:put:	Set the signed long at a one-based index.
_longAtOffset:	Answer the signed long at a zero-based offset.
_longAtOffset:put:	Set the signed long at a zero-based offset.
_long64AtOffset:	Answer the signed 64-bit integer at a zero-based offset.
_long64AtOffset:put:	Set the signed 64-bit integer at a zero-based offset.
_wordAtOffset:	Answer the signed word at a zero-based offset.
_wordAtOffset:put:	Set the signed word at a zero-based offset.

Pseudo Messages in MemoryManager class

Table 0-6 Address Manipulations

Method	Description
byteAtAddress:	Answer the byte at a specified memory address.
wordAtAddress:	Answer the 16-bit word at a specified memory address.

atAddress:	Answer the signed long at a specified memory.
atAddress:put:	Set the signed long at a specified memory address.
atAddress:add:	Add a signed long at a specified memory address.
atAddress:sub:	Substract a signed long at a specified memory address.
atAddress:bitXor:	Performs an XOR at a specified memory address.
atAddress:bitOr:	Performs an OR at a specified memory address.
atAddress:bitAnd:	Performs an AND at a specified memory address.
atAddress:addByte:	Add a byte at a specified byte memory address.
atAddress:putByte:	Set the byte at a specified memory address.
atAddress:putWord:	Set the word at a specified memory address.
atAddress:addLong:	Add a signed integer to the 32 bit contents at an address in memory.
atAddress:subLong:	Substract a signed integer from the 32 bit contents at an address in memory.
atAddress:putUnsigned:	Set the 32 bit contents at an address in memory to an unsigned integer.

Function and Method Calls

These messages allow you to call a Smalltalk method, an external function, or send a message to an object. Because the messages are inlined (except for `call:withArguments:`), performance is optimal.

Table 0-7 MemoryManager class Messages

Method	Description
call: [receiver:] {with:|with*nn*:}	Call a method given its address. `with:` precedes a normal parameter, `with`*`nn`*`:` defines a multi-byte parameter passed by value.
call:withArguments:	Call the routine at an address with the arguments given in an Array.
callAPI: {with:}	Call an external function using the WINAPI calling convention.
stdcallAPI:	Call an external function using the C calling convention.

Table 0-8 Object Messages

Method	Description
_perform: {with:}	Performs a selector symbol with zero or more arguments.
__perform:	Performs a selector, identified by its symbol id, with zero or more arguments.

Type Conversion Messages

Type conversion messages convert an entity between two known types. Type conversion messages are used when both the receiver entity and the expected result are known in advance.

Table 0-9 Type Conversion Messages

Method	Description
_asShort	Specifies that a raw 32 bit value is a short integer. The receiver is usually the result of an API call. The message is an optional optimization.
_asInteger	Specifies that the receiver is a **SmallInteger** instance. This message is often used to simplify the conversion of a SmallInteger to a raw 32-bit value used in a function call. The message is an optional optimization.
_asObject	Converts an address to an object. The receiver must be a raw 32-bit value specifying the address of an object. This message optimizes `Object>>_fromAddress:`.
_makeObject	Specifies that the receiver is the address of an object. The receiver must be an **Integer** instance. This message optimizes `Object>>_fromAddress:`.

Iteration Messages

Inline iteration messages iterate over a pointer object, an OrderedCollection or a MappingTable. If the receiver is not of the expected type, an error is raised (full optimization suppresses the assertion). Because the iteration is inlined, these constructs can be substantially faster than a traditional block iteration.

Table 0-10 Iteration Messages

Method	Description
basicDo: [:i :e | ...]	Evaluates the specified block for each element of the receiver. The first parameter is the one-based index of the element, the second the element. This message works with all pointer objects, not just arrays.
_do: [:e | ...]	Evaluates the specified block for each element of the receiver. The receiver must be an **OrderedCollection**.
_keysValuesDo: [:key :value | ...]	Evaluates the specified block for each key - value pair of the receiver. The receiver must be a **MappingTable**.

Appendix Glossary

A

Abnormal termination
In exception handling, the condition that occurs when a program leaves the try block of a try-finally statement with a return statement.

Abstract class
A class whose purpose is to provide common functionality to concrete subclasses. An abstract class is not meant to be instantiated and usually raises an error when receiving the message `new`. For example, Collection is an abstract class.

Accelerator
An accelerator key is a sequence of keystrokes that invoke a particular command in an application window. See also Accelerator table.

Accelerator table
A data structure (usually a resource) that contains a list of accelerator keys and the command identifiers associated with them.

ActiveX
A set of technologies that enable software components to interoperate in a networked environment. The ActiveX technologies are enhancements to OLE, Microsoft's component software technology. ActiveX Controls are small, efficient modules that implement specific, specialized functions.

Ambient property
OLE: A run-time property that is managed and exposed by the container. Typically, an ambient property represents a characteristic of a form, such as a background color, that needs to be communicated to a control so that the control can assume the look and feel of its surrounding environment.

ANSI character set
An 8-bit character set that contains the 7-bit ASCII standard character set as well as currency and mathematical symbols, accented characters, and other characters not normally found on the keyboard.

ANSI string
A string composed of characters from the ANSI character set. See also String.

Apartment thread
A thread used to execute calls to objects of components configured for apartment-model threading. Each object "lives in an apartment" (thread) for the life of the object. All calls to that object execute on the apartment thread. This threading model is used, for example, for component implementations that keep the object state in thread-local storage (TLS). A component's objects can be distributed over one or more apartments.

API (Application Programming Interface)
A set of standard functions exposed by Windows or third party dynamic link libraries. Typically, APIs are used to carry out low level tasks.

Application
A unit of software that performs a distinct task. An application is composed of a set of collaborating classes that provide the functionality.

Typically, an application encapsulates the initialization, running, and termination of a Windows-based application.

Argument
An object made accessible to the receiver of a message.

Array
A collection of objects that can be accessed from one to the size of the collection.

Assignment
A statement that assigns a value to a variable. An assignment statement is usually composed of a destination variable, the assignment operator (=), and an expression to be assigned.

Asynchronous call
A call to a function that is executed separately so that the caller can continue processing instructions without waiting for the function to return. Contrast with Synchronous call.

Automation
A way to manipulate an application's objects from outside the application. Automation is typically used to create applications that expose objects to programming tools and macro languages, create and manipulate one application's objects from another applications, or to create tools for accessing and manipulating objects.

B

Background color
The color of the client area of an empty window or display screen, on which all drawing and color display take place.

Behavior
Usually refers to the set of messages to which an object can respond. Occasionally refers to the internal data and methods an object uses to carry out its external behavior.
Smalltalk: the class used to represent behavior.

Binary message
A message with a single argument whose selector is not a keyword. For example, arithmetic messages, such as `1 + 2`, are binary messages.

Block
A set of deferred expressions delimited by square brackets. Blocks are evaluated when they receive the message `value`.

Boolean
A binary algebra that uses the logical operators AND, OR, XOR, and NOT, and whose outcomes consist of logical values (either TRUE or FALSE). The keyword boolean indicates that the expression or constant expression associated with the identifier takes the value TRUE or FALSE. See also conditional expression.
Smalltalk: boolean values are represented by the objects **true** and **false**, respectively the sole instances of the classes True and False.

Breakpoint

A location in a program where execution is stopped to allow the developer to examine the program's code, variables, register values and, as necessary, to make changes, continue execution, or terminate execution.

Smalltalk: debugger-level breakpoints are `self halt`, which stops the current thread and opens a debugger, and `self stop`, which raises a continuable exception. The low-level breakpoint `self _break` can be used to open an external COFF debugger.

Browser

A graphical tool that allows you to visually inspect and edit source code and objects.

Build

The process of compiling and linking source code to generate an executable program or dynamic link library.

Byte

A unit of information consisting of 8 bits. A byte, or "binary term," is the smallest collection of bits that can be accessed directly.

Byte order mark (BOM)

The Unicode character U+FEFF—or its non-character mirror-image, U+FFFE—used to indicate the byte order, or its non-character mirror image, of a text stream. The presence of a BOM is a strong clue that a file is encoded in Unicode.

Bytecode

Machine-independent code generated by the Java compiler and executed by the Java interpreter or compiled at the last minute by a JIT compiler.

C

C calling convention

The C standard for calling a function. Arguments are pushed onto the stack from right to left (in reverse order from the way they appear in the argument list). After the function returns, the calling function removes the arguments from the stack. The C calling convention permits a variable number of arguments to be passed.

Call stack

An ordered list of methods that have been called but have not returned.

Callback

A Smalltalk method that can be called (directly or indirectly) from a procedure outside of Smalltalk, such as a Windows API. For example, Windows messages are callbacks because they are executed in response to GUI events. See also exported method.

Caller

A client that invokes a method of an object.

Calling convention

A convention that determines the order in which arguments passed to functions are pushed on the stack (the calling sequence) and whether the calling or called function removes the arguments from the stack.

Cascaded message

Cascaded messages are a series of messages to the same receiver. Cascaded messages are separated by semicolons. A period indicates the end of the cascaded message group.

Change log

A file (`CHANGES.SL`) that maintains a continuous record of source code changes, project installations, image save timestamp messages, and evaluations.

Character

A member of the alphabet, a number, a punctuation mark, or any other mark used in a writing system.

Child window

A window that has the WS_CHILD or WS_CHILDWINDOW style and is confined to the client area of its parent window, which initiates and defines the child window. Typically, an application uses child windows to divide the client area of a parent window into functional areas.

Class

An object that defines the interface of a particular kind of object. A class definition defines instance variables and methods, class variables and methods, and specifies the immediate superclass. The class specifies external behavior as well as internal knowledge and methods.

Class factory

OLE: A COM object that implements the IClassFactory interface and that creates one or more instances of an object identified by a given class identifier(CLSID). See also class identifier.

Class hierarchy

A collection of classes that share a common ancestor. The descendants of a class are its subclasses. The immediate ancestor of a class is its superclass.

Class identifier (CLSID)

OLE: A globally unique identifier (GUID) associated with an OLE class object. If a class object will be used to create more than one instance of an object, the associated server application should register its CLSID in the system registry so that clients can locate and load the executable code associated with the object(s). Every OLE server or container that allows linking to its embedded objects must register a CLSID for each supported object definition.

Class instance variable

A variable defined to be part of a class. Subclasses inherit the name of the variable, but not its value.

Class method

A method invoked by sending a message to a class, rather than an instance of a class.

Class variable

A single variable whose name and value are shared by a class and its subclasses, as well as the instances of the class and subclasses.

Client area

Or client rectangle. The portion of a window where the application displays output such as text or graphics.

Client coordinates

An ordered pair (x,y) of numbers, relative to the origin (usually the upper-left corner of a window's client area), that designates a point in the client area.

Clipboard

An area of storage, or buffer, where data objects or their references are placed when a user carries out a cut or copy operation.

COFF

A format for executable and object files that is portable across platforms. The Microsoft implementation of COFF is derived from the UNIX specification for COFF, but includes additional headers for compatibility with MS-DOS and 16-bit Windows. This Microsoft version is sometimes called the "portable executable (PE) file format."

COFF Symbol

The COFF debug section contains the debugging information of an executable image. The debug section contains the COFF Symbol Table, which defines a symbol or name and its location in the image.

COM (Component Object Model)

An open architecture for cross-platform development of client/server applications based on object-oriented technology. Clients have access to an object through interfaces implemented on the object. COM is language neutral, so any language that produces ActiveX components can also produce COM applications.

Command identifier (ID)

An identifier that associates a command message with the user-interface object (such as a menu item, toolbar button, or accelerator key) that generated the command. Typically, command IDs are named for the functionality of the user-interface object they are assigned to. For example, a Clear All item in the Edit menu might be assigned an ID such as ID_EDIT_CLEAR_ALL.

Command line

A string of text typed at the command prompt, or executed from a command file, that specifies a task or tasks for the operating system or an application to perform.

Common dialogs

Standard dialog boxes defined by Windows—such as Open, Save As, Print, and Find—that applications can use.

Compiler

A program that translates source code into directly executable machine code.

Component

An object that encapsulates both data and code, and provides a well-specified set of publicly available services.

Component Object Model (COM)

OLE: The OLE object-oriented programming model that defines how objects interact within a single process or between processes. In COM, clients have access to an object through interfaces implemented on the object.

Concrete class

A class that provides a concrete implementation and that can have instances.

Conditional expression

An expression that yields a Boolean value (true or false). Such expressions can involve comparisons, using relational operators such as the less-than (<) and greater-than (>) operators, and logical combination of Boolean expressions, using Boolean operators such as bitwise AND (&) and logical OR (||).

Console

An interface that provides input and output to character-mode applications.
The Windows subsystem that runs character-based applications, as opposed to applications that have a graphical user interface (GUI).

Console application

A character-mode application that uses a console window for its input and output. If necessary, the operating system will create a new console window, which exists until the application terminates.

Control

Windows: A child window an application uses in conjunction with another window to perform simple input and output (I/O) tasks. Controls are most often used within dialog boxes, but they can also be used in other windows.
OLE: An embeddable, reusable COM object that supports, at a minimum, the IOleControl interface. Controls are typically associated with the user interface. They also support communication with a container and can be reused by multiple clients.

Control container

OLE: An application that supports embedding of controls.

Control identifier (ID)

A 16-bit value that an application uses to uniquely identify a child control. This ID is used in notification messages to the parent window when events, such as input from the user, occur in the control.

Control property

OLE: A run-time property that is exposed and managed by the control itself. For example, the font and text size used by the control are control properties.

Critical section

A segment of code which is not reentrant; that is, it does not support concurrent access by multiple threads. Often, a critical section is used to protect shared resources.

Control structure

A language construct that controls the flow of processing such as looping and branching. Control structures are implemented in Smalltalk using messages with blocks as arguments or receivers.

D

DDE

A form of interprocess communications that uses shared memory to exchange data between applications. DDE can be used for one-time data transfers and for ongoing exchanges by applications that send updates to one another as new data becomes available.

Deadlock

A situation in which two or more threads are permanently blocked (waiting), with each thread waiting for a resource exclusively held by one of the other threads that is blocked.

Debugger

A program designed to help find errors in another program by allowing the programmer to step through the program, examine data, and check conditions.

Decorated name

A string generated by a C++ compiler that contains an undecorated (literal) name followed by a string of characters that the compiler and linker use to retain type information.

Default window procedure

A system-defined function that defines certain fundamental behavior shared by all windows. A default window procedure provides default processing for nonclient–area messages, system commands, system keystrokes, and other messages that the application-defined window procedure does not specifically handle.

Device context

A data structure defining the graphic objects, their associated attributes, and the graphic modes affecting output on a device.

Dialog box

In Windows, a window used to retrieve user input. A dialog box usually contains one or more controls, such as buttons, list boxes, combo boxes, and edit boxes, with which the user enters text, chooses options, or directs the action of the command.

Dialog template

A template used by Windows to create a dialog window and display it. The template specifies the characteristics of the dialog box, including its overall size, initial location, and style, and the types and positions of its controls. A dialog template can be stored as a resource or directly in memory.

Dialog unit

A unit of horizontal or vertical distance within a dialog box. A horizontal DLU is the average width of the current dialog-box font divided by 4. A vertical DLU is the average height of the current dialog-box font divided by 8. Dialog units allow a font-independent layout of dialog boxes.

Dictionary

An object that contains associations between keys and values.

Dispatch interface

OLE: the external programming interface of some grouping of functionality exposed by the automation server. See also Dual interface.

Distributed COM (DCOM)

DCOM is an object protocol that enables ActiveX components to communicate directly with each other across a network. DCOM is language neutral, so any language that produces ActiveX components can also produce DCOM applications.

DLL

See Dynamic-link library file.

Do-it

Evaluates the current selection in the active text pane as a Smalltalk expression. See also Show-it.

Drag and drop

An operation in which the end user uses the mouse or other pointing device to move data to another location in the same window or another window.

Dual interface

An interface that derives from IDispatch and supports both late-binding via IDispatch and early-binding (VTBL binding) via direct COM methods for each of its automation methods.

Dynamic-link library file (DLL)

A file that contains one or more functions that are compiled, linked, and stored separately from the processes that use them. In Win32, the operating system maps the dynamic-link libraries (DLLs) into the address space of a process when the process is starting up or while it is running. The process then executes functions in the DLL. Dynamic-link library files usually have a `.DLL` filename extension.

E

Encapsulation

In object-oriented programming, the process of hiding the internal workings of a class to support or enforce abstraction. A class's interface, which is public, describes what a class can do, while the implementation, which is private or protected, describes how it works.

Entry point

A starting address for a function or method, executable file, or dynamic-link library.

Event

An action or occurrence, often generated by the user, to which a program might respond. Typical events include keystrokes, mouse movements, and button clicks.

Event object

A synchronization object that allows one thread to notify another that an event has occurred. Event objects are useful when a thread needs to know when to perform its task. For example, a thread that copies data to a data archive would need to be notified when new data is available. By using an event object to notify the copy thread when new data is available, the thread can perform its task as soon as possible.

Exception

An abnormal condition or error that occurs during the execution of a program and that requires the execution of software outside the normal flow of control. Examples of exceptions are running out of memory, resource allocation errors, and failure to find files.

Exception handler

A block of code that reacts to a specific type of exception. If the exception is for an error from which the program can recover, the program can resume executing after the exception handler has executed. In this case, execution will resume where the exception was handled, not at the place where it was generated.

Executable file

A program file created from one or more source code files translated into machine code and linked together. The MS-DOS, Windows, and Windows NT operating systems use the `.EXE` filename extension to indicate that the file is a runnable program.

Exported method

A method called by non-Smalltalk code. The following cases arise:

- Class methods can be exported from a Smalltalk DLL.
- Instance methods can be called as part of a COM interface.
- Both class methods and blocks can be called by non-Smalltalk code, for example by passing a block or the address of a class method to a system function.

Expression

A message expression is a request to an object (the receiver of the message) to perform a computation and return an object as the answer. There are three kinds of message expressions: unary, binary, keyword.

F

F1 Help

Context-sensitive Windows Help that the user obtains by pressing the F1 key. F1 Help opens Help on a topic associated with the currently selected item in the application.

Fatal error

Or unrecoverable error, catastrophic error. An error that causes the system or a program to fail abruptly with no hope of recovery. An example of a fatal error is an uncaught exception that cannot be handled.

File-in

A file-in is Smalltalk source code that can be added to the image. See also file-out.

File-out

The process of saving Smalltalk source code to disk. The source code can include single methods, classes as well as entire hierarchies and pool dictionaries.

Focus

A temporary property of a user-interface object, such as a window, view, dialog box, or button, that permits the object to receive keyboard input from the user. The focus is usually conveyed through highlighting. See also top-level window.

Font

Any of numerous sets of graphical representations of characters that can be installed on a computer or a printer.

Frame

The top-level application window. The frame window provides a visible frame around a view, with an optional status bar and standard window controls such as a control menu, buttons to minimize and maximize the window, and controls for resizing the window. The frame window is responsible for managing the layout of its child windows and other client-area elements such as control bars and views. The frame window also forwards commands to its views and can respond to notification messages from control windows.
OLE: The part of a container application responsible for negotiating menus, accelerator keys, toolbars, and other shared user-interface elements with an embedded COM object or a document object.

Framework

A reusable design; a set of classes, usually including abstract classes, that provides the skeleton of an application. Classes in a framework collaborate in ways that are not specific to any particular application, but rather define a structure that a variety of applications could have.

Function profiling

A run-time analysis of code execution by function, in which the profiler detects inefficiencies by counting and timing functions.

G

Garbage collector

A process that periodically frees the memory used by objects that are no longer needed.

Global variable

A variable that is accessible from anywhere in an image.

Guarded block of code

A block for which an exception or termination handler provides protection.

H

Handle

A 32-bit value that represents a Windows object such as a file, bitmap, or window.

Hash value

The value of a key that has been numerically manipulated to directly calculate either the location of its associated record in a table or the starting point for a search for the associated record. If the key value is a character string, each possible character is assigned a numeric code to permit the numerical manipulation. The manipulation performed on the key value is known as the hashing function.
Smalltalk: instances of Dictionary, MappingTable etc. use hashing to locate the value associated with a key, and Set uses hashing to quickly find elements. Hashing improves the time it takes to locate an element.

Heap

A portion of memory reserved for a program to use for the temporary storage of data structures whose existence or size cannot be determined until the program is running. The program can request free memory from the heap to hold such elements, use it as necessary, and later free the memory.

Help file

A file that contains text and graphics needed to communicate online information about an application. Each help file contains one or more topics a user can select by clicking hot spots, using the keyword search, or browsing through topics.

Hexadecimal

Or hex. The base-16 counting system, whose digits are 0 through F. The letters A through F represent the decimal numbers 10 through 15.

HRESULT

OLE: An opaque result handle used by OLE functions to indicate success or an error condition.

HTML
A markup language derived from SGML. Used to create a text document with formatting specifications that tells a software browser how to display the page or pages included in the document.

HTML Help
A new Help system based on Microsoft's HTML browser control rather than on the traditional WinHelp API. See also Help file.

I

Icon
A bitmap that is used as a visual mnemonic for an application or message. An icon is usually stored in an application's resource-definition file.

IDL
The OSF-DCE standard language for specifying the interface for remote procedure calls.

Idle time
The period during which the application has an empty message queue. Idle time permits the processing of background tasks.

Image list
A Windows object that represents a collection of same-sized images in a single, wide bitmap. Image lists are used to efficiently manage large sets of icons or bitmaps.

Inheritance
In object-oriented programming, a method for deriving new classes from existing classes. The derived class inherits the description of its ancestor class, but can be extended by adding new variables and methods.

Inline assembler
Assembly-language code that is inserted into Smalltalk source code. The `_asm` keyword preceding an assembly-language statement or block of statements invokes the inline assembler at compile time.

In parameter
OLE: A parameter that is allocated, set, and freed by the caller of a function or interface method. An In parameter is not modified by the called function. See also In/Out parameter and Out parameter.

In/Out parameter
OLE: A parameter that is initially allocated by the caller of a function or interface method, and set, freed, and reallocated, if necessary, by the process that is called. See also In parameter and Out parameter.

In-place activation
OLE: Editing an embedded object within the window of its container, using tools provided by the server.

In-process server
A server implemented as a DLL that runs in the process space of the client. DLL startup and shutdown is managed by class InProcessServer or subclasses.

Inspect-it
Evaluate the current selection in the active text pane as a Smalltalk expression, and open an Inspector on the result.

Inspector
A graphical tool used to examine and change objects in the system.

Instance
An object of a particular class.

Instance method
A method invoked by sending a message to an instance of a class.

Instance variable
A variable defined to be part of each instance of a class. Instances of the class share the name of the variable, but not its value.

Instantiation
The act of creating an object of a data type, usually a class.

Interface
A group of semantically related functions that provide access to a COM object. Each OLE interface defines a contract that allows objects to interact according to the Component Object Model (COM). An interface is identified by a globally unique identifier (GUID).

J

JIT compiler
Just-In-Time compiler. A JIT compiler takes machine-independent bytecode and compiles it on demand into native code for the target machine, giving faster execution.

K

Keyword expression
A keyword expression sends a single keyword message with one or more arguments.

L

Licensing
A COM feature that provides control over object creation. Licensed objects can be created only by clients that are authorized to use them. Licensing may afford different levels of functionality depending on the type of license.

Literal
A value, used in a program statement, that is expressed directly rather than as a named constant or the contents of a variable. A literal generally defines an object of class Number, String, Character, Symbol, or Array.

Load time

The amount of time required to place a program's executable files into memory prior to execution, or the point in time when the files are loaded.

Local server
OLE: An out-of-process server implemented as an .EXE application running on the same machine as its client application. See also In-process server, Out-of-process server, and Remote server.

Local variable
A variable declared within a method or block that exists only within the scope of the method or block. See also Temporary variable.

Locale
The national and cultural environment in which a system or program is running. The locale determines the language used for messages and menus, the sorting order of strings, the keyboard layout, and date and time formatting conventions.

Localization
The process of adapting a program for a specific international market, which includes translating the user interface, resizing dialog boxes, customizing features (if necessary), and testing results to ensure that the program still works.

M

Machine code
The ultimate result of the compilation of a high-level language. It consists of sequences of bytes that are loaded and executed by a microprocessor.

Maximized window
An application window enlarged to fill the entire desktop, a document window enlarged to fill the entire application workspace, or a client window enlarged to fill the client area of the client window.

MDI
The standard user-interface architecture for Windows-based applications. A multiple document interface application enables the user to work with more than one document at the same time. Each document is displayed within the client area of the application's main window.

Message loop
A program loop that retrieves messages from a thread's message queue and dispatches them to the appropriate window procedures. See also message queue.

Message pump
A program loop that retrieves messages from a thread's message queue, translates them, offers them to the dialog manager, informs the MDI manager about them, and dispatches them to the application. See also message queue.

Message queue
A repository for window messages awaiting processing by a thread. The system message queue holds mouse and keyboard input waiting to be passed to a thread's message queue. A thread's message queue holds messages waiting to be retrieved by a thread's message loop.

Message table

A resource that stores alert and error messages that can contain replacement parameters.

Method

In object-oriented programming, a procedure that provides access to an object's data.

Mnemonics

In a graphical user interface, underlined letters in the text of menu and dialog-box items. When the menu and dialog-box items are active, the user can select the item by pressing the key that corresponds to the underlined letter.

Modal

A restrictive or limiting interaction created by a given condition of operation. Modal often describes a secondary window that restricts a user's interaction with other windows. A secondary window can be modal with respect to its primary window or to the entire system. A modal dialog box must be closed by the user before the application continues. See also modeless.

Modeless

Not restrictive or limiting interaction. "Modeless" often describes a secondary window that does not restrict a user's interaction with other windows. A modeless dialog box stays on the screen and is available for use at any time but also permits other user activities. See also modal.

Most recently used (MRU)

The most recently used files. FrameWindow can maintain a list of MRU files. The file list can be read from or written to the registry.

Mouse capture

The act of channeling mouse input to a specific window without regard to the position of the mouse-cursor hot spot.

Mouse event

An input event that occurs whenever the user moves the mouse, or presses or releases a mouse button. Windows converts mouse input events into messages and posts them to the appropriate thread's message queue.

MRU

See most recently used.

Multiple document interface

See MDI.

N

Named instance variable

An instance variable that is identified by a name, as defined in the class declaration. Instance variables either have a name or are referred to with an integer index.

Nonclient area

The parts of a window that an application does not use when displaying output such as text or graphics. A window's nonclient area consists of the border, menu bar, title bar, scroll bar, Control menu, Minimize button, and Maximize button. See also client area.

Nonresumable exception
An exception that must return from the message that created the exception handler, because it cannot continue execution.

Notification message
A message that a control sends to its parent window when events, such as input from the user, occur.

Number
An abstract class used to compare, count, and measure instances of its numeric subclasses.

O

Object
A programming structure encapsulating both data and functionality that are defined and allocated as a single unit and for which the only public access is through the programming structure's interfaces. OLE: A COM object must support, at a minimum, the IUnknown interface, which maintains the object's existence while it is being used and provides access to the object's other interfaces. See also COM and Interface.

OLE
Microsoft's object-based technology for sharing information and services across process and machine boundaries.

OLE Automation
See Automation.

OLE control
See Control.

Optimization
Compiler fine tuning to increase program performance or reduce program size.

Out-of-process server
A server, implemented as an `.EXE` application, which runs outside the process of its client, either on the same machine or a remote machine. See also Local server and Remote server.

Out parameter
A parameter that is allocated and freed by the caller, but its value is set by the function being called. See also In parameter and In/Out parameter.

Overlapped window
A style of window meant to serve as an application's main window. Other windows can overlap the window's space on the screen.

Owned window
A window that has an owner. An owned window always appears in front of its owner window, is hidden when its owner window is minimized, and is destroyed when its owner window is destroyed. See also owner window.

Owner window

A window that owns another window, thus affecting aspects of the owned window's appearance and behavior. See also owned window.

Owner-draw control
In Windows, a control that is customized for a specific application. Owner-draw controls are similar to predefined controls in that Windows will handle the control's functionality and process input from the mouse and keyboard. However, the programmer is responsible for the appearance of the owner-draw control in its various states.

P

Parent window
A window that has one or more child windows.

Persistent
Lasting between program sessions, or renewed when a new program session is begun.

Persistent storage
Storage of a file or object in a medium such as a file system or database so that the object and its data persist when the file is closed and then re-opened at a later time.

Pointer
A data type that can contain the address of another data type such as a variable, or of a function or class. Smalltalk: instances of Pointer often refer to structures. In such a case, the Pointer can read and write fields of its associated structure.

Polymorphism
The ability of objects to respond to the same message in different ways.

Pool dictionary
A dictionary that maps symbolic names to immediate values. Pool variables defined in a pool dictionary are only used at compile-time to replace occurrences with their associated value. Pool variables appear in the class declaration and are inherited by subclasses.

Pool variable
Pool variables are the keys defined in a pool dictionary. See Pool dictionary.

Pop-up menu
A menu that is hidden until the user performs an action (such as clicking the right mouse button) that causes Windows to display the menu. The pop-up menu contains commands that are relevant to the selection or the active window. See also drop-down menu.

Portable Executable (PE) Image
The standard Win32 executable format.

Process
An executing application that consists of a private virtual address space, code, data, and other operating-system resources, such as files, pipes, and synchronization objects that are visible to the process. A process also contains one or more threads that run in the context of the process.

Profiler

A development tool for analyzing the run-time behavior of programs.

Property

Information that is associated with an object.
Smalltalk: subclasses of FrameWindow maintain a property dictionary that can be used to store information.
OLE: properties fall into two categories: run-time properties and persistent properties. Run-time properties are typically associated with control objects or their containers. For example, background color is a run-time property set by a control's container. Persistent properties are associated with stored objects.

Proxy

An interface-specific object that packages parameters for that interface in preparation for a remote method call. A proxy runs in the address space of the sender and communicates with a corresponding stub in the receiver's address space.

Q

Queue

A data structure in which elements are added to the end of a list and removed from the head of the list.

R

RCDATA resource

A custom Windows resource element. Smalltalk dialogs may use RCDATA structures to store information such as framing parameters, OLE properties and so forth.

Reentrant

Code written so that it can be shared by several programs (or processes within a single program) at the same time. When code is reentrant, one program or process can safely interrupt the execution of another program or process, execute its own code, and then return control to the first program or process in such a way that the first program or process does not fail or behave in an unexpected way.

Reference counting

Keeping a count of each interface pointer held on an object to ensure that the object is not destroyed before all references to it are released.

Registry

A Windows system database in which configuration information is registered.

Registry key

A unique identifier assigned to each piece of information in the system registration database.

Remote Server

A server application, implemented as an EXE, running on a different machine from the client application using it. See also In-process server, Local server, and Out-of-process server.

Resource

An element, such as a string, icon, bitmap, cursor, dialog, accelerator, or menu, that is included in a Win32 resource (.RC) file.

Resource compiler
A program that creates a binary resource file based on the resource-definition (.RC) file.

Resumable exception
An exception that can return from the message that signaled it and continue execution.

Return value
The value that a method evaluates to in its calling expression.

Rubber-band selection
A selection method that allows a user to select multiple items by dragging a sizing rectangle around the items to be selected. Items within the region can be manipulated by the user.

Running Object Table (ROT)
OLE: A globally accessible table on each computer that keeps track of all COM objects in the running state that can be identified by a moniker. Moniker providers register an object in the table, which increments the object's reference count. Before the object can be destroyed, its moniker must be released from the table.

S

Scope
The extent to which a given identifier (variable, pool variable) can be referenced within a program.

Screen coordinates
A means of specifying the position of a point on the display screen in terms of vertical (y-coordinate) and horizontal (x-coordinate) displacement from the upper-left corner of the screen (origin).

SEH (Structured Exception Handling)
A mechanism for handling hardware- and software-generated exceptions that gives developers complete control over the handling of exceptions, provides support for debuggers, and is usable across all programming languages and computers.

Self-registration
The process by which a server can perform its own registry operations.

Server application
The term server application often refers to an executable that runs unattended and responds to client requests over the network.
OLE: An application that can create COM objects. Container applications can then embed or link to these objects.

Semantics
The relationships between words or symbols and their intended meanings, or the rules governing these relationships. See also syntax.

Semaphore

A synchronization object that maintains a count between zero and a specified maximum value. A semaphore's state is signaled when its count is greater than zero and nonsignaled when its count is zero. The semaphore object is useful in controlling a shared resource that can support a limited number of users. It acts like a gate that counts the threads as they enter and exit a controlled area and that limits the number of threads sharing the resource to a specified maximum number.

Serialization

The process of writing or reading an object to or from a persistent storage medium, such as a disk file. Later, the object can be re-created by reading, or deserializing, the object's state from storage.

Show-it

An operation in which the current selection in the active text pane is evaluated as a Smalltalk expression. The result is printed and the resulting string inserted into the text pane.

Shortcut key

A keyboard combination that activates a program command directly, as an alternative to activating the command through the program menus.

Sibling

A node in a tree that is descended from the same immediate ancestor(s) as other nodes.

Single threading model

A model in which all objects are executed on a single thread.

Smalltalk heap

The memory that has been reserved and possibly committed for regular Smalltalk objects. By default, all new objects are created in the Smalltalk heap. See also Heap.

Source character set

The set of legal characters that can appear in source files. For Smalltalk, the source set is the standard ASCII character set.

Source code

Human-readable statements written in a high-level programming language.

SQL

A database sublanguage used to query, update, and manage relational databases.

Stack frame

An area of the stack memory that temporarily holds the arguments to a method or function as well as any local variables.

Stack overflow

An error condition caused by attempting to push an item onto a stack that is full, meaning that all of the memory allocated for that stack has been used.

String

A string is an instance of class String containing a sequence of instances of class Character.

Subclass

A class that is derived from another class. A subclass inherits state and behavior from its superclass in the form of variables and methods.

Subclassing

Smalltalk: deriving a class through inheritance from another class.
Windows: intercepting messages from and to another window in order to change the behavior of the window.

Symbol

Smalltalk: An instance of class Symbol. A Symbol is characterized by its identifier (ID), which is uniquely associated with a String that represents the symbol's name.
More generally, a variable, method name, or other identifier of an executable program. See symbolic-debugging information.

Symbolic-debugging information

A map of the source code and all the identifiers (variables, method or function names, and so on) created at compile time for use by a debugger. See also COFF.

Synchronous call

A function call that does not allow further instructions in the calling process to be executed until the function returns. See also Asychronous call.

Syntax

The grammar of a particular language, the rules governing the structure and content of the statements. See also semantics.

System registry

A system-wide repository of information supported by Windows, which contains information about the system and its applications, including OLE clients and servers.

T

Tab order
The order in which the TAB key moves the input focus from one control to the next within a dialog box. Usually, the tab order proceeds from left to right in a dialog box, and from top to bottom in a radio group.

Tab stop
One of the points in a line of text or a control in a group of controls (in a dialog box, for example) that the user can move to by pressing the TAB key. See also tab order.

Temporary variable
Or local variable, block-local variable. A variable defined as part of a method or block, that exists while the method or block executes.

Termination handler
A block that is executed, regardless of how a guarded block finishes executing. See also SEH.

Thread
The basic entity to which the operating system allocates CPU time. A thread can execute any part of the application's code, including a part currently being executed by another thread. All threads of a process share the virtual address space, global variables, and operating-system resources of the process.

Thread local storage (TLS)
A Win32 mechanism that allows multiple threads of a process to store data that is unique for each thread. Smalltalk MT supports thread local variables, whose values are specific to each thread.

Thunk
A small section of code that performs a translation or conversion during a call or indirection. For example, a thunk is used to change the size or type of function parameters when calling between 16- and 32-bit code.

Timestamp
A value that specifies the time data was created, modified, accessed, or received. Method timestamps record when a method was created or modified.

TLS
See Thread local storage.

Tool tip
A tiny pop-up window that presents a short description of a toolbar button's action. Tool tips are displayed when the user positions the mouse over a button for a period of time.

Top-level window
A window that has no parent window, or whose parent is the desktop window.

Trace message
Or trace output. An error or diagnostic message employed in debugging to provide information about where in the program execution a problem occurred. In some cases, trace output can provide advance warning about problems that are about to occur. Trace output can be viewed on an external debugger or an application such as DBMON (on Windows NT).

Transcript
The initial workspace that is displayed in the development environment. See also Workspace.

Try block

A guarded body of code in a try-except frame-based exception handler or try-finally termination handler.

Type information

Information about an object's class provided by a type library. To provide type information, a COM object implements the IProvideClassInfo interface.

U

Unary expression
A unary expression sends a series of unary messages.

Unary message
A unary message is a message that has no arguments.

Unicode
A 16-bit character set capable of encoding all known characters and used as a worldwide character-encoding standard. Windows NT uses Unicode exclusively at the system level.

Uniform Data Transfer
In OLE, a set of interfaces that allow data to be sent and received in a standard fashion, regardless of the actual method chosen to transfer the data.

User-defined message
Any message that is not a standard Windows message. Typically, a user-defined message starts at WM_USER.

User-interface thread
In Windows, a thread that handles user input and responds to user events independently of threads executing other portions of the application. User-interface threads have a message pump and process messages received from the system.

V

Variable
A named storage location that references a single object. The reference can be changed at runtime using variable assignments. See also Assignment.

Version resource
A resource that contains either text or binary data about an application's version information.

Virtual Table (VTBL)
An array of pointers to interface method implementations. See also Interface.

W

Walkback
A Walkback dialog pops up automatically when an unhandled exception occurs. When you need more information than provided in the Walkback, you explicitly request a Debugger window by clicking the Debug button.

Wide character
Or Unicode character. A 2-byte multilingual character code. Smalltalk Character instances contain Unicode characters.

Win32 API
The set of 32-bit functions supported by Windows. See Win32 platform.

Win32 platform

A platform that supports the Win32 API. These platforms include Intel Win32s, Windows NT, Windows 95/98, MIPS Windows NT, DEC Alpha Windows NT, and Power PC Windows NT.

Window class

In the Win32 API, a set of attributes that Microsoft Windows uses as a template to create a window in an application. Windows requires that an application supply a class name, the window-procedure address, and an instance handle. Other elements may be used to define default attributes for windows of the class, such as the shape of the cursor and the content of the menu for the window.

Window handle

In the Win32 API, a 32-bit value (assigned by Windows) that uniquely identifies a window. An application uses this handle to direct the actions of functions to the window.

Window procedure

A function, called by the operating system, that controls the appearance and behavior of its associated windows. The procedure receives and processes all messages to these windows.

Wizard

A special form of user assistance that guides the user through a difficult or complex task within an application.
Smalltalk: a dialog class that represents a Windows wizard.

Workspace

A window with a single text control in which text can be edited and from which Smalltalk expressions can be evaluated. The Transcript is the initial workspace window that is displayed in the development environment.

Workspace variable

A variable associated with a workspace and only accessible within that workspace. A workspace variable exists as long as the workspace remains open.

Z

Z order

Indicates a window's position in a stack of overlapping windows. This window stack is oriented along an imaginary axis, the z-axis, extending outward from the screen.

Index

_

_asCinteger, 202, 374, 376, 396
_asm, 427
_doesNotUnderstand, 426
_isKindOf, 435
_overflow, 435
_performIntSelector, 433
_performSelector, 433
_performUSelector, 434
_stalloc, 434

A

About Box, 102
Accelerator Table, 103
Accelerators, 222, 233
ActiveX
  implementing components, 333
  using, 316
Alias (name space), 163
ANSI, 397
  monitoring string conversions, 399
Apartment Model, 312
API
  aliasing, 389
  ANSI and Unicode functions, 388
  binding with, 372
  decorated functions, 388
  examples, 381
  passing arguments, 374
  passing parameters, 374
  referencing arguments, 378
  return types, 379
Application, 344
  Application Icon, 102
  Application Project, 345
  Window, 88
ApplicationProcess, 198, 229, 230
Array, 183, 189
  literal, 167
asCinteger, 374
asm, 427
Assembler, 425
  addressing modes, 427
  instructions, 427
  object types, 431
  routines, 433
  Smalltalk calling convention, 430
Asynchroneous processing, 210

B

Behavior, 182
Binary Message, 158, 159
Block, 199
  context, 170
  inlining, 423
  optimizing contexts, 422
  reentrancy issues, 171
  static, 170
  static blocks, 422
  types of, 170
  variables, 165
Boolean Expressions, 176
Browsing, 13
  messages, 14, 37, 61
  methods, 37
  unimplemented messages, 14
Build Process, 360
Button, 124, 277
Byte Object, 154
ByteArray, 183, 189

C

Callbacks, 389
Cascaded Message, 158
Categories, 178, 414
Change Browser, 15
Changes (source code), 72
Changes File, 58
Changes.sl, 72
Character, 166
  literal, 171
Child Identifiers, 93
Child windows
  accessing, 242, 247
Chunk Format, 8
Class, 183
  changing the superclass, 21
  deleting, 21
  finding, 20
  initializing, 52
  renaming, 21
  types of, 155
Class Hierarchy Browser, 17
Class Instance Variable, 164
Class Section, 67
Class Variable, 164
Clipboard, 322
  using, 321
closeWindow, 245
Code Section, 66
Code Size (reserved by the compiler), 67
Collections, 183
COM, 308
ComboBox, 124
Common Controls, 281
Common Dialog Boxes, 258
Compiler
  accessing classes, 415
  accessing pool dictionaries, 416
  accessing variables, 414
  browsing, 414
  browsing example, 417
  compiling, 415
  method categories, 414
  programming interface, 413
  settings, 67
  working with files, 417
Conditional Compilation, 181
Console, 363
  control handlers, 364
  creating, 363
  processing command line arguments, 364
Constant Expressions, 171
ContainerWindow, 303
Context Menu, 219
ContextBlock, 170
Control of Flow, 173
Controls, 121, 276
CriticalSection, 199
currentThread, 231
Custom Controls
  CustomControl class, 303
  using external controls, 124

D

Data Section, 67
Debug Output, 232
Debugger, 24
  call frames, 26
  customizing windows, 26
  using out-of-process debuggers, 79
Debugging, 412
Desktop
  customizing fonts, 12
  multiple desktops, 10
  saving and restoring, 10
destroyWindow, 245
DeviceContext, 235
Diagnostic Messages, 80
DialogBox, 89, 120, 249
  creating an application, 357
  owner, 249
Dictionary, 184
  alias name, 163
DLL, 368
  binding with, 59
  loading resources, 345
  unloading a Smalltalk DLL, 231
  updating ANSI and Unicode bindings, 398
  using, 372
dllEntryPoint, 358
Drag and Drop, 325
DragListBox, 283
Dynamic Code Section, 67

E

Edit, 124
Edit Controls, 279
Editions of Smalltalk MT, XVIII
Entry Point, 230, 354
Escape Codes, 168
Evaluating Code, 4

context of evaluation, 8
errors, 5
Events, 214, 276
overview, 211
Exceptions, 200, 391
common exceptions, 392
development exceptions, 395
runtime exceptions, 392
exitApplication, 234
exitInstance, 234
Exports, 389

F

File
alias name, 163
File Handling Framework, 220
Filing in Code, 8
Finalization, 408
Float, 193
FrameWindow, 88, 114, 245, 246
Framing Parameters, 98
Free-threading Model, 311, 312

G

Garbage Collector, 419
Generic Sample, 132, 349
getCommandArguments, 231, 232
getMessages, 231, 232
getStartupDirectory, 232
getStartupPath, 232
Global Variable, 163
Graphics, 235
GUID, 314

H

HeaderControl, 285
Headless Applications, 360
Heap Sizes, 71
Hello World Application, 347
Help, 226
commands, 227
topics, 227, 228
hInstance, 232
hiword, 192
HotKey, 285

I

IClassFactory, 333
IDataObject, 321
IDataObjectEx, 322
IdentityDictionary, 183, 187
IDispatch, 319
Idle Processing, 210
ifEvents, 217
ifFalse, 173
ifTrue, 173
ihiword, 192
iloword, 192
Image
backing up, 71
binding with libraries, 59
image differences, 72
restoring, 73
saving, 58, 59
sections, 65
version, 3
Image Lists
using, 286
Image Maintenance, 58
Image Properties, 27
diagnostic messages, 80
optimization page, 28
Implementors of a message, 37
Import Section, 67
Imports
reserved entries, 61
using, 62, 63
Indexed Class, 154
initApplication, 234
Initialization of Classes, 52
initInstance:, 234
initPosition, 245
initWindow, 245
Inline Assembler, 425
InProcessServer, 334
Inspectors, 33
inspecting objects, 33
subclassing, 36
Installation, 2
Instance Variable, 165
Integer, 191
signed / unsigned, 191
Interface Builder
code generation, 112
Controls, 94, 121
custom events, 131
duplicating and deleting, 95
extensions, 129
general properties, 122
inserting, 95

placement, 95
properties, 98
selecting, 94
Smalltalk property page, 123
creating a dialog, 90
creating a window, 88
dialog boxes, 120
editing a window, 89
editing menus, 92
frame windows, 114
framing parameters, 98
generating a resource script, 105
identifiers, 93
menus, 89
mnemonics, 100
pool dictionaries, 93
resource scripts, 101
saving a window, 89
splitter bars, 126
tab ordering, 99
Interval, 190
IOStream, 196
IOWideStream, 196

K

Keyword Message, 158, 159, 160

L

LargeInteger, 191
List Controls, 277
ListBox, 124
ListView, 125, 286
Literals, 166
Loader Program, 58, 59
Local Variables, 165
Loop Expressions, 175
loword, 192

M

Main Application Window, 233
MappingTable, 184, 185
MDIFrameWindow, 263
Memory
reducing allocations, 421
reserving, 65
sections, 65
Memory-mapped Files, 402
Menu, 91, 274
creating, 274
hints, 226
modifying, 275
popup menu, 274
Message File, 102
Message Loop, 214
Message Pump, 206
Messages, 157
processing Windows messages, 209
Metaclass, 182
Method Explorer, 37
Method Statistics, 22
Method Timestamp, 21
Methods (overview), 156
Most Recently Used File Framework, 221
Multiprocessing Classes, 198

N

Name Spaces, 163
Notation, XVI
Notifications, 276
Number, 191
literals, 173

O

Object
structure of, 154
OLE
controls, 316
data transfer, 321
events, 316
methods, 317
OleControl, 126, 335
properties, 317
property pages, 337
troubleshooting, 340
OleControl, 335
OleControlContainer, 304, 316
Online Help, 226
Operating System Version, 71
Optimization
optimization levels, 27, 31, 32
tail recursion, 31
OrderedCollection, 184, 190
OrderedMappingTable, 187
outputDebugLine:, 232
outputDebugPrefix, 232
OutputDebugString API, 232
outputDebugString:, 232
Overlapped Windows, 245

P

Performance, 419
Picture, 125
Pointer, 202
Pointer Object, 154
Pool Dictionaries, 93, 180, 416
  converting between ANSI and Unicode, 398
  editing, 53
  loading from disk, 54
Pool Dictionary, 180
Popup Menu, 274
Process attributes, 231
Process Properties, 69
  heap, 71
  resources, 71
  Smalltalk heap reserve, 70
  stack, 70
  version, 71
Processor, 198, 229
Progress Bar Sample, 144
ProgressBar, 289
Project
  defining, 42
  defining a Windows application, 44
  fails to find sub-project, 76
  how to design, 51
  syntax error while filing in, 76
Project Browser, 38
  defining a project, 42
  defining source path of projects, 51
  opening a project, 40
Prompter, 269
  check list prompter, 272
  list prompter, 271
  multi-list prompter, 271
  templates, 102
  text prompter, 270
Property Sheet, 265
Proxy
  implementing, 437

R

Real-time Applications, 419
ReBar, 290
Reentrancy of Blocks, 171
registerClass, 240
registerObject, 378
Registry, 224, 234
releaseObject, 378
Resources, 101
  about box, 102
  accelerator table, 103
  application icon, 102
  default module, 233
  integrating into image, 71
  loading, 345
  message file, 102
  prompter templates, 102
  sample, 348
  script generation, 105
  string table, 103
  tool bar, 102
  using, 52
  version resource, 103
Resources, 358
RichEdit, 291
run, 357
Runtime.sm, 360

S

Sample
  Generic, 349
  Generic Application, 132
  installing a sample application, 51
  Progress Bar, 144
  Slider Sample, 144
  Spin Button, 144
  Splitter Bars, 147
  ToolBar, 141
  using DLL resources, 348
Scroll Boxes, 279
SDIFrame, 264
Sections, 65
  class section, 67
  code section, 66
  data section, 67
  import section, 67
Selectors, 157
Semaphore, 199
Senders of a message, 37
SequenceableCollection, 184, 188
Serialization, 401
  architecture, 401
  overview, 402
  using memory-mapped files, 402
Set, 184, 190
Setup, 2
sleep, 231
Slider Sample, 144
Smalltalk Controls, 125, 303
Smalltalk Heap Reserve, 70

SortedCollection, 184, 190
Source Code
  filing in, 8
  filing out, 9
Source Database, 58
Source File, 8
  allocating, 68
  corruption, 83
Source Management, 178
Sources.bin, 58, 68
Spin Button Sample, 144
SplitPane, 129, 305
Splitter Bars Sample, 147
Stack, 190
Stack Sizes, 70
StaticBlock, 170
StatusBar, 292
StatusWindow, 292
stdin, 363
stdout, 363
STMT.EXE, 58, 59
Stream, 195
String, 190, 197
  escape codes, 168
  literal, 168
  StringA, 198
  StringW, 197
String Table (resource), 103
StringDictionary, 188
StringTable, 188
Struct, 200
  accessing, 201
  defining for ANSI and Unicode, 399
  initializing, 201
  using a pointer on, 202
Structures, 200
Styles, 281
Subclassing
  Window subclassing, 244
Subsystem Version, 71
Symbol, 198
  literal, 169
  using at runtime, 359
Symbol Editor, 53
Synchronization, 199
System Classes, 198
System Requirements, XIX
SystemDictionary, 188
SystemException, 200

T

TabControl, 293
Tail Recursion, 31
terminateThread, 231
Thread, 199, 231
  killing, 57
  stopping, 57
  suspending and resuming, 57
Thread Local Variable, 164
Thread Viewer, 56
timesRepeat, 175
ToolBar, 102, 294
  sample, 141
ToolTip, 226, 298
TrackBar, 300
Transcript
  evaluating code, 4
  window, 55
TreeView, 300
Troubleshooting, 75

U

Unary Message, 158, 159
unhandledException:, 234
Unicode, 397
UpDown, 302
User Interface
  customizing, 10, 26
  defining defaults, 11
  window placement preferences, 11

V

Validation, 251
Variables, 162, 414
  block local, 165
  class instance, 164
  instance, 165
  locals, 165
  restrictions, 165
Version Resource, 103

W

Weak Pointers, 408
WinApplication, 229, 232
WinDbg (using WinDbg or MSVC), 79
WinDocument, 264
Window, 209, 238
  creating, 241

enumerating, 241
handle, 241
items, 242
procedure, 206
properties, 243
registration, 239
subclassing, 244
Window Classes, 237
Windows Events, 214
Windows Messages, 214
winMain, 354
Wizards, 265
WM_HELP, 227
WndProc, 206
WordArray, 184, 189
literal, 167
Workspace, 56
Workspace Variables, 4

X

XFactory, 337