



# MiniGUI Programming Guide

Version 2.0 (revised edition 4)  
For MiniGUI Version 2.0.4/1.6.10

Beijing Feynman Software Technology Co. Ltd.

August, 2007



# Introduction

MiniGUI, developed by Beijing Feynman Software Technology Co. Ltd., originates from a world famous free software project, which is initiated by Wei Yongming. MiniGUI aims to provide a fast, stable and lightweight graphics user interface (GUI) support system for real-time embedded systems. MiniGUI is “a cross-operating-system graphics user interface support system for embedded devices”, and “an embedded graphics middleware”; it can run on Linux/uClinux, eCos, VxWorks, pSOS, ThreadX, Nucleus, OSE, and even uC/OS-II, also on Win32 platform.

Currently, the latest freely downloadable stable releases (GPL releases) of MiniGUI are MiniGUI-STR V1.6.2 and MiniGUI V1.3.3. The source code, development documents and sample programs are available from the download area (<http://www.minigui.com/download/index.shtml>) of the website of Beijing Feynman software Technology Co. Ltd. It should be noted that the freely downloadable MiniGUI could be only used for developing GPL or other application software with open source code. If you are using MiniGUI for developing commercial, proprietary, or other software not covered by the terms listed in GPL, you must have a commercial license for MiniGUI. Beijing Feynman software Technology Co. Ltd. will provide the latest value-added release and related porting and development technology support service for the users who have buy the MiniGUI commercial licenses.

This guide describes in detail the foundation knowledge of MiniGUI V2.0.4/1.6.10 on developing embedded application software, technical documents and development skills, the content of which involves various aspects of MiniGUI programming, include message looping, window procedure, dialog box, controls, graphics interfaces, and so on. Please refer to *MiniGUI API Reference* for complete description of MiniGUI APIs (Application Programming Interfaces).

## Copyright Claim

*MiniGUI Programming Guide* Version 2.0 (revised edition 4) for MiniGUI Ver 2.0.4/1.6.10.

Copyright © 2003~2007, Beijing Feynman Software Technology Co., Ltd.  
All rights reserved.

**By whatever means you get the entire or partial text or photograph data in this guide, no matter mechanical or electronic, you are only authorized by Beijing Feynman Software Technology Co., Ltd. the reading right. Any format conversion, redistribution, dissemination, and copying its entire or partial content, or using text or photograph therein for commercial purpose without written permission will be regarded as tortuous, and may result in severe civil or criminal punishment.**

# Contents

Introduction .....	I
Copyright Claim .....	II
1 Preface .....	1
1.1 Relevant Documents .....	1
1.2 Organization of this Guide .....	2
1.3 Obtaining Samples of the Guide .....	3
1.4 Compiling and Running Environment of Samples .....	4
1.5 Copyright and Licensing .....	5
I Foundation of MiniGUI Programming .....	7
2 Beginning MiniGUI Programming .....	9
2.1 Basic Programming Concepts .....	9
2.1.1 Event Driven Programming .....	9
2.1.2 Three Runtime Modes of MiniGUI .....	10
2.2 A Simple MiniGUI Program .....	12
2.2.1 Header Files .....	14
2.2.2 Entrance of the Program .....	15
2.2.3 Join a Layer in MiniGUI-Processes .....	15
2.2.4 Creating and Showing Main Window .....	16
2.2.5 Entering Message Loop .....	18
2.2.6 Window Procedure Function .....	20
2.2.7 Screen Output .....	21
2.2.8 Exit of the Program .....	21
2.3 Compiling, Linking and Running .....	22
2.3.1 Compiling MiniGUI Program .....	22
2.3.2 Function Libraries of MiniGUI .....	23
2.4 Writing Automake/Autoconf Scripts for MiniGUI Application .....	23
3 Window and Message .....	31
3.1 Window System and Window .....	31
3.1.1 What Is Window System .....	31
3.1.2 Concept of Window .....	32
3.2 Window of MiniGUI .....	33
3.2.1 Window Type .....	33

3.2.2 Main Window.....	35
3.2.3 Window Style.....	36
3.2.4 Destroying Main Window.....	37
3.2.5 Dialog Box.....	38
3.2.6 Control and Control Class.....	38
3.2.7 Input Method Window.....	41
<b>3.3 Message and Message Handling .....</b>	<b>43</b>
3.3.1 Message.....	43
3.3.2 Message Type.....	45
3.3.3 Message Queue.....	45
3.3.4 Message Handling.....	47
3.3.5 Sending and Posting Message.....	49
3.3.6 Message Handling Function Specific to MiniGUI-Processes .....	51
<b>3.4 Several Important Messages and Corresponding Handling .....</b>	<b>52</b>
3.4.1 MSG_NCCREATE.....	53
3.4.2 MSG_SIZECHANGING.....	53
3.4.3 MSG_SIZECHANGED and MSG_CSIZECHANGED .....	54
3.4.4 MSG_CREATE.....	54
3.4.5 MSG_FONTCHANGING.....	55
3.4.6 MSG_FONTCHANGED.....	55
3.4.7 MSG_ERASEBKGD.....	56
3.4.8 MSG_PAINT.....	57
3.4.9 MSG_CLOSE.....	58
3.4.10 MSG_DESTROY.....	58
<b>3.5 Common Window Operation Functions .....</b>	<b>59</b>
<b>4 Foundation of Dialog Box Programming .....</b>	<b>63</b>
4.1 Main Window and Dialog Box .....	63
4.2 Dialog Box Template.....	63
4.3 Dialog Box Callback Function .....	65
4.4 MSG_INITDIALOG Message.....	66
4.5 Modal and Modeless Dialog Box .....	68
4.6 Control Styles and Operations Relevant to Dialog Box.....	69
<b>5 Foundation of Control Programming.....</b>	<b>73</b>
5.1 Control and Control Class.....	73
5.2 Creating Control Instance by Using Predefined Control Class .....	75

5.3 Topics Involved in Control Programming.....	77
5.4 Operations Specific to Control .....	81
<b>6 Advanced Programming of Control.....</b>	<b>83</b>
6.1 User-Defined Control .....	83
6.2 Subclassing of Control .....	83
6.3 Combined Use of Controls .....	84
<b>7 Menu.....</b>	<b>89</b>
7.1 Concept of Menu.....	89
7.2 Creating and Handling Menu .....	89
7.2.1 Creating Normal Menu.....	89
7.2.2 Creating Popup Menu.....	90
7.2.3 MENUITEMINFO Structure.....	91
7.2.4 Operating on Menu Item.....	93
7.2.5 Deleting and Destroying Menu or Menu Item .....	94
7.2.6 MSG_ACTIVEMENU Message.....	94
7.3 Sample Program .....	95
<b>8 Scrollbar .....</b>	<b>99</b>
8.1 Concept of Scrollbar .....	99
8.2 Enabling or Disabling a Scrollbar .....	100
8.3 Range and Position of a Scrollbar.....	100
8.4 Messages of Scrollbar .....	102
8.5 Sample Program .....	104
<b>9 Keyboard and Mouse .....</b>	<b>109</b>
9.1 Keyboard.....	109
9.1.1 Keyboard Input.....	109
9.1.2 Key Stroke Message.....	110
9.1.3 Character Message.....	111
9.1.4 Key Status.....	112
9.1.5 Input Focus.....	113
9.1.6 Sample Program.....	114
9.2 Mouse.....	114
9.2.1 Mouse Input.....	114
9.2.2 Mouse Message.....	115
9.2.3 Capture of Mouse.....	118
9.2.4 Tracking Mouse Cursor.....	120

9.3 Event Hook.....	123
10 Icon, Cursor, and Caret.....	127
10.1 Icon.....	127
10.1.1 Loading and Displaying Icon.....	127
10.1.2 Destroying Icon.....	130
10.1.3 Creating Icon.....	130
10.1.4 Using System Icons.....	133
10.2 Cursor.....	135
10.2.1 Loading and Creating Cursor.....	136
10.2.2 Destroying Cursor.....	137
10.2.3 Positioning and Displaying Cursor .....	137
10.2.4 Clipping Cursor.....	139
10.2.5 Using System Cursors.....	140
10.2.6 Sample Program.....	142
10.3 Caret .....	145
10.3.1 Creating and Destroying Caret.....	145
10.3.2 Showing and Hiding Caret.....	146
10.3.3 Positioning Caret.....	146
10.3.4 Changing Blink Time of Caret.....	147
10.3.5 Sample Program.....	147
11 Using MiniGUIExt Library .....	153
11.1 User Interface Wrap Functions.....	153
11.2 Skin.....	156
11.2.1 Form of skin.....	157
11.2.2 Skin Window.....	161
11.2.3 Use of callbacks.....	162
11.2.4 Skin Operations.....	164
11.2.5 Normal Label.....	165
11.2.6 Image Label.....	166
11.2.7 Command Button.....	167
11.2.8 Check Button.....	167
11.2.9 Normal Slider.....	168
11.2.10 Rotate Slider.....	169
11.2.11 MiniGUI Control.....	170
11.2.12 Sample Program.....	170



11.3 Color Selection Dialog Box .....	173
11.4 New Open File Dialog Box .....	175
<b>12 Other Programming Topics .....</b>	<b>177</b>
12.1 Timer .....	177
12.2 Dynamic Change of Color of Window Element .....	180
12.3 Clipboard .....	181
12.3.1 Creating and Destroying Clipboard .....	182
12.3.2 Transferring Data to Clipboard .....	183
12.3.3 Getting Data from Clipboard .....	183
12.4 Reading/Writing Configuration File .....	184
12.5 Writing Portable Program .....	188
12.5.1 Using Endian-Specific Read/Write Functions of MiniGUI .....	189
12.5.2 Using Condition Compilation to Write Portable Code .....	191
12.6 Fixed-Point Computing .....	193
<b>II MiniGUI Graphics Programming .....</b>	<b>195</b>
<b>13 Graphics Device Interfaces .....</b>	<b>197</b>
13.1 Architecture of MiniGUI Graphics System .....	197
13.1.1 GAL and GDI .....	197
13.1.2 New GAL .....	198
13.2 Painting and Updating of a Window .....	200
13.2.1 When to Paint a Window? .....	200
13.2.2 MSG_PAINT Message .....	200
13.2.3 Valid and Invalid Region .....	201
13.3 Graphics Device Context .....	203
13.3.1 Abstraction of Graphics Device .....	203
13.3.2 Getting and Releasing of Device Context .....	205
13.3.3 Saving and Restoring of Device Context .....	209
13.3.4 Device Context in Memory .....	209
13.3.5 Screen Device Context .....	209
13.4 Mapping Mode and Coordinate Space .....	210
13.4.1 Mapping Mode .....	210
13.4.2 Viewport and Window .....	211
13.4.3 Conversion of Device Coordinates .....	213
13.4.4 The Deviation and Zoom of Coordinate System .....	214
13.5 Rectangle and Region Operations .....	215

13.5.1 Rectangle Operations.....	215
13.5.2 Region Operations.....	216
<b>13.6 Basic Graphics Drawing .....</b>	<b>218</b>
13.6.1 Basic Drawing Attributes.....	218
13.6.2 Basic Drawing Functions.....	219
13.6.3 Clipping Region Operations.....	219
<b>13.7 Text and Font .....</b>	<b>220</b>
<b>13.8 Bitmap Operations.....</b>	<b>221</b>
13.8.1 Concept of Bitmap.....	221
13.8.2 Bitmap Color.....	222
13.8.3 Device-Dependent Bitmap and Device-Independent Bitmap .....	224
13.8.4 Loading a Bitmap from File.....	226
13.8.5 Filling Block.....	228
13.8.6 Bit Blitting.....	230
<b>13.9 Palette .....</b>	<b>233</b>
13.9.1 Why Is the Palette Needed?.....	233
13.9.2 Using Palette.....	235
<b>14 Handling and Rendering of Text.....</b>	<b>237</b>
14.1 Charset and Encoding .....	237
14.2 Device Font.....	239
14.3 Logical font.....	240
14.4 Text Analysis .....	242
14.5 Text Transformation .....	243
14.6 Text Output.....	244
14.7 Special Render Effects of Font Glyph.....	248
<b>15 Advanced GDI Functions Based on NEWGAL .....</b>	<b>251</b>
15.1 New Region Implementation.....	251
15.2 Raster Operations .....	252
15.3 Memory DC and BitBlt .....	253
15.4 Enhanced BITMAP Operations .....	256
15.5 New GDI functions .....	258
<b>15.6 Advanced GDI functions .....</b>	<b>258</b>
15.6.1 Image Scaling Functions.....	258
15.6.2 Image Rotation Functions.....	259
15.6.3 Rounded Corners Rectangle.....	261

15.7 Curve Generators .....	261
15.7.1 Line Clipper and Line Generator .....	262
15.7.2 Circle Generator.....	263
15.7.3 Ellipse Generator.....	263
15.7.4 Arc Generator.....	263
15.7.5 Vertical Monotonous Polygon Generator .....	264
15.7.6 General Polygon Generator.....	265
15.7.7 Flood Filling Generator.....	265
15.7.8 Using Curve Generator.....	266
15.8 Plotting Complex Curve .....	268
15.9 Filling Enclosed Curve .....	269
15.10 Building Complex Region.....	270
15.11 Visiting Frame Buffer Directly .....	272
15.12 YUV Overlay and Gamma Correction .....	273
15.12.1 YUV Overlay.....	273
15.12.2 Gamma Correction.....	276
15.13 Advanced Two-Dimension GDI Functions .....	277
15.13.1 Pen and Its Properties.....	277
15.13.2 Brush and Its Properties.....	280
15.13.3 Advanced Two-Dimension Drawing Functions .....	281
15.13.4 Using Advanced Two-Dimension GDI Functions .....	282
15.14 Support for Slave Screens.....	285
15.14.1 Creating Slave Screen.....	285
15.14.2 Destroying Slave Screen.....	285
<b>III MiniGUI Advanced Programming Topics.....</b>	<b>287</b>
16 Inter-Process Communication and Asynchronous Event Process .....	289
16.1 Asynchronous Event Process .....	289
16.2 MiniGUI-Processes and Inter-Process Communication .....	291
16.2.1 Multi-Process Model under MiniGUI-Processes .....	292
16.2.2 Simple Request/Response Processing .....	293
16.2.4 Wraps for UNIX Domain Socket.....	296
17 Developing Customized MiniGUI-Processes Server Program.....	299
17.1 Mginitt of MDE.....	299
17.1.1 Initializing Itself as the Server of MiniGUI-Processes .....	299
17.1.2 Displaying Copyright Information .....	305

17.1.3	Creating Taskbar.....	305
17.1.4	Startup the Default Program.....	306
17.1.5	Entering Message Loop.....	307
17.2	A Simple Mginit Program .....	307
17.3	Functions Specific to Clients of MiniGUI-Processes .....	311
17.4	Other Functions and Interfaces Specific to Mginit.....	312
18	GAL and IAL Engines .....	314
18.1	Shadow NEWGAL Engine.....	315
18.2	CommLCD NEWGAL Engine .....	315
18.3	The IAL Interface of MiniGUI .....	317
18.4	Comm Input Engine .....	321
18.5	Developing IAL Engine for a Specific Embedded Device .....	322
IV	MiniGUI Control Programming .....	328
19	Static Control.....	330
19.1	Types and Styles of Static Control.....	330
19.1.1	Standard Type.....	330
19.1.2	Bitmap Type.....	332
19.1.3	Group Box.....	333
19.1.4	Other static control types.....	334
19.2	Static Control Messages .....	334
19.3	Notification Codes of Static Control .....	335
19.4	Sample Program .....	335
20	Button Control .....	338
20.1	Types and Styles of Button .....	338
20.1.1	Push button.....	338
20.1.2	Check Box.....	340
20.1.3	Radio Button.....	341
20.2	Messages of Button .....	343
20.3	Notification Codes of Button.....	344
20.4	Sample Program .....	345
21	List Box Control.....	350
21.1	Types and Styles of List Box.....	350
21.2	Messages of List Box.....	352
21.2.1	Adding Item into List Box.....	352
21.2.2	Deleting Item from List Box.....	354

21.2.3	Selecting and Getting Item.....	354
21.2.4	Searching Item Including a Text String .....	356
21.2.5	Setting/Getting the Status of Check Mark .....	356
21.2.6	Setting the Bold Status of Item .....	357
21.2.7	Setting/Getting the Disable Status of Item .....	357
21.2.8	Adding Multiple Items into List Box .....	357
21.2.9	Other Messages.....	358
21.3	Notification Codes of List Box .....	359
21.4	Sample Program .....	360
22	Edit Box Control .....	365
22.1	Styles of Edit Box.....	366
22.2	Messages of Edit Box .....	367
22.2.1	Getting/Setting Caret Position .....	368
22.2.2	Setting/Getting Selection of Text .....	368
22.2.3	Copy, Cut, and Past.....	369
22.2.4	Setting/Getting Properties of Line Height and Others .....	370
22.2.5	Setting Limit of Text.....	371
22.2.6	Setting or Canceling Read-only Status .....	371
22.2.7	Setting/Getting Password Character .....	371
22.2.8	Setting Title and Tip Text.....	372
22.2.9	Setting End of Line Symbol.....	373
22.2.10	Setting End of Line.....	373
22.2.11	Getting Paragraphs Information .....	374
22.3	Notification Codes of Edit Box.....	375
22.4	Sample Program .....	375
23	Combo Box Control .....	379
23.1	Types and Styles of Combo Box .....	379
23.1.1	Simple Combo Box, Pull-Down Combo Box, and Spin Combo Box .....	379
23.1.2	Digital Spin Box.....	381
23.2	Messages of Combo Box .....	382
23.2.1	Messages of Simple, Pull-Down, and Spin Combo Box .....	382
23.2.2	Messages of Digital Spin Box.....	384
23.3	Notification Codes of Combo Box.....	385
23.4	Sample Program .....	386
24	Menu Button Control .....	391

24.1 Styles of Menu Button .....	391
24.2 Messages of Menu Button .....	392
24.2.1 Adding Items to Menu Button Control .....	392
24.2.2 Deleting Items from Menu Button Control .....	392
24.2.3 Deleting All Items in the Menu .....	393
24.2.4 Setting Current Selected Item.....	393
24.2.5 Getting Current Selected Item.....	393
24.2.6 Getting/Setting Data of Menu Item .....	393
24.2.7 Other Messages.....	394
24.3 Notification Codes of Menu Button.....	394
24.4 Sample Program .....	395
25 Progress Bar Control .....	399
25.1 Styles of Progress Bar.....	399
25.2 Messages of Progress Bar .....	400
25.2.1 Setting Range of Progress Bar.....	400
25.2.2 Setting Step Value of Progress Bar .....	400
25.2.3 Setting Position of Progress Bar .....	400
25.2.4 Setting Offset Based-on Current Position .....	401
25.2.5 Advancing Position by One Step .....	401
25.3 Notification Codes of Progress Bar.....	401
25.4 Sample Program .....	402
26 Track Bar Control.....	405
26.1 Styles of Track Bar .....	405
26.2 Messages of Track Bar.....	406
26.3 Notification Codes of Track Bar .....	407
26.4 Sample Program .....	407
27 Toolbar Control.....	409
27.1 Creating Toolbar Control.....	409
27.2 Styles of Toolbar .....	410
27.3 Messages of Toolbar .....	411
27.3.1 Adding an Item.....	411
27.3.2 Getting/Setting Information of an Item .....	413
27.3.3 NTBM_SETBITMAP Message.....	414
27.4 Notification Codes of Toolbar .....	414
27.5 Sample Program .....	414

<b>28 Property Sheet Control .....</b>	<b>419</b>
28.1 Styles of Property Sheet.....	420
28.2 Messages of Property Sheet .....	420
28.2.1 Adding Property Page.....	420
28.2.2 Procedure Function of Property Page .....	420
28.2.3 Deleting Property Page.....	423
28.2.4 Handle and Index of Property Page .....	423
28.2.5 Messages Relevant Property Page .....	424
28.3 Notification Codes of Property Sheet.....	425
28.4 Sample Program .....	425
<b>29 Scroll Window Control .....</b>	<b>431</b>
29.1 Scrollable Window .....	431
29.2 General Scroll Window Messages.....	432
29.2.1 Get/Set the Range of Content Area and Visible Area .....	432
29.2.2 Get Position Information and Set Current Position .....	433
29.2.3 Get/Set Scroll Properties.....	434
29.3 Message of Scroll Window Control .....	435
29.3.1 Add Child Control.....	435
29.3.2 Get Handle of Child Control.....	436
29.3.3 Container (Content) Window Procedure .....	436
29.4 Sample Program .....	437
<b>30 Scroll View Control .....</b>	<b>441</b>
30.1 Styles of Scroll View Control .....	441
30.2 Messages of Scroll View Control.....	442
30.2.1 Draw of List Item.....	442
30.2.2 Set Operation Functions of List Item .....	443
30.2.3 Operations on List Item.....	444
30.2.4 Get/Set Current Highlighted Item .....	446
30.2.5 Selection and Display of List Item .....	446
30.2.6 Optimization of Display.....	447
30.2.7 Set Range of Visible Area.....	448
30.3 Notification Codes of Scroll View Control .....	449
30.4 Sample Program .....	449
<b>31 Tree View Control .....</b>	<b>453</b>
31.1 Styles of Tree View Control .....	453

31.2 Messages of Tree View Control.....	454
31.2.1 Creating and Deleting Node Item .....	454
31.2.2 Setting/Getting Properties of Node Item .....	455
31.2.3 Selecting and Searching a Node Item .....	456
31.2.4 Comparing and Sorting.....	458
31.3 Notification Codes of Tree View Control .....	459
31.4 Sample Program .....	459
32 List View Control .....	463
32.1 Styles of List View Control .....	463
32.2 Messages of List View Control.....	464
32.2.1 Operations on Columns.....	464
32.2.2 Operations on List Item.....	467
32.2.3 Selecting, Displaying, and Searching List Item .....	472
32.2.4 Comparing and Sorting.....	474
32.2.5 Operation of Tree View Node.....	475
32.3 Handling of Key Messages.....	476
32.4 Notification Codes of List View Control .....	477
32.5 Sample Program .....	477
33 Month Calendar Control .....	483
33.1 Styles of Month Calendar.....	483
33.2 Messages of Month Calendar .....	483
33.2.1 Getting Date.....	483
33.2.2 Setting Date.....	485
33.2.3 Adjusting Colors.....	486
33.2.4 Size of Control.....	487
33.3 Notification Codes of Month Calendar.....	487
33.4 Sample Program .....	487
34 Spin Box Control .....	489
34.1 Styles of Spin Box .....	489
34.2 Messages of Spin Box .....	490
34.2.1 Setting/Getting Position.....	490
34.2.2 Disabling and Enabling.....	491
34.2.3 Target Window.....	491
34.3 Notification Codes of Spin Box.....	492
34.4 Sample Program .....	492



35 Cool Bar Control .....	495
35.1 Styles of Cool Bar .....	495
35.2 Messages of Cool Bar .....	495
35.3 Sample Program .....	497
36 Animation Control .....	499
36.1 ANIMATION Object .....	499
36.2 Styles of Animation Control .....	501
36.3 Messages of Animation Control .....	502
36.4 Sample Program .....	502
37 GridView Control .....	505
37.1 Styles of Gridview .....	505
37.2 Messages of GridView .....	505
37.2.1 Column Operations .....	506
37.2.2 Row Operations .....	509
37.2.3 Cell Operations .....	511
37.3 Other Messages .....	512
37.4 Notification Codes of GridView .....	513
37.5 Sample Program .....	514
38 IconView Control .....	519
38.1 Styles of IconView .....	519
38.2 Messages of IconView .....	520
38.2.1 Icon operation .....	520
38.3 Notification Codes of IconView .....	523
38.4 Sample Program .....	523
Appendix A Universal Startup API for RTOSes .....	528
A.1 Malloc Initialization Interface .....	528
A.2 Standard Output Initialization Interface .....	529
A.3 POSIX Threads Initialization Interface .....	529



# 1 Preface

MiniGUI is a cross operating system and lightweight GUI support system for real-time embedded systems, and is especially suitable for embedded systems based on Linux/uClinux, VxWorks, pSOS, eCos, uC/OS-II, ThreadX, OSE, and Nucleus. Launched at the end of 1998, the software has, over nine years of development, become a stable and reliable one securing widespread applications in a variety of products and projects. At present, the latest stable version of MiniGUI is 2.0.4/1.6.10. This handbook is the programming guide for the MiniGUI version 2.0.4/1.6.10<sup>1</sup>, which describes how to develop applications based on MiniGUI.

This handbook is a complete guide on MiniGUI programming, which describes the foundation knowledge of MiniGUI programming and various programming methods and skills, and detailed describes the main API functions. Though the handbook tries to detailed describe various aspects of MiniGUI programming, it is not a complete reference manual about MiniGUI APIs; please refer to *MiniGUI API Reference* for relevant information.

## 1.1 Relevant Documents

Except for this guide, Feynman Software also ships the following printed manual with the MiniGUI-VAR product:

- *MiniGUI User Manual* Version 2.0-4. This manual describes the compile-time configuration options and run-time configuration options of MiniGUI version 2.0.4/1.6.10.

In the directory `minigui/docs/` of MiniGUI-VAR CD-ROM, you can find the document files for this guide and *MiniGUI User Manual* Version 2.0-4 in PDF

---

<sup>1</sup> MiniGUI V2.0.x provides support for multi-process-based operating systems, like Linux; MiniGUI V1.6.x provides support for traditional real-time embedded operating systems, which are multi-thread- or multi-task- based. Except the difference of runtime modes supported, these two versions have the almost same features.

format. Besides these files, there are the following documents (in PDF format) in the above directory:

- *MiniGUI API Reference* for MiniGUI Version 2.0.4. This manual describes the APIs of MiniGUI V2.0.4 (MiniGUI-Processes runtime mode) in detail<sup>2</sup>.
- *MiniGUI API Reference* for MiniGUI Version 1.6.10. This manual describes the APIs of MiniGUI V1.6.10 (MiniGUI-Threads runtime mode) in detail<sup>3</sup>.
- *MiniGUI Technology Whitepaper for V2.0.4/1.6.10 and Datasheet for MiniGUI V2.0.4/1.6.10*.

**README** file located in the product CD-ROM root directory describes the file name and the location of above documents. There is also **ReleaseNotes.pdf** file in this directory. This file describes the new features, enhancements, and optimizations in this release. Please pay special attention to the backward compatibility issues.

Please visit <http://www.minigui.com/product/index.shtml> to get the complete products and purchase information.

## 1.2 Organization of this Guide

Except this preface, this text is divided into four parts with thirty-eight chapters in total:

- Part One: Foundation of MiniGUI Programming, from Chapter 2 to Chapter 12. We describe the foundation concepts for using MiniGUI programming in this part.
- Part Two: MiniGUI Graphics Programming, from Chapter 13 to Chapter 15. We describe the use and concepts of MiniGUI graphics related APIs in this part.
- Part Three: MiniGUI Advanced Programming Topics, from Chapter 16 to

---

<sup>2</sup> Only English edition in HTML format and Windows CHM format

<sup>3</sup> Only English edition in HTML format and Windows CHM format

Chapter 18. We describe the concepts of MiniGUI-Processes related advanced programming and the development of customizing GAL and IAL engine in this part.

- Part Four: MiniGUI Control programming, from Chapter 19 to Chapter 38 We describe the use of various controls provided by MiniGUI in this part.

Note: Feynman Software no longer releases MiniGUI-VAR for Linux/uClinux V1.6.10 based on MiniGUI-Lite mode. For uClinux, Feynman Software only releases MiniGUI-VAR for uClinux V1.6.10, which provides the support for MiniGUI-Threads and MiniGUI-Standalone runtime mode but no MiniGUI-Processes mode.

### 1.3 Obtaining Samples of the Guide

Some sample programs in the guide are from MDE (MiniGUI Demonstration Environment). We organize the other sample programs into a complete Autoconf/Automake project package called `mg-samples`, and save the package in the product CD-ROM.

For MiniGUI-VAR V2.0.4, the source packages are located in the directory `minigui/2.0.x` of the product CD-ROM. The packages are listed as follow:

- `libminigui-2.0.4-<os>.tar.gz`: The source package of MiniGUI V2.0.4 for `<os>` (like linux) operating system. MiniGUI is composed of three libraries: `libminigui` (source is in `src/`), `libmgext` (`ext/`), and `libvcongui` (`vcongui/`). `Libminigui` is the core library, which provides window management support and graphics interfaces as well as standard controls. `Libmgext` is an extension library of `libminigui`; it provides some useful controls and convenient functions, such as 'Open File Dialog Box'. `Libvcongui` provides a virtual console window in which you can run programs. `Libmgext` and `libvcongui` have already been contained in this package.
- `minigui-res-2.0.4.tar.gz`: Runtime resources required by MiniGUI including fonts, icons, bitmaps, and cursors.

- `mg-samples-2.0.4.tar.gz`: The sample program package for *MiniGUI Programming Guide*.
- `mde-2.0.4.tar.gz`: The MiniGUI demo program package, which provides some complex demo applications, such as notebook, housekeeper, and minesweeper.

For MiniGUI-VAR V1.6.10, the source packages are located in the directory `minigui/1.6.x` of the product CD-ROM. The packages are listed as follow:

- `libminigui-1.6.10-<os>.tar.gz`: The source package of MiniGUI V1.6.10 for `<os>` (like VxWorks) operating system. MiniGUI is composed of three libraries: `libminigui` (source is in `src/`), `libmgext` (`ext/`), and `libvcongui` (`vcongui/`). `Libminigui` is the core library, which provides window management support and graphics interfaces as well as standard controls. `Libmgext` is an extension library of `libminigui`; it provides some useful controls and convenient functions, such as 'Open File Dialog Box'. `Libvcongui` provides a virtual console window in which you can run programs. `Libmgext` and `libvcongui` have already been contained in this package.
- `minigui-res-1.6.10.tar.gz`: Runtime resources required by MiniGUI including fonts, icons, bitmaps, and cursors.
- `mg-samples-1.6.10.tar.gz`: The sample program package for *MiniGUI Programming Guide*.
- `mde-1.6.10.tar.gz`: The MiniGUI demo program package, which provides some complex demo applications, such as notebook, housekeeper, and minesweeper.

## 1.4 Compiling and Running Environment of Samples

This guide assumes that you run MiniGUI on Linux operating system. Therefore, some samples are described on the assumption that you are using Linux and GNU toolchain. However, most of the samples in this guide can run on other operating systems. For more information to compile the samples for an operating system other than Linux, please refer to *MiniGUI User Manual*.

We recommend the following PC computer configuration for running Linux:

- Pentium III or above CPU;
- 256MB or above memory;
- At least 15GB free disk space;
- Mouse using USB/PS2 interface (PS2 or IMPS2 mouse protocol);
- VESA2-compatible video card, ensure achieve to 1024x768 resolution, 16-bit color;
- Choose Red Hat Linux 9 as the development platform, and install all the software packages (need at least 5GB disk space for /usr file system); Fedora 3 or Debian is also good for the development platform.
- Partition your hard disk reasonably, mount `/usr`, `/usr/local`, `/home`, `/var`, `/opt` etc. on different disk partition and make sure 3GB space for `/usr/local` and `/opt` file system separately.

## 1.5 Copyright and Licensing

Note that although Feynman Software provides the complete MiniGUI source code for you, you are only permitted to add new graphics engine and input engine in order to support different hardware; you are not permitted to modify other source code of MiniGUI. Other rights are all reserved by Feynman Software.

Feynman Software releases the sample code (mde and mg-samples) under GPL. The original text of GPL license can be obtained by following means:

- `COPYING` file in mde and/or mg-samples software packages
- Visiting the website of <http://www.gnu.org/licenses/licenses.html>





# I Foundation of MiniGUI Programming

- Beginning MiniGUI Programming
- Window and Message
- Foundation of Dialog Box Programming
- Foundation of Control Programming
- Advanced Control Programming
- Menu
- Scrollbar
- Keyboard and Mouse
- Icon, Cursor and Caret
- Using MiniGUIExt Library
- Other Programming Topics



## 2 Beginning MiniGUI Programming

We describe the basic concepts and foundation knowledge of MiniGUI programming with a simple MiniGUI program in this chapter.

### 2.1 Basic Programming Concepts

#### 2.1.1 Event Driven Programming

MiniGUI is a graphics user interface support system, and general GUI programming concepts are all suitable for MiniGUI programming, such as window and event driven programming etc.

In a conventional GUI graphics system model, actions of keyboard or mouse generate events that are continuously checked by application. These events are generally sent to the focused window, and are transferred by application to procedures related to this window for handling. These window procedures are generally defined by application, or one of some standard procedures.

Operating system, event-handling procedure of other windows, and application code all can generate events.

The window procedure for handling events generally identifies one “window class”, and windows with the same window procedure are regarded as belonging to the same window class.

Concept of focus and cursor is used for managing the transfer of input devices and input events. Mouse cursor is a small bitmap drawn on the screen, which indicates the current mouse position. It is the responsibility of the windowing system to draw the bitmap in some non-subversive form, but the application can control which bitmap to draw and whether the cursor to be shown. The application can catch mouse cursor and get cursor event even if the cursor goes beyond the display region of the application window. Keyboard input has similar concept of input focus and caret. Only the window having input focus

can get the keyboard event. Changing the window focus is completed by combination of special keys or by mouse cursor event. The window having input focus usually draws a keyboard caret. The existence, form, position, and control of the caret are all performed by window event handling procedures.

Application can require redrawing the whole or part of the window by calling some system functions, which are usually handled by window procedures.

### **2.1.2 Three Runtime Modes of MiniGUI**

MiniGUI can be configured and compiled into three versions, which are completely different in architecture - MiniGUI-Processes, MiniGUI-Threads and MiniGUI-Standalone. We call these the different runtime modes.

- **MiniGUI-Threads.** Programs run in MiniGUI-Threads can create multiple windows in different threads, but all the windows are run in one process or address space. This runtime mode is greatly suitable for most of the conventional embedded operating systems, for example, uC/OS-II, eCos, VxWorks, pSOS etc. Certainly, MiniGUI can be run in the runtime mode of MiniGUI-Threads on Linux and uClinux.
- **MiniGUI-Processes.** In opposition to MiniGUI-Threads, a program running on MiniGUI-Processes is an independent process, which can also create multiple windows. MiniGUI-Processes are fit for full-featured UNIX-like operating systems, such as Linux.
- **MiniGUI-Standalone.** Under this runtime mode, MiniGUI can be run in independent process form, without support of multiple threads and multiple processes, this runtime mode is suitable for applications of single function. For example, in some embedded products using uClinux, thread library support is short of because of various reasons, at this time, MiniGUI-Standalone can be used to develop applications.

Compared to UNIX-like operating systems such as Linux, traditional embedded operating systems have some particularity. Say for example, operating systems such as uClinux, uC/OS-II, eCos, VxWorks and the like are generally run on the CPU without MMU (memory management unit, used to provide

support of virtual memory), here there is usually not the concept of process but the concept of thread or task, thus, runtime environments of GUI system are quite different. Therefore, to be suitable for different operating systems environments, we can configure MiniGUI into the above three runtime modes.

Generally speaking, the runtime mode of MiniGUI-Standalone has the widest adaptability, and can support almost all the operating systems, even including operating systems similar to DOS; the adaptability of the runtime mode of MiniGUI-Threads takes second place, and can real-time embedded operating systems with support of multiple tasks, or normal operating systems with complete UNIX properties; the adaptability of runtime mode of MiniGUI-Processes is smaller, and it is only suitable for normal operating system with complete UNIX properties.

The early version of MiniGUI (namely MiniGUI-Threads) adopts the mechanisms of message post based on POSIX thread and window management, which provides maximum data share, but also causes weakness in MiniGUI system architecture. The whole system will be affected if one thread terminates due to illegal data accessing. To solve this problem and make MiniGUI more appropriate for application requirements of embedded system based on Linux, we released MiniGUI-Lite mode from version 0.98. The MiniGUI-Lite do not use the thread mechanism and pthread library, thus MiniGUI is more stable and on embedded Linux. By using MiniGUI-Lite, we can run several client processes on the basis of efficient client/server architecture, and make use of superior features like address space protection. Thus, with MiniGUI-Lite runtime mode, the flexibility, stability, and scalability of embedded system based on MiniGUI will improve greatly. For example, we can run several MiniGUI client processes on MiniGUI-Lite, and if one process terminates abnormally, other processes will still run well. Moreover, on MiniGUI-Lite, it is convenient for us to integrate third-party applications. Actually, this is why many embedded device developers use Linux as their operating systems.

Although MiniGUI-Lite runtime mode provides support for multi-process, it

cannot manage windows created by different processes at one time. Therefore, MiniGUI-Lite distinguishes windows in different processes by layers. This method fits for the most embedded devices with low-resolution screen, but brings some problems for application development.

MiniGUI V2.0.x solves this problem completely. A window created by a client of MiniGUI-Lite is not a global object, i.e., a client does not know the windows created by others. However, windows created by MiniGUI-Processes are all global objects, and windows created by MiniGUI-Processes can clip each other. Thus, MiniGUI-Processes is a successor of MiniGUI-Lite; It offers full-featured support for multi-process embedded operating systems, such as Linux.

We can run multiple MiniGUI applications simultaneously with the MiniGUI-Processes mode. First, we start up a server program called `mginit`, and then we can start up other MiniGUI applications as clients. Server is not affected and can continue to run if the client terminates due to some reason.

In this guide, the demo programs assume that the MiniGUI-Processes version is configured and installed. Before running these demo applications, you should run the `mginit` program first, which can be a user-defined `mginit` program or an `mginit` program provided by MDE. We have coded carefully to ensure that each demo program can be compiled and run on MiniGUI-Processes, MiniGUI-Threads, or MiniGUI-Standalone.

Further more, MiniGUI provides APIs in Win32 style. Readers familiar with Win32 programming can grasp the basic methods and APIs quickly.

## **2.2 A Simple MiniGUI Program**

The quickest approach to understand the basic programming method of MiniGUI is to analyze structure of a simple program. List 2.1 shows a “Hello World” program of MiniGUI, which will be discussed in detail.

## List 2.1 helloworld.c

```

#include <stdio.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>

static int HelloWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    switch (message) {
        case MSG_PAINT:
            hdc = BeginPaint (hWnd);
            TextOut (hdc, 60, 60, "Hello world!");
            EndPaint (hWnd, hdc);
            return 0;

        case MSG_CLOSE:
            DestroyMainWindow (hWnd);
            PostQuitMessage (hWnd);
            return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

int MiniGUIMain (int argc, const char* argv[])
{
    MSG Msg;
    HWND hMainWnd;
    MAINWINCREATE CreateInfo;

#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER, "helloworld", 0, 0);
#endif

    CreateInfo.dwStyle = WS_VISIBLE | WS_BORDER | WS_CAPTION;
    CreateInfo.dwExStyle = WS_EX_NONE;
    CreateInfo.spCaption = "HelloWorld";
    CreateInfo.hMenu = 0;
    CreateInfo.hCursor = GetSystemCursor(0);
    CreateInfo.hIcon = 0;
    CreateInfo.MainWindowProc = HelloWinProc;
    CreateInfo.lx = 0;
    CreateInfo.ty = 0;
    CreateInfo.rx = 240;
    CreateInfo.by = 180;
    CreateInfo.iBkColor = COLOR_lightwhite;
    CreateInfo.dwAddData = 0;
    CreateInfo.hHosting = HWND_DESKTOP;

    hMainWnd = CreateMainWindow (&CreateInfo);

    if (hMainWnd == HWND_INVALID)
        return -1;

    ShowWindow(hMainWnd, SW_SHOWNORMAL);

    while (GetMessage(&Msg, hMainWnd)) {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }

    MainWindowThreadCleanup (hMainWnd);

    return 0;
}

#ifdef _LITE_VERSION
#include <minigui/dti.c>
#endif

```

As shown in Figure 2.1, the program creates an application window with size of 320x240 pixels, and displays "Hello world!" at the center of the window client region.



Figure 2.1 The window created by helloworld program

### 2.2.1 Header Files

The four header files included in the beginning of helloworld.c, namely `minigui/common.h`, `minigui/minigui.h`, `minigui/gdi.h`, and `minigui/window.h`, should be included for all MiniGUI applications.

- `common.h` Includes definitions of macros and data types commonly used in MiniGUI.
- `minigui.h` Includes definitions of global and general interface functions and some miscellaneous functions.
- `gdi.h` Includes definitions of interfaces of MiniGUI graphics functions.
- `window.h` Includes definitions of macros, data types, data structures, which are relative to the windows and declarations of function interfaces.

Another header file must be included for MiniGUI applications using predefined controls -- `minigui/control.h`:

- `control.h` Includes interface definitions of all the built-in controls in minigui library.

Therefore, a MiniGUI program usually includes the following MiniGUI related header files in the beginning:



```
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>
```

### 2.2.2 Entrance of the Program

The entrance of a C program is the `main` program, while the entrance of a MiniGUI program is `MiniGUIMain`, and the prototype of the program is as follows:

```
int MiniGUIMain (int argc, const char* argv[])
```

The `main` function is already defined in the MiniGUI function library, and the function calls the `MiniGUIMain` function after some initialization of MiniGUI. So the entrance of each MiniGUI application (no matter server function `mginit` or client application) is the `MiniGUIMain` function. Arguments of `argc` and `argv` have the same meaning as that of the `main` function in C program, which are the number of command line arguments and the string array pointer to the arguments, respectively.

### 2.2.3 Join a Layer in MiniGUI-Processes

```
#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER , "helloworld" , 0 , 0);
#endif
```

`JoinLayer` is a special function of MiniGUI-Processes, and is therefore included in the conditional compilation of `_MGRM_PROCESSES`. In the runtime mode MiniGUI-Processes, each MiniGUI client program must call this function to join a layer (or create a new layer) before calling other MiniGUI functions<sup>4</sup>.

If the program is the server of MiniGUI-Processes, you should call `ServerStartup` instead:

```
if (!ServerStartup (0 , 0 , 0)) {
```

<sup>4</sup> The old function `SetDesktopRect` of Version 1.6.x has been deleted.

```
forintf (stderr.  
    "Can not start the server of MiniGUI-Processes: mginit.\n");  
return 1;  
}
```

We will give detailed description on interfaces specific to MiniGUI-Processes in Chapter 17.

Note that MiniGUI defines different macros for three runtime modes:

- MiniGUI-Threads: `_MGRM_THREADS`
- MiniGUI-Processes: `_MGRM_PROCESSES` and `_LITE_VERSION`
- MiniGUI-Standalone: `_MGRM_STANDALONE`, `_LITE_VERSION`, and `_STAND_ALONE`

## 2.2.4 Creating and Showing Main Window

```
hMainWnd = CreateMainWindow (&CreateInfo);
```

The initial user interface of each MiniGUI application is usually a main window; you can create a main window by calling `CreateMainWindow` function. The argument of `CreateMainWindow` function is a pointer to `MAINWINCREAT` structure, which is `CreateInfo` in this example, and the return value is the handle to the created main window. `MAINWINCREAT` structure describes the properties of a main window, and you need to set its properties before using `CreateInfo` to create a main window.

```
CreateInfo.dwStyle = WS_VISIBLE | WS_BORDER | WS_CAPTION;
```

The above statement sets the style of the main window, herein the window is set to be visible initially, and to have border and caption bar.

```
CreateInfo.dwExStyle = WS_EX_NONE;
```

The above statement sets the extension style of the main window, and the window has no extension style.

```
CreateInfo.spCaption = "HelloWorld";
```

This statement sets the caption of the main window to be "HelloWorld".

```
CreateInfo.hMenu = 0;
```

This statement sets the main menu of the main window, and the window has no main menu.

```
CreateInfo.hCursor = GetSystemCursor(0);
```

This statement sets the cursor of the main window, and the cursor for this window is the default system cursor.

```
CreateInfo.hIcon = 0;
```

This statement sets the icon of the main window, and the window has no icon.

```
CreateInfo.MainWindowProc = HelloWinProc;
```

This statement sets the window procedure function of the main window to be **HelloWinProc**, and all the messages sent to the window are handled by this function.

```
CreateInfo.lx = 0;  
CreateInfo.ty = 0;  
CreateInfo.rx = 320;  
CreateInfo.by = 240;
```

The above statements set the position and the size of the main window on the screen, and the upper-left corner and lower-right corner are located in (0, 0) and (320, 240), respectively.

```
CreateInfo.iBkColor = PIXEL_lightwhite;
```

This statement sets the background color of the main window to be white, and **PIXEL\_lightwhite** is the pixel value predefined by MiniGUI.

```
CreateInfo.dwAddData = 0;
```

This statement sets the additional data of the main window, and the window has no additional data.

```
CreateInfo.hHosting = HWND_DESKTOP;
```

This statement sets the hosting window of the main window to be the desktop window.

```
ShowWindow(hMainWnd, SW_SHOWNORMAL);
```

**ShowWindow** function needs to be called to show the created window on the screen after the main window is created. The first argument of **ShowWindow** is the handle of the window to be shown, and the second argument specifies the style of showing the window (show or hide). **SW\_SHOWNORMAL** means showing the main window and setting it to be the top-most window.

### 2.2.5 Entering Message Loop

The main window will be displayed on the screen when **ShowWindow** function is called. Like other GUI, it is time to go into the message loop. MiniGUI maintains a message queue for each MiniGUI program. After an event happens, MiniGUI transforms the event into a message, and put the message into the message queue of the target window. Then the task of the application is to execute the following message loop code to get the message from the message queue continuously and handle them.

```
while (GetMessage(&Msg, hMainWnd)) {  
    TranslateMessage(&Msg);  
    DispatchMessage(&Msg);  
}
```

The type of **Msg** variable is a **MSG** structure, which is defined in **minigui/window.h** as follows:

```
typedef struct _MSG  
{  
    HWND        hwnd;  
    int         message;  
    WPARAM      wParam;  
    LPARAM      lParam;
```

```
    unsigned int    time;  
#ifndef _LITE_VERSION  
    void*          pAdd;  
#endif  
} MSG;  
  
typedef MSG* PMSG;
```

**GetMessage** function gets a message from the message queue of the application.

```
GetMessage( &Msg, hMainWnd);
```

The second argument of this function is the handle to the main window, and the first argument is the pointer to a **MSG** structure. **GetMessage** function fills the fields of the **MSG** structure with the message gotten from the message queue, which includes:

- **hwnd**: The handle of the window to which hwnd message is sent. The value is the same with **hMainWnd** in the **helloworld.c** program.
- **message**: Message identifier. This is an integer for identifying a message. Each messages has a corresponding predefined identification, these identification are defined in **minigui/window.h** and with **MSG\_** prefix.
- **wParam**: The first 32-bit message parameter, the meaning and value of which is different for different message.
- **lParam**: The second 32-bit message parameter, the meaning and value of which depends on the message.
- **time**: The time when the message is put into the message queue.

If only the message gotten from the message queue is not **MSG\_QUIT**, **GetMessage** will return a non-zero value, and the message loop will be continued. **MSG\_QUIT** message makes the **GetMessage** return zero, and results in the termination of the message loop.

```
TranslateMessage (&Msg);
```

**TranslateMessage** function translates a keystroke message to a **MSG\_CHAR** message, and then sends the message to the window procedure function.

```
DispatchMessage (&Msg);
```

**DispatchMessage** function will finally send the message to the window procedure of the target window, and let it handle the message. In this example, the window procedure is **HelloWinProc**. That is to say, MiniGUI calls the window procedure function (callback function) of the main window in the **DispatchMessage** function to handle the messages sent to this main window. After handling the messages, the window procedure function of the application would return to **DispatchMessage** function, while **DispatchMessage** function will return to the application code in the end, and the application begins a new message loop by calling the next **GetMessage** function.

## 2.2.6 Window Procedure Function

Window procedure function is the main body of MiniGUI program. The most work of an application happens in the window procedure function actually, because the main task of GUI program is to receive and handle various messages received by the window.

The window procedure is the function named as **HelloWinProc** in **helloworld.c** program. Programs may name the window procedure function arbitrarily. **CreateMainWindow** function creates the main window according to the window procedure specified in **MAINWINCREATE** structure.

The window procedure function always has the following prototype:

```
static int HelloWinProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
```

The four arguments of the window procedure are the same as the first four fields of **MSG** structure. The first argument **hWnd** is the handle of the window to receive messages, which is the same with the return value of **CreateMainWindow** function and indicates the specific window to receive the message. The second argument is the same as the **message** field of **MSG** structure, which is an integer value indicating the received message. The last

two arguments are both 32-bit message parameters, which provide special information related to the message.

The window procedure function is usually not called by the program directly, but is called by MiniGUI, that is, it is a callback function.

The messages, which are not handled by the window procedure function, should be passed to `DefaultMainWinProc` function to perform default handling, the return value of `DefaultMainWinProc` must be returned by the window procedure.

### 2.2.7 Screen Output

The program executes screen output when responding to the `MSG_PAINT` message. The application gets the device context handle first by calling `BeginPaint` function, and uses it to call GDI functions to execute drawing operations. Herein, the program uses `TextOut` function to display a "Hello world!" string centered at the client region of the window. The application calls `EndPaint` function to release the device context handle after the drawing is completed.

We will give detailed description on MiniGUI graphics device interfaces in Part II of this guide.

### 2.2.8 Exit of the Program

The window procedure function will receive a `MSG_CLOSE` message when the user clicks the close button on the upper-right corner of the window. Helloworld program calls `DestroyMainWindow` function to destroy the main window when it receives the `MSG_CLOSE` message, and calls `PostQuitMessage` function to throw a `MSG_QUIT` message into the message queue. `GetMessage` function will return 0 when receiving the message `MSG_QUIT`, and finally results in exit of the program from the message loop.

The program calls `MainWindowThreadCleanup` to clean the system resource such as message queue used by the main window and returns from `MiniGUIMain` finally.

## 2.3 Compiling, Linking and Running

### 2.3.1 Compiling MiniGUI Program

You can input the following command in the command line to compile `helloworld.c`, and link it to generate the executive file `helloworld`:

```
$ gcc -o helloworld helloworld.c -lminigui -ljpeg -lpng -lz
```

Following compiling options are needed if MiniGUI is configured to be MiniGUI-Threads:

```
$ gcc -o helloworld helloworld.c -lpthread -lminigui -ljpeg -lpng -lz
```

Here, `-o` option tells `gcc` the target file name to be generated, which is herein `helloworld`; `-l` option specifies the library needed to be linked when generating `helloworld`, which is herein minigui library and pthread library will be linked if MiniGUI is configured to be MiniGUI-Threads. Libpthread is the library to provide POSIX compatible thread support, which must be linked when compiling a MiniGUI-Threads program; The program compiled here only uses the functions in MiniGUI core library libminigui, and does not use the functions provided by other MiniGUI libraries (such as libmgext or libvcongui), therefore only minigui library is needed to be linked and the linking option is `-lminigui`; other function libraries such as jpeg, png, z and so on are the function libraries which are relied on by MiniGUI internally (herein we presume that you have enabled JPEG and PNG image support when configuring MiniGUI).

Provided that you configure MiniGUI to be MiniGUI-Processes, you must first ensure the server program `mginit` of MiniGUI have been started up before



running `helloworld` program. For example, you can start up `mginit` program of MDE, and then enter the directory where `helloworld` file exists, and input `./helloworld` in the command line to start up the `helloworld` program.

```
$ ./helloworld
```

Result of running the program is as shown in Fig. 2.1.

**[Prompt] If MiniGUI has already been configured as MiniGUI-Threads, you need not start up `mginit` programs. One MiniGUI-Threads application can be run directly from the console.**

### 2.3.2 Function Libraries of MiniGUI

Besides core library `libminigui`, MiniGUI also includes two other libraries: `libvcongui` and `libmgext`. `Libmgext` includes some useful controls and convenient graphics user interface routines, such as “open file” dialog, and `libvcongui` is virtual console support library. If you use functions provided by these libraries in your program, you may need to include the corresponding header files in your program and link the corresponding libraries when compiling the application.

## 2.4 Writing Automake/Autoconf Scripts for MiniGUI Application

We have already known that Autoconf/Automake is the best tool for maintaining a software project in UNIX system, which can help us be disengaged from work of inputting repeated command lines and maintain a project, and can even help us complete cross compiling of programs. MDE, which is released together with MiniGUI, is such a software project organized with Autoconf/Automake.

We will build project script for `helloworld` referring to Autoconf/Automake script of MDE. We will not describe in detail on working mechanism of

Autoconf/Automake, and relevant information can be obtained referring to books about Linux programming or the info pages of these two programs.

Considering that the project built in this section can also be used for organizing or maintaining the demo programs of the succeeding sections in this guide, we make `samples` directory as the root directory of the project in an appropriate directory of the system, and name the project as `samples`. For example:

```
$ mkdir -p ~/minigui/samples
```

**[Prompt] It is presumed that the source code of MiniGUI and MDE are located in the minigui directory of your home directory, which are `~/minigui/libminigui-2.0.x` and `~/minigui/mde-2.0.x`, respectively.**

Then make `src` directory in `samples` to store the source code of the helloworld program. Save `helloworld.c` into `samples/src/` directory, and then copy `configure.in` file from `mde-2.0.x`.

**[Prompt] Saving source files in a single directory can help us manage the project files better. As a rule, source code of a project should be saved in the directory `src/`, and global header files should be saved in the directory `include/`.**

We will modify the `samples` project based on MDE configuration script below. It should be noted that these scripts need Autoconf 2.53 and Automake 1.6 or higher version, and using lower version (such as Red Hat 7.x or lower) Autoconf and Automake will result in error.

First, we modify `configure.in` file. The modified file is as follows (Note the comments in the text. We only modify the commented macros):

```
dnl Process this file with autoconf to produce a configure script.
AC_PREREQ(2.13)

dnl Specify a source file of the project in the following macro
AC_INIT(src/helloworld.c)
```

```

dnl =====
dnl needed for cross-compiling
AC_CANONICAL_SYSTEM

dnl =====
dnl Checks for programs.
AC_PROG_MAKE_SET
AC_PROG_CC

dnl Specify project name (samples) and project version (1.0) in the following macro
AM_INIT_AUTOMAKE(samples,1.0)

dnl =====
dnl Checks for typedefs, structures, and compiler characteristics.
AC_C_CONST

dnl =====
dnl Checks for header files.
AC_HEADER_STDC
AC_HEADER_SYS_WAIT
AC_HEADER_TIME
AC_CHECK_HEADERS(sys/time.h unistd.h)

dnl =====
dnl check for libminigui
have_libminigui="no"
AC_CHECK_HEADERS(minigui/common.h, have_libminigui=yes, foo=bar)

dnl =====
dnl check for runtime mode of MiniGUI
dnl =====
threads_version="no"
AC_CHECK_DECLS(_MGRM_THREADS, threads_version="yes", foo=bar, [#include <minigu
i/common.h>])

procs_version="no"
AC_CHECK_DECLS(_MGRM_PROCESSES, procs_version="yes", foo=bar, [#include <minigu
i/common.h>])

standalone_version="no"
AC_CHECK_DECLS(_MGRM_STANDALONE, standalone_version="yes", foo=bar, [#include <
minigui/common.h>])

dnl =====
dnl check for newgal or oldgal interface.
use_newgal="no"
AC_CHECK_DECLS(_USE_NEWGAL, use_newgal="yes", foo=bar, [#include <minigui/common.h>])

dnl =====
dnl Write Output

if test "$ac_cv_prog_gcc" = "yes"; then
    CFLAGS="$CFLAGS -Wall -Wstrict-prototypes -pipe"
fi

if test "x$threads_version" = "xyes"; then
    CFLAGS="$CFLAGS -D_REENTRANT"
    LIBS="$LIBS -lpthread -lminigui"
else
    LIBS="$LIBS -lminigui"
fi

AC_CHECK_DECLS(_HAVE_MATH_LIB, LIBS="$LIBS -lm", foo=bar, [#include <minigui/co
mmon.h>])
AC_CHECK_DECLS(_PNG_FILE_SUPPORT, LIBS="$LIBS -lpng", foo=bar, [#include <minig
ui/common.h>])
AC_CHECK_DECLS(_JPG_FILE_SUPPORT, LIBS="$LIBS -ljpeg", foo=bar, [#include <mini
gui/common.h>])
AC_CHECK_DECLS(_TYPE1_SUPPORT, LIBS="$LIBS -lt1", foo=bar, [#include <minigui/c
ommon.h>])
AC_CHECK_DECLS(_TTF_SUPPORT, LIBS="$LIBS -lttf", foo=bar, [#include <minigui/co
mmon.h>])

dnl First comment the following four macros, which will be opened in the future
dnl AM_CONDITIONAL(MGRM_THREADS, test "x$threads_version" = "xyes")

```

```
dn1 AM_CONDITIONAL(MGRM_PROCESSES, test "$procs_version" = "xves")
dn1 AM_CONDITIONAL(MGRM_STANDALONE, test "$standalone_version" = "xyes")
dn1 AM_CONDITIONAL(USE_NEWGAL, test "$use_newgal" = "xyes")

dn1 List the Makefile file to be generated in the following macro
AC_OUTPUT(
  Makefile
  src/Makefile
)

if test "$have_libminigui" != "xyes"; then
  AC_MSG_WARN([
    MiniGUI is not properly installed on the system. You need MiniGUI Ver 2.0.2
    or later for building this package. Please configure and
    install MiniGUI Ver 2.0.x first.
  ])
fi
```

Following works can be done with the configuration script generated by **configure.in** and **Makefile** file:

- Generate the configuration script that appropriate for cross compilation.
- Check whether MiniGUI has been installed in the system.
- Check whether the installed MiniGUI in the system is configured to be MiniGUI-Processes, MiniGUI-Threads, or MiniGUI-Standalone, and set appropriately the function libraries to be linked into the program.
- Determine other dependent function libraries to be linked according to the configuration options of MiniGUI.
- Generate Makefile in the root directory of the project and Makefiles in the **src/** subdirectory.

Next, we create **Makefile.am** file in the root directory of the project, the content of which is as follows:

```
SUBDIRS = src
```

Above file content tells Automake to enter **src/** directory to handle sequentially.

Then we create **Makefile.am** file in the **src/** subdirectory, and the content of the file is as follows:

```
noinst_PROGRAMS=helloworld
helloworld_SOURCES=helloworld.c
```

The file content above tells Automake to generate a Makefile, which can be used to create `helloworld` program from `helloworld.c`.

Finally, we return to the root directory of the project and create an `autogen.sh` file, and the content of the file is as follows:

```
#!/bin/sh
aclocal
automake --add-missing
autoconf
```

This file is a shell script, which calls `aclocal`, `automake`, and `autoconf` command successively. Note that after the file being created, `chmod` command should be run to make it into an executable.

```
$ chmod +x autogen.sh
```

Up to now, we can run the following command to generate the Makefile file required by the project:

```
$ ./autogen.sh
$ ./configure
```

**[Prompt] `./autogen.sh` command should be run to refresh configure script and makefiles after `configure.in` file has been modified.**

After having run above commands, you will find many file automatically generated in the root directory of the project. It is unnecessary to care the purpose of these files. Ignore them, and run the `make` command:

```
[weiy@adsl samples]$ make
Making all in src
make[1]: Entering directory `/home/weiy/minigui/samples/src'
source='helloworld.c' object='helloworld.o' libtool=no \
depfile='.deps/helloworld.Po' tmpdepfile='.deps/helloworld.TPo' \
depmode=gcc3 /bin/sh ../depcomp \
gcc -DPACKAGE_NAME=\"\" -DPACKAGE_TARNAME=\"\" -DPACKAGE_VERSION=\"\" -DPACKAGE_STRING=
\"\" -DPACKAGE_BUGREPORT=\"\" -DPACKAGE=\"samples\" -DVERSION=\"0.1\" -DSTDC_HEADERS=1
-DHAVE_SYS_WAIT_H=1 -DTIME_WITH_SYS_TIME=1 -DHAVE_SYS_TYPES_H=1 -DHAVE_SYS_STAT_H=1 -DH
AVE_STDLIB_H=1 -DHAVE_STRING_H=1 -DHAVE_MEMORY_H=1 -DHAVE_STRINGS_H=1 -DHAVE_INTTYPES_H
=1 -DHAVE_STDINT_H=1 -DHAVE_UNISTD_H=1 -DHAVE_SYS_TIME_H=1 -DHAVE_UNISTD_H=1 -DHAVE_MIN
```

```
IGUI_COMMON_H=1 -DHAVE_DECL_MGRM_PROCESSES=1 -DHAVE_DECL_MGRM_THREADS=0 -DHAVE_DECL_MGRM_STANDALONE=0 -DHAVE_DECL_USE_NEWGAL=1 -I. -I. -g -O2 -Wall -Wstrict-prototype
s -pipe -D_REENTRANT -c `test -f 'helloworld.c' || echo './`helloworld.c
gcc -g -O2 -Wall -Wstrict-prototypes -pipe -D_REENTRANT -o helloworld helloworld.o -
lpthread -lminigui -ljpeg -lpng -lz -lt1 -lttf
make[1]: Leaving directory `/home/weiyminigui/samples/src'
make[1]: Entering directory `/home/weiyminigui/samples'
make[1]: Nothing to be done for `all-am'.
make[1]: Leaving directory `/home/weiyminigui/samples'
```

If you have a careful look at above output, you can find that `make` command enters `src/` subdirectory first, and call `gcc` to compile `helloworld.c` into the object file `helloworld.o`, and then call `gcc` again to generate `helloworld` program. Notice that `gcc` links the functions in the libraries of `pthread`, `minigui`, `jpeg`, `png` etc. (`-lpthread -lminigui`) when generating `helloworld` program. The reason is because that the author has configured MiniGUI to be MiniGUI-Threads runtime mode, linking `pthread` library is needed for generating MiniGUI-Threads application, and MiniGUI provides the support for JPEG and PNG picture by using `jpeg` and `png` libraries.

If the scale of the `helloworld` program is very huge, and thus source code are placed into different source files, you need only modify the `Makefile.am` file in `src/`, append the names of these source files to the `helloworld_SOURCES`, and then run `make` command again in the root directory of the project. For example:

```
noinst_PROGRAMS=helloworld
helloworld_SOURCES=helloworld.c helloworld.h module1.c module2.c
```

**[Prompt] Please list source files and header files on which the program depends behind `foo_SOURCES`.**

Demo programs of other chapters in this guide can be added to this project conveniently. For example, in order to add `foo` program, we can modify the `Makefile.am` file as follows:

```
noinst_PROGRAMS=helloworld foo
helloworld_SOURCES=helloworld.c
foo_SOURCES=foo.c
```

Thus, two program files, which are `helloworld` and `foo` respectively, will be

generated during compilation.

**[Prompt] Foo is generally used to specify a hypothetical object or name, which should be replaced by the real name in an actual project. Demo programs after this section can all be added into samples project in this way.**

Having such a simple project frame and Automake/Autoconf script template, we can enrich these scripts based on our requirements. These scripts can help us perform many works, the most important among which is to configure cross compilation option to help us porting our application into the target system. You can refer to *MiniGUI User Manual* to get knowledge about cross compilation of MiniGUI application.

Full demo program package of this guide is `mg-samples-2.0.x.tar.gz`, which includes all the demo programs of this guide and includes complete Autoconf/Automake for your reference.





## 3 Window and Message

Window and message/event are two important concepts in graphics user interface programming. A window is a rectangle region on the screen, and the application uses windows to display the output or accept the user input. Popular GUI programming generally adopts the event driven mechanism. Meaning of event driven is that, program process is not the serial executive line with only an entrance and an exit; in contrast, program will be in a loop status all the time. Program gets some events continuously from outside and inside, such as keystrokes or cursor moving by the user, then responds based on these events and achieves certain functions. The loop would not terminate until it receives a certain message. Generally said, “message queue” and “message loop” are the bottom facility of “event driven”.

This chapter will describe in detail the window model and the message handling mechanism of MiniGUI, several important functions for handling messages, and the difference between MiniGUI-Threads and MiniGUI-Processes in implementation of message loop.

### 3.1 Window System and Window

#### 3.1.1 What Is Window System

Computers with graphics user interface manage the display on the screen of an application by the window system. The components of a graphics user interface usually have the relationship as shown in Fig.3.1.

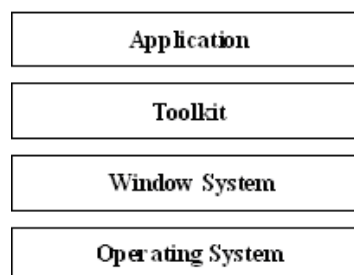


Fig. 3.1 Components of graphics user interface

The window system is a software system, which helps the user managing and controlling different display environments by dividing the display screen into different parts. The window system provides a working model based on windows, and each window is a rectangle region on the screen and is parallel to the boundary of screen. Applications can own one or more windows. The window system generally adopts the concept and mechanism of “overlap windows” to manage windows display and individual windows are overlapped on the screen. The window system overcomes the shortcomings of older terminal machines that only one job can be done on one screen; The window system makes the user able to monitor several jobs simultaneously on one screen, and switch between the jobs conveniently as well.

### **3.1.2 Concept of Window**

A window is a rectangle region on the screen. In traditional windowing system model, the vision part is comprised of one or more windows. Each window represents one drawing region on the screen, and the windowing system controls the mapping of the drawing region to the actual screen, that is, to control the position, size, and vision region of the window. Each window is allocated a screen drawing region to display part or whole of the window, even if it is not allocated screen region at all (this window is overlapped and hidden by other overlap windows).

Overlapping windows on the screen generally has the following relationship:

- Windows are generally organized in the form of hierarchy (or form of tree structure).
- The root window is the ancestor of all windows, and occupies the whole screen; it is also referred as desktop window.
- All the other windows have parent windows except the root window, and each window may have child windows, brother windows, ancestor windows, or descendant windows etc.
- A child window is contained in a parent window, and the child windows of the same parent window are regarded as windows of the same level.
- The visibility of overlapped windows depends on their relationship. A

window is visible only when its parent window is visible, and its parent window can clip a child window.

- Windows of the same level can be overlapped, but only one window can be outputted to the overlapped region at a time.
- A frame window/main window comprises a usable client region and a decorate region (also referred as “non client region”) managed by the windowing system.
- A child window of the desktop window is generally a frame window.
- Windows have subordinate relationship, that is, their owner determines the life cycle and visibility of some windows. A parent window generally owns their child windows.

An application window usually includes the following parts:

- A visible border.
- A window ID, which is used by programs to operate the window and is referred as “window handle” in MiniGUI.
- Some other characteristics, such as, height, width and background color etc.
- Additional window elements such as menu and scrollbars may exist.

## 3.2 Window of MiniGUI

### 3.2.1 Window Type

MiniGUI has three window types: main window, dialog box, and control window (child window). Each MiniGUI application generally creates a main window, which is used as the main interface or start interface of the application. The main window generally includes some child windows, and these child windows are generally control windows or user-defined window classes. An application can also create windows of other types such as dialog box or message box. A dialog box is actually a main window, and the application usually prompts the user to perform input operation within a dialog box. Message box is built-in dialog box used to display some prompt or warning for the user.

Fig 3.2 is a typical main window of MiniGUI; Fig. 3.3 is a typical dialog box of

MiniGUI.

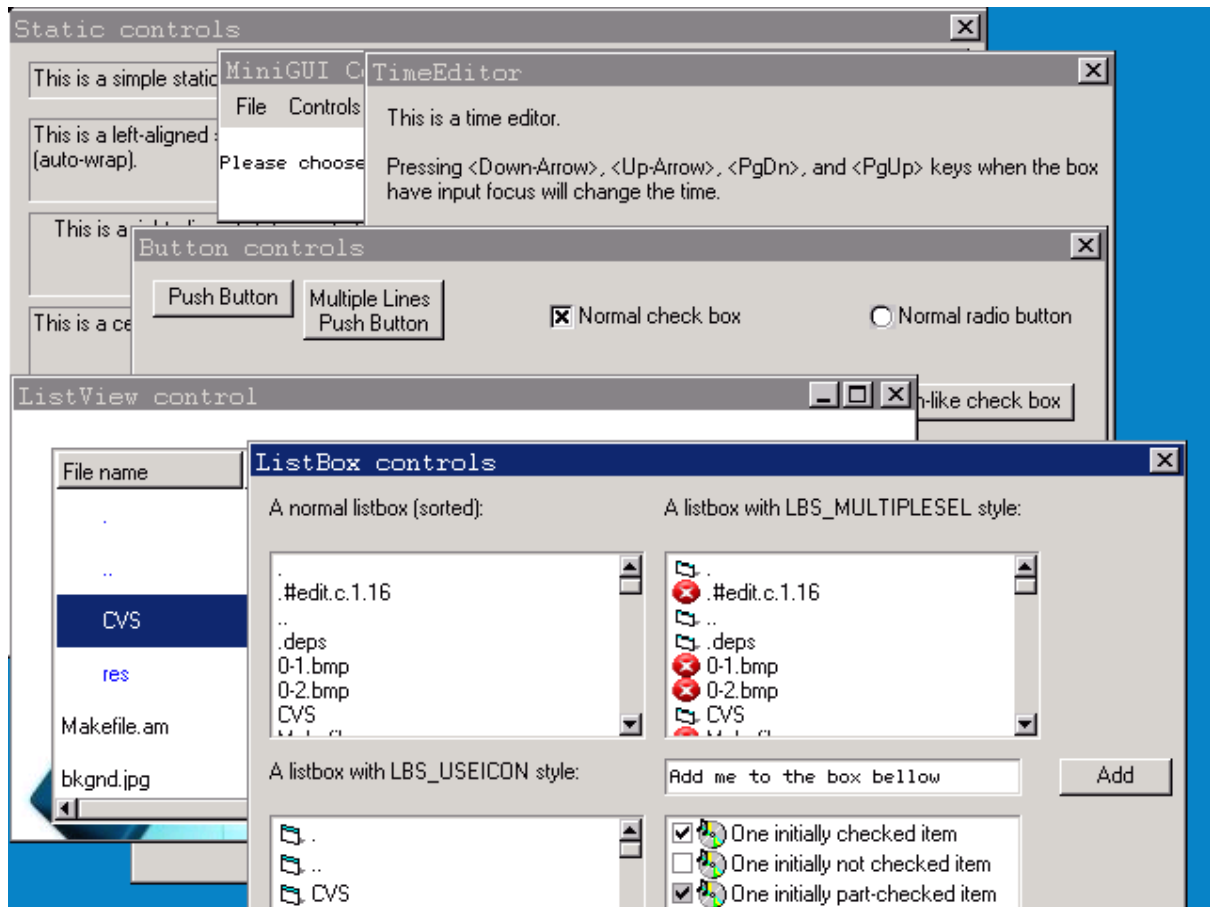


Fig. 3.2 Typical main windows (classic style) of MiniGUI

**[Prompt]** You can change the appearance style of MiniGUI window when configuring MiniGUI. Fig. 3.2, Fig. 3.3 illustrates display effect with different styles. Please refer to *MiniGUI User Manual* for relative information about MiniGUI appearance styles.

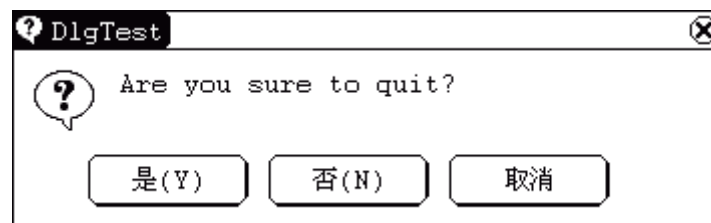


Fig. 3.3 A typical dialog box (flat style) of MiniGUI

### 3.2.2 Main Window

The main window in MiniGUI has no concept of window class; you should initialize a **MAINWINCREATE** structure, and then call **CreateMainWindow** function to create a main window. Meanings of the members of **MAINWINCREATE** structure are as follow:

<code>CreateInfo.dwStyle</code>	the style of the window
<code>CreateInfo.spCaption</code>	the caption of the window
<code>CreateInfo.dwExStyle</code>	the extended style of the window
<code>CreateInfo.hMenu</code>	the handle of the menu attached to the window
<code>CreateInfo.hCursor</code>	the handle of the cursor of the window
<code>CreateInfo.hIcon</code>	the icon of the window
<code>CreateInfo.MainWindowProc</code>	the window procedure of the window
<code>CreateInfo.lx</code>	absolute x-coordinate of upper-left corner of window relative to screen in pixel
<code>CreateInfo.ty</code>	absolute y-coordinate of upper-left corner of window relative to screen in pixel
<code>CreateInfo.rx</code>	absolute x-coordinate of lower-right corner of window relative to screen in pixel
<code>CreateInfo.by</code>	absolute y-coordinate of lower-right corner of window relative to screen in pixel
<code>CreateInfo.iBkColor</code>	the background pixel value of the window
<code>CreateInfo.dwAddData</code>	a 32-bit additional value attached to the window
<code>CreateInfo.hHosting</code>	the handle of the hosting window of the window

Wherein the following members need to be illustrated specially:

1. **CreateInfo.dwAddData**: During C language coding, static variables should be reduced as far as possible, but how to transfer parameters to window without using static variables? At this situation, you can use this field. This field is a 32-bit value; therefore all the parameters needed to transfer to the window can be organized into a structure, and assigns the pointer to this structure to this field. In a window procedure, the pointer can be obtained by using **GetWindowAdditionalData** function, so that the parameters that needed to be transferred can be obtained.
2. **CreateInfo.hHosting**: This field means the message queue of which main window is used for the main window to be created. We call the main window as a "hosted" main window when it uses message queue of other main window. Concept of host is very important in MiniGUI, and the following rules should generally be obeyed:
  - **MiniGUI-Threads**: The hosting window of the first main window created by each thread in MiniGUI-Threads must be the desktop,

namely `HWND_DESKTOP`. Other windows of this thread must adopt the existed main window belonging to the same thread as the hosting window. The system creates a new message queue when the hosting window is `HWND_DESKTOP`, while uses the message queue of the hosting window when a non-desktop window is specified as hosting window. That is to say, all the main windows of the same thread should use the same message queue.

- All the main windows in MiniGUI-Processes should specify hosting window according to similar rules, therefore it is enough to regard all main window as belonging to the same thread.

### 3.2.3 Window Style

Window style is used to control some appearances and behaviors of windows, such as border type of window, visibility of window, and usability of window etc. In MiniGUI, window styles can be divided to normal style and extended style, which is specified by `dwStyle` and `dwExStyle` when calling `CreateMainWindow` or `CreateWindowEx` to create a main window. We will describe the special style of controls in following Part III of this guide. Some common styles are listed in Table 3.1. The identifiers for these styles are defined in `minigui/window.h`, with prefix of `WS_` or `WS_EX`.

Table 3.1 Common styles of a window

Style identifier	Meaning	Comment
<code>WS_NONE</code>	no any style	
<code>WS_VISIBLE</code>	creating initially visible window	
<code>WS_DISABLED</code>	creating initially disabled window	
<code>WS_CAPTION</code>	creating main window with caption	limited to main window
<code>WS_SYSMENU</code>	creating main window with system menu	limited to main window
<code>WS_BORDER</code>	creating window with border	
<code>WS_THICKFRAME</code>	creating window with thick border	
<code>WS_THINFRAME</code>	creating window with thin border	
<code>WS_VSCROLL</code>	creating window with vertical scroll bar	
<code>WS_HSCROLL</code>	creating window with horizontal scroll bar	
<code>WS_MINIMIZEBOX</code>	caption bar with the minimization button	limited to main window
<code>WS_MAXIMIZEBOX</code>	caption bar with maximization button	limited to main window
<code>WS_EX_NONE</code>	no extended style	
<code>WS_EX_USEPRIVATECDC</code>	using private DC	limited to main window
<code>WS_EX_TOPMOST</code>	creating main window always on	limited to main window

	top-most	
<b>WS_EX_TOOLWINDOW</b>	creating ToolTip main window	limited to main window. ToolTip main window wouldn't own input focus, but can receive mouse information
<b>WS_EX_TRANSPARENT</b>	transparent window	limited to partial controls, such as the List Box, List View and Tree View.
<b>WS_EX_USEPARENTFONT</b>	using the font of the parent as the default font	
<b>WS_EX_USEPARENTCURSOR</b>	using the icon of the parent as the default icon	
<b>WS_EX_NOCLOSEBOX</b>	caption bar without close button	
<b>WS_EX_CTRLMAINWIN</b>	creating control which can be displayed out of main window	limited to specific controls.
<b>WS_EX_CLIPCHILDREN</b>	the region of the child will be clipped when paint the parent's client area.	

### 3.2.4 Destroying Main Window

**DestroyMainWindow** function can be used to destroy a main window. This function sends **MSG\_DESTROY** message to the window procedure, and terminates the destroy process when the message return a non-zero value.

Application generally calls this function to destroy a main window when receiving **MSG\_CLOSE** message in main window procedure, and then calls **PostQuitMessage** message to terminate the message loop, as shown below:

```
case MSG_CLOSE:
    // Destroy the main window
    DestroyMainWindow (hWnd);
    // Post a MSG_QUIT message
    PostQuitMessage (hWnd);
    return 0;
```

**DetroyMainWindow** destroys a main window, but does not destroy the message queue used by the main window and the window object itself. So, the application should use **MainWindowCleanup** to finally destroy the message queue used by the main window and the window object itself at the end of thread or process.

MiniGUI will call **DetroyMainWindow** to destroy all hosted windows when a main window is destroyed.

### 3.2.5 Dialog Box

A dialog box is a special main window, which is usually created by `DialogBoxIndirectParam` function in an application:

```
int GUIAPI DialogBoxIndirectParam (PDLGTEMPLATE pDlgTemplate,
                                   HWND hOwner, WNDPROC DlgProc, LPARAM lParam);
```

The dialog box created by this function is called as model dialog box. Users need to prepare dialog box template and window procedure function of dialog box for this function.

We will describe the basic programming technique for dialog box in Chapter 4 in this guide.

### 3.2.6 Control and Control Class

Each control of MiniGUI is an instance of a certain control class, and each control class has a corresponding control procedure, which is shared by all control instance of the same class.

The definition of control class in MiniGUI is as follows:

```
typedef struct _WNDCLASS
{
    /** the class name */
    char*   spClassName;

    /** internal field, operation type */
    DWORD   opMask;

    /** window style for all instances of this window class */
    DWORD   dwStyle;

    /** extended window style for all instances of this window class */
    DWORD   dwExStyle;

    /** cursor handle to all instances of this window class */
    HCURSOR hCursor;

    /** background color pixel value of all instances of this window class */
    int     iBkColor;

    /** window callback procedure of all instances of this window class */
    int     (*WinProc) (HWND, int, WPARAM, LPARAM);

    /** the private additional data associated with this window class */
    DWORD   dwAddData;
} WNDCLASS;
```



```
typedef WNDCLASS* PWNDCLASS;
```

The main elements of control class are as follow:

- Class name **spClassName**: class name different from other control classes.
- Window procedure function **winProc**: all control instances of this class use this window procedure function, which handles all the messages posted/sent to the control and define appearance, behavior, and features of the controls.
- Class style **dwStyle**: define the style such as the appearance and behavior, and all instances of this class have this normal style.
- Extension style **dwExStyle**: define the extension style of the window, and all instances of this class have this extension style.
- Class cursor **hCursor**: define the shape of the cursor of the control class.
- Background color **ibkColor**: define the pixel value of the background color of the control class.
- Class private additional data **dwAddData**: additional space reserved for the class by MiniGUI.

Functions related to control class operation in MiniGUI are as follow:

```
BOOL GUIAPI RegisterWindowClass (PWNDCLASS pWndClass) ;
```

This function registers a control class.

```
BOOL GUIAPI UnregisterWindowClass (const char *szClassName) ;
```

This function unregisters a control class.

```
const char* GUIAPI GetClassName (HWND hWnd) ;
```

This function retrieves the class name of a specific control.

```
BOOL GUIAPI GetWindowClassInfo (PWNDCLASS pWndClass) ;
```

This function retrieves the class information of a specific control class.

```
BOOL GUIAPI SetWindowClassInfo (const WNDCLASS *pWndClass) ;
```

This function sets the class information of a specific control class.

The following code illustrates how to use **WNDCLASS** structure, **RegisterWindowClass** function, and **UnregisterWindowClass** function to register a user-defined control class in application:

```
/* Define the name of control class */
#define MY_CTRL_NAME "mycontrol"

static int MyControlProc (HWND hwnd, int message, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;

    switch (message) {
        case MSG_PAINT:
            /* Only output "hello, world! - from my control" */
            hdc = BeginPaint (hwnd);
            TextOut (hdc, 0, 0, "Hello, world! - from my control");
            EndPaint (hwnd, hdc);
            return 0;
    }

    return DefaultControlProc (hwnd, message, wParam, lParam);
}

/* This function registers "mycontrol" control into MiniGUI */
static BOOL RegisterMyControl (void)
{
    WNDCLASS MyClass;

    MyClass.spClassName = MY_CTRL_NAME;
    MyClass.dwStyle      = 0;
    MyClass.hCursor      = GetSystemCursor (IDC_ARROW);
    MyClass.iBkColor      = COLOR_lightwhite;
    MyClass.WinProc      = MyControlProc;

    return RegisterWindowClass (&MyClass);
}

/* Unregister the control from system */
static void UnregisterMyControl (void)
{
    UnregisterWindowClass (MY_CTRL_NAME);
}
```

The control class created above only implements a task, namely displaying "Hello, world!" after creating a control instance. The general process of using the user-defined control class in a user's application is as follows:

```
/* Register the control class */
RegisterMyControl();

...
```

```
/* Creat an instance of the control class in a certain main window */  
hwnd = CreateWindow (MY_CTRL_NAME, "", WS_VISIBLE, IDC_STATIC, 0, 0, 200, 20, parent, 0);  
  
...  
  
/* Destroy the control and unregister the control class after using it */  
DestroyWindow (hwnd);  
UnregisterMyControl();
```

The window shown in Fig. 3.4 creates an instance of the user-defined control class described above, in which "Hello, world! - from my control." is displayed. Please refer to `mycontrol.c` file in the demo program package `mg-sample` of this guide to get the complete list of this program.



Fig. 3.4 Display "Hello, world!" using the user-defined control

We will describe in detail the foundation knowledge of control programming in Chapter 5; and discusses the advanced programming techniques related to controls in Chapter 6. We will introduce all predefined controls of MiniGUI in Part IV in this guide.

### 3.2.7 Input Method Window

Input method is the mechanism introduced by MiniGUI for supporting multi-byte char-sets such as Chinese, Korean, and Japanese etc. It is similar to the input method in windows system. Input method generally appears in a form of a window in the top-most, catches the key down message of the system and performs appropriate handling, and sends the translated characters to the current active window. MiniGUI internally provides the input

method window to implement GB2312 (Simplified Chinese) input method, and you can also write your user-defined input method.

MiniGUI-Processes provides the following function for application to use GB2312 input method:

```
HWND GBIMEWindowEx ( HWND hosting, int lx, int ty, int rx, int by, BOOL two_lines );
```

This function creates a GB2312 input method window. You must create a main window as the hosting window of IME window before calling this function.

The meanings of each argument of **GBIMEWindowEx** function is as follows:

- **hosting**: The hosting window of IME window, which can not be **HWND\_DESKTOP**
- **lx, ty, rx, by**: The size and position of IME window
- **two\_lines**: Determining whether IME window is divided into two lines

**GBIMEWindowEx** returns the handle of IME window.

MiniGUI-Threads defines an entrance function for GB2312 input method:

```
HWND GBIMEWindow (HWND hosting);
```

This function creates a GB2312 input method window, which can be dragged.

Besides GB2312 input method described above, MiniGUI provides the following more general input method interfaces:

```
int GUIAPI RegisterIMEWindow ( HWND hWnd );
```

This function registers the specified window **hWnd** as IME window of MiniGUI, then keyboard input will be sent to IME window first. Note that only one IME window can be registered.

```
int GUIAPI UnregisterIMEWindow ( HWND hWnd );
```

This function unregisters an IME window.

```
int GUIAPI SetIMEStatus ( int StatusCode, int Value );
```

This function sets the status of current IME window.

For GB2312 input method, `statusCode` can be one of the following values:

- **IS\_ENABLE**: Enable or disable the IME window
- **IS\_FULLCHAR**: Whether translate half character to full character?
- **IS\_FULLPUNC**: Whether translate half punctuate mark to full mark?
- **IS\_METHOD**: Which input method? For example, Internal code, Quanpin, 5-stroke, etc.

`value` is the status value.

```
int GUIAPI GetIMEStatus ( int StatusCode );
```

This function obtains the status of current IME window. `statusCode` specifies what to obtain, and `GetIMEStatus` returns the status value of the IME window.

Input method window is generally created by `mginit` program in MiniGUI-Processes. Application can create its own input method window, and then calls `RegisterIMEWindow` function to register the window to be input method window. After that, MiniGUI will send all the keystroke messages to the input method window, which in turn is handled by the input method window and forwarded to the target window or active client.

## 3.3 Message and Message Handling

### 3.3.1 Message

MiniGUI application communicates with the outside by receiving messages.

Messages are generated by the system or the application itself. The system generates messages for input events, and also generates messages for corresponding to applications. Applications can complete a certain task or communicate with other windows by sending messages. Generally speaking, MiniGUI is a message driven system, in which all the operations are performed around messages.

System sends messages to the window procedure of application. Window procedure has four arguments: window handle, message identifier and two 32-bit message parameters. Window handle determines the target window to which messages are sent, through which MiniGUI determines to which window procedure the messages are sent. Message identifier is an integer constant, which identifies the type of a message. If window procedure receives a message, it determines the type of the message and how to handle it according to the message identifier. Message parameters give further description of the message contents, meaning of which usually depends on the message itself. It can be an integer, a flag or a pointer to a data structure. For example, for mouse message, `lParam` usually includes the position information of the mouse, while `wParam` parameter includes the correspondence status information of SHIFT keys when the message happens. For other different message types, `wParam` and `lParam` also have clear definition. Applications usually need to check message parameters to determine how to handle the message.

It has been mentioned in Chapter 2 that a message is defined in MiniGUI as follows (`minigui/window.h`):

```
typedef struct _MSG
{
    HWND          hwnd;
    int           message;
    WPARAM        wParam;
    LPARAM        lParam;
    unsigned int   time;
#ifdef _LITE_VERSION
    void*         pAdd;
#endif
}MSG;

typedef MSG* PMSG;
```

Members of **MSG** message structure include the window that the message belonging to (**hwnd**), message identifier (**message**), **WPARAM** type parameter (**wParam**) of the message, **LPARAM** type parameter (**lParam**) of message and the time when the message happens.

### 3.3.2 Message Type

The predefined general messages of MiniGUI have the following types:

- System messages: include **MSG\_IDLE**, **MSG\_TIMER**, and **MSG\_FDEVENT** etc.
- Dialog box messages: include **MSG\_COMMAND**, **MSG\_INITDIALOG**, **MSG\_ISDIALOG**, **MSG\_SETTEXT**, **MSG\_GETTEXT**, and **MSG\_FONTCHANGED** etc.
- Window painting messages: include **MSG\_PAINT** and **MSG\_ERASEBKGD** etc.
- Window creating and destroying messages: include **MSG\_CREATE**, **MSG\_NCCREATE**, **MSG\_DESTROY**, and **MSG\_CLOSE** etc.
- Keyboard and mouse event messages: include **MSG\_KEYDOWN**, **MSG\_CHAR**, **MSG\_LBUTTONDOWN**, and **MSG\_MOUSEMOVE** etc.
- Keyboard/mouse post-handling messages: include **MSG\_SETCURSOR**, **MSG\_SETFOCUS**, **MSG\_KILLFOCUS**, **MSG\_MOUSEMOVEIN** etc., which is the window event message caused by mouse/keyboard messages.

You can define messages yourself, and define the meaning of **wParam** and **lParam**. To let users be able to define messages themselves, MiniGUI defines **MSG\_USER** macro, and application can define its own messages as follow:

```
#define MSG_MYMESSAGE1    (MSG_USER + 1)
#define MSG_MYMESSAGE2    (MSG_USER + 2)
```

You can use the user-defined message in your programs, and transfer data by using your customized messages.

### 3.3.3 Message Queue

MiniGUI has the following two methods for sending messages to a window procedure:

- Post the message to a first-in-first-out message queue, which is a memory region for storing messages in system with each message stored in a message structure;
- Or send the message directly to window procedure, i.e. call window procedure function directly through message sending function.

Messages posted into message queue are mainly the mouse/keyboard event messages from mouse and keyboard input, such as `MSG_LBUTTONDOWN`, `MSG_MOUSEMOVE`, `MSG_KEYDOWN`, and `MSG_CHAR` etc. Messages posted into message queue also include time message `MSG_TIMER`, paint message `MSG_PAINT` and exit message `MSG_QUIT` etc.

Why need message queue? We know that system displays multiple application windows simultaneously, and device driver generates mouse and keyboard messages continuously when the user moves the mouse or clicks the keyboard. These messages need to be sent to the corresponding application and window for handling. With message queue, system can manage various events and messages better, and the communication between system and applications becomes more convenient.

System posts messages to a message queue of the application by filling a `MSG` structure and then copying it to the message queue. Information in the `MSG` structure is as described above, including the target window handle of the message, the message identifier, two message parameters, and message time.

An application can get a message from its message queue through `GetMessage` function; this function fills a `MSG` message structure with information of the message. An application can also call `HavePendingMessage` function to check whether there exists any message in the message queue, which is not gotten out.

```
int GUIAPI GetMessage (PMSG pMsg, HWND hWnd);
BOOL GUIAPI HavePendingMessage (HWND hWnd);
```

Messages that don't queue up are sent to the window procedure of the target



window directly without through message queue. System generally completes some events that need to be handled immediately by sending messages that don't queue up, such as `MSG_ERASEBKGD` message.

### 3.3.4 Message Handling

An application must handle the messages that are posted to its message queue in time. Application generally handles the messages in the message queue through message loop in `MiniGUIMain` function.

A message loop is a loop, in which the program gets messages from the message queue continuously, then dispatches the message to a specified window through `DispatchMessage` function, namely, calls the window procedure of the specified window and passes the message parameters. A typical message loop is as follows:

```
MSG Msg;
HWND hMainWnd;
MAINWINCREATE CreateInfo;

InitCreateInfo (&CreateInfo);

hMainWnd = CreateMainWindow (&CreateInfo);
if (hMainWnd == HWND_INVALID)
    return -1;

while (GetMessage (&Msg, hMainWnd)) {
    TranslateMessage (&Msg);
    DispatchMessage (&Msg);
}
```

As illustrated above, an application starts a message loop after creating the main window. `GetMessage` function gets a message from the message queue to which the `hMainWnd` window belongs, and then calls `TranslateMessage` function to translate the keystroke messages of `MSG_KEYDOWN` and `MSG_KEYUP` into character message `MSG_CHAR`, and finally calls `DispatchMessage` function to dispatch the message to the specific window in the end.

`GetMessage` will not return until getting a message from the message queue, and generally return a non-zero value; it will return 0 if the gotten message is `MSG_QUIT`, thereby terminates the message loop. Terminating the message

loop is the first step to close an application, and the application generally quits message loop by calling **PostQuitMessage** function.

Under MiniGUI-Threads, we can use **HavePendingMessage** function when we need to return immediately to handle other works during waiting messages. For example:

```
do {
    /* It is time to read from master pty, and output. */
    ReadMasterPty (pConInfo);

    if (pConInfo->terminate)
        break;

    while (HavePendingMessage (hMainWnd)) {
        if (!GetMessage (&Msg, hMainWnd))
            break;
        DispatchMessage (&Msg);
    }
} while (TRUE);
```

When the program above cannot get any message or get a **MSG\_QUIT** message, it will return immediately and call **ReadMasterPty** function to read data from a certain file descriptor.

Under MiniGUI-Thread, each GUI thread, which creates window, has its own message queue; furthermore, all windows belonging to the same thread share the same message queue. So **GetMessage** function will get all messages of the windows belonging to the same thread with **hMainWnd** window. Whereas, there is only a message queue under MiniGUI-Processes, **GetMessage** gets all messages from this message queue, and ignores **hMainWnd** parameter. A message queue needs only a message loop, no matter how many windows the application has. **DispatchMessage** function can dispatch the message to its target window because the **MSG** message structure includes the target window handle of a message.

What **DispatchMessage** does is to get the window procedure of the target window of the message, and then call the window procedure function directly to handle the message.

The window procedure is a function of special type, which is used for receiving

and handling all the messages sent to the window. Each control has a window procedure, all controls belonging to the same control class share the same window procedure to handle messages.

Generally, window procedure must pass a message to system for default handling if the window procedure does not handle this message. The main window procedure generally calls `DefaultMainWinProc` (`PreDefMainWinProc` as `default`) to complete the default handling work for the message, and returns the return value of the function:

```
int PreDefMainWinProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam);
```

Most window procedures only handle messages of several types, and other messages are handled by system through `DefaultMainWinProc`.

The default message handling of dialog boxes is completed by `DefaultDialogProc` (`PreDefDialogProc` as `default`) function:

```
int PreDefDialogProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam);
```

The default message handling of a control is completed by `DefaultControlProc` (`PreDefControlProc` as `default`) function:

```
int PreDefControlProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam);
```

### 3.3.5 Sending and Posting Message

Posting (mailing) a message is to copy the message into message queue, and sending a message is to send the message to window procedure function.

Several important message handling functions in MiniGUI are listed below.

**PostMessage:** This function posts a message to the message queue of a specific window and then returns immediately. This form is called "posting message". If the posting message buffer of the message queue is full, the

function will return an error value. The window will not handle the message until **GetMessage** function gets the message. **PostMessage** is usually used for posting some non-key messages. For example, mouse and keyboard messages are posted by **PostMessage** function in MiniGUI.

**SendMessage**: Application usually informs the window procedure to complete a certain task immediately by sending a message. This function is different from **PostMessage** function in that it sends a message to the window procedure of a specific window and will not return until the window procedure handles the message. When it needs to know the handling result of a certain message, you should use this function to send the message, and then perform handling according to the return value of the function. In MiniGUI-Threads, if the thread to send messages is not the same as the thread to receive messages, the sender thread will be blocked and wait the handling result of another thread, and then continue to run; if the thread sender is the same as the receiver thread, the window procedure function of the window which receives message is called directly like **SendMessage** in MiniGUI-Processes.

**SendMessage**: Similar to **PostMessage** message, this function returns immediately without waiting the message to be handled. But it is different from **PostMessage** message in that the messages sent by this function will not be lost due to full of the buffer because system adopts a linked-list to maintain messages of this type. The messages sent by this function are called "notification message", which is generally used to send notification messages from a control to its parent window.

**PostQuitMessage**: This message will set a **QS\_QUIT** flag in the message queue. **GetMessage** will check the flag when getting a message from a specific message queue. If the flag is **QS\_QUIT**, **GetMessage** message will return zero; thus this return value can be used to terminate the message loop.

The following are some other message handling functions:

```
int GUIAPI BroadcastMessage ( int iMsg, WPARAM wParam, LPARAM lParam );
```

This function broadcasts a message to all main windows on the desktop.

```
int GUIAPI ThrowAwayMessages ( HWND pMainWnd );
```

This function removes all the messages in the message queue associated with a window, and returns the number of removed messages.

```
BOOL GUIAPI WaitMessage ( PMSG pMsg, HWND hMainWnd );
```

This function waits for the message in the message queue of the main window, and it will return as soon as there is a message in the queue. Unlike the **GetMessage** function, this function doesn't remove the message from the message queue.

### 3.3.6 Message Handling Function Specific to MiniGUI-Processes

MiniGUI defines some message handling functions specific to MiniGUI-Processes, which can be used to send messages from the MiniGUI-Processes server program to other client programs.

```
int GUIAPI Send2Client ( MSG * msg, int cli );
```

**Send2Client** function sends a message to a specified client. The function is defined for MiniGUI-Processes and can only be called by the server program **mginit**.

**msg** is a pointer to a message structure; **cli** can either be the identifier of the target client or one of the following values:

- **CLIENT\_ACTIVE**: The current active client on the topmost layer.
- **CLIENTS\_TOPMOST**: All clients in the topmost layer.
- **CLIENTS\_EXCEPT\_TOPMOST**: All clients except clients in the topmost layer.
- **CLIENTS\_ALL**: All clients.

Return values:

- **SOCKERR\_OK**: Read data successfully.
- **SOCKERR\_IO**: There are some I/O errors occurred.
- **SOCKERR\_CLOSED**: The peer has closed the socket.
- **SOCKERR\_INVARG**: You passed invalid arguments

```
BOOL GUIAPI Send2TopMostClients ( int iMsg, WPARAM wParam, LPARAM lParam );
```

**Send2TopMostClients** sends a message to all clients in the topmost layer. The function is defined for MiniGUI-Processes and can only be called by the server program **mginit**.

```
BOOL GUIAPI Send2ActiveWindow (const MG_Layer* layer,
                               int iMsg, WPARAM wParam, LPARAM lParam);
```

**Send2ActiveWindow**<sup>5</sup> function sends a message to the current active window in the specific layer. The function is defined for MiniGUI-Processes and can only be called by the server program **mginit**.

Generally speaking, the message sent from the server to a client will be sent to the virtual desktop of the client in the end, and handled by the virtual desktop window procedure, which is similar to that MiniGUI-Threads program receives the events from the keyboard and mouse.

MiniGUI-Processes also defines a special message - **MSG\_SRVNOTIFY**. The server can send this message and its parameters to the specified client, and the desktop of the client will broadcast the message to all main windows of the client after receiving the message.

### 3.4 Several Important Messages and Corresponding Handling

Several import messages need to be handled carefully in the life cycle of windows (include main windows and child windows). Concepts and typical handling of these messages will be described below.

---

<sup>5</sup> The old function **Send2ActiveClient** in MiniGUI V1.6.x is obsolete.

### 3.4.1 MSG\_NCCREATE

This message is sent to window procedure during MiniGUI creates a main window. The parameter `lParam` is the pointer to `pCreateInfo` structure passed to `CreateMainWindow`. You can modify some values in `pCreateInfo` structure during the message is being handled. It should be noted that the window object has not been created when the system sends the message to the window procedure, so the window device context can not be gotten by functions, such as `GetDC` etc., during the messages is being handled, and child window can not be created in `MSG_NCCREATE` message.

For the input method window, you must register the input method window during handling the message, for example:

```
case MSG_NCCREATE:
    if (hz_input_init())
        /* Register before show the window. */
        SendMessage (HWND_DESKTOP, MSG_IME_REGISTER, (WPARAM)hWnd, 0);
    else
        return -1;
    break;
```

### 3.4.2 MSG\_SIZECHANGING

This message is sent to the window procedure to determine the size of the window when the size of the window is changing or a window is being created. The parameter `wParam` includes the expected size of the window, while `lParam` is used to save result value. The default handling of MiniGUI is as follows:

```
case MSG_SIZECHANGING:
    memcpy ((PRECT)lParam, (PRECT)wParam, sizeof (RECT));
    return 0;
```

You can catch the message handling to let the window to be created locate in the specific position or has a fixed size, for example, the spin box control handle the message in order to have a fixed size:

```
case MSG_SIZECHANGING:
{
    const RECT* rcExpect = (const RECT*) wParam;
    RECT* rcResult = (RECT*) lParam;

    rcResult->left = rcExpect->left;
```

```
rcResult->top = rcExpect->top;
rcResult->right = rcExpect->left + _WIDTH;
rcResult->bottom = rcExpect->left + _HEIGHT;
return 0;
}
```

### 3.4.3 MSG\_SIZECHANGED and MSG\_CSIZECHANGED

This message is sent to the window procedure to determine the size of client region of the window when the size of a window is changed. The parameters of this message are similar to `MSG_SIZECHANGING`. `WParam` includes the size of the window, while `lParam` is a pointer to a `RECT` object used to save the size of client region of the window and has a default value. If handling of the message returns a non-zero value, the size value in the `lParam` will be used as the size of the client region; otherwise, the message handling will be ignored. For example, the message is handled to let the client region occupy the entire window rectangle in spin box control:

```
case MSG_SIZECHANGED
{
    RECT* rcClient = (RECT*) lParam;

    rcClient->right = rcClient->left + _WIDTH;
    rcClient->bottom = rcClient->top + _HEIGHT;
    return 0;
}
```

`MSG_CSIZECHANGED` message is a notification message sent to the window after the size of the client area of the window has changed. You can handle this message to reflect the new size of the client area. The parameters `wParam` and `lParam` of this message contain the new width and new height of the client area respectively.

### 3.4.4 MSG\_CREATE

This message is sent to the window procedure after the window is created successfully and added to the window management module of MiniGUI. At this time, applications can create child windows. If the message returns non-zero value, the newly created window will be destroyed.



### 3.4.5 MSG\_FONTCHANGING

The message is sent to the window procedure when the application calls **SetWindowFont** to change the default font of the window. In general, the application will pass this message to the default window procedure; but if the window does not allow the user to change the default font, the message can be caught and a non-zero value is returned. For example, the simple edit box of MiniGUI can only deal with equal width font, so the message can be handled as follows:

```
case MSG_FONTCHANGING:
    return -1;
```

**SetWindowFont** function will terminate handling and return after application handles the message and returns a non-zero value, that is to say, the default font of the window will not be changed.

### 3.4.6 MSG\_FONTCHANGED

This message will be sent to the window procedure when the application calls **SetWindowFont** and has changed the default font of the window. At this time, the window procedure should perform some handling to reflect the new font set. For example, the edit box of MiniGUI will deal with this message, and redraw the edit box ultimately:

```
case MSG_FONTCHANGED:
{
    sled = (PSLEDITDATA) GetWindowAdditionalData2 (hWnd);

    sled->startPos = 0;
    sled->editPos = 0;
    edtGetLineInfo (hWnd, sled);

    /* Recreate the caret appropriate for the new font */
    DestroyCaret (hWnd);
    CreateCaret (hWnd, NULL, 1, GetWindowFont (hWnd)->size);
    SetCaretPos (hWnd, sled->leftMargin, sled->topMargin);
    /* Invalidate the control to redraw the content */
    InvalidateRect (hWnd, NULL, TRUE);
    return 0;
}
```

### 3.4.7 MSG\_ERASEBKGND

This message is sent to the window procedure when the system needs to erase the window background. In general, when the application calls functions such as `InvalidateRect` or `UpdateWindow` etc. and pass `TRUE` to `bErase` argument, the system will send this message to inform the window to erase the background. The default window procedure will refresh the client region of the window with the background color. For some special windows, which usually refresh the entire client region in `MSG_PAINT` message, you can ignore the handling of the message in this case:

```
MSG_ERASEBKGND:
    return 0;
```

There are also some windows, which expect to fill the background with a bitmap, which can be done during handling `MSG_ERASEBKGND` message:

```
MSG_ERASEBKGND:
    HDC hdc = (HDC)wParam;
    const RECT* clip = (const RECT*) lParam;
    BOOL fGetDC = FALSE;
    RECT rcTemp;

    if (hdc == 0) {
        hdc = GetClientDC (hDlg);
        fGetDC = TRUE;
    }

    if (clip) {
        rcTemp = *clip;
        ScreenToClient (hDlg, &rcTemp.left, &rcTemp.top);
        ScreenToClient (hDlg, &rcTemp.right, &rcTemp.bottom);
        IncludeClipRect (hdc, &rcTemp);
    }

    /* Fill the background with a BITMAP object */
    FillBoxWithBitmap (hdc, 0, 0, 0, 0, &bmp_bkgnd);

    if (fGetDC)
        ReleaseDC (hdc);
    return 0;
```

Please refer to `bmppkgnd.c` program in the sample package of this guide for the complete implementation of filling the window background with a bitmap, the effect of the program is as shown in Fig. 3.5.



Fig 3.5 Using image as window background

### 3.4.8 MSG\_PAINT

This message is sent to the window procedure when the window needs to be repainted. MiniGUI determines whether to repaint by judging whether the window has invalid region. When a window is initially displayed, changes from hidden status to visible status, or from partially invisible to visible, or the application calls `InvalidateRect` function to invalidate a certain rectangle region, the window will have invalid region. At this time, MiniGUI will handle the invalid region and send `MSG_PAINT` message to the window procedure after finishing handling all the posted messages and notification messages. The typical handling of the message is as follows:

```
case MSG_PAINT:
{
    HDC hdc;

    hdc = BeginPaint (hWnd);

    /* Paint window with hdc */
    ...

    EndPaint (hWnd, hdc);
    return 0;
}
```

```
}
```

It should be noted that, application should return directly after handling the message, and should not pass this message to the default window procedure. Device contexts and graphics APIs will be described in detail in Part II of this guide.

### 3.4.9 MSG\_CLOSE

MiniGUI sends MSG\_CLOSE message to the window procedure when the user click the "Close" button on the window caption bar. The application should call `DestroyMainWindow` to destroy the main window during handling the message. If the window has the style of `WS_MINIMIZEBOX` and `WS_MAXIMIZEBOX`, the caption bar of the window should display "Minimization" and "Maximization" buttons. At present, MiniGUI has not realized handling of these buttons, but application can utilize these two styles to display other buttons, such as "Ok" and "Help" button, and then deal with `MSG_MINIMIZE` and `MSG_MAXIMIZE` messages in the window procedure.

### 3.4.10 MSG\_DESTROY

This message is sent to the window procedure when `DestroyMainWindow` or `DestroyWindow` is called, which is used to inform the system to destroy a window. You can destroy your private objects when receiving this message. If a non-zero value is returned for handling this message, the destroy process will be canceled.

When the application destroys a certain hosting main window, `DestroyMainWindow` will destroy the hosted main windows first. Certainly, when modal dialog box is used, logic of modal dialog box will ensure that there is no any hosted main window existing when destroying the hosting main window. But when using modeless dialog box or normal main window, application should deal with the hosted main windows according to the following rules, in order that the hosted main window and its related resource can be destroyed correctly when the user destroys a certain hosting main

window:

- The application should destroy the resource of the hosted main window, such as bitmaps and fonts etc., in the **MSG\_DESTROY** message:

```
case MSG_DESTROY:
    DestroyIcon (icon1);
    DestroyIcon (icon2);
    DestroyAllControls (hWnd);
    return 0;
```

- **DestroyMainWindow** and **MainWindowCleanup** functions should be called when the hosted main window responds to **MSG\_CLOSE** message:

```
case MSG_CLOSE:
    DestroyMainWindow (hWnd);
    MainWindowCleanup (hWnd);
    return 0;
```

- Handle **MSG\_CLOSE** message and call **DestroyMainWindow** function in the hosting main window. We also can release the resource of the hosting main window when handling **MSG\_DESTROY** message.

Thus, no matter the user closes the hosting main window or the hosted main window, the window itself and its related resource both can be released completely.

### 3.5 Common Window Operation Functions

MiniGUI provides some general window operation functions, which can be used for the main windows or controls, as shown in Table 3.2. In this guide, we use the term “window” to refer generally to main windows or controls. The functions for windows can be used for main windows or controls if no particular note.

Table 3.2 Common window operation functions

Function	Purpose	Note
<b>UpdateWindow</b>	Update the whole window immediately	
<b>ShowWindow</b>	Show or hide a window	
<b>IsWindowVisible</b>	Determine whether a window is visible	
<b>EnableWindow</b>	Enable or disable a window	
<b>IsWindowEnabled</b>	Determine whether a window is enabled	

<b>GetClientRect</b>	Get the client rectangle of a window.	
<b>GetWindowRect</b>	Get the window rectangle of a window	The result is in the parent coordinates
<b>GetWindowBkColor</b>	Get the background pixel value of a window	
<b>SetWindowBkColor</b>	Set the background pixel value of a window	
<b>GetWindowFont</b>	Get the default font of a window	
<b>SetWindowFont</b>	Set the default font of a window	
<b>GetWindowCursor</b>	Get the cursor of a window	
<b>SetWindowCursor</b>	Set the cursor of a window	
<b>GetWindowStyle</b>	Get the style of a window	
<b>GetWindowExStyle</b>	Get the extended style of a window	
<b>GetFocusChild</b>	Get the focus child of a window	
<b>SetFocusChild</b>	Set the focus child of a window	
<b>GetWindowCallbackProc</b>	Get the window procedure of a window	
<b>SetWindowCallbackProc</b>	Set the window procedure of a window	
<b>GetWindowAdditionalData</b>	Get the first additional data attached to a window	
<b>SetWindowAdditionalData</b>	Set the first additional data attached to a window	
<b>GetWindowAdditionalData2</b>	Get the second additional data attached to a window	Dialog box and controls has already used the second additional data internally, and reserve the first data for application.
<b>SetWindowAdditionalData2</b>	Set the second additional data attached to a window	
<b>GetWindowCaption</b>	Get the caption of a window	
<b>SetWindowCaption</b>	Set the caption of a window	
<b>InvalidateRect</b>	Makes a rectangle region in the client area of a window invalid	Would result in repainting of a window
<b>GetUpdateRect</b>	Retrieves the bounding box of the update region of a window	
<b>ClientToScreen</b>	Converts the client coordinates of a point to screen coordinates	
<b>ScreenToClient</b>	Converts the screen coordinates of a point to client coordinates	
<b>WindowToScreen</b>	Converts the window coordinates of a point to screen coordinates	
<b>ScreenToWindow</b>	Converts the screen coordinates of a point to window coordinates	
<b>IsMainWindow</b>	Determines whether a window is a main window	
<b>IsControl</b>	Determines whether a window is a control	
<b>IsDialog</b>	Determines whether a window is a dialog box	
<b>GetParent</b>	Retrieves the handle to a child window's parent window	The parent of main window always be <b>HWND_DESKTOP</b>
<b>GetMainWindowHandle</b>	Retrieves the handle to the main window contains a window	
<b>GetNextChild</b>	Retrieves the next control in a window	used for traveling all the children of a windows

<b>GetNextMainWindow</b>	Retrieves the next main window in the system	used for traveling all the main windows
<b>GetHosting</b>	Retrieves the hosting main window of a main window	
<b>GetFirstHosted</b>	Retrieves the first hosted main window of a main window	used for traveling all hosted main window a hosting main window
<b>GetNextHosted</b>	Retrieves the next hosted main window of a main window	
<b>GetActiveWindow</b>	Retrieves the main window handle to the active main window	
<b>SetActiveWindow</b>	Sets a main window to be the active main window	
<b>GetCapture</b>	Retrieves the handle to the window (if any) that has captured the mouse	Relevant contents will be described in Chapter 9
<b>SetCapture</b>	Sets the mouse capture to the specified window	
<b>ReleaseCapture</b>	Releases the mouse capture from a window and restores normal mouse input processing	
<b>MoveWindow</b>	Changes the position and size of a window	
<b>ScrollWindow</b>	Scrolls the content of a window's client area	





## 4 Foundation of Dialog Box Programming

The dialog box programming is a technique to construct a user interface quickly. In general, when we write a simple user interface, we can directly create all the needed child windows, i.e. the controls, by calling `CreateWindow` function. But when the user interface is complex, it is undesirable to call the `CreateWindow` function each time a control is created, and pass many complex arguments. One of the main reasons is that the program code is mixed with the data for creating controls, which is not good for maintaining. Therefore, a usual GUI system provides a mechanism, you can create corresponding main windows and controls according to the specified template by using this mechanism. MiniGUI also provides this method, and can create a modal or modeless dialog box by using a dialog box template.

This chapter first describes the different between main window and dialog box, and then describes the definition of the dialog box template, the dialog box callback function, and usage of some important messages. At last, this chapter illustrates the programming techniques of modal and modeless dialog boxes and the difference between them.

### 4.1 Main Window and Dialog Box

In MiniGUI, a dialog box is a special main window, which pays attention only to the interaction with the user: it displays the output information and more often receives the input from the user. The dialog box can be interpreted as a subclassed main window class. It is specially designed for the specialty of the dialog box (that is to interact with the user). For example, the user can use `TAB` key to travel the controls and can use `ENTER` key to indicate the default input.

### 4.2 Dialog Box Template

In MiniGUI, two structures are used to denote the dialog box template

(`minigui/window.h`), shown as follows:

```
typedef struct
{
    char*      class_name;           // control class
    DWORD      dwStyle;              // control style
    int        x, y, w, h;           // control position in dialog
    int        id;                   // control identifier
    const char* caption;             // control caption
    DWORD      dwAddData;            // additional data

    DWORD      dwExStyle;            // control extended style
} CTRLDATA;

typedef CTRLDATA* PCTRLDATA;

typedef struct
{
    DWORD      dwStyle;              // dialog box style
    DWORD      dwExStyle;            // dialog box extended style
    int        x, y, w, h;           // dialog box position
    const char* caption;             // dialog box caption
    HICON      hIcon;                // dialog box icon
    HMENU      hMenu;                // dialog box menu
    int        controlnr;             // number of controls
    PCTRLDATA  controls;             // pointer to control array
    DWORD      dwAddData;            // additional data, must be zero
} DLGTEMPLATE;

typedef DLGTEMPLATE* PDLGTEMPLATE;
```

The structure `CTRLDATA` is used to define controls, and `DLGTEMPLATE` is used to define the dialog box itself. You should first use `CTRLDATA` to define all the controls in the dialog box and express them with a structure array. The order of controls in the array is the switch order when the user presses `TAB` key. And then, you define the dialog box, specify the number of controls in the dialog box, and let the member `controls` in `DLGTEMPLATE` structure point to the array defining the controls, as shown in List 4.1.

List 4.1 Definition of dialog box template

```
static DLGTEMPLATE DlgInitProgress =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    120, 150, 400, 130,
    "VAM-CNC is initializing",
    0, 0,
    3, NULL,
    0
};

static CTRLDATA CtrlInitProgress [] =
{
    {
        "static",
        WS_VISIBLE | SS_SIMPLE,
        10, 10, 380, 16,
        IDC_PROMPTINFO,
        "Initializing...",
    }
};
```

```

    0
  },
  {
    "progressbar",
    WS_VISIBLE,
    10, 40, 380, 20,
    IDC_PROGRESS,
    NULL,
    0
  },
  {
    "button",
    WS_TABSTOP | WS_VISIBLE | BS_DEFPUSHBUTTON,
    170, 70, 60, 25,
    IDOK,
    "OK",
    0
  }
};

```

**[Note]** Data variables for defining the dialog box template in the program should be defined as static data, so that the data definition is validate only in its file, and it can be avoided causing potential compiling or linking errors due to the pollution of the name space.

### 4.3 Dialog Box Callback Function

You need define the callback function of the dialog box after defining its template data, and call `DialogBoxIndirectParam` function to create a dialog box, as shown in List 4.2. The running effect of the created dialog box is as shown in Fig. 4.1. Please refer to `dialogbox.c` of the sample program package `mg-samples` for this guide to get the complete source code of this program.

List 4.2 Defining the callback function of a dialog box and creating the dialog box

```

/* Define the dialog box callback function */
static int InitDialogBoxProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_INITDIALOG:
            return 1;

        case MSG_COMMAND:
            switch (wParam) {
                case IDOK:
                case IDCANCEL:
                    EndDialog (hDlg, wParam);
                    break;
            }
            break;
    }
    return DefaultDialogProc (hDlg, message, wParam, lParam);
}

static void InitDialogBox (HWND hWnd)
{

```

```
/* Associate the dialog with the control structure array */
DlgInitProgress.controls = CtrlInitProgress;

DialogBoxIndirectParam (&DlgInitProgress, hWnd, InitDialogBoxProc, 0L);
}
```

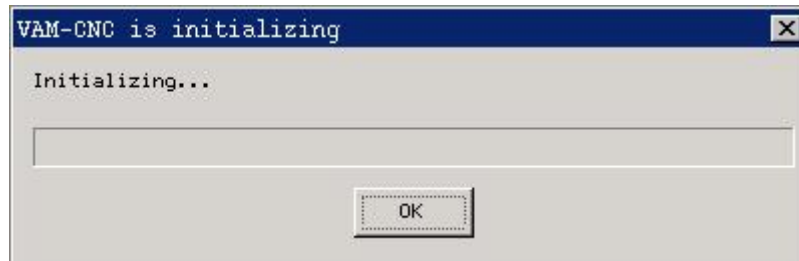


Fig. 4.1 The dialog box created by program in List 4.2

The prototypes of `DialogBoxIndirectParam` and related functions are listed as follow:

```
int GUIAPI DialogBoxIndirectParam (PDLGTEMPLATE pDlgTemplate,
                                   HWND hOwner, WNDPROC DlgProc, LPARAM lParam);
BOOL GUIAPI EndDialog (HWND hDlg, int endCode);
void GUIAPI DestroyAllControls (HWND hDlg);
```

When calling `DialogBoxIndirectParam`, you should specify the dialog box template (`pDlgTemplate`), the handle of the hosting main window of the dialog box (`hOwner`), the callback function address of the dialog box (`DlgProc`), and the parameter value (`lParam`) to be passed to the dialog box callback procedure. `EndDialog` is used to terminate the dialog box. `DestroyAllControls` is used to destroy all the child controls of the dialog box (This function can be used by a main window).

The dialog box callback function does not perform any substantial work in List 4.2, and calls `EndDialog` function to return directly when the user click "OK" button.

## 4.4 MSG\_INITDIALOG Message

The dialog box callback function is a special main window procedure function. When you define your own dialog box callback function, `MSG_INITDIALOG` message needs to be handled. This message is sent to the dialog box after

MiniGUI creates the dialog box and controls according to the dialog box template. **LParam** parameter of this message includes the value that passed to the dialog box callback function by the fourth argument of **DialogBoxIndirectParam** function. The user can use this value to initialize the dialog box or save it for future use. For example, the program in List 4.3 saves **LParam** parameter of **MSG\_INITDIALOG** message to the additional data of the dialog box window, so that this data can be readily gotten from the additional data of the dialog box window whenever needed.

List 4.3 Handling of MSG\_INITDIALOG message

```
static int DepInfoBoxProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    struct _DepInfo *info;

    switch(message) {
    case MSG_INITDIALOG:
    {
        /* Get lParam parameter passed to this dialog box, and assign it
         * to the first additional data associated with the dialog box
         * for future use. */
        info = (struct _DepInfo*)lParam;

        /* Use the data in the info structure to initialize the dialog box */
        .....

        SetWindowAdditionalData (hDlg, (DWORD)lParam);
        break;
    }

    case MSG_COMMAND:
    {
        /* Get the parameter from the first additional data
         * associated with the dialog box */
        info = (struct _DepInfo*) GetWindowAdditionalData (hDlg);

        switch(wParam) {
        case IDOK:
            /* Use the data in the info structure. */
            .....

            case IDCANCEL:
                EndDialog(hDlg,wParam);
                break;
        }
    }
    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}
```

Generally, the parameter passed to the dialog box callback function is a pointer to a structure. The structure includes some data for initializing the dialog box, and can also save the input data of the dialog box and pass it outside of the dialog box for use.

If the dialog box callback function returns a nonzero value when handling `MSG_INITDIALOG` message, MiniGUI will set the input focus to the first control with `WS_TABSTOP` style.

## 4.5 Modal and Modeless Dialog Box

At a word, the modal dialog box is a dialog box that the user cannot switch to other main windows once this dialog box has been displayed. The user can only use other main windows when closing the modal dialog box. In MiniGUI, the dialog box created by `DialogBoxIndirectParam` function is a modal dialog. In fact, the dialog box first creates a dialog box according to the template, then disables its hosting main window and creates controls in `MSG_CREATE` message of the main window, and then sends `MSG_INITDIALOG` message to the callback function, and lastly enters a new message loop until the program calls `EndDialog` function.

In fact, we also can create a normal main window using the dialog box template in MiniGUI, i.e. the modeless dialog box. Here we use `CreateMainWindowIndirect` function. The following is the prototypes of this function and its related functions (`minigui/window.h`):

```
HWND GUIAPI CreateMainWindowIndirect (PDLGTEMPLATE pDlgTemplate,  
                                     HWND hOwner, WNDPROC WndProc);  
BOOL GUIAPI DestroyMainWindowIndirect (HWND hMainWin);
```

The main window created by `CreateMainWindowIndirect` according to the dialog template is not different from a normal main window at all. However, `CreateMainWindowIndirect` is different from `DialogBoxIndirectParam` function as follows:

- After creating the main window with the data in the dialog box template, `CreateMainWindowIndirect` function will return immediately, while `DialogBoxIndirectParam` will enters a new message loop.

The program in List 4.4 creates a main window with the dialog box template in List 4.1

List 4.4 Creating a main window with a dialog box template

```

/* Define the window callback function */
static int InitWindowProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_COMMAND:
            switch (wParam) {
                case IDOK:
                case IDCANCEL:
                    DestroyMainWindowIndirect (hWnd);
                    break;
            }
            break;
    }

    return DefaultWindowProc (hDlg, message, wParam, lParam);
}

...

{
    HWND hwnd;
    MSG Msg;

    /* Associate the dialog box template with control array */
    DlgInitProgress.controls = CtrlInitProgress;

    /* Create the main window */
    hwnd = CreateMainWindowIndirect (&DlgInitProgress, HWND_DESKTOP, InitWindowProc);

    if (hwnd == HWND_INVALID)
        return -1;

    while (GetMessage (&Msg, hwnd)) {
        TranslateMessage (&Msg);
        DispatchMessage (&Msg);
    }
}

```

The program in List 4.4 will create a main window, which is completely the same with the dialog box in Fig. 4.1.

## 4.6 Control Styles and Operations Relevant to Dialog Box

Some general window styles are only valid for child controls in the dialog box; Table 4.1 summarizes these styles. The default window procedure of the dialog box will handle control with these styles.

Table 4.1 Some styles used only for the controls in a dialog box

Style Identifier	Purpose	Comment
WS_GROUP	Controls with this style will be the leader of the same group.	Controls from this control to the control before next control with WS_GROUP style, or to the control before the control with different class, belong to the same group
WS_TABSTOP	Have TAB-stop function	Indicates the user can set the input focus

		to the control by using Tab key
--	--	---------------------------------

MiniGUI provides some operation function for the dialog box, summarized as in Table 4.2. It should be noted that, these functions are not limited to be used in dialog boxes although their name having prefix of "Dlg". For example, `GetDlgItemText` function can be used to get the text of the child controls if only the handle of parent window and the identifier of child controls are known.

Table 4.2 Operation functions of the dialog box

Function	Purpose	Comment
<b>DestroyAllControls</b>	Destroys all controls in a window	
<b>GetDlgCtrlID</b>	Gets the integer identifier of a control	
<b>GetDlgItem</b>	Retrieves the handle to a control in a dialog box	
<b>GetDlgItemInt</b>	Translates the text of a control in a dialog box into an integer value	
<b>SetDlgItemInt</b>	Sets the text of a control in a dialog box to the string representation of a specified integer value	
<b>GetDlgItemText</b>	Retrieves the title or text associated with a control in a dialog box	Function is same as <code>GetWindowText</code>
<b>GetDlgItemText2</b>	Retrieves the title or text associated with a control in a dialog box	Automatically allocate memory according to the text length, application is responsible for releasing these memory.
<b>SetDlgItemText</b>	Sets the title or text of a control in a dialog box	Function is same as <code>SetWindowText</code>
<b>GetNextDlgGroupItem</b>	Retrieves the handle to the first control in a group of controls that precedes (or follows) the specified control in a dialog box	Used for traveling the same group controls, please refer to <code>WS_GROUP</code> style
<b>GetNextDlgTabItem</b>	Retrieves the handle to the first control that has the <code>WS_TABSTOP</code> style that precedes (or follows) the specified control	Used for TAB key traveling controls, please refer to <code>WS_TABSTOP</code> style
<b>SendDlgItemMessage</b>	Sends a message to the specified control in a dialog box	Function is same as <code>SendMessage</code>
<b>CheckDlgButton</b>	Changes the check status of a button control	
<b>CheckRadioButton</b>	Adds a check mark to (checks) a specified radio button in a group	



	and removes a check mark from (clears) all other radio buttons in the group	
<b>IsDlgButtonChecked</b>	Determines whether a button control has a check mark next to it or whether a three-state button control is grayed, checked, or neither	
<b>GetDlgDefPushButton</b>	Gets the default push button control in a window	



## 5 Foundation of Control Programming

There are predefined controls in a relatively complex GUI system, and they are the main elements for the human-machine interaction. This chapter will illustrate what is the control and control class, and briefly introduce the predefined control classes in MiniGUI.

### 5.1 Control and Control Class

The concept of controls (or widgets) is well known. The control can be interpreted as the child window in the main window. The behaviors of these child windows, as well as that of the main window, both can accept the exterior input such as the keyboard and the mouse, and can output in their own regions - but all the actions are restricted within the main window. MiniGUI also supports child windows, and can create nested child windows in the child window. We refer to all the child windows in MiniGUI as controls.

In Window or X Window, the system will predefine some control classes. When you use a certain control class to create controls, all the controls belonging to this control class will have the same behaviors and appearances. With this technique, the consistent human-machine interaction interface can be ensured, and the program can construct the graphics user interface as building toy bricks. MiniGUI uses the concept of control and control class, and can conveniently overload the existed control to make it have some special effects. For example, when an edit box which only allows digit to input in, you can realize it by overloading the existed EDIT control class, instead of developing a new control class.

If you have the experience of developing a Windows application, you should remember that before creating a new window, you must assure that the corresponding window class of this new window has existed in the system. In Windows operating system, each window created by the program corresponds to a certain window class. This concept is similar to the relationship between

the class and the object in the object-oriented programming. Referring to the terms of object oriented, each window in Windows is an instance of a certain window class. Similar concepts are present in the X Window programming; for example, each widget created is actually an instance of a certain widget class.

Thus, if a program needs to create a window, it should first be sure to select the correct window class, because each window class determines the appearances and behaviors of the corresponding window instances. Herein the presentation means the appearance of the window, such as the width of the window border, whether or not the caption bar exists, etc. The behavior means the response of the window to the user input. Each GUI system will predefine some window classes. Examples of the common window classes include the button, list box, scroll bar, edit box, etc. If the window to be created by the program is very special, you can first register your own window class, and then create an instance of this window class. Thus the code reusability is improved substantially.

In MiniGUI, the main window is generally considered as a special window. Since the reusability of the main window code is usually very low, it will cause additional unnecessary memory space if we register a window class for each main window in a usual way. So we do not provide window class support in the main window, but all the child windows, i.e. controls, in the main window support the concept of the window class (control class). MiniGUI provides the predefined control classes commonly used, including button (including radio button, check box), static control, list box, progress bar, track bar, edit box, etc. A program can customize its own control class, register the control class and then create the corresponding control instances. Table 5.1 lists the predefined control classes and the corresponding class names in MiniGUI.

Table 5.1 Control class and corresponding class name predefined in MiniGUI

Control Class	Class Name	C Macro for Control Name	Note
Static Frame	"static"	CTRL_STATIC	
Button	"button"	CTRL_BUTTON	
Single-Line Edit Box	"sedit"	CTRL_SLEDIT	Can deal with variable-width font, and support any character set.
Multi-Line Edit Box	"mledit"	CTRL_MLEDIT	

Text Edit Box	"textedit"	CTRL_TEXTEDIT	
List Box	"listbox"	CTRL_LISTBOX	
Progress Bar	"progressbar"	CTRL_PROGESSBAR	
Trackbar	"trackbar"	CTRL_TRACKBAR	
Combo Box	"combobox"	CTRL_COMBOBOX	
New Toolbar	"newtoolbar"	CTRL_NEWTOOLBAR	
Menu Button	"menubutton"	CTRL_MENUBUTTON	
Property Sheet	"propsheet"	CTRL_PROPSHEET	
Scrollable Window	"ScrollWnd"	CTRL_SCROLLWND	
Scrollable View	"ScrollView"	CTRL_SCROLLVIEW	
Tree View	"treeview"	CTRL_TREEVIEW	Include in mgext library, namely MiniGUI extension library.
List View	"listview"	CTRL_LISTVIEW	
Month Calendar	"MonthCalendar"	CTRL_MONTHCALENDAR	
SpinBox	"SpinBox"	CTRL_SPINBOX	
Cool Bar	"CoolBar"	CTRL_COOLBAR	
Icon View	"IconView"	CTRL_ICONVIEW	
Grid View	"gridview"	CTRL_GRIDVIEW	
Animation	"Animation"	CTRL_ANIMATION	

## 5.2 Creating Control Instance by Using Predefined Control Class

In MiniGUI, by calling the `CreateWindow` function (the `CreateWindow` is in fact a macro of the `CreateWindowEx` function) can create an instance of a certain control class. The control class can be either the predefined MiniGUI control classes in Table 5.1, or a user-defined control class. Following are the prototypes of several functions related to `CreateWindow` function

(`minigui/window.h`):

```

HWND GUIAPI CreateWindowEx (const char* spClassName, const char* spCaption,
                           DWORD dwStyle, DWORD dwExStyle, int id,
                           int x, int y, int w, int h, HWND hParentWnd, DWORD dwAddData);
BOOL GUIAPI DestroyWindow (HWND hWnd);

#define CreateWindow(class_name, caption, style, id, x, y, w, h, parent, add_data) \
  CreateWindowEx(class_name, caption, style, 0, id, x, y, w, h, parent, add_data)

```

The `CreateWindow` function creates a child window, i.e. a control. It specifies the control class (`class_name`), the control caption (`caption`), the control style (`style`), the control identifier (`id`), and the initial position and size of the control (`x`, `y`, `w`, `h`). This function also specifies the parent window (`parent`) of the child window. The parameter `add_data` is used to pass a special data to the control, and the data structures pointed to by this pointer are different for different control classes.

The affection of the `CreateWindowEx` function is same as `CreateWindow` function; however, the `CreateWindowEx` can be used to specify the extended style (`dwExStyle`) of the control.

The `DestroyWindow` function is used to destroy the control or the child window created by above two functions.

The program in List 5.1 creates several types of controls by using the predefined control classes: static control, button and single-line edit box. Among them `hStaticWnd1` is the static control created in the main window `hWnd`; `hButton1`, `hButton2`, `hEdit1`, and `hStaticWnd2` are several controls created in `hStaticWnd1`, and existing as the child controls of `hStaticWnd1`; While `hEdit2` is the child control of `hStaticWnd2`, and is the grandchild control of `hStaticWnd1`.

List 5.1 Creating controls using the predefined control classes

```
#define IDC_STATIC1      100
#define IDC_STATIC2      150
#define IDC_BUTTON1      110
#define IDC_BUTTON2      120
#define IDC_EDIT1        130
#define IDC_EDIT2        140

/* Create a static control */
hStaticWnd1 = CreateWindow (CTRL_STATIC,
    "This is a static control",
    WS_CHILD | SS_NOTIFY | SS_SIMPLE | WS_VISIBLE | WS_BORDER,
    IDC_STATIC1,
    10, 10, 180, 300, hWnd, 0);

/* Create two button controls in hStaticWnd1 */
hButton1 = CreateWindow (CTRL_BUTTON,
    "Button1",
    WS_CHILD | BS_PUSHBUTTON | WS_VISIBLE,
    IDC_BUTTON1,
    20, 20, 80, 20, hStaticWnd1, 0);
hButton2 = CreateWindow (CTRL_BUTTON,
    "Button2",
    WS_CHILD | BS_PUSHBUTTON | WS_VISIBLE,
    IDC_BUTTON2,
    20, 50, 80, 20, hStaticWnd1, 0);

/* Create one edit control in hStaticWnd1 */
hEdit1 = CreateWindow (CTRL_EDIT,
    "Edit Box 1",
    WS_CHILD | WS_VISIBLE | WS_BORDER,
    IDC_EDIT1,
    20, 80, 100, 24, hStaticWnd1, 0);

/* Create a static control in hStaticWnd1 */
hStaticWnd2 = CreateWindow (CTRL_STATIC,
    "This is child static control",
    WS_CHILD | SS_NOTIFY | SS_SIMPLE | WS_VISIBLE | WS_BORDER,
    IDC_STATIC1,
```

```

        20, 110, 100, 50, hStaticWnd1, 0):

/* Create an edit box hEdit2 in hStaticWnd2,
 * thus hEdit2 is the grandchild window of hStaticWnd1 */
hEdit2  = CreateWindow (CTRL_EDIT,
        "Edit Box 2",
        WS_CHILD | WS_VISIBLE | WS_BORDER,
        IDC_EDIT2,
        0, 20, 100, 24, hStaticWnd2, 0);

```

### 5.3 Topics Involved in Control Programming

In control programming, besides creating and destroying of the control, following subjects are usually involved:

- The control has its own window style definition. You need to specify the needed style when creating the control. Different styles will result in different appearance and behavior.
- Getting or setting the status and contents and so on of the control. In general, sending some general or special messages to the control can complete this. In addition, the general functions for windows are usually suitable for the controls, such as **ShowWindow**, **MoveWindow**, **EnableWindow**, **SetWindowFont**, etc.
- Informing the parent window by notification messages when a certain event happens in the control. Notification messages are generally sent by **MSG\_COMMAND** messages. **WParam** parameter of this message consists of the window identifier and the notification code, and **lParam** includes the handle of the control sending the notification messages. For example, when users change the contents of an edit box, the edit box will send **EN\_CHANGE** notification message to the parent window. If the window procedure of the parent window needs to know this change, the notification message should be handled in the window procedure of the parent window as follows:

```

switch (message) {
    case MSG_COMMAND:
    {
        int id = LOWORD(wParam);
        int nc = HIWORD(wParam);
        if (id == ID_MYEDIT && nc == EN_CHANGE) {
            /* The user has changed the content of ID_MYEDIT edit box
             * of the child window, and further handling is being done now. */
        }
    }
    break;
}

```

- MiniGUI V1.2.6 introduces **SetNotificationCallback** function for the notification message handling of control. This function can set a callback function of the notification message for a control. When the control has a notification message, it will call the callback function, instead of sending the notification message to the parent window. New applications should use this function as far as possible to handle notification messages of a control to get a good program structure. All the sample programs for this guide use this interface to handle notification messages of a control.

The function in List 5.2 creates a simple dialog box by using the predefined control classes. When the user inputs data in unit of millimeter (mm) into the edit box, the system will display the corresponding data in unit of inch in the static control below the edit box, and return the data input by the user to the function calling the dialog box when the user select "OK" button.

List 5.2 Realizing a simple input dialog box using the predefined controls

```
#include <stdio.h>
#include <stdlib.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

/* Define dialog box template */
static DLGTEMPLATE DlgBoxInputLen =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    120, 150, 400, 160,
    "Please input the length",
    0, 0,
    4, NULL,
    0
};

#define IDC_SIZE_MM    100
#define IDC_SIZE_INCH  110
/*
 * The dialog includes four controls in total, which are used to
 * display prompt information, input value, display the
 * transformed length value, and show a "OK" button to close the program
 */
static CTRLDATA CtrlInputLen [] =
{
    {4
        CTRL_STATIC,
        WS_VISIBLE | SS_SIMPLE,
        10, 10, 380, 18,
        IDC_STATIC,
        "Please input the length (Unit: mm)",
        0
    },
}
```



```

{
    CTRL_EDIT,
    WS_VISIBLE | WS_TABSTOP | WS_BORDER,
    10, 40, 380, 24,
    IDC_SIZE_MM,
    NULL,
    0
},
{
    CTRL_STATIC,
    WS_VISIBLE | SS_SIMPLE,
    10, 70, 380, 18,
    IDC_SIZE_INCH,
    "Equivalent to 0.00 inches",
    0
},
{
    CTRL_BUTTON,
    WS_TABSTOP | WS_VISIBLE | BS_DEFPUSHBUTTON,
    170, 100, 60, 25,
    IDOK,
    "OK",
    0
}
};
/* This is the notification callback function of the input box. */
static void my_notif_proc (HWND hwnd, int id, int nc, DWORD add_data)
{
    /* When the value in the input box is changed,
     * get the value, transform it into inch, and display it in the inch box
     */
    if (id == IDC_SIZE_MM && nc == EN_CHANGE) {
        char buff [60];
        double len;
        GetWindowText (hwnd, buff, 32);
        len = atof (buff);
        len = len / 25.4;

        sprintf (buff, "Equivalent to %.5f inches ", len);
        SetDlgItemText (GetParent (hwnd), IDC_SIZE_INCH, buff);
    }
}

/* The dialog box callback function */
static int InputLenDialogBoxProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_INITDIALOG:
            /*
             * Save the pointer passed by the last parameter of DialogBoxIndirectParam
             * in the form of window additional data for future use.
             */
            SetWindowAdditionalData (hDlg, lParam);

            /* Set the notification callback function for the edit box.
             */
            SetNotificationCallback (GetDlgItem (hDlg, IDC_SIZE_MM), my_notif_proc);
            return 1;

        case MSG_COMMAND:
            switch (wParam) {
                case IDOK:
                {
                    char buff [40];
                    /* Get data from the input box, and save it in the pointer passed in.
                     */
                    double* length = (double*) GetWindowAdditionalData (hDlg);
                    GetWindowText (GetDlgItem (hDlg, IDC_SIZE_MM), buff, 32);
                    *length = atof (buff);
                }
                case IDCANCEL:
                    EndDialog (hDlg, wParam);
                    break;
            }
            break;
    }
}

```

```

    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}

static void InputLenDialogBox (HWND hWnd, double* length)
{
    DlgBoxInputLen.controls = CtrlInputLen;

    DialogBoxIndirectParam (&DlgBoxInputLen, hWnd, InputLenDialogBoxProc, (LPARAM)length
);
}

int MiniGUIMain (int argc, const char* argv[])
{
    double length;

#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER , "input" , 0 , 0);
#endif

    InputLenDialogBox (HWND_DESKTOP, &length);

    /* Print the value input by the user in the dialog box to the terminal
    */
    printf ("The length is %.5f mm.\n", length);

    return 0;
}

#ifdef _LITE_VERSION
#include <minigui/dti.c>
#endif

```

The running effect of the program in List 5.2 is shown in Fig. 5.1. Please refer to the `input.c` file of the sample program package for this guide to get the complete source code of the program.

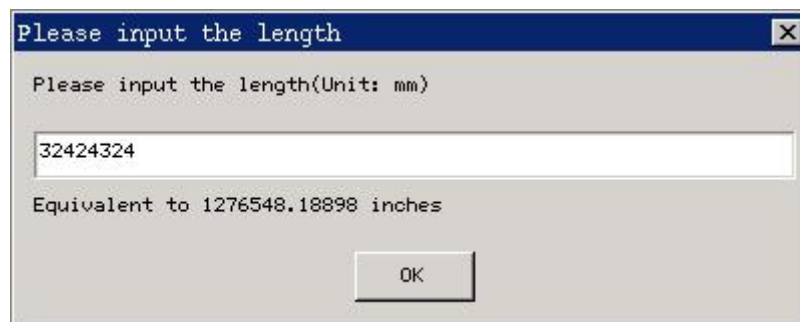


Fig. 5.1 A simple input dialog box

In the Part IV, we will introduce the predefined controls of MiniGUI. We will describe all the predefined controls in three aspects: the purpose and usage of the control, the styles, messages, and notification codes. We will also give the sample code of the controls.

## 5.4 Operations Specific to Control

MiniGUI provides some operation functions specific to control, as show in Table 5.2.

Table 5.2 Operation functions specific to control

Function	Purpose	Comment
<b>GetNotificationCallback</b>	Gets the notification callback procedure of a control	Appear in MiniGUI version 1.2.6
<b>SetNotificationCallback</b>	Sets the notification callback procedure of a control	
<b>NotifyParentEx</b>	Sends a notification message to the parent	



## 6 Advanced Programming of Control

### 6.1 User-Defined Control

You can also register your own control class by calling `RegisterWindowClass` function, and create an instance of your control class. If the program will not use a certain user-defined control class any more, it should use `UnregisterWindowClass` function to unregister this user-defined control class. Please refer to “control class” of section 3.2.4 of this guide for the usage of the above two functions.

### 6.2 Subclassing of Control

Using the framework of a control class and control instance can not only improve the code reusability, but also extend the existing control class conveniently. For example, when you need to create an edit box which only allows inputting digits, you can realize by overriding the existing edit box control class rather than write a new control class. In MiniGUI, this technique is called subclassing or window derived. Methods of subclassing have three types:

- The first one is to perform subclassing on the created control instance, and the result of subclassing only affects this control instance.
- The second one is to perform subclassing on a certain control class, which will affect all the control instances created later of this control class.
- The last one is to register a subclassing control class based on a certain control class, which will not affect the original control class. In Windows, this technique is also called super-subclassing.

In MiniGUI, the subclassing of a control is actually implemented by replacing the existing window procedure. The codes in List 6.1 create two subclassing edit boxes by subclassing: one only allows inputting digits, and the other only allows inputting letters:

## List 6.1 Subclassing of control

```
#define IDC_CTRL1      100
#define IDC_CTRL2      110
#define IDC_CTRL3      120
#define IDC_CTRL4      130

#define MY_ES_DIGIT_ONLY    0x0001
#define MY_ES_ALPHA_ONLY    0x0002

static WNDPROC old_edit_proc;

static int RestrictedEditBox (HWND hwnd, int message, WPARAM wParam, LPARAM lParam)
{
    if (message == MSG_CHAR) {
        DWORD my_style = GetWindowAdditionalData (hwnd);
        /* Determine the key-pressed type being shielded */
        if ((my_style & MY_ES_DIGIT_ONLY) && (wParam < '0' || wParam > '9'))
            return 0;
        else if (my_style & MY_ES_ALPHA_ONLY)
            if (!((wParam >= 'A' && wParam <= 'Z') || (wParam >= 'a' && wParam <= 'z')))
                /* Receive the key-pressed message being shielded, and returns directly */
                return 0;
    }
    /* Handle the other messages by the old window procedure */
    return (*old_edit_proc) (hwnd, message, wParam, lParam);
}

static int ControlTestWinProc (HWND hwnd, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_CREATE:
        {
            HWND hWnd1, hWnd2, hWnd3;
            CreateWindow (CTRL_STATIC, "Digit-only box:", WS_CHILD|WS_VISIBLE|SS_RIGHT, 0,
                10, 10, 180, 24, hwnd, 0);
            hWnd1 = CreateWindow (CTRL_EDIT, "", WS_CHILD|WS_VISIBLE|WS_BORDER, IDC_CTRL1,
                200, 10, 180, 24, hwnd, MY_ES_DIGIT_ONLY);
            CreateWindow (CTRL_STATIC, "Alpha-only box:", WS_CHILD|WS_VISIBLE|SS_RIGHT, 0,
                10, 40, 180, 24, hwnd, 0);
            hWnd2 = CreateWindow (CTRL_EDIT, "", WS_CHILD|WS_BORDER|WS_VISIBLE, IDC_CTRL2,
                200, 40, 180, 24, hwnd, MY_ES_ALPHA_ONLY);
            CreateWindow (CTRL_STATIC, "Normal edit box:", WS_CHILD|WS_VISIBLE|SS_RIGHT, 0,
                10, 70, 180, 24, hwnd, 0);
            hWnd3 = CreateWindow (CTRL_EDIT, "", WS_CHILD|WS_BORDER|WS_VISIBLE, IDC_CTRL2,
                200, 70, 180, 24, hwnd, MY_ES_ALPHA_ONLY);
            CreateWindow ("button", "Close", WS_CHILD|BS_PUSHBUTTON|WS_VISIBLE, IDC_CTRL4,
                100, 100, 60, 24, hwnd, 0);
            /* Replace the window procedure of the edit box with the
             * user-defined window procedure, and save the old window procedure. */
            old_edit_proc = SetWindowCallbackProc (hWnd1, RestrictedEditBox);
            SetWindowCallbackProc (hWnd2, RestrictedEditBox);
            break;
        }
    }
    return DefaultMainWinProc (hwnd, message, wParam, lParam);
}
```

## 6.3 Combined Use of Controls

We can also combine two different controls together to achieve a certain special effect. In fact, the predefined control class of the combo box is a typical one of combining controls. When we combine different controls, we can encapsulate and register the combined control to be a new control class, and

can also use it directly without encapsulation.

To illustrate the method to combine controls more clearly, we can assume that we want to implement a time editor. This time editor displays the time in form of "08:05:30", and we need further add a method to edit the time neatly according the user's requirement. To meet this requirement, we combine the edit box and the spin box together which implement the following functions, respectively:

- The edit box displays the time in form of "HH:MM:SS".
- When the input focus is in the edit box, the user can not edit the time directly, but must control the time value where the caret is with the arrow keys and **PageDown** and **PageUp** keys. So we must subclass this edit box to catch the key-pressed in it and perform the appropriate handling.
- Place a spin box beside the edit box. The user can adjust the time element where the caret is to increase or decrease by clicking the spin box. To achieve this goal, we can use the function of the spin box, and set the handle of the target window to be the edit box.

Thus, the time editor can work normally. Partial codes of this program are listed in List 6.2, and please refer to `timeeditor.c` file of the sample program package of this guide for the complete source code. Fig 6.1 shows the running effect of the `timeeditor`.

List 6.2 Time Editor

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>
#include <minigui/mgext.h>

#define IDC_EDIT 100
#define IDC_SPINBOX 110

static PLOGFONT timefont;
static WNDPROC old_edit_proc;
```

```
static void on_down_up (HWND hwnd, int offset)
{
    char time [10];
    int caretpos;
    int hour, minute, second;

    GetWindowText (hwnd, time, 8);
    caretpos = SendMessage (hwnd, EM_GETCARETPOS, 0, 0);

    hour = atoi (time);
    minute = atoi (time + 3);
    second = atoi (time + 6);

    if (caretpos > 5) { /* change second */
        second += offset;
        if (second < 0)
            second = 59;
        if (second > 59)
            second = 0;
    }
    else if (caretpos > 2) { /* change minute */
        minute += offset;
        if (minute < 0)
            minute = 59;
        if (minute > 59)
            minute = 0;
    }
    else { /* change hour */
        hour += offset;
        if (hour < 0)
            hour = 23;
        if (hour > 23)
            hour = 0;
    }

    sprintf (time, "%02d:%02d:%02d", hour, minute, second);
    SetWindowText (hwnd, time);

    SendMessage (hwnd, EM_SETCARETPOS, 0, caretpos);
}

static int TimeEditBox (HWND hwnd, int message, WPARAM wParam, LPARAM lParam)
{
    if (message == MSG_KEYDOWN) {
        switch (wParam) {
            case SCANCODE_CURSORBLOCKUP:
                on_down_up (hwnd, 1);
                return 0;
            case SCANCODE_CURSORBLOCKDOWN:
                on_down_up (hwnd, -1);
                return 0;
            case SCANCODE_PAGEUP:
                on_down_up (hwnd, 10);
                return 0;
            case SCANCODE_PAGEDOWN:
                on_down_up (hwnd, -10);
                return 0;

            case SCANCODE_CURSORBLOCKLEFT:
            case SCANCODE_CURSORBLOCKRIGHT:
                break;
            default:
                return 0;
        }
    }

    if (message == MSG_KEYUP || message == MSG_CHAR)
        return 0;

    return (*old_edit_proc) (hwnd, message, wParam, lParam);
}

static int TimeEditorWinProc (HWND hwnd, int message, WPARAM wParam, LPARAM lParam)
{

```



```

switch (message) {
case MSG_CREATE:
{
    HWND hwnd;
    HDC hdc;
    HWND timeedit, spin;
    SIZE size;

    hwnd = CreateWindow (CTRL_STATIC,
        "This is a time editor.\n\n"
        "Pressing <Down-Arrow>, <Up-Arrow>, <PgDn>, and <PgUp> keys"
        " when the box has input focus will change the time.\n\n"
        "You can also change the time by clicking the SpinBox.\n",
        WS_CHILD | WS_VISIBLE | SS_LEFT,
        IDC_STATIC,
        10, 10, 220, 200, hwnd, 0);

    timefont = CreateLogFont (NULL, "Arial", "ISO8859-1",
        FONT_WEIGHT_BOOK, FONT_SLANT_ROMAN, FONT_SETWIDTH_NORMAL,
        FONT_SPACING_CHARCELL, FONT_UNDERLINE_NONE, FONT_STRUCKOUT_NONE,
        30, 0);

    hdc = GetClientDC (hwnd);
    SelectFont (hdc, timefont);
    GetTextExtent (hdc, "00:00:00", -1, &size);
    ReleaseDC (hdc);

    timeedit = CreateWindow (CTRL_SLEDIT,
        "00:00:00",
        WS_CHILD | WS_VISIBLE | ES_BASELINE,
        IDC_EDIT,
        40, 220, size.cx + 4, size.cy + 4, hwnd, 0);

    SetWindowFont (timeedit, timefont);

    old_edit_proc = SetWindowCallbackProc (timeedit, TimeEditBox);

    spin = CreateWindow (CTRL_SPINBOX,
        "",
        WS_CHILD | WS_VISIBLE,
        IDC_SPINBOX,
        40 + size.cx + 6, 220 + (size.cy - 14) / 2, 0, 0, hwnd, 0);

    SendMessage (spin, SPM_SETTARGET, 0, timeedit);
    break;
}

case MSG_DESTROY:
    DestroyAllControls (hwnd);
    DestroyLogFont (timefont);
    return 0;

case MSG_CLOSE:
    DestroyMainWindow (hwnd);
    PostQuitMessage (hwnd);
    return 0;
}

return DefaultMainWinProc (hwnd, message, wParam, lParam);
}

...

```

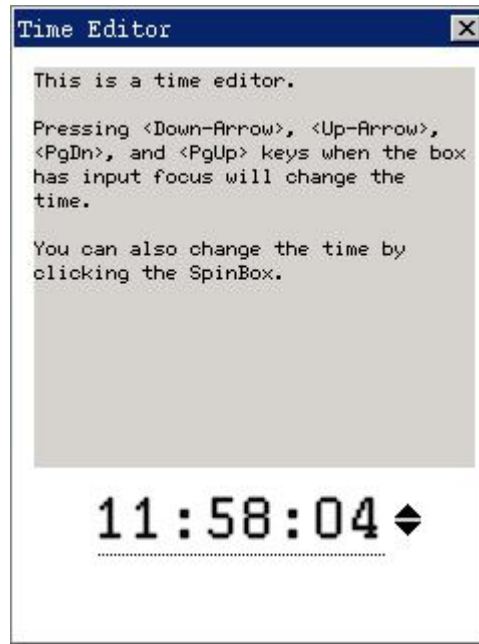


Fig. 6.1 Running effect of time editor

## 7 Menu

### 7.1 Concept of Menu

The menu is generally associated with a window (called normal menu), or presented in independent, popup form (called popup menu). The menu mainly provides the user a shortcut way to make a selection.

### 7.2 Creating and Handling Menu

#### 7.2.1 Creating Normal Menu

In a program, we should first create a menu, and then pass the handle of the menu to `CreateMainWindow` function, which creates a main window. When the main window is shown, the created menu is shown under the caption bar. When the user active and select a menu item using the mouse or the `Alt` key, the window with which the menu associates will receive `MSG_COMMAND` message.

Two procedures are needed when creating a menu:

- Creating the menu bar.
- Creating the submenu of each menu in the menu bar.

First, we call `CreateMenu` to create an empty menu, and then call `InsertMenuItem` function to add menu items to the empty menu, as shown in the following:

```
HMENU hmnu;
MENUITEMINFO mii;

hmnu = CreateMenu();
memset (&mii, 0, sizeof(MENUITEMINFO));
mii.type      = MFT_STRING ;
mii.state     = 0;
mii.id       = IDM_ABOUT_THIS;
mii.typedata  = (DWORD)"File...";
InsertMenuItem(hmnu, 0, TRUE, &mii);
```

If this menu item has its submenu, you can specify the submenu of this menu

item by setting **hsubmenu** variable of this menu item.

```
mii.hsubmenu = create_file_menu();
```

The method to create a submenu is similar to the procedure to create a menu bar, but you should call **CreatePopupMenu** function when creating an empty submenu, as shown in the following:

```
HMENU hmnu;
MENUITEMINFO mii;
memset (&mii, 0, sizeof(MENUITEMINFO));
mii.type = MFT_STRING;
mii.id = 0;
mii.typedata = (DWORD)"File";
hmnu = CreatePopupMenu (&mii);

memset (&mii, 0, sizeof(MENUITEMINFO));
mii.type = MFT_STRING;
mii.state = 0;
mii.id = IDM_NEW;
mii.typedata = (DWORD)"New";
InsertMenuItem(hmnu, 0, TRUE, &mii);
```

**Hmnu** handle in the above code can be used as the handle of the submenu of the higher-level menu item.

### 7.2.2 Creating Popup Menu

The use of a popup menu is different from a menu bar. The popup menu is usually used to respond to the user's clicking on the right mouse button, and generally called "context menu".

The method to create a popup menu is same as the method to create a submenu. You need to call **CreatePopupMenu** function. To display this popup menu, **TrackPopupMenu** function should be called:

```
int WINAPI TrackPopupMenu (HMENU hmnu, UINT uFlags, int x, int y, HWND hwnd);
```

**x** and **y** arguments are the screen coordinates of popup menu, its detail meaning is related to **uFlags** argument.

**uFlags** argument can take the following values:

- **TPM\_LEFTALIGN**: Menu is horizontally left aligned based on (x, y) point, that is to say, x argument specifies the left position of menu.
- **TPM\_CENTERALIGN**: horizontally center aligned.
- **TPM\_RIGHTALIGN**: horizontally right aligned.
- **TPM\_TOPALIGN**: vertically top aligned.
- **TPM\_VCENTERALIGN**: vertically center aligned.
- **TPM\_BOTTOMALIGN**: vertically bottom aligned.

If we want to popup a normal menu, **uFlags** generally takes the value of **TPM\_LEFTALIGN | TPM\_TOPALIGN**, if we want to popup a popup menu, **uFlags** generally takes the value of **TPM\_LEFTALIGN | TPM\_BOTTOMALIGN**.

The following code calls **CreatePopupMenu** function to create a popup menu. It calls **StripPopupHead** function. This function is used to strip the head of the popup menu of MiniGUI. The header of a popup menu is similar to the caption bar of a main window, and the head information of a popup menu will be striped after the function is called.

```

HMENU hNewMenu;
MENUITEMINFO mii;
HMENU hMenuFloat;
memset (&mii, 0, sizeof(MENUITEMINFO));
mii.type      = MFT_STRING;
mii.id        = 0;
mii.typedata   = (DWORD) "File";

hNewMenu = CreatePopupMenu (&mii);

hMenuFloat = StripPopupHead(hNewMenu);

TrackPopupMenu (hMenuFloat, TPM_CENTERALIGN, 40, 151, hWnd);

```

### 7.2.3 MENUITEMINFO Structure

**MENUITEMINFO** structure is the core data structure for operating a menu item, and its definition is as follows:

```

typedef struct _MENUITEMINFO {
    UINT      mask;
    UINT      type;
    UINT      state;
    int       id;
    HMENU     hsubmenu;
    PBITMAP   uncheckedbmp;
    PBITMAP   checkedbmp;
    DWORD     itemdata;
    DWORD     typedata;
}

```

```

    UINT          cch:
} MENUITEMINFO;
typedef MENUITEMINFO* PMENUITEMINFO;

```

Illustrations for these members are as follow:

- **mask**: used by `GetMenuItemInfo` and `SetMenuItemInfo` functions, can be OR'ed with the following values:
  - **MIIM\_STATE**: Get/Set the state of the menu item.
  - **MIIM\_ID**: Get/Set the identifier of the menu item.
  - **MIIM\_SUBMENU**: Get/Set the sub-menu of the menu item.
  - **MIIM\_CHECKMARKS**: Get/Set the check/uncheck bitmap.
  - **MIIM\_TYPE**: Get/Set the type and the type data of the menu item.
  - **MIIM\_DATA**: Get/Set the private data of the menu item.

These macros are used to define which items of the menu item `GetMenuItemInfo` and `SetMenuItemInfo` functions operate on.

- **type**: type of menu items, which can be one of the following values:
  - **MFT\_STRING**: A normal text string menu item.
  - **MFT\_BITMAP**: A bitmap menu item.
  - **MFT\_BITMAPSTRING**: A bitmap menu item followed by a string.
  - **MFT\_SEPARATOR**: A separator in the menu.
  - **MFT\_RADIOCHECK**: A normal text string with a dot check mark.
- **state**: state of the menu item, which can be one of the following values:
  - **MFS\_GRAYED**: The menu item is disabled.
  - **MFS\_DISABLED**: The menu item is disabled.
  - **MFS\_CHECKED**: The menu item is checked. It could be displayed a hook when the **type** is **MFT\_STRING** and displayed a dot when **type** is **MFT\_RADIOCHECK**.
  - **MFS\_ENABLED**: The menu item is enabled.
  - **MFS\_UNCHECKED**: The menu item is unchecked.
- **id**: the integer identifier of the menu item
- **hsubmenu**: the handle of the sub-menu if this menu contains a sub menu.
- **uncheckedbmp**: the pointer to a **BITMAP** object used for unchecked

bitmap menu item.

- **checkedbmp**: the pointer to a **BITMAP** object used for checked bitmap menu item.
- **itemdata**: the private data attached to the menu item
- **typedata**: the data of this menu item, used to pass the string of the menu item.
- **cch**: used by **GetMenuItemInfo** function to indicate the maximal length of the string

When the menu item is **MFT\_BITMAP** type, the member **typedata** will be the pointer to the normal **BITMAP** object, the member **uncheckedbmp** will be the pointer to the highlighted **BITMAP** object, and the member **checkedbmp** will be the pointer to the checked **BITMAP** object. When the menu item is **MFT\_BMPSTRING** type, the member **typedata** will be the pointer to the text string, the member **uncheckedbmp** will be the pointer to the normal **BITMAP** object, and the member **checkedbmp** will be the pointer to the checked **BITMAP** object.

#### 7.2.4 Operating on Menu Item

You can get the menu item properties of interest by **GetMenuItemInfo** function, and can also set the menu item properties of interest by **SetMenuItemInfo** function:

```
int GUIAPI GetMenuItemInfo (HMENU hmenu, int item, BOOL flag, PMENUITEMINFO pmii);  
int GUIAPI SetMenuItemInfo (HMENU hmenu, int item, BOOL flag, PMENUITEMINFO pmii);
```

These two functions are used to get or set the properties of a menu item in **hmenu** menu. At this time, we need an approach to locate the menu item in the menu. MiniGUI provides the following two ways:

- The argument **flag** is **MF\_BYCOMMAND**: The position value item is based on the command identifier. At this time, the **item** argument of the two functions above is the command identifier of the item.
- The argument **flag** is **MF\_BYPOSITION**: The position value item is based on the position in the menu. At this time, the **item** argument of the two

functions above is the position index value of the menu item in the menu, and is zero for the first menu item.

When set or get properties of menu, you should set the **mask** of **MENUIITEMINFO**. You can reference 7.2.3.

MiniGUI also provides some other functions to get or set the menu item properties, and all these functions use the above methods to locate a menu item. These functions include **GetSubMenu**, **SetMenuItemBitmaps**, **GetMenuItemID**, **EnableMenuItem**, etc. Functions of all these can actually be implemented by the above two functions, and we will not describe them once more.

### 7.2.5 Deleting and Destroying Menu or Menu Item

MiniGUI provides the following function to delete menu items from a menu or destroy a menu:

- **RemoveMenu**: This function deletes a specified menu item from a menu. If the menu item includes submenu, this function detaches the submenu from the specified menu item, but does not delete the submenu.
- **DeleteMenu**: This function deletes a specified item from a menu. If the menu item includes submenu, the function will destroy the submenu simultaneously.
- **DestroyMenu**: This function destroys the whole menu.

### 7.2.6 MSG\_ACTIVEMENU Message

When the user activates a popup menu in the menu bar, MiniGUI will send **MSG\_ACTIVEMENU** message to the window procedure in which the menu bar lies. The first parameter of this message is the position of the activated submenu, and the second parameter is the handle of the activated submenu. The application can use this message to handle the menu, for example, change the selection state of some menu items according to the running state of the program. The following code is from **libvcongui** of MiniGUI, and it sets the



selection state of the menu item correspondingly according to the user's selections (the size of the virtual terminal and the character set):

```
case MSG_ACTIVEMENU:
    if (wParam == 2) {
        CheckMenuRadioItem ((HMENU)lParam,
            IDM_40X15, IDM_CUSTOMIZE,
            pConInfo->termType, MF_BYCOMMAND);
        CheckMenuRadioItem ((HMENU)lParam,
            IDM_DEFAULT, IDM_BIG5,
            pConInfo->termCharset, MF_BYCOMMAND);
    }
    break;
```

Note that in the above code, calling `CheckMenuRadioItem` function twice sets the size of current terminal and the character set, respectively.

### 7.3 Sample Program

List 7.1 is a fragment of code illustrating normal menu operations. This example is a part of `notebook` in MDE. For the length consideration, here we only give the part about the menu. The effect of the menu created by this program is shown in Fig. 7.1

List 7.1 Sample of operating normal menu

```
/* Create a "file" menu */
static HMENU createpmenufile (void)
{
    HMENU hmnu;
    MENUITEMINFO mii;
    memset (&mii, 0, sizeof(MENUITEMINFO));
    mii.type      = MFT_STRING;
    mii.id        = 0;
    mii.typeData  = (DWORD)"文件"; /* file */
    hmnu = CreatePopupMenu (&mii);

    memset (&mii, 0, sizeof(MENUITEMINFO));
    mii.type      = MFT_STRING;
    mii.state     = 0;
    mii.id        = IDM_NEW;
    mii.typeData  = (DWORD)"新建"; /* new */
    InsertMenuItem(hmnu, 0, TRUE, &mii);

    mii.type      = MFT_STRING;
    mii.state     = 0;
    mii.id        = IDM_OPEN;
    mii.typeData  = (DWORD)"打开..."; /* open */
    InsertMenuItem(hmnu, 1, TRUE, &mii);

    mii.type      = MFT_STRING;
    mii.state     = 0;
    mii.id        = IDM_SAVE;
    mii.typeData  = (DWORD)"保存"; /* save */
    InsertMenuItem(hmnu, 2, TRUE, &mii);
}
```

```

    mii.type      = MFT_STRING;
    mii.state     = 0;
    mii.id        = IDM_SAVEAS;
    mii.typepdata = (DWORD)"另存为..."; /* save as */
    InsertMenuItem(hmnu, 3, TRUE, &mii);

    mii.type      = MFT_SEPARATOR;
    mii.state     = 0;
    mii.id        = 0;
    mii.typepdata = 0;
    InsertMenuItem(hmnu, 4, TRUE, &mii);

    mii.type      = MFT_SEPARATOR;
    mii.state     = 0;
    mii.id        = 0;
    mii.typepdata = 0;
    InsertMenuItem(hmnu, 5, TRUE, &mii);

    mii.type      = MFT_STRING;
    mii.state     = 0;
    mii.id        = IDM_EXIT;
    mii.typepdata = (DWORD)"退出"; /* quit */
    InsertMenuItem(hmnu, 6, TRUE, &mii);

    return hmnu;
}

/* Create menu bar */
static HMENU createmenu (void)
{
    HMENU hmnu;
    MENUITEMINFO mii;

    hmnu = CreateMenu();

    memset (&mii, 0, sizeof(MENUITEMINFO));
    mii.type      = MFT_STRING;
    mii.id        = 100;
    mii.typepdata = (DWORD)"文件"; /* file */
    mii.hsubmenu  = createmenufile ();

    InsertMenuItem(hmnu, 0, TRUE, &mii);

    ...

    return hmnu;
}

/* Handle MSG_ACTIVEMENU to ensure to set the selection state of menu items correctly */
case MSG_ACTIVEMENU:
    if (wParam == 2) {
        /* Set the checked state of the menu items by CheckMenuRadioItem */
        CheckMenuRadioItem ((HMENU)lParam,
            IDM_40X15, IDM_CUSTOMIZE,
            pNoteInfo->winType, MF_BYCOMMAND);
        CheckMenuRadioItem ((HMENU)lParam,
            IDM_DEFAULT, IDM_BIG5,
            pNoteInfo->editCharset, MF_BYCOMMAND);
    }
    break;

/* Handle MSG_COMMAND message to handle the commands of each menu item */
case MSG_COMMAND:
    switch (wParam) {
        case IDM_NEW:
            break;

        case IDM_OPEN:
            break;

        case IDM_SAVE:
            break;

        case IDM_SAVEAS:

```

```
}; break;
```



Fig. 7.1 Menu created by the notebook program

List 7.2 illustrates the sample program for the popup menu.

List 7.2 Sample program of popup menu

```
static HMENU CreateQuickMenu (void)
{
    int i;
    HMENU hNewMenu;
    MENUITEMINFO mii;
    HMENU hMenuFloat;

    char *msg[] = {
        "A",
        "F",
        "H",
        "L",
        "P",
        "S",
        "X"
    };

    memset (&mii, 0, sizeof(MENUITEMINFO));
    mii.type = MFT_STRING;
    mii.id = 0;
    mii.typedata = (DWORD) "File";

    hNewMenu = CreatePopupMenu (&mii);

    for ( i = 0; i < 7; i ++ ) {
        memset (&mii, 0, sizeof(MENUITEMINFO));
        mii.type = MFT_STRING;
        mii.id = 100 + i;
        mii.state = 0;
        mii.typedata = (DWORD) msg[i];
```

```

    InsertMenuItem ( hNewMenu, i, TRUE, &mii );
}

hMenuFloat = StripPopupHead(hNewMenu);

TrackPopupMenu (hMenuFloat, TPM_CENTERALIGN | TPM_LEFTBUTTON , 40, 151, hWnd);
}

```

The popup menu created by the program in List 7.2 is as shown in Fig. 7.2.



Fig.7.2 Popup menu

## 8 Scrollbar

### 8.1 Concept of Scrollbar

Scrollbar is one of the best functions in GUI, which is easy to use and provides good effect of vision feedback. You can use scrollbar to display any thing - whatever text, graph, table, database record, image, or web page, if only the needed space of which exceeds the display region of the window.

Both vertical-direction (for moving up and down) and horizontal-direction (for moving right and left) scrollbars are available. The user can use mouse to click on the arrows or in the region between the arrows of the scrollbar, here, the movement of the thumb in the scrollbar is proportional to the movement of the displayed information in the whole file. The user can also use mouse to drag the thumb to a specified position. Fig. 8.1 shows the suggested using method of the vertical scrollbar.

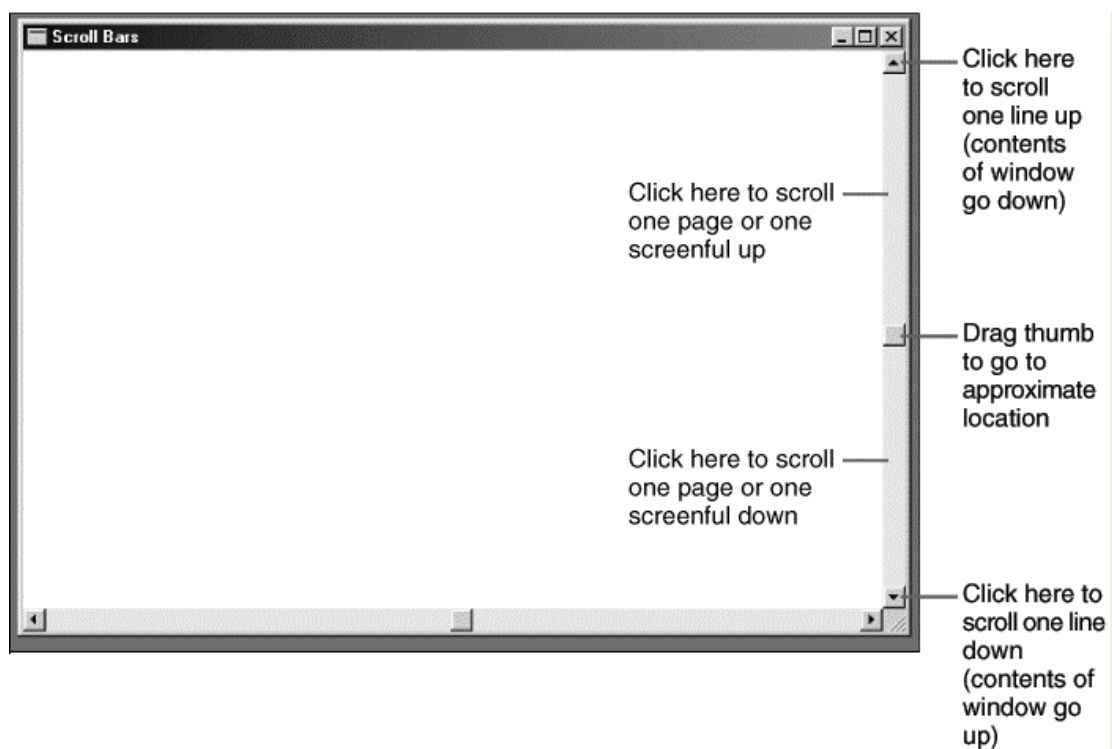


Fig. 8.1 Suggested using method of the vertical scrollbar

Sometimes, the programs fell difficult to understand the scrolling concept, because their opinion is different from that of the users: the user intends to see the lower part of the file by scrolling downward; however, the program actually moves the file upward relative to the display window. MiniGUI is according to the user's opinion: scrolling upward means moving to the start of the file; scrolling downwards means moving to the end of the file.

It is easy to include horizontal or vertical scrollbar in the application. The program need only to include window style `WS_VSCROLL` and/or `WS_HSCROLL` in the third argument of `CreateWindow` or `CreateMainWindow`. These scrollbars are commonly placed in the right side and bottom side of the window, extending to the whole length or width of the display region. The client area does not include the space filled by scrollbar.

In MiniGUI, mouse click is automatically transformed into corresponding message; however, you must treat the keyboard message by yourself.

## 8.2 Enabling or Disabling a Scrollbar

```
EnableScrollBar (hWnd, SB_HORZ, TRUE);  
EnableScrollBar (hWnd, SB_VERT, FALSE);
```

`EnableScrollBar` function can be used to enable or disable the scrollbar, and its second parameter specifies which (vertical or horizontal) scrollbar will be operated.

## 8.3 Range and Position of a Scrollbar

Each scrollbar has a related "range" (which is a pair of integer, representing the minimum and the maximum) and "position" (which is the position of thumb in the range). When the thumb is positioned at the top (or left) of the scrollbar, the position of the scroll bar is the minimum of the range; when the thumb is positioned at the bottom (or right) of the scrollbar, the position of the scroll bar is the maximum of the range.

In a default case, the range of the scrollbar is from 0 (top or left) to 100 (bottom or right), while we can also change the range to values, which are easy to treat by a program:

```
SCROLLINFO si;  
  
si.nMax = 100; // The maximal position  
si.nMin = 0;  // The minimal position  
si.nPage = 10; // this variable determines the length of the thumb  
               // this length is dependent on the value of nPage/(nMax-nMin)  
si.nPos = 0;   // the present position of the thumb, which must be between nMin and nMax  
  
SetScrollInfo (hWnd, Bar, &si, bRedraw);
```

The argument `Bar` is `SB_VERT` or `SB_HORZ`, and `nMin` and `nMax` is the minimum and maximum of the range, respectively. If you want MiniGUI redraw the scrollbar according to the new range, `bRedraw` should be set to `TRUE` (if you recall other functions which influence the position of the scrollbar after recalling `SetScrollRange`, `bRedraw` should be set to `FALSE` to prevent the scrollbar from being redraw excessively). The position of the thumb is always discrete integer. For example, a scrollbar with range from 0 to 4 has five positions of thumb, as shown in Fig. 8.2.

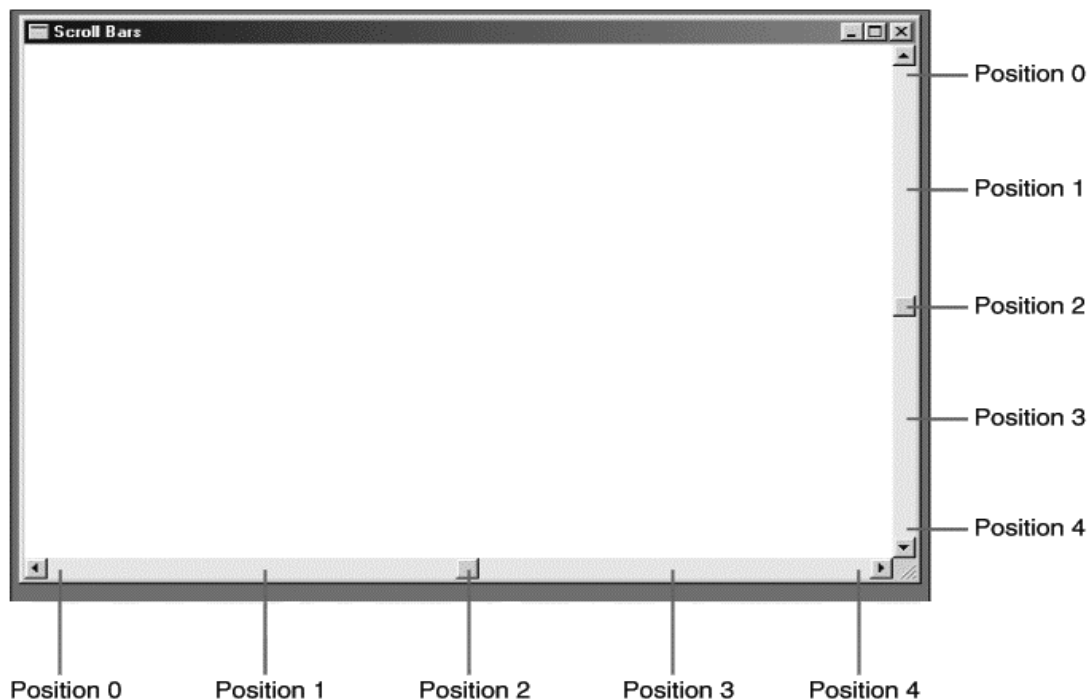


Fig. 8.2 A scrollbar with range from 0 to 4 has five positions of thumb

When the scrollbar is used in a program, the program together with MiniGUI is in charge of maintaining the scrollbar and updating the position of the thumb. Following is the handling of the scrollbar by MiniGUI:

- Handling all the scrollbar mouse events.
- When the user drags the thumb in the scrollbar, move the thumb.
- Sending scrollbar message to the window procedure of the window containing the scrollbar.

Following are the tasks remained for the program:

- Initializing the range and position of the scrollbar.
- Handling the scrollbar message in the window procedure.
- Refreshing the position of the scroll thumb in the scrollbar.
- Changing the content in the display region responding the change of the scrollbar.

## 8.4 Messages of Scrollbar

When the user clicks the scrollbar or drags the thumb by mouse, MiniGUI sends message of `MSG_VSCROLL` (for moving up and down) and `MSG_HSCROLL` (for moving left and right) to the window procedure.

Like all other messages, `MSG_VSCROLL` and `MSG_HSCROLL` also have message parameter `lParam` and `wParam`. In most cases, we can ignore the parameter `lParam`.

`wParam` is a value, which specifies the operation to the scrollbar by the mouse. The value is regarded as a "Notification code". The notification code is defined with a prefix "`SB_`". Table 8.1 illustrates the notification codes defined by MiniGUI.

Table 8.1 The scrollbar notification codes defined by MiniGUI

Notification code identifier	Meaning
<code>SB_LINEUP</code>	Click once the up arrow in the vertical scrollbar by mouse.
<code>SB_LINEDOWN</code>	Click once the down arrow in the vertical scrollbar by mouse.



<b>SB_LINELEFT</b>	Click once the left arrow in the horizontal scrollbar by mouse.
<b>SB_LINERIGHT</b>	Click once the right arrow in the horizontal scrollbar by mouse.
<b>SB_PAGEUP</b>	Click once the region between the up arrow and the thumb of the vertical scrollbar by mouse.
<b>SB_PAGEDOWN</b>	Click once the region between the down arrow and the thumb of the vertical scrollbar by mouse.
<b>SB_PAGELEFT</b>	Click once the region between the left arrow and the thumb of the horizontal scrollbar by mouse.
<b>SB_PAGERIGHT</b>	Click once the region between the right arrow and the thumb of the horizontal scrollbar by mouse.
<b>SB_THUMBTRACK</b>	Message received by the window when the user drags the thumb. It should be noted that the value passed through <b>lParam</b> presents the corresponding numerical value.
<b>SB_THUMBPOSITION</b>	The user finished dragging the thumb.

The identifiers including **LEFT** and **RIGHT** are used to horizontal scrollbar, and the identifiers including **UP** and **DOWN** is used to vertical scrollbar. The identifiers generated by the mouse clicking in different regions of the scrollbar are shown in Fig. 8.3.

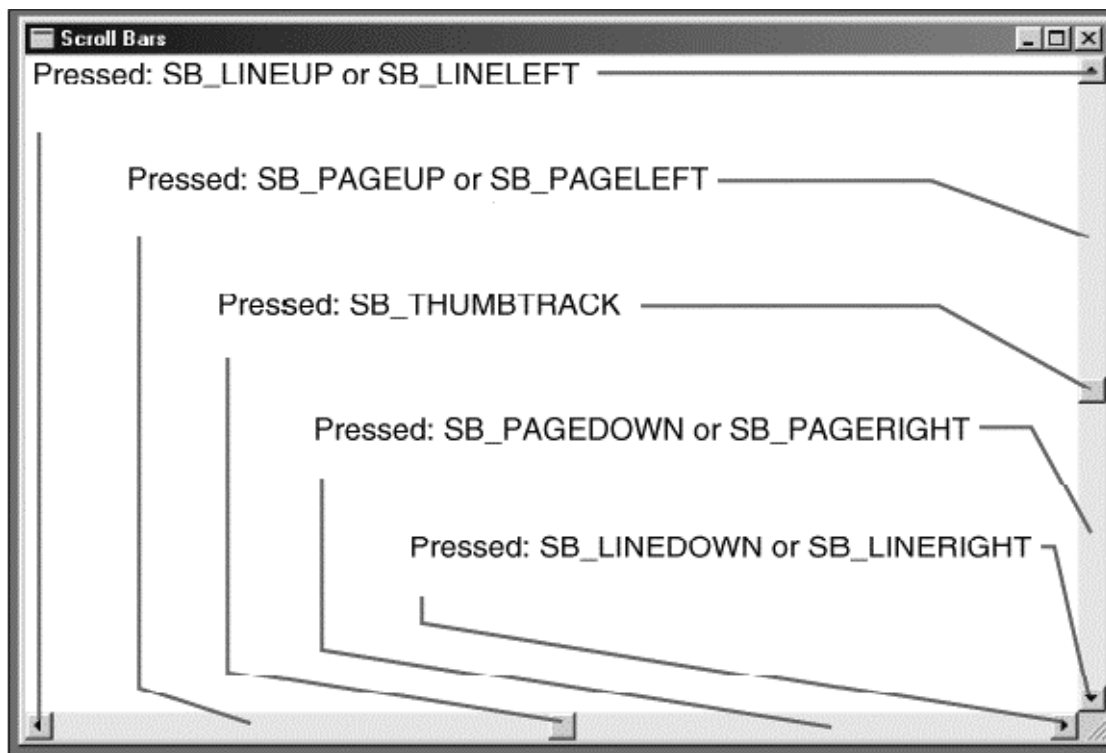


Fig. 8.3 Identifiers generated by the mouse clicking in different regions of the scrollbar

When you put the mouse cursor on the thumb and press the mouse button, you can move the thumb. Scrollbar message including notification code of **SB\_THUMBTRACK** is thus generated. When **wParam** is **SB\_THUMBTRACK**, **lParam** is the present position when the user drags the thumb. This position is between

the minimum and the maximum. As for other scrollbar operations, `lParam` should be ignored.

For providing feedback to the user, window system moves the thumb when you move it with mouse, and at the same time your program will receive message of `SB_THUMBTRACK`. However, if you do not recall `SetScrollPos` to handle messages of `SB_THUMBTRACK` or `SB_THUMBPOSITION`, the thumb will jump back to its original position after the user releases the mouse button.

Program can handle message of `SB_THUMBTRACK`. If the message of `SB_THUMBTRACK` is handled, you need move the content in the display region when the user drags the thumb.

## 8.5 Sample Program

List 8.1 presents a simple scrollbar-handling program. The complete code of this program is available in `scrollbar.c` in the sample program package `mg-samples`.

List 8.1 Scrollbar and its handling

```
#include <stdio.h>
#include <string.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>

/* define the string array to be displayed in the window */
static char* strLine[] = {
    "This is the 1st line.",
    "This is the 2nd line.",
    "This is the 3rd line.",
    "This is the 4th line.",
    "This is the 5th line.",
    "This is the 6th line.",
    "This is the 7th line.",
    "This is the 8th line.",
    "This is the 9th line.",
    "This is the 10th line.",
    "This is the 11th line.",
    "This is the 12th line.",
    "This is the 13th line.",
    "This is the 14th line.",
    "This is the 15th line.",
    "This is the 16th line.",
    "This is the 17th line."
};
```

```

/* The window procedure */
static int ScrollWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    static int iStart = 0;
    static int iStartPos = 0;

    switch (message) {
        case MSG_CREATE:
            /* Create caret */
            if (!CreateCaret (hWnd, NULL, 8, 14))
                fprintf (stderr, "Create caret error!\n");
            break;

        case MSG_SHOWWINDOW:
            /* Disable the horizontal scrollbar */
            EnableScrollBar (hWnd, SB_HORZ, FALSE);
            /* set scrolling range of the vertical scrollbar: 0~20 */
            SetScrollRange (hWnd, SB_VERT, 0, 20);
            ShowCaret (hWnd);
            break;

        case MSG_SETFOCUS:
            /* activate and display the caret when obtaining input focus */
            ActiveCaret (hWnd);
            ShowCaret (hWnd);
            break;

        case MSG_KILLFOCUS:
            /* hide the caret when losing input focus */
            HideCaret (hWnd);
            break;

        case MSG_LBUTTONDOWN:
            /* modify the position of the caret when mouse clicking */
            SetCaretPos (hWnd, LOWORD (lParam), HIWORD (lParam));
            break;

        case MSG_LBUTTONDBLCLK:
            /* hide the horizontal scrollbar when left mouse button double clicked */
            ShowScrollBar (hWnd, SB_HORZ, FALSE);
            break;

        case MSG_RBUTTONDBLCLK:
            /* show the horizontal scrollbar when right mouse button double clicked */
            ShowScrollBar (hWnd, SB_HORZ, TRUE);
            break;

        /* handle the horizontal scrollbar message */
        case MSG_HSCROLL:
            if (wParam == SB_LINERIGHT) {
                if (iStartPos < 5) {
                    iStartPos ++;
                    /* scrolling to the right, one character width once */
                    ScrollWindow (hWnd, -GetSysCharWidth (), 0, NULL, NULL);
                }
            }
            else if (wParam == SB_LINELEFT) {
                if (iStartPos > 0) {
                    iStartPos --;

                    /* scrolling to the left, one character width once */
                    ScrollWindow (hWnd, GetSysCharWidth (), 0, NULL, NULL);
                }
            }
            break;

        /* handle the vertical scrollbar message */
        case MSG_VSCROLL:
            if (wParam == SB_LINEDOWN) {
                if (iStart < 12) {
                    iStart ++;
                    /* scrolling upwards, 20 pixels once */
                    ScrollWindow (hWnd, 0, -20, NULL, NULL);
                }
            }
    }
}

```

```

    }
    else if (wParam == SB_LINEUP) {
        if (iStart > 0) {
            iStart --;

            /* scrolling downwards, 20 pixels once */
            ScrollWindow (hWnd, 0, 20, NULL, NULL);
        }
    }
    /* refresh the scrollbar position */
    SetScrollPos (hWnd, SB_VERT, iStart);
    break;

case MSG_PAINT:
{
    HDC hdc;
    int i;
    RECT rcClient;
    GetClientRect (hWnd, &rcClient);
    hdc = BeginPaint (hWnd);
    /* output 17 strings according to the present scrolling position */
    for (i = 0; i < 17 - iStart; i++) {
        rcClient.left = 0;
        rcClient.right = (strlen (strLine [i + iStart]) - iStartPos)
            * GetSysCharWidth ();
        rcClient.top = i*20;
        rcClient.bottom = rcClient.top + 20;

        TextOut (hdc, 0, i*20, strLine [i + iStart] + iStartPos);
    }

    EndPaint (hWnd, hdc);
}
return 0;

/* destroy the caret and the main window */
case MSG_CLOSE:
    DestroyCaret (hWnd);
    DestroyMainWindow (hWnd);
    PostQuitMessage (hWnd);
    return 0;
}

return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

static void InitCreateInfo(PMAINWINCREATE pCreateInfo)
{
    /* specify having horizontal and vertical scrollbar in window style */
    pCreateInfo->dwStyle = WS_BORDER | WS_CAPTION | WS_HSCROLL | WS_VSCROLL;
    pCreateInfo->dwExStyle = WS_EX_NONE | WS_EX_IMECOMPOSE;
    pCreateInfo->spCaption = "The Scrollable Main Window" ;
    pCreateInfo->hMenu = 0;
    pCreateInfo->hCursor = GetSystemCursor(0);
    pCreateInfo->hIcon = 0;
    pCreateInfo->MainWindowProc = ScrollWinProc;
    pCreateInfo->lx = 0;
    pCreateInfo->ty = 0;
    pCreateInfo->rx = 400;
    pCreateInfo->by = 280;
    pCreateInfo->iBkColor = COLOR_lightwhite;
    pCreateInfo->dwAddData = 0;
    pCreateInfo->hHosting = HWND_DESKTOP;
}

int MiniGUIMain(int args, const char* arg[])
{
    MSG Msg;
    MAINWINCREATE CreateInfo;
    HWND hMainWnd;

#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER, "scrollbar", 0, 0);
#endif

```

```
InitCreateInfo(&CreateInfo);  
  
hMainWnd = CreateMainWindow(&CreateInfo);  
if (hMainWnd == HWND_INVALID)  
    return 1;  
  
ShowWindow(hMainWnd, SW_SHOW);  
  
while (GetMessage(&Msg, hMainWnd) ) {  
    DispatchMessage(&Msg);  
}  
  
MainWindowThreadCleanup(hMainWnd);  
return 0;  
}  
  
#ifndef _LITE_VERSION  
#include <minigui/dti.c>  
#endif
```

Fig. 8.4 is the running effect of the program in List 8.1.



Fig. 8.4 Handling of scrollbar



## 9 Keyboard and Mouse

The application receives the user's input from keyboard and mouse (or other pointing device, such as touch-screen). Application of MiniGUI receives keyboard and mouse input by handling messages sent to window. We will describe generation of keyboard and mouse message, and how application receives and handles messages of keyboard and mouse in this chapter.

### 9.1 Keyboard

#### 9.1.1 Keyboard Input

Fig. 9.1 illustrates the process of handling keyboard input in MiniGUI.

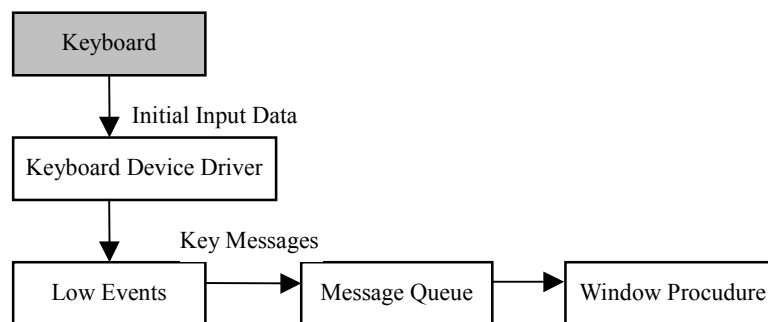


Fig. 9.1 Keyboard input in MiniGUI

MiniGUI receives original input event or data from keyboard through keyboard device driver, and transforms it into MiniGUI abstract keyboard event and data. Related handling procedures of bottom layer event transform these keyboard events into key press/release messages of top layer and put them in the message queue. Application gets these messages through message loop, and dispatches them to window procedure for handling. MiniGUI can support 255 keys, and each key corresponds to a unique "scan code", scan code less than 129 is used to define keys on PC keyboard. The following is the definition of some scan codes in `minigui/common.h`:

```
#define MGUI_NR_KEYS      255
#define NR_KEYS           128
```

```
#define SCANCODE_USER (NR_KEYS + 1)
#define SCANCODE_ESCAPE 1
#define SCANCODE_1 2
#define SCANCODE_2 3
#define SCANCODE_3 4
#define SCANCODE_4 5
#define SCANCODE_5 6
#define SCANCODE_6 7
#define SCANCODE_7 8
#define SCANCODE_8 9
#define SCANCODE_9 10
#define SCANCODE_0 11
#define SCANCODE_MINUS 12
#define SCANCODE_EQUAL 13
#define SCANCODE_BACKSPACE 14
#define SCANCODE_TAB 15
...
```

### 9.1.2 Key Stroke Message

When a key is pressed down, application will receive a message **MSG\_KEYDOWN** or **MGS\_SYSKEYDOWN**; and releasing a key will generate a message **MSG\_KEYUP** or **MGS\_SYSKEYUP**.

Pressing key and releasing key messages are usually appear in pairs; while the user presses a key and does not release it, auto repeating characteristic of the keyboard will be started up after a period of time, and the system will generate a series of messages **MSG\_KEYDOWN** or **MGS\_SYSKEYDOWN**; and a message **MSG\_KEYUP** or **MSG\_SYSKEYUP** will not be generated till the user releases this key.

In MiniGUI, when the user presses one key as **ALT** being pressed down, system keystroke messages **MSG\_SYSKEYDOWN** and **MSG\_SYSKEYUP** will be generated.

Non-system keystroke generates non-system keystroke messages **MSG\_KEYDOWN** and **MSG\_KEYUP**. System keystroke messages are used to control the activation of menu in MiniGUI, and non-system keystroke messages are mainly used in application. If a system keystroke message is handled by application, this message should be transferred to the function **DefaultMainWinProc** for handling furthermore. Otherwise, system operation will be disabled.



Keystroke messages' **wParam** parameter presents the scan code of the key, and **lParam** parameter includes the status flags of special keys such as **SHIFT**, **ALT**, **CTRL**, etc.

### 9.1.3 Character Message

Usually, a typical window procedure does not handle directly the key stroke messages of character keys, but handles character message **MSG\_CHAR** of the character key, and **wParam** parameter of message **MSG\_CHAR** presents the encode value of this character.

**MSG\_CHAR** is usually generated by **TranslateMessage** function, which checks the scan code and the status of the corresponding key when receiving messages **MSG\_KEYDOWN** and **MSG\_SYSKEYDOWN**. If it can be transformed to a character, **MSG\_CHAR** or **MSG\_SYSCHAR** message of the corresponding character will be generated, and will be sent to the target window of the keystroke message. The application generally calls **TranslateMessage** function before **DispatchMessage** in a message loop is as follows:

```
while (GetMessage(&Msg, hMainWnd)) {  
    TranslateMessage(&Msg);  
    DispatchMessage(&Msg);  
}
```

Because **TranslateMessage** function handles messages **MSG\_KEYDOWN** and **MSG\_SYSKEYDOWN** before dispatching message, generates character message, and transfers it directly to window procedure, window procedure will receive the character message of character key first and then receive keystroke message.

Usually, parameter **wParam** of character message includes the ISO8859-1 encode of the character, and the content of parameter **lParam** is the same as the parameter of keystroke message generating the character message.

The basic rule for handling key stroke message and character message is: You can handle message **MSG\_CHAR** if you want to get keyboard character input into

window; and you can handle message `MSG_KEYDOWN` when you want to get cursor keys, function keys, `Del`, `Ins`, `Shift`, `Ctrl` and `Alt` key strokes.

#### 9.1.4 Key Status

Meanwhile handling keyboard messages, application need determine the current status of special shift keys (`Shift`, `Ctrl` and `Alt`) or switch keys (`CapsLock`, `NumLock`, and `ScrollLock`). Parameter `lParam` includes the status flags of special keys, and application can determine the status of a certain key by using `AND` operation of specific macro with this parameter. For example, if (`lParam & KS_LEFTCTRL`) is `TRUE`, left `CTRL` key is pressed down. Key status macros defined by MiniGUI include:

<code>KS_CAPSLOCK</code>	CapsLock key is locked
<code>KS_NUMLOCK</code>	NumLock key is locked
<code>KS_SCROLLLOCK</code>	ScrollLock key is locked
<code>KS_LEFTCTRL</code>	Left Ctrl key is pressed down
<code>KS_RIGHTCTRL</code>	Right Ctrl key is pressed down
<code>KS_CTRL</code>	One of Ctrl keys is pressed
<code>KS_LEFTSHIFT</code>	Left Shift key is pressed down
<code>KS_RIGHTSHIFT</code>	Right Shift key is pressed down
<code>KS_SHIFT</code>	One of Shift keys is pressed down
<code>KS_IMEPOST</code>	The message is posted by IME window
<code>KS_LEFTBUTTON</code>	Left mouse button is preessed down
<code>KS_RIGHTBUTTON</code>	Right mouse button is preessed down
<code>KS_MIDDLBUTTON</code>	Middle mouse button is preessed down
<code>KS_CAPTURED</code>	Mouse is captured by the target window

Except `KS_CAPTURED` can only be used for mouse message, other macros can be used both for handling keystroke messages and mouse messages.

Application can get key status value through `GetShiftKeyStatus` function:

```
DWORD WINAPI GetShiftKeyStatus (void);
```

The return value of this function includes the status of all above keys.

Application can determine a certain key status by using `AND` operation of specific macro with the return value. For example, using `GetShiftKeyStatus()` & `KS_CTRL` to determine whether the left or right `ctrl` key is pressed, if it is the value of expression above is `TRUE`.

Application can also get the status of a certain key on the keyboard through

**GetKeyStatus** function:

```
BOOL WINAPI GetKeyStatus (UINT uKey);
```

Parameter **uKey** presents the scan code of the key to be required. If this key is pressed down, **GetKeyStatus** returns **TRUE**, otherwise returns **FALSE**.

### 9.1.5 Input Focus

System sends keyboard messages to the message queue of the window currently having input focus. Window with input focus can receive keyboard input, such windows are usually active main window, child window of active main window or grandchild window of active main window, etc. Child window usually indicates to have input focus by displaying a blinking caret. Please refer to Chapter 10 “*Icon, Cursor, and Caret*” for information about caret.

Input focus is a property of window, and system can make keyboard shared to all the windows displayed on the screen through changing the input focus among these windows. The user can move input focus from one window to another. If input focus is moved from one window to another, system sends message **MSG\_KILLFOCUS** to the window losing focus, and sends message **MSG\_SETFOCUS** to the window gaining focus.

Application can get the handle of the child window having input focus of a certain window by calling **GetFocusChild** function:

```
#define GetFocus GetFocusChild  
HWND WINAPI GetFocusChild (HWND hWnd);
```

Parent window can put input focus to any one of its child windows by calling **SetFocusChild** function:

```
#define SetFocus SetFocusChild  
HWND WINAPI SetFocusChild (HWND hWnd);
```

### 9.1.6 Sample Program

The code in List 9.1 illustrates simple concept of keyboard input. The complete code of this program is included in `simplekey.c` of sample program package `mg-samples` for this guide.

List 9.1 simplekey.c

```
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>

static int SimplekeyWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_KEYDOWN:
            /* Print the scan code of the key pressed */
            printf ("MGS_KEYDOWN: key %d\n", LOWORD(wParam));
            break;

        case MSG_KEYUP:
            /* Print the scan code of the key released */
            printf ("MGS_KEYUP: key %d\n", LOWORD(wParam));
            break;

        case MSG_CHAR:
            /* Print the encode of the translated character */
            printf ("MGS_CHAR: char %d\n", wParam);
            break;

        case MSG_CLOSE:
            DestroyAllControls (hWnd);
            DestroyMainWindow (hWnd);
            PostQuitMessage (hWnd);
            return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

/* the below code for creating main window are omitted */
```

Window procedure of above program prints the parameter `wParam` of each message `MSG_KEYDOWN`, `MSG_KEYUP`, and `MGS_CHAR` it received. The value of this parameter may be scan code of the pressed/released key (`MSG_KEYDOWN` and `MSG_KEYUP`), and may also be encode value of the character (`MSG_CHAR`).

## 9.2 Mouse

### 9.2.1 Mouse Input

The handling for mouse is similar to that for keyboard in MiniGUI, as shown in

Fig. 9.2.

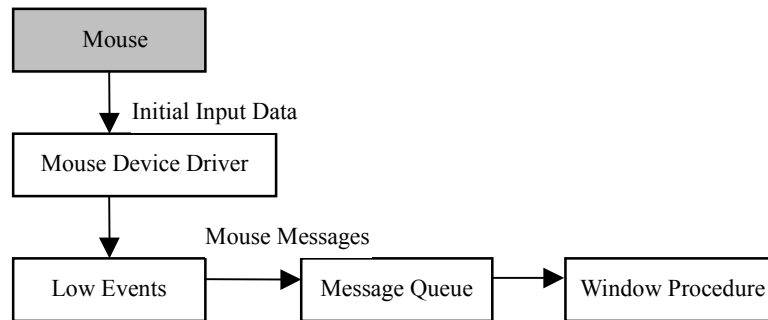


Fig. 9.2 Mouse input in MiniGUI

MiniGUI receives original input event or data through mouse device driver, and transforms it into MiniGUI abstract mouse event and data. Related bottom event handling procedures transform these mouse events into mouse messages of top layer and put them into the message queue. Application gets these messages through message loop, and dispatches them to window procedure for handling.

When the user moves the mouse, system moves an icon called mouse cursor. Mouse cursor includes a pixel point called hotspot, which is used by system to track and identify the mouse position. If a mouse event occurs, the window under the hotspot usually receives related mouse messages. Window, which can receive mouse messages, are not always active window or window-having keyboard input focus. Please refer to Chapter 10 “*Icon, Cursor, and Caret*” for information about mouse cursor.

### 9.2.2 Mouse Message

As long as the user moves mouse, presses down or releases mouse button, a mouse event will be generated. MiniGUI transforms mouse input events of bottom layer into mouse messages and posts them into the message queue of corresponding window. When mouse cursor is within a window, or a mouse event occurs when the window captures the mouse, the window will receive mouse message, regardless whether this window is active or has input focus.

Mouse messages can be divided into two groups: client area messages and non-client area messages. Application usually treats only client area mouse messages and ignores non-client area mouse messages.

If mouse cursor is in client area of the window when mouse event occurs, the window will receive a client area mouse message. If the user moves mouse in client area, system posts a message **MSG\_MOUSEMOVE** to the window. When cursor is in client area, the following message is sent if the user presses down or releases a mouse button:

<b>MSG_LBUTTONDOWN</b>	left mouse button is pressed down
<b>MSG_LBUTTONUP</b>	left mouse button is released
<b>MSG_RBUTTONDOWN</b>	right mouse button is pressed down
<b>MSG_RBUTTONUP</b>	right mouse button is released
<b>MSG_LBUTTONDBLCLK</b>	left mouse button is double clicked
<b>MSG_RBUTTONDBLCLK</b>	right mouse button is double clicked

Parameter **lParam** of mouse messages in client area indicates the position of hotspot, the low word is the x-coordinate of the point and the high word is the y-coordinate. These two positions are both given in client area coordinate, i.e., coordinates relative to upper-left (0, 0) of client area of the window. It should be noted that the position coordinates of above messages are given in screen coordinate when a window captures the mouse.

Parameter **lParam** is the key status value discussed in previous section, which indicates the status of other buttons of mouse and shift keys of **CTRL** and **SHIFT** etc., when mouse event occurs. You must check these flags when you need to handle mouse messages according to other buttons or status of **CTRL** and **SHIFT** keys. For example, if **(lParam & KS\_SHIFT)** is **TRUE**, then a mouse event takes place when **SHIFT** key is pressed down.

If mouse event occurs in the non-client area of the window, such as caption bar, menu and window scrollbar, the window will receive a mouse message of non-client area, which includes:

<b>MSG_NCLBUTTONDOWN</b>	left mouse button is pressed down
<b>MSG_NCLBUTTONUP</b>	left mouse button is released
<b>MSG_NCRBUTTONDOWN</b>	right mouse button is pressed down
<b>MSG_NCRBUTTONUP</b>	right mouse button is released

MSG_NCLBUTTONDBLCLK	left mouse button is double clicked
MSG_NCRBUTTONDBLCLK	right mouse button is double clicked

Usually, application need not handle non-client area mouse messages, but leaves them to system for default handling; thereby some system function can be implemented.

Parameter **lParam** of non-client area mouse message includes x-coordinate at low word and y-coordinate at high word, which are both window coordinate.

Parameter **wParam** of non-client area mouse message indicates the position of non-client area when moving or clicking mouse button, which is one of the identifiers beginning with **HT** defined in `minigui/window.h`:

```
#define HT_UNKNOWN          0x00
#define HT_OUT              0x01
#define HT_MENUBAR         0x02
#define HT_TRANSPARENT     0x03
#define HT_BORDER_TOP      0x04
#define HT_BORDER_BOTTOM   0x05
#define HT_BORDER_LEFT     0x06
#define HT_BORDER_RIGHT    0x07
#define HT_CORNER_TL       0x08
#define HT_CORNER_TR       0x09
#define HT_CORNER_BL       0x0A
#define HT_CORNER_BR       0x0B
#define HT_CLIENT           0x0C

#define HT_NEEDCAPTURE     0x10
#define HT_BORDER          0x11
#define HT_NCLIENT         0x12
#define HT_CAPTION         0x13
#define HT_ICON            0x14
#define HT_CLOSEBUTTON     0x15
#define HT_MAXBUTTON       0x16
#define HT_MINBUTTON       0x17
#define HT_HSCROLL         0x18
#define HT_VSCROLL         0x19
```

Above identifiers are called hit-testing code, and the hotspot position identified includes caption bar, menu bar, and client area etc.

If a mouse event occurs, system will send a **MSG\_HITTEST** (**MSG\_NCHITTEST**) message to the window having cursor hotspot or the window capturing the mouse, and MiniGUI determines whether send the mouse message to the client area or non-client area.

**wParam** parameter of **MSG\_HITTEST** message is the x-coordinate of the cursor hotspot, and **lParam** parameter is the y-coordinate of the cursor hotspot. The

default mouse message procedure in MiniGUI handles `MSG_HITTEST` message, checks the pair of coordinate and returns a hit-testing code of the cursor hotspot. If the cursor hotspot is in the client area of the window, the hit-testing code of `HT_CLIENT` is returned, and MiniGUI switched the screen coordinate into client area coordinate, and then post client area mouse messages to corresponding window process. If the cursor hotspot is in the non-client area of the window, other hit-testing codes are returned, and MiniGUI post non-client area mouse messages to window procedure, and place the hit-testing code into `wParam` parameter, and place the cursor coordinate into `lParam` parameter.

### 9.2.3 Capture of Mouse

The window procedure usually received mouse messages only when the mouse cursor is in the client area or non-client area, i.e., the system only sends mouse messages to a window under the cursor hotspot. While in some cases the application may need to receive mouse messages, even if the cursor hotspot is outside its window area. In this case, we can use `SetCapture` function to make a certain window capture the mouse, so that this window will receive all mouse messages before the application calls `ReleaseCapture` to recover normal mouse handling mode:

```
HWND GUIAPI SetCapture(HWND hWnd);  
void GUIAPI ReleaseCapture(void);  
HWND GUIAPI GetCapture(void);
```

At a certain time, only one window can capture the mouse, and the application can use `GetCapture` function to determine which window has captured the mouse at present.

Generally speaking, the mouse is only captured when the mouse button is pressed down in the client area, and releases the mouse capture when the mouse button is released.

The code in List 9.2 creates a simple button-like control, and its response to



the mouse action is like a button and can demonstrate the necessity of the mouse capture, although it can do nothing and does not look like a button very much. The complete code is available in program `capture.c` in the example program package `mg-samples` for this guide.

List 9.2 capture.c

```
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

#define IDC_MYBUTTON    100

/* the callback function of the simple button control class */
static int MybuttonWindowProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    /*
     * used to save the button status.
     * Noet: you should not use a static variable to save
     * the information of a control instance.
     */
    static int status = 0;

    switch (message) {
    case MSG_LBUTTONDOWN:
        /* set the pressed status */
        status = 1;
        /* capture the mouse */
        SetCapture (hWnd);
        /* invalidate the control, causing to repaint the button */
        InvalidateRect (hWnd, NULL, TRUE);
        break;

    case MSG_LBUTTONUP:
        if (GetCapture() != hWnd)
            break;
        /* set the released status */
        status = 0;
        /* release the mouse */
        ReleaseCapture ();
        /* invalidate the control, causing to repaint the button */
        InvalidateRect (hWnd, NULL, TRUE);
        break;

    case MSG_PAINT:
        hdc = BeginPaint (hWnd);
        /* perform different repainting according to the pressed or released status */
        if (status) {
            SetBkMode(hdc, BM_TRANSPARENT);
            TextOut(hdc, 0, 0, "pressed");
        }
        EndPaint(hWnd, hdc);
        return 0;

    case MSG_DESTROY:
        return 0;
    }

    return DefaultControlProc (hWnd, message, wParam, lParam);
}

/* this function registers simple button control class */
BOOL RegisterMybutton (void)
{
    WNDCLASS WndClass;
```

```

    WndClass.spClassName = "mybutton";
    WndClass.dwStyle      = 0;
    WndClass.dwExStyle    = 0;
    WndClass.hCursor      = GetSystemCursor(0);
    WndClass.iBkColor     = PIXEL_lightgray;
    WndClass.WinProc      = MybuttonWindowProc;

    return RegisterWindowClass (&WndClass);
}
/* main window proc */
static int CaptureWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_CREATE:
            /* register the simple button control class */
            RegisterMybutton();
            /* creates an instance of the simple button cotrol class */
            CreateWindow ("mybutton", "", WS_VISIBLE | WS_CHILD, IDC_MYBUTTON,
                30, 50, 60, 20, hWnd, 0);
            break;

        case MSG_CLOSE:
            /* destroys the control and the main window */
            DestroyAllControls (hWnd);
            DestroyMainWindow (hWnd);
            PostQuitMessage (hWnd);
            return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

/* the code below for creating the main window are omitted */

```

### 9.2.4 Tracking Mouse Cursor

Tracking the position of the mouse cursor is one of the tasks the GUI application usually needs to do. For example, the position of the mouse cursor needs to be tracked when the painting program performs painting operation, so that the user can paint in the client area of the window by dragging the mouse.

**MSG\_LBUTTONDOWN**, **MSG\_LBUTTONUP**, and **MSG\_MOUSEMOVE** messages need to be handled for tracking the mouse cursor. Generally, the window procedure begins to track the mouse cursor when **MSG\_LBUTTONDOWN** message occurs, determines the current position of the cursor through handling **MSG\_MOUSEMOVE** message, and ends tracking the mouse cursor when **MSG\_LBUTTONUP** occurs.

The code in List 9.3 is a simple painting program, which allows you to paint arbitrarily in the client area of the window, and clear the screen when the user clicks the right button. The complete code of this program is available in program **painter.c** in the example program package **mg-samples** for this

guide.

List 9.3 painter.c

```
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

static int PainterWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    /* Use static variables to save the running status
     * and the position where the mouse is pressed
     */
    static BOOL bdraw = FALSE;
    static int pre_x, pre_y;

    switch (message) {
    case MSG_LBUTTONDOWN:
        /* enter painting status; capture the mouse and write the position
         * where the mouse is pressed
         */
        bdraw = TRUE;
        SetCapture(hWnd);
        pre_x = LOWORD (lParam);
        pre_y = HIWORD (lParam);
        break;

    case MSG_MOUSEMOVE:
    {
        int x = LOWORD (lParam);
        int y = HIWORD (lParam);

        if (bdraw) {
            /* if it is in painting status, the moused is indicated to be captured,
             * and therefore the mouse coordinates need to be converted from
             * screen coordinate to client coordinate
             */
            ScreenToClient(hWnd, &x, &y);

            /* get the graphics device context of the client area and begin to paint */
            hdc = GetClientDC(hWnd);
            SetPenColor(hdc, PIXEL_red);

            /* paint a straight line from the previous position
             * to the current mouse position
             */
            MoveTo(hdc, pre_x, pre_y);
            LineTo(hdc, x, y);
            ReleaseDC(hdc);
            /* update the previous position with the current mouse position */
            pre_x = x;
            pre_y = y;
        }
        break;
    }

    case MSG_LBUTTONUP:
        /* quit painting status and release the mouse */
        bdraw = FALSE;
        ReleaseCapture();
        break;

    case MSG_RBUTTONDOWN:
        /* click the right button of the mouse to clear the window */
        InvalidateRect(hWnd, NULL, TRUE);
        break;
    }
}
```

```
case MSG_CLOSE:
    DestroyAllControls (hWnd);
    DestroyMainWindow (hWnd);
    PostQuitMessage (hWnd);
    return 0;
}

return DefaultMainWinProc (hWnd, message, wParam, lParam);
}

/* the code below for creating main window are omitted */
```



Fig. 9.3 A simple painting program

The painting flag is set to be **TRUE** when painter handles **MSG\_LBUTTONDOWN** message, so that the program can paint when receiving **MSG\_MOUSEMOVE** message. In order to prevent the mouse from being released in other window and prevent the window procedure from receiving **MSG\_LBUTTONUP** message, painter calls **SetCapture** function to capture the mouse when **MSG\_LBUTTONDOWN** message occurs.

The program performs painting in **MSG\_MOUSEMOVE** message, ends painting when receiving **MSG\_LBUTTONUP**, and calls **ReleaseCapture** to release the mouse capture.

The program calls **InvalidateRect** function to clear the content in the client region when receiving **MSG\_RBUTTONDOWN**.

## 9.3 Event Hook

In common cases, keyboard events and mouse events are transferred from the bottom layer device to the application window procedure in a normal approach for handling. MiniGUI provides a kind of mechanism so that we can capture these events before they are transformed to corresponding message queue and transferred to specific windows. Then there are two choices available: Let the event transferred in the normal approach; or cancel the normal event handling process. This is the hook mechanism. The hook is a callback function in nature, and if the application registers a hook, the system will call this callback function in the midway of transferring messages, and then determine whether to continue to transfer the message according to the return value of the callback function.

The prototype of the hook callback function defined for MiniGUI-Threads and MiniGUI-Standalone is as follows:

```
typedef int (* MSGHOOK)(void* context, HWND dst_wnd, int msg,
                        WPARAM wParam, LPARAM lParam);
```

Wherein **context** is a context data passed in when registering the hook, which may be a pointer; **dst\_wnd** is the target main window of this message; **msg** is the message identifier; **wparam** and **lparam** are the two parameters of the message. The returned value of the hook function determines whether the system continues to transfer the event: continues to transfer the event when returning **HOOK\_GOON**; and stops transferring the event when returning **HOOK\_STOP**.

The application may call the following two functions to register the hook function of the keyboard and the mouse events for MiniGUI-Threads and MiniGUI-Standalone, respectively:

```
MSGHOOK WINAPI RegisterKeyMsgHook (void* context, MSGHOOK hook);
MSGHOOK WINAPI RegisterMouseMsgHook (void* context, MSGHOOK hook);
```

You need only pass the context information and the pointer of the hook callback function when calling these two functions. If you want to unregister

the hook function previously registered, you need only transfer `NULL` in, as described below:

```
int my_hook (void* context, HWND dst_wnd, int msg, WPARAM wParam, LPARAM lParam)
{
    if (...)
        return HOOK_GOON;
    else
        return HOOK_STOP;
}

MSGHOOK old_hook = RegisterKeyMsgHook (my_context, my_hook);

...

/* Restore old hook */
RegisterKeyMsgHook (0, old_hook);
```

Event hook mechanism is very import to some applications, for example, you can use keyboard hook when the application needs to capture some global keys.

When you handle the event hook under MiniGUI-Threads, the following fact must be noted:

**[NOTE] The hook callback function is called by the desktop thread, i.e., the hook callback function is executed in the desktop thread, therefore, you can not send messages to other threads through `SendMessage` in the hook callback function, which may cause the occurrence of a possible deadlock.**

Besides the hook mechanism for MiniGUI-Threads and MiniGUI-Standalone described above, MiniGUI provides a different hook mechanism for MiniGUI-Processes.

For a client of MiniGUI-Processes, you can use the following functions to register a hook window:

```
HWND WINAPI RegisterKeyHookWindow (HWND hwnd, DWORD flag);
HWND WINAPI RegisterMouseHookWindow (HWND hwnd, DWORD flag);
```

Wherein `hwnd` is the handle to a window of a client, `flag` indicates whether stop or continue handling the hooked messages: `HOOK_GOON` to continue and

`HOOK_STOP` to stop.

After registering a key/mouse hook window, the server of MiniGUI-Processes will post the key/mouse messages to the window of the client first. It will determine whether stop handling the messages or continue the normal handling by the `flag` of the hook.

MiniGUI provides another hook mechanism for the server of MiniGUI-Processes. The event can be received in the sever process, captured before it is transferred to its desktop window or a certain client for further handling. On the event-transferring path, the position of this hook is earlier than that of the hook mentioned above.

The prototype of the hook callback function prepared for the sever process of MiniGUI-Processes is:

```
typedef int (* SRVEVTHOOK) (PMSG pMsg);
```

Here, `pMsg` is the pointer to the message structure to be transferred. The server can modify arbitrarily the value in the message structure, which this pointer points to. The sever process will continue the normal handling of the event when the callback function returns `HOOK_GOON`, and cancels the handling when `HOOK_STOP` is returned.

`Mginit` program can register hook function in the system through the following function:

```
SRVEVTHOOK GUIAPI SetServerEventHook (SRVEVTHOOK SrvEvtHook);
```

This function returns the old hook function.

By using the hook function, we can monitor the idle of the system, and start up the screen saver when the system idle time is up to the specified value.





## 10 Icon, Cursor, and Caret

### 10.1 Icon

An icon is a small image, and is usually used to present an application, or used in the windows such as warning message box. It consists of a bitmap and a mask bitmap, and can produce transparent image region. An icon file may include more than one icon image, and the application can choose an appropriate one to use according to the size and color bit number of each icon image.

MiniGUI provides the support to load, display, create and destroy of mono-color, 16-color and 256-color Windows icons.

#### 10.1.1 Loading and Displaying Icon

The application can use the `LoadIconFromFile` function to load an icon file, and the prototype of the `LoadIconFromFile` function is as follows:

```
HICON GUIAPI LoadIconFromFile (HDC hdc, const char* filename, int which);
```

The argument meanings are as follow:

- `hdc`: The device context.
- `filename`: The file name of the icon file
- `which`: The index value of the selected icon.

`LoadIconFromFile` function loads an icon from a Windows icon file (\*.ico), and the icon can be mono-color, 16-color or 256-color icon. Some Windows icon files contain two icons in different sizes. You can tell this function to load which icon though the argument of `which`, 0 for the first icon, and 1 for the second icon. `LoadIconFromFile` function reads in the information of the icon size, color bit number and bitmap image pixel data, etc., calls `CreateIcon` function to create an icon object, and then returns an icon handle which

presents the loaded icon object.

You can also loads an icon from a memory area by `LoadIconFromMem` function, the prototype of which is as follows:

```
HICON GUIAPI LoadIconFromFile (HDC hdc, const void* area, int which);
```

The memory area pointed to by `area` must have the same layout as a Windows ICO file.

After the application has loaded the icon object, you can call `DrawIcon` function to draw an icon in the specified position. `DrawIcon` function draws an icon object in a rectangle region. The prototype of `DrawIcon` function is as follows:

```
void GUIAPI DrawIcon (HDC hdc, int x, int y, int w, int h, HICON hicon);
```

The argument meanings are as follow:

- `hdc`            The device context.
- `x, y`            The x and y coordinates of the upper-left corner of the rectangle.
- `w, h`            The width and height of the rectangle, respectively
- `hicon`           The handle of the icon object.

The code in List 10.1 illustrates how to load an icon from an icon file, and then draw an icon in the client area of the window. The complete code of this program can be referred to program `drawicon.c` in sample program package `mg-samples` for this guide.

List 10.1 Loading and drawing an icon

```
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

static int DrawiconWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
```

```

static HICON mvicon_small, mvicon_large;
HDC hdc;

switch (message) {
case MSG_CREATE:
    /* call LoadIconFromFile function to load two icons with different size
     * from myicon.ico file.
     */
    myicon_small = LoadIconFromFile(HDC_SCREEN, "myicon.ico", 0);
    if (myicon_small == 0)
        fprintf(stderr, "Load icon file failure!");
    myicon_large = LoadIconFromFile(HDC_SCREEN, "myicon.ico", 1);
    if (myicon_large == 0)
        fprintf(stderr, "Load icon file failure!");
    break;

case MSG_PAINT:
    /* display two icons in different positions of the window */
    hdc = BeginPaint(hWnd);
    if (myicon_small != 0)
        DrawIcon(hdc, 10, 10, 0, 0, myicon_small);
    if (myicon_large != 0)
        DrawIcon(hdc, 60, 60, 0, 0, myicon_large);
    EndPaint(hWnd, hdc);
    return 0;

case MSG_CLOSE:
    /* destroys the icon and the main window itself */
    DestroyIcon(myicon_small);
    DestroyIcon(myicon_large);
    DestroyMainWindow(hWnd);
    PostQuitMessage(hWnd);
    return 0;
}

return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

/* the code below for creating main window are omitted */

```

The output of the program is as shown in Fig. 10.1.

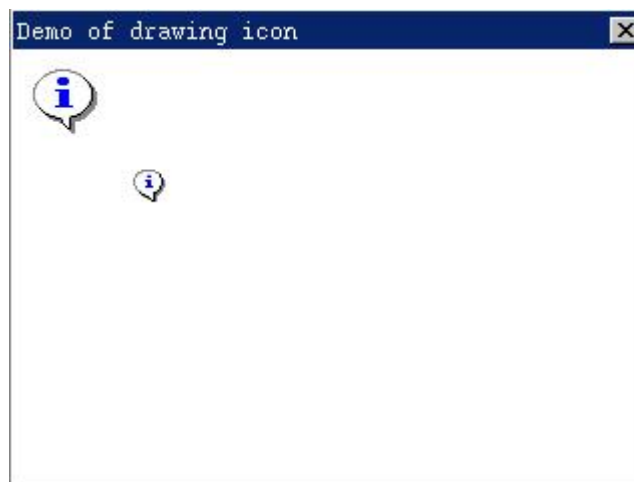


Fig. 10.1 Drawing icon

The program described above loads two 16-color icons of 32x32 pixels and 16x16 pixels, respectively. The icon with number 0 is the big icon with 32x32

pixels, and is saved in the icon object `myicon_large`; and the icon with number 1 is the small one with 16x16 pixels, and is saved in the icon object `myicon_small`. The program uses `DrawIcon` function to draw these two icons in the client area of the window when handling `MSG_PAINT` message. Note that `w` and `h` arguments passed to `DrawIcon` function are both zero when calling it, and in this case `DrawIcon` function will draw the icon according to its original size without scaling.

### 10.1.2 Destroying Icon

The application should destroy an icon when the application does not need it any more. The program above destroys the two icons previously loaded using `LoadIconFromFile` function by calling `DestroyIcon` function when it is going to quit (when handling `MSG_CLOSE` message). `DestroyIcon` function destroys the icon handle, and releases the memory it used. This function's prototype is as follows:

```
BOOL GUIAPI DestroyIcon(HICON hicon);
```

`DestroyIcon` function has only one parameter `hicon`, which specify the icon object to be destroyed. The icon loaded by `LoadIconFromFile` function needs to be destroyed by `DestroyIcon` function. We will see that `DestroyIcon` can also destroy the icon created dynamically by the application using `CreateIcon` function.

### 10.1.3 Creating Icon

Besides loading an icon from an icon file, the application can also use `CreateIcon` function to create an icon dynamically at run time. The icon created by this function also needs to be destroyed by calling `DestroyIcon` function. The prototype of `CreateIcon` function is as follows:

```
HICON GUIAPI CreateIcon (HDC hdc, int w, int h, const BYTE* pAndBits,
                        const BYTE* pXorBits, int colornum)
```

The argument meanings are as follow:

- **hdc**: The device context.
- **w, h**: The width and height of the icon, respectively.
- **pAndBits**: The pointer to the AND bits of the icon.
- **pXorBits**: The pointer to the XOR bits of the icon.
- **colormap**: The bit-per-pixel of XOR bits.

**CreateIcon** creates an icon according to the specified data such as icon size, color and the bit-mask bitmap image. The icon width and height specified by arguments of **w** and **h** respectively must be sizes supported by the system, for example 16x16 pixels or 32x32 pixels. **pAndBits** points to a byte array, which comprises the image data of the AND bit-mask bitmap of the icon, and the AND bit-mask bitmap should be mono-color. **pXorBits** points to a byte array, which comprises the image data of the XOR bit-mask bitmap of the icon, and the XOR bit-mask bitmap can be a mono-color or colorful. MiniGUI currently supports mono-color icon, 16-color icon and 256-color icon. **colormap** specifies the number of color bits of the icon, i.e., the number of the color bits of the XOR bit-mask bitmap. For mono-color icon, it should be 1, for 16-color icon, it should be 4, for 256-color icon, it should be 8.

The code in List 10.2 describes how to use **CreateIcon** function to create a user-defined icon during runtime. The complete code of this program is available in program **createicon.c** in the example program package **mg-samples**.

List 10.2 Creating an icon

```

/*
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

/* define the AND mask data and XOR mask data of the icon */
static BYTE ANDmaskIcon[] = {
    0xff, 0x9f, 0x00, 0x00,
    0xff, 0x1f, 0x00, 0x00,
    0xfc, 0x1f, 0x00, 0x00,
    0xf0, 0x1f, 0x00, 0x00,
    0xe0, 0x0f, 0x00, 0x00,
    0xc0, 0x07, 0x00, 0x00,
    0x80, 0x03, 0x00, 0x00,
    0x80, 0x03, 0x00, 0x00,
    0x00, 0x01, 0x00, 0x00,
    0x00, 0x01, 0x00, 0x00,
    0x00, 0x01, 0x00, 0x00,

```

```

0x00, 0x01, 0x00, 0x00,
0x80, 0x03, 0x00, 0x00,
0x80, 0x03, 0x00, 0x00,
0xc0, 0x07, 0x00, 0x00,
0xf0, 0x1f, 0x00, 0x00
};

static BYTE XORmaskIcon[] = {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x0f, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0xff, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x8f, 0xff, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x8f, 0xff, 0xff, 0x00, 0x00, 0x00,
    0x00, 0x08, 0xff, 0xf8, 0xff, 0xf8, 0x00, 0x00,
    0x00, 0xff, 0xff, 0x80, 0x8f, 0xff, 0xf0, 0x00,
    0x00, 0xff, 0xff, 0xf8, 0xff, 0xff, 0xf0, 0x00,
    0x0f, 0xff, 0xff, 0xf0, 0xff, 0xff, 0xff, 0x00,
    0x0f, 0xff, 0xff, 0xf8, 0x00, 0xff, 0xff, 0x00,
    0x0f, 0xff, 0xf0, 0x0f, 0x00, 0xff, 0xff, 0x00,
    0x00, 0xff, 0xf8, 0x00, 0x00, 0x08, 0xff, 0xf0, 0x00,
    0x00, 0x8f, 0xff, 0x80, 0x8f, 0xff, 0xf0, 0x00,
    0x00, 0x00, 0x8f, 0xff, 0xff, 0xf0, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0xff, 0x9f, 0x00, 0x00, 0x00, 0xff, 0x1f, 0x00, 0x00,
    0xfc, 0x1f, 0x00, 0x00, 0xf0, 0x1f, 0x00, 0x00,
    0xe0, 0x0f, 0x00, 0x00, 0xc0, 0x07, 0x00, 0x00,
    0x80, 0x03, 0x00, 0x00, 0x80, 0x03, 0x00, 0x00,
    0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00,
    0x00, 0x01, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00,
    0x80, 0x03, 0x00, 0x00, 0x80, 0x03, 0x00, 0x00,
    0xc0, 0x07, 0x00, 0x00, 0xf0, 0x1f, 0x00, 0x00,
    0x26, 0x00, 0x00, 0x00, 0xf4, 0xd9, 0x04, 0x08,
    0xa8, 0xf8, 0xff, 0xbf, 0xc0, 0xf7, 0xff, 0xbf,
    0x20, 0x00, 0x00, 0x00, 0x10, 0x00, 0x00, 0x00,
    0xc0, 0x00, 0x00, 0x00, 0x0e, 0x03, 0x00, 0x00,
    0x28, 0x01, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00,
    0x10, 0x00, 0x00, 0x00, 0x10, 0x00, 0x00, 0x00,
    0xf0, 0x10, 0x04, 0x00, 0x70, 0xe1, 0x04, 0x08,
    0xd8, 0xf8, 0xff, 0xbf, 0x41, 0x90, 0x04, 0x08
};

static int CreateIconWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    static HICON new_icon;
    HDC hdc;

    switch (message) {
        case MSG_CREATE:
            /* create an icon using the user-defined data */
            new_icon = CreateIcon(HDC_SCREEN, 16, 16, ANDmaskIcon, XORmaskIcon, 4);
            break;

        case MSG_PAINT:
            hdc = BeginPaint(hWnd);
            if (new_icon != 0) {
                /* display the icon with its actual size */
                DrawIcon(hdc, 0, 0, 0, 0, new_icon);
                /* display the scaled icon */
                DrawIcon(hdc, 50, 50, 64, 64, new_icon);
            }
            EndPaint(hWnd, hdc);
            return 0;

        case MSG_CLOSE:
            /* destroy the icon and the main window */
            DestroyIcon(new_icon);
            DestroyMainWindow(hWnd);
            PostQuitMessage(hWnd);
            return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

/* the code below for creating main window are omitted */

```

The output of the program is as shown in Fig. 10.2.

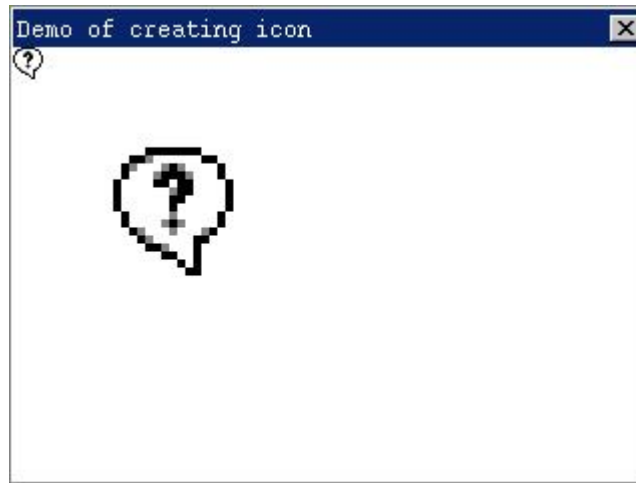


Fig. 10.2 Creating and drawing icon

The code in List 10.2 creates a user-defined icon `new_icon` by calling `CreateIcon` according to the data in the `ANDmaskIcon` and `XORmaskIcon` bit-mask byte array. The size of the icon is 16x16 pixels, and the number of color bits is 4. And then the program draws the created “interrogation” icon in the client area of the window with its original size and scaled size. Finally, the program calls `DestroyIcon` function to destroy the user-defined icon created by `CreateIcon` function when handling `MSG_CLOSE` message.

#### 10.1.4 Using System Icons

The `iconinfo` section in the configuration file `MiniGUI.cfg` of MinGUI defines all the icons the system uses and provides, shown as follow:

```
[iconinfo]
# Edit following line to specify icon files path
iconpath=/usr/local/lib/minigui/res/icon/
# Note that max number defined in source code is 7.
iconnumber=7
icon0=form.ico
icon1=w95mbx01.ico
icon2=w95mbx02.ico
icon3=w95mbx03.ico
icon4=w95mbx04.ico
# default icons for TREEVIEW control
icon5=fold.ico
icon6=unfold.ico
```

**[Note] The maximum number of the icon files the system uses is 7, which is defined in the source code of MiniGUI. Therefore, if you modify the iconnumber item in the configuration file, its value should be smaller than or equal to 7, and the icons with number larger than 7 will be ignored.**

MiniGUI loads all the system icons from icon files to memory according to the setting of the `iconinfo` section in the configuration file when initializing MiniGUI. The application can get the built-in system icons through `GetLargeSystemIcon` function and `GetSmallSystemIcon` function. The prototypes of these two functions are as follow:

```
HICON GUIAPI GetLargeSystemIcon (int id);
HICON GUIAPI GetSmallSystemIcon (int id);
```

`GetLargeSystemIcon` is used to get a large system icon with 32x32 pixels, and `GetSmallSystemIcon` is used to get a small system icon with 16x16 pixels. These two functions return the handle of the built-in system icon object. The obtained icon is one of the seven system icons, which is specified by `id`. `id` is an integer value, and may be one of the following values:

<code>IDI_APPLICATION</code>	Application icon
<code>IDI_STOP / IDI_HAND</code>	Stoping icon
<code>IDI_QUESTION</code>	Question icon
<code>IDI_EXCLAMATION</code>	Exclamation icon
<code>IDI_INFORMATION / IDI_ASTERISK</code>	Information icon

These identifiers are defined in `minigui/window.h` as follow:

```
#define IDI_APPLICATION      0
#define IDI_HAND             1
#define IDI_STOP             IDI_HAND
#define IDI_QUESTION         2
#define IDI_EXCLAMATION      3
#define IDI_ASTERISK         4
#define IDI_INFORMATION      IDI_ASTERISK
```

It can be seen that they present the five icon files with sequence number from 0 to 4 in `MiniGUI.cfg`. The icon files with index numbers 5 and 6 are used by TREEVIEW control.

The icons obtained by `GetLargSystemIcon` and `GetSamllSystemIcon` functions



are system-predefined icons, which are system sharing resources and do not need to be destroyed by the application.

In addition, the application may also use `LoadSystemIcon` function to load the required icon directly from the icon file defined in `MiniGUI.cfg` configuration file:

```
HICON GUIAPI LoadSystemIcon (const char* szItemName, int which);
```

Here `szItemName` argument specifies the symbol name of the icon file defined in the `iconinfo` section of `MiniGUI.cfg`, for example, `icon0` presents `form.ico` icon file. The argument of `which` specifies which icon is to be loaded. This function returns the handle to the obtained icon object.

Actually, `LoadSystemIcon` loads an icon by calling `LoadIconFromFile`. Apparently, the icon created by `LoadSystemIcon` must be destroyed using `DestroyIcon` function when it is not be required any more.

## 10.2 Cursor

The pointing device such as mouse controls a cursor, which is a small bitmap and its position in the screen. It is used to indicate the position of the pointing device. When the user moves the mouse, the cursor will move on the screen accordingly. If the cursor moves into different region or different window, the system possibly changes the shape of the cursor. A pixel called hotspot in the cursor indicates the accurate position of the cursor on the screen. The system uses this point to trace and identify the cursor position. For example, the hotspot of a arrow cursor is commonly its arrowhead position. The cursor hotspot is commonly the cursor focus. If a mouse input event occurs, the system will send the mouse containing the hotspot coordinates to the window, which the hotspot is located in, or to the window, which captures the mouse.

MiniGUI provides functions for loading, creating, displaying, destroying, and moving mono- and 16-color cursors. Currently, MiniGUI does not support

256-color or animation cursor.

### 10.2.1 Loading and Creating Cursor

The application may use **LoadCursorFromFile** function to load a cursor from a Windows cursor file. The prototype of this function is as follows:

```
HCURSOR WINAPI LoadCursorFromFile (const char* filename);
```

**LoadCursorFromFile** function reads in information of the size of the cursor, the hotspot position, the number of color bits, and the bitmap image data, etc., creates a cursor object, and returns a cursor handle presenting this cursor object.

**LoadCursorFromMem** function loads a cursor from memory:

```
HCURSOR WINAPI LoadCursorFromMem (const void* area);
```

This function loads a cursor from a specified memory area, the cursor memory area to which area points should be the same as the layout of the Windows cursor file.

The application may also create dynamically a cursor by calling **CreateCursor** function during run time. The prototype of **CreateCursor** function is as follows:

```
HCURSOR WINAPI CreateCursor (int xhotspot, int yhotspot, int w, int h,  
                             const BYTE* pANDBits, const BYTE* pXORBits, int colornum);
```

The argument meanings are as follow:

- **xhotspot, yhotspot**: the horizontal and vertical positions of the cursor hotspot
- **w, h**: the width and height of the cursor
- **pAndBits**: The pointer to the AND bits of the cursor.
- **pXorBits**: The pointer to the XOR bits of the cursor.

- **colornum**: The bit-per-pixel of XOR bits.

Similar to the method of creating an icon by `CreateIcon` function, `CreateCursor` function creates a cursor according to the data of specified cursor size, color and bit-mask bitmap image in memory. The difference is that the hotspot position of the cursor to be created must be specified when using `CreateCursor` function. `xhotspot` and `yhotspot` arguments specify the horizontal and vertical positions of the hotspot of the created cursor, respectively. The cursor width and height specified by `w` and `h` must be the size supported by the system, and MiniGUI can only use cursor of 32x32 pixels. Therefore, the value of `w` and `h` arguments can only be 32. `pAndBits` points to a byte array of a AND bit-mask bitmap image data containing a cursor, and AND bit-mask bitmap is a mono-color bitmap. `pXorBits` points to a byte array of a XOR bit-mask bitmap image data containing a cursor, and XOR bit-mask bitmap may be either a mono-color bitmap or a color bitmap. `colornum` specifies the number of color bits of the cursor, or the number of color number of the XOR bit-mask bitmap. For a mono-color cursor, it should be 1, and for 16-color cursor, it should be 4.

### 10.2.2 Destroying Cursor

The application should destroy the cursor when not requiring it any more. `DestroyCursor` function can destroy the cursor created by `LoadCursorFromFile` and `CreateCursor` functions, and release the memory used by the cursor object. The prototype of this function is as follows:

```
BOOL WINAPI DestroyCursor (HCURSOR hcsr);
```

The argument `hcursor` of `DestroyCursor` function specifies the cursor object to be destroyed.

### 10.2.3 Positioning and Displaying Cursor

If the system includes a mouse, then the system will display it automatically, and update its position on the screen according to the position of mouse and

repaint the cursor. The application can get the current screen position of the cursor by calling `GetCursorPos` function, and move the cursor to the specified position on the screen by calling `SetCursorPos` function:

```
void WINAPI GetCursorPos (POINT* ppt);  
void WINAPI SetCursorPos (int x, int y);
```

The application can get the current cursor handle by calling `GetCurrentCursor` function, and set the current cursor by calling `SetCursorEx` function. The prototypes of these two functions are as follow:

```
HCURSOR WINAPI GetCurrentCursor (void);  
HCURSOR WINAPI SetCursorEx (HCURSOR hcsr, BOOL set_def);
```

`SetCursorEx` function sets the cursor specified by `hcsr` as the current cursor. If `set_def` is TRUE, `SetCursorEx` will further set this cursor as the default cursor, which is the cursor displayed when moved the mouse to the desktop. This function returns old cursor handle. `SetCursorEx` has other two simplified versions: `SetCursor` and `SetDefaultCursor`. `SetCursor` sets the current cursor and does not change the default cursor; and `SetDefaultCursor` sets the specified cursor as the current cursor and the default cursor:

```
#define SetCursor(hcsr) SetCursorEx (hcsr, FALSE);  
#define SetDefaultCursor(hcsr) SetCursorEx (hcsr, TRUE);
```

MiniGUI will send `MSG_SETCURSOR` to the window under the cursor when the user moves the mouse, and the application can change the current cursor when handling `MSG_SETCURSOR` message. If the window procedure function changes the cursor when handling this message, it should be returned immediately.

The system displays the class cursor related to the control where the cursor is. The application can assign a class cursor to the control class when registering the control class. After registering this control class, each control of this control class has the same class cursor, i.e., the cursor displayed by the system is the same specified cursor when the mouse moves onto these controls. The application can get the current cursor of the specified window

through `GetWindowCursor`, and sets a new window cursor through `SetWindowCursor`.

```
HCURSOR WINAPI GetWindowCursor (HWND hWnd);  
HCURSOR WINAPI SetWindowCursor (HWND hWnd, HCURSOR hNewCursor);
```

The code below is from `listview.c` in the MiniGUI source code, which illustrates how to assign a cursor handle to `hCursor` member of the `WNDCLASS` structure to specify the class cursor for a control class.

```
WNDCLASS WndClass;  
  
WndClass.spClassName = CTRL_LISTVIEW;  
WndClass.dwStyle = WS_NONE;  
WndClass.dwExStyle = WS_EX_NONE;  
WndClass.hCursor = GetSystemCursor (0);  
WndClass.lBkColor = PIXEL_lightwhite;  
WndClass.WinProc = sListViewProc;  
  
return RegisterWindowClass (&WndClass);
```

The class cursor in the code above is the system default cursor gotten by `GetSystemCursor` function, i.e., an arrow cursor. `GetSystemCursor (0)` and `GetSystemCursor (IDC_ARROW)` are the same.

The application can show or hide the cursor by calling `ShowCursor` function.

```
int WINAPI ShowCursor (BOOL fShow);
```

`ShowCursor` function hides the cursor when parameter `fShow` is FALSE, and shows the cursor when `fShow` is TRUE. `ShowCursor` does not change the shape of the current cursor. This function internally uses a cursor showing counter to determine whether showing or hiding the cursor. Each calling of `ShowCursor` function to show the cursor will increase this counter by one, and each calling of `ShowCursor` function to hide the cursor will decrease this counter by one. The cursor is visible only when the counter is larger than or equal to 0.

#### 10.2.4 Clipping Cursor

The application can use `ClipCursor` function to clip the cursor within a certain

rectangle area, which is usually used to correspond to an event in a certain clipping rectangle. The prototype of this function is as follows:

```
void GUIAPI ClipCursor (const RECT* prc);
```

The specified clipping rectangle is pointed by parameter **prc**. If **prc** is **NULL**, **clipCursor** will disable cursor clipping. When **clipCursor** function clips the cursor in a certain rectangular area on the screen, it moves the cursor to the center of the rectangular area.

**GetClipCursor** function gets the current cursor clipping rectangle, and can be used to save the original clipping rectangle before setting a new clipping rectangle and use it to restore the original area when required. The prototype of this function is as follows:

```
void GUIAPI GetClipCursor (RECT* prc);
```

## 10.2.5 Using System Cursors

The **cursorinfo** section in the configuration file of MiniGUI, **MiniGUI.cfg**, defines all the cursors provide by the system, as shown below:

```
[cursorinfo]
# Edit following line to specify cursor files path
cursorpath=/usr/local/lib/minigui/res/cursor/
cursornumber=23
cursor0=d_arrow.cur
cursor1=d_beam.cur
cursor2=d_pencil.cur
cursor3=d_cross.cur
cursor4=d_move.cur
cursor5=d_sizenew.cur
cursor6=d_sizens.cur
cursor7=d_sizenwse.cur
cursor8=d_sizewe.cur
cursor9=d_uparrow.cur
cursor10=d_none.cur
cursor11=d_help.cur
cursor12=d_busy.cur
cursor13=d_wait.cur
cursor14=g_rarrow.cur
cursor15=g_col.cur
cursor16=g_row.cur
cursor17=g_drag.cur
cursor18=g_nodrop.cur
cursor19=h_point.cur
cursor20=h_select.cur
cursor21=ho_split.cur
cursor22=ve_split.cur
```

The maximum number of the cursors used by the system defined in MiniGUI is  $(\text{MAX\_SYSCURSORSINDEX} + 1)$ . **MAX\_SYSCURSORSINDEX** is the maximum system cursor index value, defined as 22; therefore the maximum cursor number predefined by the system is 23.

MiniGUI loads all the system cursors from the specified cursor files to memory according to the setting of the **cursorinfo** section in the configuration file when initializing the system. The application can get the built-in system cursor through **GetSystemCursor** function. The prototype of this function is as follows:

```
HCURSOR WINAPI GetSystemCursor (int csrid);
```

**GetSystemCursor** function returns the handle to the cursor object in memory. The obtained cursor is one of the possible 23 system system-predefined cursors, and is specified by identifier **csrid**. Parameter **csrid** is an integer value, and may be one of the following values:

IDC_ARROW	System default arrow cursor
IDC_IBEAM	'I' shaped cursor, indicating an input filed
IDC_PENCIL	Pencil-shape cursor
IDC_CROSS	Cross cursor
IDC_MOVE	Moving cursor
IDC_SIZENWSE	Sizing cursor, along north-west and south-east
IDC_SIZENESW	Sizing cursor, along north-east and south-west
IDC_SIZEWE	Sizing cursor, along west and east
IDC_SIZENS	Sizing cursor, along north and south
IDC_UPARROW	Up arrow cursor
IDC_NONE	None cursor
IDC_HELP	Arrow with question
IDC_BUSY	Busy cursor
IDC_WAIT	Wait cursor
IDC_RARROW	Right arrow cursor
IDC_COLOMN	Cursor indicates column
IDC_ROW	Cursor indicates row
IDC_DRAG	Draging cursor
IDC_NODROP	No dropping cursor, used in dragging operation
IDC_HAND_POINT	Hand point cursor
IDC_HAND_SELECT	Hand selection cursor
IDC_SPLIT_HORZ	Horizontal splitting cursor
IDC_SPLIT_VERT	Vertical splitting cursor

The definitions of these cursor index values are as follow:

```
/* System cursor index. */
#define IDC_ARROW      0
#define IDC_IBEAM      1
#define IDC_PENCIL     2
#define IDC_CROSS      3
#define IDC_MOVE       4
```

```
#define IDC_SIZENWSE 5
#define IDC_SIZENESW 6
#define IDC_SIZEWE 7
#define IDC_SIZENS 8
#define IDC_UPARROW 9
#define IDC_NONE 10
#define IDC_HELP 11
#define IDC_BUSY 12
#define IDC_WAIT 13
#define IDC_RARROW 14
#define IDC_COLOMN 15
#define IDC_ROW 16
#define IDC_DRAG 17
#define IDC_NODROP 18
#define IDC_HAND_POINT 19
#define IDC_HAND_SELECT 20
#define IDC_SPLIT_HORZ 21
#define IDC_SPLIT_VERT 22
```

They present the 23 system-predefined cursors with sequence number from 0 to 22 in MiniGUI, respectively.

The cursor gotten through `GetSystemCursor` function is the system-predefined cursor, and belongs to the system sharing resource; therefore it needs not to be destroyed by the application.

### 10.2.6 Sample Program

The code in List 10.3 illustrates the use of the cursor in MiniGUI. The complete code of this program is available in program `cursordemo.c` in the sample program package `mg-samples`.

List 10.3 Using cursor

```
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

#define IDC_TRAP 100

static HWND hTrapWin, hMainWnd;
static RECT rcMain, rc;

/* window procedure of "trap" control class */
static int TrapwindowProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    static BOOL bTrapped = FALSE;

    switch (message) {
        case MSG_MOUSEMOVE:
            /* further clipping in the range of the control
             * when the mouse enters the range of this control
             */
            if (!bTrapped) {
```



```

        GetWindowRect(hWnd, &rc);
        ClientToScreen(hMainWnd, &rc.left, &rc.top);
        ClientToScreen(hMainWnd, &rc.right, &rc.bottom);
        ClipCursor(&rc);
        bTrapped = TRUE;
    }
    break;

case MSG_DESTROY:
    return 0;
}

return DefaultControlProc(hWnd, message, wParam, lParam);
}

/* register "trap" control class */
BOOL RegisterTrapwindow (void)
{
    WNDCLASS WndClass;

    WndClass.spClassName = "trap";
    WndClass.dwStyle      = 0;
    WndClass.dwExStyle    = 0;
    WndClass.hCursor      = GetSystemCursor(IDC_HAND_POINT);
    WndClass.hIcon         = 0;
    WndClass.hIconSm      = 0;
    WndClass.lpszMenuName  = 0;
    WndClass.lpszName      = 0;
    WndClass.lpfnWndProc   = TrapwindowProc;
    WndClass.hInstance     = 0;
    WndClass.hbrBackground = 0;

    return RegisterWindowClass (&WndClass);
}

static int CursordemoWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_CREATE:
            /* register "trap" control class */
            RegisterTrapwindow();
            /* create an instance of "trap" control class */
            hTrapWin = CreateWindow("trap", "", WS_VISIBLE | WS_CHILD, IDC_TRAP,
                                   10, 10, 100, 100, hWnd, 0);
            break;

        case MSG_LBUTTONDOWN:
            /* Clip the cursor to the main window when the user click the left mouse button. */
            /
            GetWindowRect(hWnd, &rcMain);
            ClipCursor(&rcMain);
            /* hide the mouse cursor */
            ShowCursor(FALSE);
            break;

        case MSG_RBUTTONDOWN:
            /* show the mouse cursor when the right mouse button clicked */
            ShowCursor(TRUE);
            break;

        case MSG_SETCURSOR:
            /* set the mouse cursor shape to be "I" shape */
            SetCursor (GetSystemCursor (IDC_IBEAM));
            return 0;

        case MSG_CLOSE:
            /* destroy the control and the main window itself */
            DestroyAllControls (hWnd);
            DestroyMainWindow (hWnd);
            PostQuitMessage (hWnd);
            return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

/* the code below for creating main window are omitted */

```

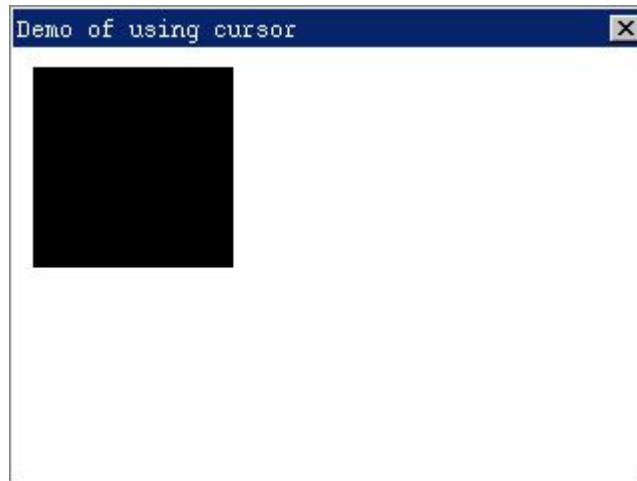


Fig. 10.3 Using cursor

The user interface of the sample program above is shown in Fig. 10.3. The procedure of the main window registers a window class "trap" by calling `RegisterTrapWindow` function, and then creates a child window of "trap" class with size of 100x100 in the up-left corner (10, 10). `RegisterTrapWindow` function sets the background color of the window to be black and sets the window class cursor to be hand-point cursor when registering the "trap" window class. The window procedure of trap window class clips the cursor within this window by calling `ClipCursor` function when handling `MSG_MOUSEMOVE` message.

When handling `MSG_LBUTTONDOWN` message, the main window procedure first gets the main window rectangle by calling `GetWindowRect`, clips the cursor within the main window rectangle by calling `ClipCursor` function, and then hides the cursor by calling `ShowCursor` function. When handling `MSG_RBUTTONDOWN` message, the main window procedure shows the cursor by calling `ShowCursor` function. The main window procedure resets the current cursor (arrow cursor) to I-shaped cursor.

During program running, the moving of the cursor is clipped within the main window and is hidden when the user clicks the mouse left button in the main window. The user can show the cursor again by clicking the right mouse button. When the cursor is moved onto the trap window, it will be "grasped" within this black window.

## 10.3 Caret

The caret is a blinking character in the window client area, and is normally used to indicate the position of keyboard input. Usual shapes of the caret are an underline, a vertical line, a block, and so on.

MiniGUI provides the application with functions of creating, destroying, showing, hiding, positioning the caret, and changing the caret blinking time.

### 10.3.1 Creating and Destroying Caret

**CreateCaret** function creates a caret and assigns it to a specified window.

```
BOOL WINAPI CreateCaret (HWND hWnd, PBITMAP pBitmap, int nWidth, int nHeight);
```

The argument meanings are as follow:

- **hWnd**: The owner of the caret.
- **pBitmap**: The bitmap shape of the caret.
- **nWidth**: The width of the caret.
- **nHeight**: The height of the caret.

If **pBitmap** is not NULL, **CreateCaret** function will create a caret according to the bitmap object. If **pBitmap** is NULL, **CreateCaret** function will create the caret by a rectangle with **nWidth** wide and **nHeight** high. The width **nWidth** and height **nHeight** of the caret are in pixels.

The caret is hidden after being created. To show the caret, you must call **ShowCaret** function to show the caret on the screen after creating it by calling **CreateCaret** function.

**DestroyCaret** function destroys the caret created by **CreateCaret** function, and its prototype is as follows:

```
BOOL WINAPI DestroyCaret (HWND hWnd);
```

**DestroyCaret** function destroys the caret of a window, and deletes it from the screen.

We can create a caret by calling **CreateCaret** in **MSG\_CREATE** message, and then destroy it by calling **DestroyCaret** function when receiving **MSG\_DESTROY** message.

### 10.3.2 Showing and Hiding Caret

At a time, only one window has the keyboard input focus. Usually, the window, which receives the keyboard input, shows the caret when receiving the input focus, and hides the caret when losing the input focus.

The system sends **MSG\_SETFOCUS** message to the window that has received the input focus and the application should show the caret by calling **ShowCaret** function when receiving this message. When the window loses the keyboard input focus, the system sends a **MSG\_KILLFOCUS** message to this window, and the application should hide the caret by calling **HideCaret** function when handling this message. The prototypes of these two functions are as follow:

```
BOOL WINAPI ShowCaret (HWND hWnd);  
BOOL WINAPI HideCaret (HWND hWnd);
```

**ShowCaret** function shows the caret of a given window, and the caret will automatically blink after appearing. **HideCaret** function deletes the caret from the screen. If the application must repaint the screen when handling **MSG\_PAINT** message, and must retain the caret, you can use **HideCaret** function to hide the caret before painting, and use **ShowCaret** function to show the caret again after painting. If the message being handled by the application is **MSG\_PAINT** message, you need not hide and show again the caret, because **BeginPaint** and **EndPaint** function will automatically finish these operations.

### 10.3.3 Positioning Caret

The application uses **GetCaretPos** function to get the position of the caret, and

uses `SetCaretPos` function to move the caret within a window.

```
BOOL WINAPI GetCaretPos (HWND hWnd, POINT pPt);  
BOOL WINAPI SetCaretPos (HWND hWnd, int x, int y);
```

The function `GetCaretPos` copies the position of the window caret, in client coordinates, to a `POINT` structure pointed to by `pPt`. `SetCaretPos` function moves the caret of the window to the position in the client area specified by `x` and `y`, regardless whether or not the caret is visible.

### 10.3.4 Changing Blink Time of Caret

The elapsed time for reversing the caret is called reversing time. Blinking time is referred to the elapsed time for showing, reversing, and recovering. The application uses `GetCaretBlinkTime` to get the blink time of the caret, in milliseconds. The blink time of the system default caret is 500 milliseconds. You can use `SetCaretBlinkTime` function to change the blink time of the caret. The blink time of the caret cannot be less than 100 milliseconds. The definitions of these two functions are as follow:

```
UINT WINAPI GetCaretBlinkTime (HWND hWnd);  
BOOL WINAPI SetCaretBlinkTime (HWND hWnd, UINT uTime);
```

### 10.3.5 Sample Program

The code in List 10.4 uses the caret functions discussed in this section to create a simple text input window, which can be considered as a simple edit box control. In “myedit” control, you can input less than 10 characters, move the caret by left and right arrow key (caret moving key), and delete the character in the window with backspace key. The complete code of this program is available in program `caretdemo.c` in the sample program package `mg-samples`.

List 10.4 Using caret

```
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>
```

```
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

#define IDC_MYEDIT    100

/* the window procedure of a simple edit box control class */
static int MyeditWindowProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    /* use a static variable to save the information of the control.
     * in a real control, you should not use static variables to save these informations,
     * because a control class may have many control instances at the same time
     */
    static char *pBuffer = NULL;
    static int pos = 0, len = 0;
    HDC hdc;

    switch (message) {
    case MSG_CREATE:
        /* set the control font to be the system default font */
        SetWindowFont(hWnd, GetSystemFont(SYSLOGFONT_WCHAR_DEF));
        /* create the caret */
        if (!CreateCaret(hWnd, NULL, 1, GetSysCharHeight())) {
            return -1;
        }
        /* allocate the edit buffur */
        pBuffer = (char *) malloc(10);
        *pBuffer = 0;
        break;

    case MSG_SETFOCUS:
        /* set the caret postion when getting the input focus */
        SetCaretPos(hWnd, pos*GetSysCharWidth(), 0);
        /* showing the caret */
        ShowCaret(hWnd);
        break;

    case MSG_KILLFOCUS:
        /* hiding the caret when losing the focus */
        HideCaret(hWnd);
        break;

    case MSG_CHAR:
        switch (wParam) {
            case '\t':
            case '\b':
            case '\n':
            {
                /* change the blink time of the caret when inputting these characters */
                SetCaretBlinkTime(hWnd, GetCaretBlinkTime(hWnd) - 100);
            }
        }
        break;

    default:
        {
            /* insert characters in the buffer area */
            char ch, buf[10];
            char *tmp;
            ch = wParam;
            if (len == 10)
                break;
            tmp = pBuffer+pos;
            if (*tmp != 0) {
                strcpy(buf, tmp);
                strcpy(tmp+1, buf);
            }
            *tmp = ch;
            pos++;
            len++;
            break;
        }
        break;
    }
}
```

```

        break;

    case MSG_KEYDOWN:
        switch (wParam) {
            case SCANCODE_CURSORBLOCKLEFT:
                /* move the caret to the left */
                pos = MAX(pos-1, 0);
                break;
            case SCANCODE_CURSORBLOCKRIGHT:
                /* move the caret to the right */
                pos = MIN(pos+1, len);
                break;
            case SCANCODE_BACKSPACE:
                {
                    /* delete the character where the caret is */
                    char buf[10];
                    char *tmp;
                    if (len == 0 || pos == 0)
                        break;
                    tmp = pBuffer+pos;
                    strcpy(buf, tmp);
                    strcpy(tmp-1, buf);
                    pos--;
                    len--;
                }
                break;
        }
        /* update the caret position and repaint the window */
        SetCaretPos(hWnd, pos*GetSysCharWidth(), 0);
        InvalidateRect(hWnd, NULL, TRUE);
        break;
    case MSG_PAINT:
        hdc = BeginPaint(hWnd);
        /* output text */
        TextOut(hdc, 0, 0, pBuffer);
        EndPaint(hWnd, hdc);
        return 0;

    case MSG_DESTROY:
        /* destroy the caret and release the buffer */
        DestroyCaret(hWnd);
        if (pBuffer)
            free(pBuffer);
        return 0;
    }

    return DefaultControlProc(hWnd, message, wParam, lParam);
}

/* register a simple edit box control */
BOOL RegisterMyedit(void)
{
    WNDCLASS WndClass;

    WndClass.spClassName = "myedit";
    WndClass.dwStyle      = 0;
    WndClass.dwExStyle    = 0;
    WndClass.hCursor      = GetSystemCursor(IDC_IBEAM);
    WndClass.hBkColor     = PIXEL_lightwhite;
    WndClass.WinProc      = MyeditWindowProc;

    return RegisterWindowClass (&WndClass);
}

/* main windoww proc */
static int CaretdemoWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    HWND hMyedit;

    switch (message) {
        case MSG_CREATE:
            /* register simple edit box control class and create an instance */
            RegisterMyedit();
            hMyedit = CreateWindow("myedit", "", WS_VISIBLE | WS_CHILD, IDC_MYEDIT,
                30, 50, 100, 20, hWnd, 0);
            SetFocus(hMyedit);
    }

```

```

        break;

    case MSG_CLOSE:
        /* destroy the control and the main window itself */
        DestroyAllControls (hWnd);
        DestroyMainWindow (hWnd);
        PostQuitMessage (hWnd);
        return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

/* the code below for creating main window are omitted */

```



Fig 10.4 A simple edit box

For simplification, we use fixed-width font in “myedit” because other fonts are more difficult to handle. The window procedure function of `myedit` uses `GetSystemFont (SYSLOGFONT_WCHAR_DEF)` to get the system default fixed-width font, and then set the font of the text input window by calling `SetWindowFont`. `Myedit` control calls `CreatCaret` function in `MSG_CREATE` message to create a caret of width 1 and the same height as the font.

The buffer area pointed to by `pBuffer` is used to save the characters input into the text window; `len` presents the number of characters, and `pos` presents the current edit position of the current caret.

The window procedure of `myedit` calls `ShowCaret` function to show the caret when receiving `MSG_SETFOCUS` message, and calls `HideCaret` to hide the caret when receiving `MSG_KILLFOCUS` message.



`Myedit` handles the input of normal characters in `MSG_CHAR` message, and adjusts the buffer and values of the `pos` and `len` correspondingly. Furthermore, each time `myedit` receives special character such as newline, `myedit` calls `SetCaretBlinkTime` to reduce the reversing time of caret by 100 ms.

`Myedit` handles the left and arrow keys and the backspace key in `MSG_KEYDOWN` message, adjusts the caret position by calling `SetCaretPos` function, paints in `MSG_PAINT`, and destroys the caret when receiving `MSG_DESTROY` message.



## 11 Using MiniGUIExt Library

MiniGUIExt library (libmgext) is an extension of the libminigui that is the core MiniGUI function library. This library is built on the basis of libminigui, mainly including the extension interfaces for some applications as follow:

- Extended control classes, such as tree view control and list view control
- The popular UI wrap functions
- Skin interface support

Before using the interfaces in MiniGUIExt library, you should first call function `InitMiniGUIExt` to initialize this function library. You should call `TermMiniGUIExt` function when you are done in order to release the resources taken up by the mgext library. When the compiler makes the executable, which uses the functions in MiniGUIExt library, you should link `mgext` library (that is, use `-lmgext` option when using `gcc`).

In Part IV of this guide, we will introduce the extension controls provided by MiniGUIExt library. we will introduce other modules of that library in this chapter, including UI wrap functions and the support for skin user interface.

### 11.1 User Interface Wrap Functions

The user interface wrap functions of MiniGUIExt library are previously provided by mywins library. In version 1.2.6, we merged the APIs of mywins library into mgext library.

As we known, the installation program of Red Hat Linux is developed on the basis of character interface function library called `newt`. Newt tries to provide a GUI-like interface under character mode. In the library there are some popular UI elements, including label, button, list box, and edit box. Most of the setup tools in Red Hat Linux are also developed on the basis of newt.

In order to use MiniGUI to realize the newt-like installation program, we develop the libmywins library, and then merge this library into MiniGUIExt library when releasing MiniGUI version 1.2.6. These interfaces can be divided into the following categories:

- Most of them are simple wrap of the existed MiniGUI functions. They can accept variable arguments, and also can format the message text like `sprintf`. The functions include:
  - `createStatusWin/destroyStatusWin`: To create and destroy the state window. State Window is used to display such text as the system is "Coping software package, please wait...".
  - `createToolTipWin/destroyToolTipWin`: To create and destroy tool tip window. Tool tip window usually is a small yellow window, which display some tip information for the user.
  - `createProgressWin/destroyProgressWin`: To create and destroy progress window. Progress window includes a progress bar, which can display the progress information.
  - `myWinHelpMessage`: A dialog box to display the help message in plain text. The box has a spin box can be used to scroll the help message.
- Comprehensive assist functions can be used to receive and return complex input information. The functions include:
  - `myWinMenu`: This function creates a list box with which users can choose one of the option in that. It is similar to the dialog box displayed by the tool of `timeconfig` of Red Hat Linux.
  - `myWinEntries`: This function creates a group of edit boxes to input. It is similar to the interface used to display information such as network IP address and network mask by the tool of `netconfig` of Red Hat Linux.
  - `OpenFileDialog/OpenFileDialogEx`: File open/save dialog box (obsolete).
  - `ShowOpenDialog`: New file open/save dialog box.
  - `ColorSelDialog`: Color selection dialog box.

We will use some examples to help you understand above functions in this section.

The following code calls `myWinHelpMessage` to create a help message dialog box. The dialog box created by this function can be seen in Fig. 11.1.

```
myWinHelpMessage (hwnd, 300, 200,
    "About SpinBox control",
    "We use the SpinBox control in this Help Message Box.\n\n"
    "You can click the up arrow of the control to scroll up "
    "the message, and click the down arrow of the control to scroll down. "
    "You can also scroll the message by typing ArrowDown and ArrowUp keys.\n\n"
    "In your application, you can call 'myWinHelpMessage' function "
    "to build a Help Message box like this.\n\n"
    "The Help Message Box is useful for some PDA-like applications.\n\n"
    "The SpinBox control allways have the fixed width and height. "
    "You can read the source of 'ext/control/spinbox.c' to know how to "
    "build such a control.\n\n"
    "If you want to know how to use this control, please read the "
    "source of 'mywindows/helpwin.c' in the MiniGUI source tree.");
```

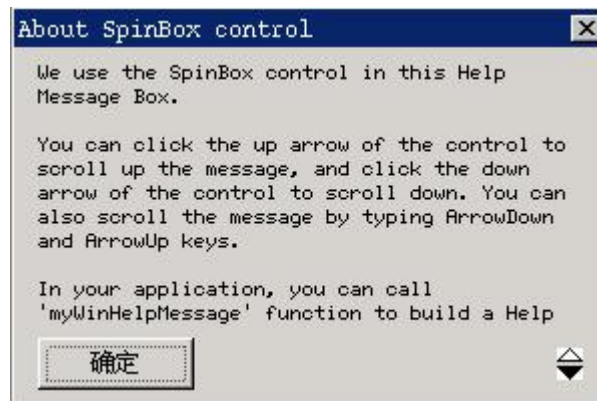


Fig. 11.1 Dialog box created by `myWinHelpMessage`

The code in List 11.1 calls function `myWinEntries` to create the dialog box with two edit boxes. They are used to input new rows and columns of a window. Dialog box created by this function can be seen in Fig. 11.2.

List 11.1 Usage of `myWinEntries`

```
char cols [10];
char rows [10];
char* newcols = cols;
char* newrows = rows;

/* Specify two properties of edit boxes, including labels and initial contents.
/* The structure array end with NULL */
myWINENTRY entries [] = {
    { "列数:" /* Columns */, &newcols, 0, 0 },
    { "行数:" /* Rows */, &newrows, 0, 0 },
    { NULL, NULL, 0, 0 }
};

/* Specify properties of two edit boxes, including labels and initial contents.
/* The structure array end with NULL */
myWINBUTTON buttons[] = {
    { "确认" /* OK */, IDOK, BS_DEFPUSHBUTTON },
    { "取消" /* Cancel */, IDCANCEL, 0 },
```

```

        { NULL, 0, 0}
    };
    int result;

    sprintf (cols, "%d", 80);
    sprintf (rows, "%d", 25);

    /* Call myWinEntries to display interface and return the data input by user */
    result = myWinEntries (HWND_DESKTOP,
        "新记事本大小" /* New size of notepad */,
        "请指定新记事本的窗口大小"
        /* Please specify the new window size of notepad */,
        240, 150, FALSE, entries, buttons);

    /*
     * The input content in two edit boxes by the user
     * will be returned by newcols and newrows
     */
    col = atoi (newcols);
    row = atoi (newrows);

    /* Don't forget to release newcols and newrows because they are allocated
     * by myWinEntries function. */
    free (newcols);
    free (newrows);

    if (result == IDOK) {
        /* Other handling work */
    }
    else
        return;

```

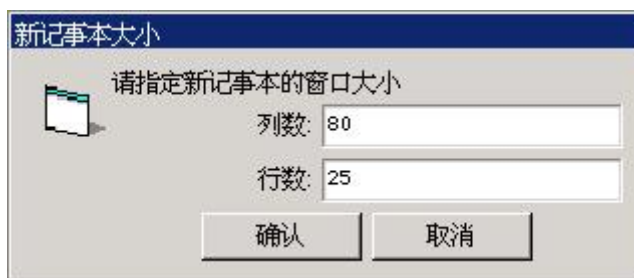


Fig. 11.2 Dialog box created by myWinEntries

## 11.2 Skin

Skin is a mechanism using a series of images to form program user interface. Skin technology allows you to design dashy software user interfaces. The application even can get different appearance by changing among multiple skins.

MiniGUI version 1.3.1 added the support for skin, which allows application to use skin technology to design relatively disengaged software interface. We will introduce how to use the skin of MiniGUIExt library to realize skin interface in this section.

### 11.2.1 Form of skin

Skin interface in MiniGUI is formed mainly by a main skin interface and kinds of skin elements included in skin window. Skin window is the window that skin is attached to. Skin cannot be displayed until attaches to a certain window.

The main skin interface is the place that skin elements attach to. Skin elements mean kinds of interface elements forming skin interface, including button, label, and slider. They are rendered by one or more images.

The following data structure `skin_head_t` is used to define a skin:

```
/** Skin header information structure */
struct skin_head_s
{
    /** The name of the skin */
    char* name;

    /** The style of the skin */
    DWORD style;

    /** Bitmap object array used by skin and skin item */
    const BITMAP* bmps;
    /** Logic font array used by skin */
    const LOGFONT* fonts;

    /** Index of skin background bitmap object in bitmap array */
    int bk_bmp_index;

    /** Number of skin items in skin */
    int nr_items;
    /** Skin item array */
    skin_item_t* items;

    /** Attached data by application */
    DWORD attached;

    // ...
};
typedef struct skin_head_s skin_head_t;
```

Before application creates a skin window, it should use this data structure to define the skin properties of a skin window, including the bitmap objects, the logical fonts, the skin elements, and the callback function.

**Name** is used to define the name of skin; **style** is skin style, currently there is only one style named `SKIN_STYLE_TOOLTIP`, which indicates that skin window has the function of displaying tip information.

**Bmps** points to a bitmap object array, which includes all bitmap objects used by

the skin; `fonts` points to a logical font array, which includes logical fonts used by the skin. Before using `skin_head_t` structure, we need to initialize the bitmaps and fonts resource of these two arrays. For example, load the bitmaps from files. `bk_bmp_index` defines the background bitmap object of skin main interface. It is the index value of the `bmps` array.

`Nr_items` and `items` respectively represent the number of elements and elements array of the skin. `Items` point to a `skin_item_t` array, which defines all skin elements in the skin. Skin elements included in skin should be defined at the same time with the `skin_head_t` structure.

After we use `skin_head_t` structure and `skin_item_t` structure to define the properties of a skin, the skin object is still incomplete. We also need call `skin_init` to initialize skin object, which allows this object to include complete external information and internal data, then we can use this skin in a MiniGUI window:

```
BOOL skin_init (skin_head_t* skin, skin_event_cb_t event_cb, skin_msg_cb_t msg_cb);
```

The arguments of `event_cb` and `msg_cb` point to the event callback function and message callback function of this skin respectively.

If we no longer need a skin object, we can use `skin_deinit` function to destroy it:

```
void skin_deinit (skin_head_t* skin);
```

Skin element is the main constitute part of a skin object. The following data structure `skin_item_t` defines properties of a skin element:

```
/** Skin element information structure */
typedef struct skin_item_s
{
    /** Identity used to identify skin element */
    int id;

    /** Style of skin element */
    DWORD style;
}
```



```

/** x coordinate of skin element in skin interface */
int x;
/** y coordinate of skin element in skin interface */
int y;

/** Hit rectangle of skin element */
RECT rc_hittest;

/** Index of skin element bitmap in skin bitmap array */
int bmp_index;

/** Tip information text */
char* tip;

/** Attached data of application */
DWORD attached;

/* Data defining the certain properties of skin element */
void* type_data;

    // ...
} skin_item_t;

```

**id** is an integer used to identify the skin element. The **id** value can be used to determine an element object in event callback function; **x** and **y** is the place of skin element in skin interface; **rc\_hittest** is the hit-test rectangle of skin element. If a mouse event occurs in the hit-test rectangle, system will emit a skin event corresponding to this skin element.

Almost every skin and skin element included in the skin display its appearance by using images. **bmp\_index** specifies the index value in bitmap object array (**bmps** in **skin\_head\_t** structure). The image resource used by skin and skin elements should be loaded by application into the bitmap object array.

**Type\_data** defines the data of special skin element properties. This pointer usually points to a data structure of skin element property. For example, as for the image label element, **type\_data** is a pointer pointing to a **si\_bmplabel\_t** structure, which gives the essential information of label text and optional text aggregation. If skin element is a MiniGUI control, **type\_data** should point to a structure with type of **CTRLDATA**.

**Attached** is the add-on data of a skin element. Application can store the add-on data related to a certain skin element in this field. The data is private for application, explained and used by application.

**style** specifies the style of a skin element, including the class of skin element,

certain skin element style and the type of hit-test region. Different styles should be combined by using OR operation.

Using corresponding element style in item style specifies the class of skin element. MiniGUI has following pre-defined skin elements:

- `SI_TYPE_NRMLABEL`: Normal label
- `SI_TYPE_BMPLABEL`: Image label
- `SI_TYPE_CMDBUTTON`: Command button
- `SI_TYPE_CHKBUTTON`: Check button
- `SI_TYPE_NRMSLIDER`: Normal slider
- `SI_TYPE_ROTSLIDER`: Rotate slider
- `SI_TYPE_CONTROL`: MiniGUI control

We will describe the usage of these skin elements in the latter of this section.

When the class of skin element is `SI_TYPE_CONTROL`, it will be a MiniGUI control, such as button, static box, or skin child window.

Shape of skin element is specified by the following styles:

- `SI_TEST_SHAPE_RECT`: Rectangle
- `SI_TEST_SHAPE_ELLIPSE`: Ellipse
- `SI_TEST_SHAPE_LOZENGE`: Lozenge
- `SI_TEST_SHAPE_LTRIANGLE`: Isosceles triangle with vertex on the left
- `SI_TEST_SHAPE_RTRIANGLE`: Isosceles triangle with vertex on the right
- `SI_TEST_SHAPE_UTRIANGLE`: Isosceles triangle with vertex on the top
- `SI_TEST_SHAPE_DTRIANGLE`: Isosceles triangle with vertex on the foot

The following specifies status of skin element

- `SI_STATUS_VISIBLE`: Visible
- `SI_STATUS_DISABLED`: Disabled
- `SI_STATUS_HIGHLIGHTED`: Highlighted

When defining a skin element, we should specify its initial status. Furthermore,

a particular skin element may have its particular status definition, and we will illustrate this in the following skin element.

### 11.2.2 Skin Window

Skin window means MiniGUI window including skin, it can be a modeless main window, a modal main window, or a child window (control).

The main difference between a skin main window and a normal MiniGUI main window lies in appearance (skin main window has no caption bar, border and system menu). Events and message callback functions of skin main window are similar to those of normal main window, but slightly different on usage. A skin child window is also a MiniGUI child window (control), and like skin main window, skin child window provides skin event callback function and MiniGUI message callback function.

Use of skin window in MiniGUI is flexible, normal MiniGUI window can comprise skin child window, and skin window also can include normal MiniGUI child window or skin child window. In other words, skin window can be used nestedly.

MiniGUI provides the following functions for creating and destroying skin window:

```
HWND create_skin_main_window (skin_head_t* skin, HWND hosting, int x, int y,  
                             int w, int h, BOOL modal);  
HWND create_skin_control (skin_head_t* skin, HWND parent, int id, int x, int y,  
                         int w, int h);  
void destroy_skin_window (HWND hwnd);
```

The `create_skin_main_window` function is used to create a main window with skin interface, and this main window has no caption bar, border and system menu. The argument `hosting` of the `create_skin_main_window` function specifies the hosting main window of skin window; the arguments of `x`, `y`, `w`, `h` specify the position and size of skin main window; the argument `skin` specifies the skin attached to this main window, and it is a pointer to `skin_head_t`

structure. The structure defines the related data of the skin object, and the skin object should be well initialize using `skin_init` function. If argument `modal` is TRUE, a modal main window is created; otherwise a modeless main window is created.

The `create_skin_control` function is used to create child window with skin interface, or in other words, skin control. The argument `parent` specifies the parent window of skin control; `id` is the control identifier; the arguments `x`, `y`, `w`, `h` specify the position and size of skin control in its parent window.

The `destroy_skin_window` function is used to destroy skin main window or child window created by `create_skin_main_window` or `create_skin_control`. It should be noted that destroying a skin window wouldn't destroy the skin object included by it.

### 11.2.3 Use of callbacks

Similar to window procedure function, callback functions are used to handle skin events and window messages of skin and skin window. When the user moves or clicks the mouse on a skin window, such as clicks a button skin element, the system will send the corresponding skin event to the event callback function, and send the window message-to-message callback function.

The event callback function and message callback of skin are specified by arguments `event_cb` and `msg_cb` when calling function `skin_create_main_window` and function `skin_create_control` to create skin window. These two functions of skin can also be changed by function `skin_set_event_cb` and `skin_set_msg_cb`:

```
skin_event_cb_t skin_set_event_cb (skin_head_t* skin, skin_event_cb_t event_cb);
skin_msg_cb_t skin_set_msg_cb (skin_head_t* skin, skin_msg_cb_t msg_cb);
```

`skin_event_cb_t` is event callback function, its prototype is as follows:

```
typedef int (* skin_event_cb_t) (HWND hwnd, skin_item_t* item, int event, void* data);
```

The argument **hwnd** is the skin window handle of the occurred event; **item** is the skin element, **event** is event type; **data** is event-related data. Usually we can judge which skin element occurs what types of events by the value **id** and **event**.

The currently defined event types are:

- **SIE\_BUTTON\_CLICKED**: click button
- **SIE\_SLIDER\_CHANGED**: the change of slider
- **SIE\_GAIN\_FOCUS**: skin element gain focus
- **SIE\_LOST\_FOCUS**: skin element lost focus

**skin\_msg\_cb\_t** is the type of message callback function, defined as follows:

```
typedef int (* skin_msg_cb_t) (HWND hwnd, int message, WPARAM wparam, LPARAM lparam, int * result);
```

Here the argument **hwnd** is skin window handle occurring message; **message** is message definition; **wparam** and **lparam** are message parameters, and **result** is used to return message related result.

If application program defines the message callback function of skin window, the window procedure function will process the message before calling the message callback function of the skin, then judge if it needs to continue according to the return value of message callback function.

The return value of message callback function includes:

- **MSG\_CB\_GOON**: Skin window procedure function will continue to process the message, result value will be ignored
- **MSG\_CB\_DEF\_GOON**: Message will be processed by procedure function of MiniGUI default window, result value will be ignored
- **MSG\_CB\_STOP**: The process of message will be stopped; skin window procedure function returns the result.

### 11.2.4 Skin Operations

We can use skin operation function to perform a series of general operations to skin or skin elements.

Function `set_window_skin` can change skin attached to a skin window, we can use this function to realize the change of the skin on the fly.

```
skin_head_t* set_window_skin (HWND hwnd, skin_head_t *new_skin);
```

Here `hwnd` is window handle of skin window; `new_skin` is the new skin object, which must have been initialized by function `skin_init`. Function `set_window_skin` returns the old skin object. We need to know that this function does not destroy the old skin object.

`get_window_skin` function is used to get skin included in skin window.

```
skin_head_t* get_window_skin (HWND hwnd);
```

We can use `skin_get_item` function to get a skin element object by passing its identifier:

```
skin_item_t* skin_get_item (skin_head_t* skin, int id);
```

Function `skin_get_item_status` gets the general state of skin element, including visible, disabled and highlighted:

```
DWORD skin_get_item_status (skin_head_t* skin, int id);
```

Function `skin_get_hilited_item` is used to get current highlighted skin element:

```
skin_item_t* skin_get_hilited_item (skin_head_t* skin);
```

Function `skin_set_hilited_item` is used to set current highlighted skin element:

```
skin_item_t* skin_set_hilited_item (skin_head_t* skin, int id);
```

Function **skin\_show\_item** is used to display or hide a skin element:

```
DWORD skin_show_item (skin_head_t* skin, int id, BOOL show);
```

Function **skin\_enable\_item** is used to disable or enable a skin element:

```
DWORD skin_enable_item (skin_head_t* skin, int id, BOOL enable);
```

### 11.2.5 Normal Label

Normal label refers to a text label using appointed logical font to display text. When we use **skin\_item\_t** structure to define a normal label element, item **style** should have **SI\_TYPE\_NRMLABEL** style; item **type\_data** points to a **si\_nrmlabel\_t** structure, which defines a normal label:

```
/** Normal label item info structure */
typedef struct si_nrmlabel_s
{
    /** label text */
    char* label;

    /** label text color in normal state */
    DWORD color;
    /** label text color in focus state */
    DWORD color_focus;
    /** label text color in clicking state */
    DWORD color_click;
    /** Logic font index of label text */
    int font_index;
} si_nrmlabel_t;
```

We can get and set the label string by using **skin\_get\_item\_label** and **skin\_set\_item\_label**:

```
const char* skin_get_item_label (skin_head_t* skin, int id);
BOOL skin_set_item_label (skin_head_t* skin, int id, const char* label);
```

These two functions are also applicable for image label.

### 11.2.6 Image Label

Image label refers to a label using images to display text or other character. When we use `skin_item_t` structure to define an image label element, item `style` should have `SI_TYPE_BMPLABEL`; item `type_data` points to a `si_bmplabel_t` structure, which defines the properties of an image label:

```
/** Bitmap label item info structure */
typedef struct si_bmplabel_s
{
    /** Label text */
    char* label;
    /** label predefined text set */
    const char* label_chars;
} si_bmplabel_t;
```

The string `label` is the text content that will be displayed by the image label; `label_chars` string includes all optional text.

The text on the label is displayed by using images, which are stored in the bitmap object appointed by `bmp_index` item of `skin_item_t` structure. Those images represented by bitmap object need to accord with the following requirements:

Text in the text image ranks horizontally with equidistant. They may have multiple rows, but each row cannot contain more than 20 characters.

Text in the text image should completely accord with those optional text stated in `label_chars`.

Let's use a simple example. If using a digital image label with digital tube style and with content of 21:30, the image comes from a character image. It can be seen in Fig. 11.3.



Fig. 11.3 Text image of a image label

The image label should be defined as follows:



```
si_bmplabel_t timelabel;  
timelabel.label = "21:30";  
label_chars = "0123456789:.";
```

Functions `skin_get_item_label` and `skin_set_item_label` can be used to get and set the label string of an image label.

### 11.2.7 Command Button

Command button is a skin element with similar function to normal button control. It has four states: normal, press-down, highlighted, and disabled. When we use `skin_item_t` structure to define a command button, item `style` should have `SI_TYPE_CMDBUTTON` style; the image represented by item `bmp_index` should include four button shapes, which rank from left to right, respectively representing four states: normal, pressed, highlighted, and disabled. It can be seen in Fig. 11.4:



Fig. 11.4 Command button picture

Command button has a special state - `SI_BTNSTATUS_CLICKED`, it represents that the button has been pressed.

### 11.2.8 Check Button

Check button has a little difference with command button. When clicked, it will toggle between checked or unchecked. It also has four states: normal, pressed, highlighted, and disabled. When we use `skin_item_t` structure to define a check button, item `style` should be `SI_TYPE_CHKBUTTON`; item `bmp_index` represents the image similar to command button.

Check button has a special state - `SI_BTNSTATUS_CHECKED`, means checked.

We can use function `skin_get_check_status` and function `skin_set_check_status` to get and set the current state of a check button:

```
BOOL skin_get_check_status (skin_head_t* skin, int id);
DWORD skin_set_check_status (skin_head_t* skin, int id, BOOL check);
```

### 11.2.9 Normal Slider

Normal slider can be used to indicate the progress information. When we use `skin_item_t` structure to define a normal slider, item `style` should be `SI_TYPE_NRMSLIDER`; item `type_data` points to a `si_nrmslider_t` typed structure, which defines the properties of a normal slider:

```
/** Normal slider item info structure */
typedef struct si_nrmslider_s
{
    /** Slide information */
    sie_slider_t    slider_info;

    /** Slide bitmap index */
    int thumb_bmp_index;
} si_nrmslider_t, si_progressbar_t;
```

`sie_slider_t` structure is used to represent slider information. When defining a normal slider, we should also define the value of its minimum position, maximum position and current position by using `sie_slider_t` structure:

```
/** Slider information structure */
typedef struct sie_slider_s
{
    /** Minimum slide position */
    int min_pos;
    /** Maximum slide positio */
    int max_pos;
    /** current slide positon */
    int cur_pos;
} sie_slider_t;
```

The bitmap of slider is appointed by item `bmp_indx` of `skin_item_t` structure. The bitmap of thumb of the slider is appointed by item `thumb_bmp_index` of `si_nrmslider_t` structure. Both of them are the array index value of skin bitmap.

Normal slider has three styles:

- **SI\_NRMSLIDER\_HORZ**: Horizontal slider
- **SI\_NRMSLIDER\_VERT**: Vertical slider
- **SI\_NRMSLIDER\_STATIC**: Progress slider

If we need a progress bar in horizontal direction, we can define the item style of **skin\_item\_t** structure to be **SI\_NRMSLIDER\_HORZ** | **SI\_NRMSLIDER\_STATIC**.

We can use function **skin\_get\_slider\_info**, function **skin\_set\_slider\_info**, and function **skin\_scale\_slide\_pos** to get and set the normal slider information:

```

BOOL skin_get_slider_info (skin_head_t* skin, int id, sie_slider_t* sie);
BOOL skin_set_slider_info (skin_head_t* skin, int id, const sie_slider_t* sie);
int skin_get_thumb_pos (skin_head_t* skin, int id);
BOOL skin_set_thumb_pos (skin_head_t* skin, int id, int pos);
int skin_scale_slider_pos (const sie_slider_t* org, int new_min, int new_max);

```

Function **skin\_get\_slider\_info** is used to get the value of minimum position, maximum position, and current position of the slider, the result is stored in a **sie\_slider\_t** typed structure; function **skin\_set\_slider\_info** set the position information of the slider; function **skin\_get\_thumb\_pos** and function **skin\_set\_thumb\_pos** are used to set and get the position information of slider; function **skin\_scale\_slider\_pos** is used to calculate the new position of slider within the zoomed scope.

### 11.2.10 Rotate Slider

Rotated slider is similar to normal slider, but its thumb slides along with arc. When we use **skin\_item\_t** structure to define a rotated slider element, item **style** should be **SI\_TYPE\_ROTSLIDER**; item **type\_data** points to a **si\_rotslider\_t** typed structure, which defines the properties of a rotated slider:

```

/** Rotation slider item info structure */
typedef struct si_rotslider_s
{
    /** Rotation radius */
    int radius;
    /** Start degree */
    int start_deg;
    /** End degree */
    int end_deg;
}

```

```

/** Current degree */
int cur_pos;

/** Slide bitmap index */
int thumb_bmp_index;
} si_rotslider_t;

```

Rotated slider has three styles:

- **SI\_ROTSLIDER\_CW**: Rotate clockwise
- **SI\_ROTSLIDER\_ANTICW**: Rotate anti-clockwise
- **SI\_ROTSLIDER\_STATIC**: Progress bar style

Similar to normal slider, we can get and set information of a rotated slider by using `skin_get_slider_info`, `skin_set_slider_info`, and `skin_scale_slide_pos`.

### 11.2.11 MiniGUI Control

MiniGUI control element represents a normal MiniGUI control. When we use `skin_item_t` structure to define a MiniGUI control element, item `style` should be **SI\_TYPE\_CONTROL**; item `type_data` points to a **CTRLDATA** typed structure. This element type is designed for you to create normal MiniGUI controls on skin window.

Following function can get window handle of MiniGUI control element by using skin element `id`.

```

HWND skin_get_control_hwnd (skin_head_t* skin, int id);

```

### 11.2.12 Sample Program

The source codes in List 11.2 create a skin interface for a MP3 player, which can respond to the basic operation of the user. The complete source code can be seen in file `skindemo.c` of program package `mg-samples` for this guide.

List 11.2 Skin Interface Example Program

```

#include <minigui/common.h>
#include <minigui/minigui.h>

```

```

#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>
#include <minigui/mgext.h>
#include <minigui/skin.h>

#define SIID_TITLE      1
#define SIID_PLAY       2
#define SIID_PAUSE      3
#define SIID_STOP       4
#define SIID_PROGRESS   5
#define SIID_SYSMENU    6
#define SIID_CLOSE      7
#define SIID_VOLUME     8
#define SIID_TIMER      9

#define DEF_WIDTH       284
#define DEF_HEIGHT     135
#define ID_TIME         100

/* Define special properties of skin item */
static si_nrmslider_t progress = { {0, 180, 0 }, 5 };
static si_nrmslider_t volume  = { {1, 100, 50}, 9 };
static si_bmplabel_t timer    = { "00:00", "0123456789:-" };

/* Defien skin item array */
static skin_item_t skin_main_items [] =
{
    {SIID_PLAY, SI_TYPE_CHKBUTTON | SI_TEST_SHAPE_RECT | SI_STATUS_VISIBLE,
     205, 106, {}, 1, "Play"},
    {SIID_PAUSE, SI_TYPE_CHKBUTTON | SI_TEST_SHAPE_RECT | SI_STATUS_VISIBLE,
     230, 106, {}, 2, "Pause"},
    {SIID_STOP, SI_TYPE_CHKBUTTON | SI_TEST_SHAPE_RECT | SI_STATUS_VISIBLE,
     254, 106, {}, 3, "Stop"},
    {SIID_PROGRESS, SI_TYPE_NRMSLIDER | SI_TEST_SHAPE_RECT | SI_STATUS_VISIBLE
     | SI_NRMSLIDER_HORZ, 8, 91, {}, 4, "Progress", 0, &progress},
    {SIID_SYSMENU, SI_TYPE_CMDBUTTON | SI_TEST_SHAPE_RECT | SI_STATUS_VISIBLE,
     9, 2, {}, 6},
    {SIID_CLOSE, SI_TYPE_CMDBUTTON | SI_TEST_SHAPE_RECT | SI_STATUS_VISIBLE,
     263, 2, {}, 7, "Close"},
    {SIID_VOLUME, SI_TYPE_NRMSLIDER | SI_TEST_SHAPE_RECT | SI_STATUS_VISIBLE
     | SI_NRMSLIDER_HORZ, 102, 55, {}, 8, "Volume", 0, &volume},
    {SIID_TIMER, SI_TYPE_BMPLABEL | SI_TEST_SHAPE_RECT | SI_STATUS_VISIBLE,
     20, 67, {}, 10, "Time", 0, &timer}
};

/* Define skin */
skin_head_t main_skin =
{
    "Skin Window",
    SKIN_STYLE_TOOLTIP, NULL, NULL,
    0, 8, skin_main_items, FALSE
};

/* Bitmap array */
const char *bmp_name[] = {
    "main.png", "play.png", "pause.png", "stop.png", "progress-bk.png", "progress.png",
    "sysmenu.png", "close.png", "volume-bk.png", "volume.png", "timer.png"
};

static int cur_pos = 0;

/* Bitmap resource load/unload function */
void load_skin_bmps ( skin_head_t *skin, BOOL load )
{
    int i, bmp_num = sizeof(bmp_name)/sizeof(char *);

    /*
     * If load is true, load bitmap to bmps array of skin, else unload bitmap in
     * bmps array code is omitted.
     */
}

/* Skin event callback function */
static int main_event_cb (HWND hwnd, skin_item_t* item, int event, void* data)
{

```

```

    if (event == SIE_BUTTON_CLICKED) {
        switch (item->id) {
            case SIID_PLAY:
/* SIE_BUTTON_CLICKED event of skin item SIID_PLAY is handled herein */
                ...
                break;
            }
            ...
        }
    }
    else if (event == SIE_SLIDER_CHANGED) {
        ...
    }

    return 1;
}

/* Skin window message callback function */
static int msg_event_cb (HWND hwnd, int message,
                        WPARAM wparam, LPARAM lparam, int* result)
{
    switch (message) {
        case MSG_TIMER:
            ...
            hostskin = get_window_skin (hwnd);
            skin_set_thumb_pos (hostskin, SIID_PROGRESS, cur_pos);
            skin_set_item_label (hostskin, SIID_TIMER, buf);
            break;
        }
    }
    return 1;
}

int MiniGUIMain (int argc, const char *argv[])
{
    MSG msg;
    HWND hWndMain;

#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER, "skindemo", 0, 0);
#endif

    if (!InitMiniGUIExt()) {
        return 2;
    }

    load_skin_bmps (&main_skin, TRUE); /* Load bitmap resource */
    if ( !skin_init (&main_skin, main_event_cb, msg_event_cb) ){/* init the skin */
        printf ("skin init fail !\n");
    }
    else{
        hWndMain = create_skin_main_window (&main_skin,
                                            HWND_DESKTOP, 100, 100, 100 + DEF_WIDTH, 100 + DEF_HEIGHT, FALSE);

        while (GetMessage (&msg, hWndMain)) {
            TranslateMessage (&msg);
            DispatchMessage (&msg);
        }

        MainWindowCleanup (hWndMain);
        skin_deinit (&main_skin); /* Cancel skin */
    }

    load_skin_bmps (&main_skin, FALSE); /* Unload bitmap resource */
    MiniGUIExtCleanUp ();

    return 0;
}

#ifdef _LITE_VERSION
#include <minigui/dti.c>
#endif

```

The program defines a skin window `main_skin`, which includes eight skin

elements: five command buttons, two normal sliders, and one image label. Firstly the program loads the bitmap resource needed by the skin window, then calls function `skin_init` to initialize the skin. If the initialization is successful, program will call function `create_skin_main_window` to create skin window and enter the message loop of skin window. When program stops, it calls function `skin_deinit` to de-initialize the skin.

The program's interface is showed as Fig. 11.5.



Fig. 11.5 Skin Interface Example

### 11.3 Color Selection Dialog Box

Color selection dialog box provides an intuitionistic manner, so that the user can select a desired value of color in RGB color space, as shown in Fig. 11.6.



Fig. 11.6 Color selection dialog box

`ColorSelDialog` function creates a color selection dialog box:

```
int ColorSelDialog (HWND hWnd, int x, int y, int w, int h, PCOLORDATA pClrData);
```

Here `hWnd` is the hosting window handle of the color selection dialog box; `x`, `y`, `w`, `h` specifies the position and size of the box; `pClrData` points to a structure of `COLORDATA` structure, which is used to store the value of color selected by the user:

```
typedef struct _COLORDATA {
    /** reserves, not used now. */
    DWORD style;
    /** the value of the color returned. */
    gal_pixel pixel;
    /** the R, G, B value of the color returned. */
    UInt8 r, g, b;
    /** the H, S, V value of the color returned. */
    UInt16 h;
    UInt8 s, v;
} COLORDATA, *PCOLORDATA;
```

The `pixel` field is the pixel value of the selected color; `r`, `g`, and `b` are RGB values of the color; `h`, `s` and `v` are the values of corresponding color in HSV color space.



If the user selects a color, `ColorSelDialog` function will return `SELCOLOR_OK`, otherwise `SELCOLOR_CANCEL`.

**[NOTE] Color selection dialog box is only provided when MiniGUI is set to use NEWGAL engine.**

## 11.4 New Open File Dialog Box

We add new open file dialog box into MiniGUIExt library during development of MiniGUI V1.6, this dialog box provides more functions and convenient interface for the user. New applications should use the new open file dialog box. Effect of the dialog box is as shown in Fig. 11.7.



Fig. 11.7 New open file dialog box

`ShowOpenDialog` function creates a file open dialog box:

```
int ShowOpenDialog (HWND hWnd, int lx, int ty, int w, int h, PNEWFILEDLGDATA pnfd);
```

Here `hWnd` is the hosting window handle of the file open dialog box. Parameter `lx`, `ty`, `w` and `h` specify the position and size of the dialog box. MiniGUI will automatically adjust the layout of controls in the dialog box according to the size passed. `pnfd` points to a structure of `NEWFILEDLGDATA` type, which is used to specify initial data and information of file selected by the user:

```
typedef struct _NEWFILEDLGDATA
```

```
{
    /** indicates to create a Save File or an Open File dialog box. */
    BOOL    IsSave;
    /** the full path name of the file returned. */
    char    filefullname[NAME_MAX + PATH_MAX + 1];
    /** the name of the file to be opened. */
    char    filename[NAME_MAX + 1];
    /** the initial path of the dialog box. */
    char    filepath[PATH_MAX + 1];
    /** the filter string, for example: All file (*.*)|Text file (*.txt;*.TXT) */
    char    filter[MAX_FILTER_LEN + 1];
    /** the initial index of the filter*/
    int     filterindex;
} NEWFILEDLGDATA;
```

- **IsSave**: Used to specify whether to open file or save file. If it is TRUE, the dialog box is used to specify name of the file to be stored.
- **filefullname**: Used to return the full path name of the file selected by the user.
- **filename**: Used to specify the name of the file to be open/save by default.
- **filepath**: Used to specify the initial value of the directory where the file is.
- **filter**: Used to specify the filter strings of files. Different filter string is separated by pipeline, for example: "All file (\*.\*)|Text file (\*.txt;\*.TXT)" specifies two filter strings used for all files and text files.
- **filterindex**: Used to specify the index of filter string effective initially, zero is the first index.

If the user selects a file, `ShowOpenDialog` function will return `IDOK`, otherwise `IDCANCEL`.

**[NOTE] File open dialog box is only provided when MiniGUI is run on UNIX operating system alike (Linux, uClinux etc.).**

## 12 Other Programming Topics

### 12.1 Timer

MiniGUI application can call function `SetTimer` to create a timer. When the created timer is expired, the target window will receive message `MSG_TIMER`, the identifier of the timer, and the system tick value that the timer is triggered. When the time is not needed, application program can call `KillTimer` to delete the timer. `SetTimerEx` and `ResetTimerEx` are provided since MiniGUI version 2.0.4/1.6.10; they support timer callback procedure. The prototypes are as follow:

```
typedef BOOL (* TIMERPROC) (HWND, int, DWORD);  
  
BOOL WINAPI ResetTimerEx (HWND hWnd, int id, unsigned int speed,  
                          TIMERPROC timer_proc);  
  
BOOL WINAPI SetTimerEx (HWND hWnd, int id, unsigned int speed,  
                       TIMERPROC timer_proc);  
  
#define SetTimer(hWnd, id, speed) \  
    SetTimerEx(hWnd, id, speed, NULL)  
  
#define ResetTimer(hWnd, id, speed) \  
    ResetTimerEx(hWnd, id, speed, (TIMERPROC) 0xFFFFFFFF)
```

The meaning of each parameter in `TIMERPROC` is as follows:

- The window handle passed when creating the timer. If no use, it can be any 32-bit value.
- The timer ID.
- The system tick value that the timer is triggered.

When `TIMERPROC` return `FALSE`, MiniGUI will delete the timer automatically. It can be used to create one-shot timer.

The timer mechanism of MiniGUI provides application with a comparatively convenient timing mechanism. However, the timer mechanism of MiniGUI has the following constraints:

- Each message queue of MiniGUI-Threads only can manage 32 timers. Note that when creating a new thread, there will be a new message queue created to correspond, that is, each thread can have maximum 32 timers.

- MiniGUI-Processes has only one message queue, which can manage maximum 32 timers.
- The process of timer message is relatively special. Its implementation is similar to the signal mechanism of UNIX operating system. System will ignore this new timer message when a certain timer message has not been processed but new timer message occurs. It is because that if the frequency of certain timer is very high while the window processing this timer responds too slowly, when the message still needs to be posted, the message queue will be finally choked up.
- Timer message has the lowest priority. When there is not other type of messages in the message queue (such as posted message, notification message, and painting message), system will go to check if there is any expired timer.

When the timer frequency is very high, it is possible that such situation like timer message lost or interval asymmetry occurs. If application needs to use relatively accurate timer mechanism, it should use `setitimer` system call of Linux/UNIX with `SIGALRM` signal. What needs to note is that the server process (`mginit` program) of MiniGUI-Processes has called `setitimer` system call to install the timer. Therefore, the program `mginit` realized by application itself cannot use `setitimer` to realize the timer. However, the client program of MiniGUI-Processes still can call `setitimer`. MiniGUI-Threads has no such constraint.

Code in List 12.1 creates a timer with interval of one second, then use current time to set static text when timer is expired so as to display the clock. Finally, program will delete the timer while closing window.

List 12.1 Use of Timer

```
#define _ID_TIMER 100
#define _ID_TIME_STATIC 100

static char* mk_time (char* buff)
{
    time_t t;
    struct tm * tm;

    time (&t);
    tm = localtime (&t);
    sprintf (buff, "%02d:%02d:%02d", tm->tm_hour, tm->tm_min, tm->tm_sec);
}
```

```

    return buff;
}

static int TaskBarWinProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    char buff [20];

    switch (message) {
    case MSG_CREATE:
    {
        CreateWindow (CTRL_STATIC, mk_time (buff),
                      WS_CHILD | WS_BORDER | WS_VISIBLE | SS_CENTER,
                      _ID_TIME_STATIC, g_rcExcluded.right - _WIDTH_TIME - _MARGIN, _MARGIN,
                      _WIDTH_TIME, _HEIGHT_CTRL, hWnd, 0);

        /* Create a timer with interval being one second.
         * Its identifier is _ID_TIMER, target window is hWnd.
         */
        SetTimer (hWnd, _ID_TIMER, 100);
        break;

    case MSG_TIMER:
    {
        /* Received a message of MSG_TIMER.
         * Application should determine whether wParam is _ID_TIMER.
         */
        SetDlgItemText (hWnd, _ID_TIME_STATIC, mk_time (buff));
        break;

    case MSG_CLOSE:
    {
        /* Delete the timer */
        KillTimer (hWnd, _ID_TIMER);
        DestroyAllControls (hWnd);
        DestroyMainWindow (hWnd);
        PostQuitMessage (hWnd);
        return 0;
    }

    return DefaultMainWinProc (hWnd, message, wParam, lParam);
}
}

```

It is necessary to explain that the third argument of `SetTimer` is used to specify the interval of timer. The default unit is 10 milliseconds.

Application can also call `ResetTimer` to reset interval of a timer. The usage of the function is similar to `SetTimer`.

Moreover, MiniGUI provides the other two functions to query system timer status. `IsTimerInstalled` is used to check whether a timer is installed in the assigned window. `HaveFreeTimer` is used to check whether system have free timer resources.

## 12.2 Dynamic Change of Color of Window Element

Prior to MiniGUI 1.6.0, color of window element (such as caption bar, border, etc.) is determined by configuration information in MiniGUI.cfg file, and all the windows use the same configuration. In MiniGUI 1.6.0 version, we add the following interfaces, so we can change color (pixel value) of some window elements of a certain window on-the-fly without influencing other windows:

```
gal_pixel GUIAPI GetWindowElementColorEx (HWND hwnd, Uint16 item);
gal_pixel GUIAPI SetWindowElementColorEx (HWND hwnd, Uint16 item, gal_pixel new_value);
```

The `GetWindowElementColorEx` function gets the color of a certain window element (specified by `item`) of a certain window (`hwnd`); `SetWindowElementColorEx` sets the color of a certain window element, and returns the old color. Window element is specified by `item`, and different value of `item` represents different window element respectively, as shown in Tab. 12.1.

Tab. 12.1 Definitions of window element

ID for window element	Meanings	Remarks
BKC_CAPTION_NORMAL	Background color of caption bar in normal state	
FGC_CAPTION_NORMAL	Foreground color of caption bar in normal state	
BKC_CAPTION_ACTIVATED	Background color of caption bar in active state	
FGC_CAPTION_ACTIVATED	Foreground color of caption bar in normal state	
BKC_CAPTION_DISABLED	Background color of caption bar in disabled state	
FGC_CAPTION_DISABLED	Foreground color of caption bar in disabled state	
WEC_FRAME_NORMAL	Color of frame in normal state	
WEC_FRAME_ACTIVATED	Color of frame in normal state	
WEC_FRAME_DISABLED	Color of frame in disabled state	
WEC_3DBOX_NORMAL	Normal color of 3D-box	Can be used to control the color of 3D-box of control with 3D-display effect such as button.
WED_3DBOX_REVERSE	Reversed color of 3D-box	
WEC_3DBOX_LIGHT	Highlight color of 3D-box	
WEC_3DBOX_DARK	Dark color of 3D-box	
WEC_FLAT_BORDER	Border color of flat box	
FGC_CONTROL_DISABLED	Foreground color of flat box	Can be used to control the color of controls with

<b>BKC_HILIGHT_NORMAL</b>	Background color of section selected by control	the color of controls with selection property such as edit box, list box etc.
<b>BKC_HILIGHT_LOSTFOCUS</b>	Background color of section selected by control when lose focus	
<b>FGC_HILIGHT_NORMAL</b>	Foreground color of section selected by control	
<b>BKC_CONTROL_DEF</b>	Background color of control by default	Used to control the color of foreground and background of normal control
<b>FGC_CONTROL_NORMAL</b>	Foreground color of control in normal state	
<b>FGC_HILIGHT_DISABLED</b>	Foreground color of section selected by control in disabled state	
<b>BKC_DESKTOP</b>	Background color of desktop	Global property
<b>BKC_DIALOG</b>	Background color by default of dialog box	

When you need to change the painting color of some element of a window (main window or control), you can write code as follows:

```

HWND hwnd = GetDlgItem (hDlg, IDC_STATIC);
gal_pixel pixel = RGB2Pixel (HDC_SCREEN, r, g, b);

SetWindowElementColorEx (hwnd, FGC_CONTROL_NORMAL, pixel);
UpdateWindow (hwnd, TRUE);

```

The code above changes the foreground color of a static box in dialog box. It should be noted that if the window is visible, then you should call **UpdateWindow** function to update the window to reflect the change.

Note that if you want to change the background color of main window of control, you should use **SetWindowBkColor** function, for example:

```

HWND hwnd = GetDlgItem (hDlg, IDC_STATIC);
gal_pixel pixel = RGB2Pixel (HDC_SCREEN, r, g, b);

SetWindowBkColor (hwnd, pixel);
InvalidateRect (hwnd, NULL, TRUE);

```

## 12.3 Clipboard

Clipboard is a tool to transfer data, and can be used for data communication among applications and application internals. Its principle is very simple, that is, a program places data on the clipboard, and another application takes the data from the clipboard. Clipboard is a data interchange station among

applications.

Edit box control in MiniGUI supports clipboard operations, when the user selects text and press down <CTRL+C> keys, the data is copied to the text clipboard by default; when the users press down <CTRL+V> keys, the data is copied from the clipboard into edit box.

### 12.3.1 Creating and Destroying Clipboard

MiniGUI provides a default text clipboard, named as `CBNAME_TEXT` (string name "text"), used for copying and pasting of text. Applications can use the clipboard directly without other additional operations. A clipboard defined by an application itself need to be created with `CreateClipboard` function, and to be destroyed with `DestroyClipboard` after using it.

There are at most `NR_CLIPBOARDS` clipboards in MiniGUI, including the system default text clipboard and clipboards defined by user. `NR_CLIPBOARDS` macro is defined to be 4 in `minigui/window.h` header file by default.

`CreateClipboard` function creates a clipboard with specified name, this name can not be same as the name of existing clipboards (system defined or user defined):

```
int GUIAPI CreateClipboard (const char* cb_name, size_t size);
```

Argument `cb_name` specifies the name of a clipboard; argument `size` specifies the size of data to be stored by the clipboard. If creates successfully, the function returns `CBERR_OK`; if the name is reduplicate, the function returns `CBERR_BADNAME`; if the memory is not enough, the function returns `CBERR_NOMEM`.

The `DestroyClipboard` function destroys a user defined clipboard created by `CreateClipboard` function:



```
int GUIAPI DestroyClipboard (const char* cb_name);
```

### 12.3.2 Transferring Data to Clipboard

**SetClipboardData** function transfers data to a specified clipboard:

```
int GUIAPI SetClipboardData (const char* cb_name, void* data, size_t n, int cbop);
```

Here, the argument **cb\_name** specifies the name of clipboard; **data** is the pointer to data buffer; **n** is the size of data; **cbop** is the operation type, which can be:

- **CBOP\_NORMAL**: Default overwriting operation. The new data overwrites the existing data on the clipboard;
- **CBOP\_APPEND**: Appending operation. The new data will be appended to the existing data on the clipboard

### 12.3.3 Getting Data from Clipboard

**GetClipboardDataLen** function is used to get the size of data on clipboard:

```
size_t GUIAPI GetClipboardDataLen (const char* cb_name);
```

**GetClipboardData** function is used to copy the data on clipboard to the specified data buffer:

```
size_t GUIAPI GetClipboardData (const char* cb_name, void* data, size_t n);
```

Here the argument **cb\_name** specifies the name of the clipboard; **data** is the pointer to the data buffer; **n** specifies the size of specified data buffer. The function returns the size of gotten data from the clipboard

Generally speaking, you can use **GetClipboardDataLen** function to get the size of data before using **GetClipboardData** function to get data of clipboard, so that you can specify an appropriate data buffer to save data.

**GetClipboardByte** function is used to get a byte from the specified position of data on clipboard.

```
int GUIAPI GetClipboardByte (const char* cb_name, int index, unsigned char* byte);
```

Here the argument **index** specifies the index position of specified data; **byte** is used to save the gotten byte data.

## 12.4 Reading/Writing Configuration File

The configuration file of MiniGUI (default as `/usr/local/etc/MiniGUI.cfg` file) uses Windows INI-like file format. This format is very simple, seen as follows:

```
[section-name1]
key-name1=key-value1
key-name2=key-value2

[section-name2]
key-name3=key-value3
key-name4=key-value4
```

The information of such configuration file is grouped by section, then uses **key=value** format to appoint parameter and its value. Application can also use this format to store some configuration information. So, MiniGUI provides following functions (`minigui/minigui.h`):

```
int GUIAPI GetValueFromEtcFile (const char* pEtcFile, const char* pSection,
                                const char* pKey, char* pValue, int iLen);

int GUIAPI GetIntValueFromEtcFile (const char* pEtcFile, const char* pSection,
                                    const char* pKey, int* value);

int GUIAPI SetValueToEtcFile (const char* pEtcFile, const char* pSection,
                              const char* pKey, char* pValue);

GHANDLE GUIAPI LoadEtcFile (const char* pEtcFile);

int GUIAPI UnloadEtcFile (GHANDLE hEtc);

int GUIAPI GetValueFromEtc (GHANDLE hEtc, const char* pSection,
                            const char* pKey, char* pValue, int iLen);

int GUIAPI GetIntValueFromEtc (GHANDLE hEtc, const char* pSection,
                              const char* pKey, int *value);

int GUIAPI SetValueToEtc (GHANDLE hEtc, const char* pSection,
                          const char* pKey, char* pValue);

int GUIAPI RemoveSectionInEtcFile (const char* pEtcFile, const char* pSection);

int GUIAPI GetValueFromEtcSec (GHANDLE hSect,
                              const char* pKey, char* pValue, int iLen);
```

```
int GUIAPI GetIntValueFromEtcSec (GHANDLE hSect,
                                const char* pKey, int* pValue);

int GUIAPI SetValueToEtcSec (GHANDLE hSect,
                            const char* pKey, char* pValue);

int GUIAPI SaveEtcToFile (GHANDLE hEtc, const char* file_name);

GHANDLE GUIAPI FindSectionInEtc (GHANDLE hEtc,
                                const char* pSection, BOOL bCreateNew);

int GUIAPI RemoveSectionInEtc (GHANDLE hEtc, const char* pSection);
```

The use of first three functions is as follows:

- **GetValueFromEtcFile**: To get a specified key value from a specified configuration file. The value of the key returns as a string.
- **GetIntValueFromEtcFile**: To get a specified key integer value from a specified configuration file. This function converts the accepted string to integer value (using `strtol` function) and then returns the value.
- **SetValueToEtcFile**: This function stores the given key value into a specified configuration file. If the configuration file does not exist, this function will create new configuration file. If the file exists, the old value will be covered.

The next five functions are new configuration file reading/writing function since MiniGUI version 1.6.x; the use of them is as follows:

- **LoadEtcFile**: Read a specified configuration file into memory and returns a configuration object handle, then the related functions can visit the configuration information in memory through this handle.
- **UnloadEtcFile**: Release the configuration information in memory.
- **GetValueFromEtc**: The way of use is similar to **GetValueFromEtcFile**; But its first argument is the configuration object handle, not the file name. This function can be used to get configuration information from memory.
- **GetIntValueFromEtc**: The way of use is similar to **GetIntValueFromEtcFile**.
- **SetValueToEtc**: Similar to **SetValueToEtcFile**, but this function only changes the configuration key value in memory, does not affect the content in the file.

The last seven functions are new configuration file reading/writing function since MiniGUI version 2.0.4/1.6.10; the use of them is as follows:

- **RemoveSectionInEtcFile**: Remove a specified section from a specified configuration file.
- **RemoveSectionInEtc**: Remove a specified section from the configuration information in memory.
- **GetValueFromEtcSec**: Get value from a specified section in memory. Similar to **GetValueFromEtc**.
- **GetIntValueFromEtcSec**: Get an integer value from specified section in memory. Similar to **GetIntValueFromEtc**.
- **SetValueToEtcSec**: Set the value in a specified section in memory. Similar to **SetValueToEtc** and **SetValueToEtc**.
- **FindSectionInEtc**: Find or create a specified section from the configuration information in memory.
- **SaveEtcToFile**: Save the configuration information in memory into a specified file.

These functions are usually used to read all information of configuration file for once. When need to get relatively more key values, this function will first use **LoadEtcFile** to read in a configuration file, and then use **GetValueFromEtc** to get key value. When there is no need to visit configuration information, use **UnloadEtcFile** to release the configuration object handle.

Assuming that certain configuration file records some application information, and has the following formats:

```
[mginit]
nr=8
autostart=0

[app0]
path=../tools/
name=vcongui
layer=
tip=Virtual&console&on&MiniGUI
icon=res/konsole.gif

[app1]
path=../bomb/
name=bomb
layer=
tip=Game&of&Minesweeper
icon=res/kmines.gif
```

```
[app2]
path=../controlpanel/
name=controlpanel
layer=
tip=Control&Panel
icon=res/kcmx.gif
```

The section `[mginit]` records the number of applications and its automatically-startup index (`autostart` key). The section `[appX]` records information of each application program, including the path, the name, and the icon of the application. Code in List 12.2 illustrates how to use the functions above to get such information (the code comes from program `mginit` of MDE).

List 12.2 Using MiniGUI configuration file functions to get information

```
#define APP_INFO_FILE "mginit.rc"

static BOOL get_app_info (void)
{
    int i;
    APPITEM* item;

    /* Get information of the number of programs */

    if (GetIntValueFromEtcFile (APP_INFO_FILE, "mginit", "nr", &app_info.nr_apps) != ETC_OK)
        return FALSE;

    if (app_info.nr_apps <= 0)
        return FALSE;

    /* Get index of autostarting application */
    GetIntValueFromEtcFile (APP_INFO_FILE, "mginit", "autostart", &app_info.autostart);

    if (app_info.autostart >= app_info.nr_apps || app_info.autostart < 0)
        app_info.autostart = 0;

    /* Calloc information structure of application */
    if ((app_info.app_items = (APPITEM*)calloc (app_info.nr_apps, sizeof (APPITEM))) == NULL) {
        return FALSE;
    }

    /* Get information of each application such as path, name and icon*/
    item = app_info.app_items;
    for (i = 0; i < app_info.nr_apps; i++, item++) {
        char section [10];

        sprintf (section, "app%d", i);
        if (GetValueFromEtcFile (APP_INFO_FILE, section, "path",
                                item->path, PATH_MAX) != ETC_OK)
            goto error;

        if (GetValueFromEtcFile (APP_INFO_FILE, section, "name",
                                item->name, NAME_MAX) != ETC_OK)
            goto error;

        if (GetValueFromEtcFile (APP_INFO_FILE, section, "layer",
                                item->layer, LEN_LAYER_NAME) != ETC_OK)
            goto error;

        if (GetValueFromEtcFile (APP_INFO_FILE, section, "tip",
                                item->tip, TIP_MAX) != ETC_OK)
            goto error;

        strsubchr (item->tip, '&', ' ');
    }
}
```

```

        if (GetValueFromEtcFile (APP_INFO_FILE, section, "icon",
                                item->bmp_path, PATH_MAX + NAME_MAX) != ETC_OK)
            goto error;

        if (LoadBitmap (HDC_SCREEN, &item->bmp, item->bmp_path) != ERR_BMP_OK)
            goto error;

        item->cdpath = TRUE;
    }
    return TRUE;
error:
    free_app_info ();
    return FALSE;
}

```

If using `LoadEtcFile`, `GetValueFromEtc`, and `UnloadEtcFile` to implement above example, the code will be as follows:

```

GHANDLE hAppInfo;
hAppInfo = LoadEtcFile (APP_INFO_FILE);
//...
get_app_info ();
//...
UnloadEtcFile (hAppInfo);

```

We also need change `GetValueFromEtcFile` of function `get_app_info` to `GetValueFromEtc`.

## 12.5 Writing Portable Program

As we know, the CPU used by most embedded system has totally different construction and characteristic from the CPU of normal desktop PC. However, operating system and advanced language can hide these differences to a great extent. With the support of advanced language programming, the compiler and operating system can help programs solve most problems related to CPU architecture and characteristic in order to save developing time and increase developing efficiency. However, application programs have to face some certain CPU characteristics; the following aspects need to be paid more attention:

- The order of byte. Generally, when CPU stores integer data of multi-bytes, it will store the low-bit byte in low address unit, such as Intel x86 series. Some CPU uses opposite order to store. For example, the popularly used PowerPC in embedded system stores low-bit byte in high address unit. The former is called little-endian system while the

latter is called big-endian system.

- The Linux kernel on some platforms may lack of some advanced system calls, the most popular one is the system calls related to virtual memory mechanism. The Linux system running on certain CPU cannot provide virtual memory mechanism because of the limitation of CPU capability. For example, CPU lack of MMU unit cannot provide the sharing memory of System V IPC mechanism.

In order to make the portable code have most popular adaptability, application programs must notice these differences and write code according to different situations. Here we will describe how to write portable code in MiniGUI applications.

### 12.5.1 Using Endian-Specific Read/Write Functions of MiniGUI

In order to solve the first problem mentioned above, MiniGUI provides several endian-related read/write functions. These functions can be divided into two categories:

- Functions used to swap the order of byte, including `ArchSwapLE16`, `ArchSwapBE16` and so on.
- Functions used to read/write standard I/O stream, including `MGUI_ReadLE16`, `MGUI_ReadBE16` and so on.

The first category is used to convert the 16-bit, 32-bit, or 64-bit integer into system native byte from certain byte order. For example:

```
int fd, len_header;
...
if (read (fd, &len_header, sizeof (int)) == -1)
    goto error;
#if MGUI_BYTEORDER == MGUI_BIG_ENDIAN
    len_header = ArchSwap32 (len_header); // If it is big-endian system, swap the order
#endif
...
```

The above code first uses `read` system call to read an integer value from the a specified file descriptor to variable `len_header`. The integer value saved in this file is in little-endian, so the byte order of this integer value has to be swapped

if this integer value is used in big-endian system. We can use **ArchSwapLE32** to convert 32-bit integer value into system native byte order. Also, we can swap the bytes only for big-endian system, and then we just need to use **ArchSwap32** function.

The functions (or macro) used to swap bytes are as follow:

- **ArchSwapLE16(X)** converts the specified 16-bit integer value (stored in little endian byte order) to system native integer value. If system is little endian, this function will directly return X; if system is big endian, the function will call **ArchSwap16** to swap the bytes.
- **ArchSwapLE32(X)** converts the specified 32-bit integer value (stored in little endian byte order) to system native integer value. If system is little endian, this function will directly return X; if system is big endian, the function will call **ArchSwap32** to swap the bytes.
- **ArchSwapBE16(X)** converts the specified 16-bit integer value (stored in big endian byte order) to system native integer value. If system is big endian, this function will directly return X; if system is little endian, the function will call **ArchSwap16** to swap the bytes.
- **ArchSwapBE32(X)** converts the specified 32-bit integer value (stored in big endian byte order) to system native integer value. If system is big endian, this function will directly return X; if system is little endian, the function will call **ArchSwap32** to swap the bytes.

The second category of functions provided by MiniGUI is used to read/write integer value from standard I/O file object. If the file is stored in little endian byte order, the function uses **MGUI\_ReadLE16** and **MGUI\_ReadLE32** to read integer value by converting integer value to system native byte order, whereas uses **MGUI\_ReadBE16** and **MGUI\_ReadBE32**. If the file is stored as little endian byte order, the function will use **MGUI\_WriteLE16** and **MGUI\_WriteLE32** to write integer value after converting integer value from system native byte order to little endian; whereas use **MGUI\_WriteBE16** and **MGUI\_WriteBE32**. The following code explains the above functions:

```
FILE* out;  
int count;  
...
```



```
MGUI WriteLE32 (out. count): // Write count to the file in little endian
```

## 12.5.2 Using Condition Compilation to Write Portable Code

When regarding problems related to portability, we can easily use the way described above to perform function wrap in order to provide well portable code. However, sometime we cannot use such way to provide the portable code, so we can only use conditional compilation. Code in List 12.3 illustrates how to use conditional compilation to ensure the program running well (the code come from MiniGUI `src/kernel/sharedres.c`).

List 12.3 The usage of conditional compilation

```
/* If system does not support memory share, define _USE_MMAP */
#undef _USE_MMAP
/* #define _USE_MMAP 1 */

void *LoadSharedResource (void)
{
#ifdef _USE_MMAP
    key_t shm_key;
    void *memptr;
    int shmid;
#elseif

    /* Load share resource*/
    ...

#endif

#ifdef _USE_MMAP /* Get object of share memory*/
    if ((shm_key = get_shm_key ()) == -1) {
        goto error;
    }
    shmid = shmget (shm_key, mgSizeRes, SHM_PARAM | IPC_CREAT | IPC_EXCL);
    if (shmid == -1) {
        goto error;
    }

    // Attach to the share memory.
    memptr = shmat (shmid, 0, 0);
    if (memptr == (char*)-1)
        goto error;
    else {
        memcpy (memptr, mgSharedRes, mgSizeRes);
        free (mgSharedRes);
    }

    if (shmctl (shmid, IPC_RMID, NULL) < 0)
        goto error;
#endif

    /* Open a file */
    if ((lockfd = open (LOCKFILE, O_WRONLY | O_CREAT | O_TRUNC, 0644)) == -1)
        goto error;

#ifdef _USE_MMAP
    /* If use mmap, write share resource into the file*/
    if (write (lockfd, mgSharedRes, mgSizeRes) < mgSizeRes)
        goto error;
    else
    {
        free(mgSharedRes);
        mgSharedRes = mmap( 0, mgSizeRes, PROT_READ|PROT_WRITE, MAP_SHARED, lockfd, 0);
    }
#endif
}
```

```

    }
#else
    /* otherwise write the object ID of share memory into the file*/
    if (write (lockfd, &shmid, sizeof (shmid)) < sizeof (shmid))
        goto error;
#endif

    close (lockfd);

#ifdef _USE_MMAP
    mgSharedRes = memptr;
    SHAREDRES_SHMID = shmid;
#else
    SHAREDRES_SEMID = semid;

    return mgSharedRes;

error:
    perror ("LoadSharedResource");
    return NULL;
}

```

The MiniGUI-Processes server program to load sharing resource uses the above code fragment. If system supports shared memory, it will initialize the shared memory object and associate the shared resource with the shared memory object, then write the shared memory object ID into a file; if system does not support shared memory, it will write all initialized sharing resource into a file. If the system support shared memory, clients can get shared memory object ID from the file and directly attach it; if the system does not support shared memory, clients can use **mmap** system call to map the file to the address space of them. Code of clients can be seen in List 12.4.

List 12.4 The usage of conditional compilation (cont.)

```

void* AttachSharedResource (void)
{
#ifdef _USE_MMAP
    int shmid;
#else
    int lockfd;
    void* memptr;

    if ((lockfd = open (LOCKFILE, O_RDONLY)) == -1)
        goto error;

#ifdef _USE_MMAP
    /* Use mmap to image share resource to process address space */
    mgSizeRes = lseek (lockfd, 0, SEEK_END);
    memptr = mmap( 0, mgSizeRes, PROT_READ, MAP_SHARED, lockfd, 0);
#else
    /* Otherwise get ID of the object of share memroy, and associate the share memory */
    if (read (lockfd, &shmid, sizeof (shmid)) < sizeof (shmid))
        goto error;
    close (lockfd);

    memptr = shmat (shmid, 0, SHM_RDONLY);
#endif
    if (memptr == (char*)-1)
        goto error;
}

```

```

return memPtr;
error:
    perror ("AttachSharedResource");
    return NULL;
}

```

## 12.6 Fixed-Point Computing

Usually when we perform math operations, we will use float-point to represent real number, and use `<math.h>` head file to calculate the float-point number. However, float-point calculation is a time-consuming process. Therefore, in order to reduce extra CPU instructions caused by float-point calculation, some three-dimension graphics application always use fixed-point number to represent real number, which will greatly accelerate the calculation of three-dimension graphical rendering. MiniGUI also provides some fixed-point computing functions, divided into the following categories:

- Conversion among integer, float-point number and fixed-point number. `itofix` converts integer to fixed-point number, while `fixtoi` converts fixed-point to integer; `ftofix` converts float-point number to fixed-point number, while `fixtof` converts fixed-point number to float-point number.
- The basic arithmetic computing such as add, subtract, multiple, and divide of fixed-point numbers: `fixadd`, `fixsub`, `fixmul`, `fixdiv`, `fixsqrt`.
- The triangle compute of fixed-point number: `fixcos`, `fixsin`, `fixtan`, `fixacos`, `fixasin`.
- Matrix and vector computing. Matrix and vector related computing are important for three-dimension graphics. Readers can refer to `minigui/fixedmath.h` for the functions.

Code in List 12.5 illustrates the use of fixed-point number. This code converts plane rectangular coordinates to screen coordinates.

List 12.5 Fixed-point computing

```

void scale_to_window (const double * in_x, const double * in_y,
                     double * out_x, double * out_y)
{
    fixed f_x0 = ftofix (get_x0());

```

```
fixed f_v0 = ftofix (get_v0());  
fixed f_in_x = ftofix (*in_x);  
fixed f_in_y = ftofix (*in_y);  
fixed f_p = ftofix (get_pixel_length());  
  
*out_x = fixtof(fixmul(fixsub(f_in_x, f_x0), f_p));  
*out_y = -fixtof(fixmul(fixsub(f_in_y, f_y0), f_p));  
}
```

The calculation of above program is very simple. The steps are as follow:

1. Converts the input parameters to fixed-point values.
2. Does the calculation by using the fixed-point values.
3. Converts the result values to float-point values.

## **II MiniGUI Graphics Programming**

- Graphics Device Interfaces
- Handling and Rendering of Text
- Advanced GDI functions based on NEWGAL



## 13 Graphics Device Interfaces

Graphics Device Interfaces (GDI) is an important part of a GUI system. Through GDI, the GUI application can execute graphics output on the screen or other display devices, including basic painting and text output. In this chapter and the two sequent chapters, we will describe in detail the important concepts of GDI, the methods of graphics programming and the main GDI functions of MiniGUI, and will illustrate the use of important functions with example.

### 13.1 Architecture of MiniGUI Graphics System

#### 13.1.1 GAL and GDI

In order to separate the bottom layer graphics device and the top layer graphics interface so as to increase the portability of the MiniGUI graphics system, MiniGUI introduces the concept of Graphics Abstract Layer (GAL). GAL defines a group of abstract interfaces, which do not depend on any special hardware, and all the top layer graphics operation are based on these abstract interfaces. The bottom layer code used to realize this abstract interface is called “graphics engine”, similar to the driver in an operating system. Using GAL, MiniGUI can run on may existed graphics function libraries, and can be readily port to other POSIX systems, only requiring to realize the new graphics engine according to our abstract layer interfaces. For example, in a system based on Linux, we can create general MiniGUI graphics engine based on Linux FrameBuffer driver. In fact, the native graphics engine included in MiniGUI 1.0.00 version is the graphics engine based on Linux FrameBuffer. Generally speaking, all the embedded systems based on Linux will provide FrameBuffer support so that the native graphics engine can be run on either a common PC or a special embedded system.

### 13.1.2 New GAL

MiniGUI version 1.1.0 makes much improvement to GAL and GDI, introducing new GAL and GDI interfaces and functions.

In the old GAL and GDI design, GAL can be considered as the graphics driver of GDI, and many graphics operations, for example drawing point, drawing line, filling rectangle, and bitmap operations, etc., are implemented through the corresponding function of GAL. The biggest problem of this design is GDI cannot be extended. For example, in order to add the ellipse drawing function, it is needed to realize the ellipse painting function in each engine. Moreover, it is the clipping region, which GDI manages, while GAL engine is based on clipping rectangle. This method also causes that GDI function cannot optimize the painting. Therefore, in the interface design of new GAL and GDI, we make restriction to GAL interface, and make many graphics input functions which are previous completed by GAL engine to be completed in top layer GDI functions. The function partition of New GAL (NEWGAL) and new GDI (NEWGDI) are as follow:

- NEWGAL is responsible for initializing the video device, and managing the use of video memory;
- NEWGAL is responsible for providing top layer GDI with linear video memory which is mapped into process address space, and other information such as palette;
- NEWGAL is responsible for realizing fast bit block operation, including rectangle filling and blitting operation, etc., and using hardware acceleration function in possible cases;
- NEWGDI function realizes advanced graphics function, including point, line, circle, ellipse, arc, spine curve, and further advanced logical pen and logical brush, and implements acceleration function by calling NEWGAL interface when it is necessary;
- Although some video devices also provide hardware support for the advanced graphics functions mentioned above, however, considering other factors, these hardware acceleration functions are not provided by NEWGAL interface, but are all realized by software.



Thus, the main painting function realized by NEWGAL is limited to bit block operation, for example, rectangle filling and bit blitting operation; and other advanced graphics functions are all realized by NEWGDI functions.

The interface of NEWGAL can effectively use video memory in video card, and sufficiently use the hardware acceleration function. As we know, current video cards commonly have more than 4MB video memory, and not all the video memory will be used in a common display mode. Therefore, NEWGAL engine can manage this unused video memory, and allocate it to the application. Thus, it is realized to save the use of system memory on one hand, and sufficiently use the acceleration function provided by video card so that it can perform fast bit block operation between different video memory areas, i.e. blitting, on the other hand.

When top layer NEWGDI interface is creating a memory DC device, it will allocate memory from video memory, and will consider to use the system memory when if it is not successful. Thus, if NEWGAL engine provides hardware acceleration function, blitting operation (i.e., GDI function BitBlt) will be run in the fastest speed between two different DC devices. Further, if the hardware supports transparent or alpha blending function, the transparent or alpha blending blitting operation will also be run in the fastest speed. NEWGAL interface can automatically use these hardware acceleration functions according to the acceleration ability of the bottom layer engine. The hardware acceleration abilities currently supported mainly include: rectangle filling, normal blitting operation, transparent and alpha blending blitting operation, etc. Certainly, if the hardware does not support these acceleration functions, NEWGAL interface can also realize these functions by software. Currently, the video cards which provide above hardware acceleration function through NEWGAL and FrameBuffer include: Matrox, and 3DFX, etc.

GDI interface based on NEWGAL are partially compatible with old GDI, but we provide some advanced functions based on NEWGAL. We will describe advanced GDI interfaces based on NEWGAL in Chapter 15.

## 13.2 Painting and Updating of a Window

### 13.2.1 When to Paint a Window?

The application uses the window as the main output device, i.e., the MiniGUI application paints only within its window.

MiniGUI manages the display output on the entire screen. If the window content should be repaint due to actions such as window movement, MiniGUI puts a flag to the area in the window to be updated, and then sends a `MSG_PAINT` message to the corresponding window. The application must perform necessary painting to update the window when receiving this message. If the window content changed is caused by the application itself, the application can make a flag to the window area to be updated, and generate a `MSG_PAINT` message.

If it is needed to paint within a window, the application needs to get the device context handle of this window first. Most painting operations of the application are executed during handling `MSG_PAINT`. At this time, the application gets the device context handle by calling `BeginPaint` function. If a certain operation of the application is required to respond immediately, for example to handle the keyboard and mouse messages, it can execute painting immediately without waiting `MSG_PAINT` message. The application can get the device context handle by calling `GetDC` or `GetClientDC` when painting at other time.

### 13.2.2 MSG\_PAINT Message

Usually, the application executes the window painting when receiving `MSG_PAINT` message. If the change of the window influences the content in the client area, or the invalid region of the window is not `NULL`, MiniGUI will send a `MSG_PAINT` message to the corresponding window procedure.

When receiving `MSG_PAINT` message, the application should call `BeginPaint` function to get the device context handle, and use it to call GDI functions to

execute painting which is necessary for updating the client area. After finishing painting, the application should call **EndPaint** function to release the device context handle.

**BeginPaint** function is used to complete the preparing work before painting the window. It first gets the device context of the window client area by calling **GetClientDC** function, and sets the clipping region of the device context to be the current invalid region of the window. Only those regions, which have, be changed need to be repainted, and any attempt to painting outside the clipping region will be clipped and will not be shown on the screen. In order not to influence the painting operation, **BeginPaint** function hides the caret. Finally, **BeginPaint** clears the invalid region of the window to prevent generating continually **MSG\_PAINT** message, and then returns the gotten device context handle.

**LParam** parameter of **MSG\_PAINT** message is the pointer to the window invalid region, and the application can use the information of the window invalid region to optimize painting, for example, limiting painting within the window invalid region. If the application output is simple, you can paint in the whole window and ignoring the invalid region, and let MiniGUI clips the unnecessary painting outside the clipping region so that only the painting within the invalid region is visible.

The application should call **EndPaint** function to end the whole painting process after finishing painting. The main work of **EndPaint** function is to call **ReleaseDC** function to release the device context gotten by **GetClientDC** function; in addition, it shows the caret hidden by **BeginPaint** function.

### 13.2.3 Valid and Invalid Region

Updating region (invalid region) is referred to the region in the window, which is outdated or invalid and need to be repainted. MiniGUI generates **MSG\_PAINT** message for the application according to the region needed to be updated, and the application can also generates **MSG\_PAINT** message by setting invalid

region.

The application can use `InvalidateRect` function to invalidate a certain rectangular region of the window. The prototype of this function is as follows:

```
BOOL WINAPI InvalidateRect (HWND hWnd, const RECT* prc, BOOL bEraseBkgnd);
```

The meaning of the arguments is as follows:

- **hWnd**: the handle of the window needed to be updated
- **prc**: pointer to invalid rectangle
- **bEraseBkgnd**: whether to clear the window background

`InvalidateRect` function adds the specified rectangle to the updating region. This function combines the specified rectangle and the previous updating region of the application window, and then posts a `MSG_PAINT` message to the message queue of this window.

If `bEraseBkgnd` is `TRUE`, the application window will receive a `MSG_ERASEBKGND` message, and the window procedure can handle this message and automatically clear the window background. If the application does not handle `MSG_ERASEBKGND` message, but passes it to `DefaultMainWinProc`, the default handling of `MSG_ERASEBKGND` by MiniGUI is to erase the background with the background color of the window.

The window background is referred to the color and style used to fill the client area before painting the window. The window background may cover the previous content in the client area of the window, and make the program output not disturbed by the existed content on the screen.

`LParam` parameter of `MSG_ERASEBKGND` message includes a `RECT` structure pointer, indicating the rectangle, which should be erased. The application can use this parameter to paint the window background. After finishing painting, the application can directly return zero without calling `DefaultMainWinProc` for default message handling. The example related to handling

`MSG_ERASEBKGD` message can be referred to the related sections of Chapter 3 of this guide.

## 13.3 Graphics Device Context

### 13.3.1 Abstraction of Graphics Device

The application usually calls the painting primitives provided by the graphics system to paint on a graphics context. The context is an object, which notes the graphics properties used by painting primitives. These properties usually include:

- Foreground color (pen), the pixel value or the image used when drawing lines.
- Background color or filling bitmap (brush), the pixel value or image used by painting primitives when filling.
- Painting mode, which describes how the foreground color and the existed screen color are combined. The usual option is to cover the existed screen content or execute "XOR" bit logical operation with the painting color and the screen color. XOR mode makes the painting object able to be reappeared through repainting.
- Filling mode, which describes how the background color or image and the screen color are combined. The usual option is transparent, i.e. ignoring the background and reserving the existed screen content.
- Color mask, which is a bitmap, used to determine the style of the influence on the screen pixel by the painting operation.
- Pen style, the width, the cap shape, and the joint type when drawing line.
- Font, which usually corresponds to a group of bitmaps for a certain character set, and is used by text output functions. Specifying the properties such as size, style, and character set usually chooses font.
- Painting region, which is in concept a viewport with arbitrary size and position mapped to the window. Changing its origin can move the viewport. The system sometimes allows the viewport to be scaled.
- Clipping region. A painting primitive is valid only when it outputs within

the clipping region. The output outside the clipping region will be clipped. The clipping region is mainly used in repainting window, and consists of the invalid regions of the window. The application can adjust the clipping region.

- Current position, for example, you can use painting primitives such as MoveTo and LineTo to draw a line.

MiniGUI adopts the concept of graphics device context (DC) commonly used in GUI systems such as Windows and X Window. Each graphics device context defines a rectangular displaying output region and its related graphics properties in graphics output device or memory. When calling the graphics output function, an initialized graphics device context needs to be specified. That is to say, all the painting operations must work in a certain graphics device context.

From the point view of a program, an initialized graphics device context defines a graphics device environment, determines some basic properties of the graphics operations on it thereafter, and keeps these properties until they are changed. These properties include: the line color, filling color, font color, font shape, and so on. However, from the point view of GUI system, the meanings presented by a graphics device context are more complex, and at least include the following contents:

- Information of the device in which the device context is (display mode, color depth, and layout of video memory, etc.);
- Information of the window presented by this device context and the clipping region of this window by other windows (called "global clipping region" in MiniGUI);
- Basic operation functions of this context (point, line, polygon, filling, block operations, etc.), and its context information;
- Local information set by the program (painting property, mapping relationship, and local clipping region, etc.).

When you want to paint on a graphics output device (e.g. the monitor screen), you must first get a device context handle and take it as a parameter in GDI function to identify the graphics device context to be used when painting.

The device context includes many current properties to determine how GDI function works on the device. These properties make that the parameter transferred to GDI function may only include the starting coordinate or size information and need not include the other information required for displaying an object, since this information is a part of the device context. When you want to change on of these properties, you can call a function which can change the device context property, and GDI function calling for this device context will use the changed property.

The device context is actually a data structure managed internally in GDI. The device context is related to the specified displaying device. Some values in the device context are graphics properties. These properties define some special contents of the working status of some GDI painting functions. For example, for TextOut function, the property of the device context determines the text color, background color, the mapping manner of the x-coordinate and y-coordinate to the window client area, and the font used for displaying the text.

When the program needs to paint, it must first get a device context handle. The device context handle is a value presenting a device context, and the GDI functions use this handle.

### **13.3.2 Getting and Releasing of Device Context**

In MiniGUI, all the functions related to painting need a device context. When the program needs to paint, it must first get a device context handle. When the program finishes painting, it must release the device context handle. The program must get and release the handle during handling a single message. That is to say, if the program gets a device context handle when handling a message, it must release this device context handle before it finishes handling this message and quits the window procedure function.

One of the commonly used methods for getting and releasing the device context is through `BeginPaint` and `EndPaint` functions. The prototypes of

these two functions are as follow (`minigui/window.h`):

```
HDC GUIAPI BeginPaint(HWND hWnd);
void GUIAPI EndPaint(HWND hWnd, HDC hdc);
```

It should be noted that these two functions can only be called when handling `MSG_PAINT` message. Then handling of `MSG_PAINT` message has usually the following form:

```
MSG_PAINT:
    HDC hdc = BeginPaint (hWnd);
    /* use GDI functions to paint */
    ...
    EndPaint (hWnd, hdc);
    return 0;
}
```

`BeginPaint` takes the window handle `hWnd` according to the window procedure function as its argument, and returns a device context handle. Then GDI function can use this device context handle for graphics operations.

In a typical graphics user interface environment (including MiniGUI), the application is usually paint text and graphics in the client area of the window. However, the graphics system does not ensure the painting content in the client area be kept all the time. If the client area of this program window is overlaid by another window, the graphics system will not reserve the content of the overlaid window region and leave repainting of the window to the application. When needing to recover some contents of the window, the graphics system usually informs the program to update this part of client area. MiniGUI informs the application to perform the painting operation of the window client area by sending `MSG_PAINT` message to the application. If program consider it is necessary to update the content of client area, it can generate a `MSG_PAINT` message on its own, so that client area is repainted.

Generally speaking, in the following case, window procedure will receive a `MSG_PAINT` message:

- When the user moves or shows a window, MiniGUI sends `MSG_PAINT` message to the previously hidden window.



- When the program uses `InvalidateRect` function to update the invalid region of the window, a `MSG_PAINT` message will be generated;
- The program calls `UpdateWindow` function to redraw the window;
- The dialog box or message box over a window is destroyed;
- Pull down or popup menu is disappeared.

In some cases, MiniGUI saves some overlaid displaying area, and recovers them when necessary, for example the mouse cursor moving.

In usual cases, the window procedure function needs only to update a part of the client area. For example, a dialog box overlays only a part of the client area of a window; when the dialog box destroyed, redrawing of the part of the client area previously overlaid by the dialog box is needed. The part of the client area needed to be repainted called "invalid region".

MiniGUI gets the client area device context through `GetClientDC` in `BeginPaint` function, and then selects the current invalid region of the window to be the clipping region of the window. While `EndPaint` function clears the invalid region of the window, and release the device context.

Because `BeginPaint` function selects the invalid region of the window to the device context, you can improve the handling efficiency of `MSG_PAINT` through some necessary optimizations. For example, if a certain program wants fill some rectangles in the window client area; it can handle as follows in `MSG_PAINT` function:

```
MSG_PAINT:
{
    HDC hdc = BeginPaint (hWnd);

    for (j = 0; j < 10; j++) {
        if (RectVisible (hdc, rcs + j)) {
            FillBox (hdc, rcs[j].left, rcs[j].top, rcs [j].right, rcs [j].bottom);
        }
    }

    EndPaint (hWnd, hdc);
    return 0;
}
```

Thereby unnecessary redrawing operation can be avoided, and the painting

efficiency is improved.

The device context can be gotten and released through `GetClientDC` and `ReleaseDC` function. The device context gotten by `GetDC` is for the whole window, while the device context gotten `GetClientDC` is for the client area of the window. That is, for the device context gotten by the former function, its origin is located in upper-left corner of the window, and its output is clipped within the window area. For the device context gotten by the latter function, its origin is located in upper-left corner of the window client area, and its output is limited within the range of the window client area. Following are the prototypes of these three functions (`minigui/gdi.h`):

```
HDC GUIAPI GetDC (HWND hwnd);
HDC GUIAPI GetClientDC (HWND hwnd);
void GUIAPI ReleaseDC (HDC hdc);
```

`GetDC` and `GetClientDC` get a currently unused device context from some DCs reserved by the system. Therefore, the following two points should be noted:

1. After finishing using a device context gotten by `GetDC` or `GetClientDC`, you should release it as soon as possible by calling `ReleaseDC`.
2. Avoid using multiple device contexts at the same time, and avoid calling `GetDC` and `GetClientDC` in a recursive function.

For programming convenience and improving the painting efficiency, MiniGUI also provides functions to set up private device context. The private device context is valid in the whole life cycle of the window, thereby avoiding the getting and releasing process. The prototypes of these functions are as follow:

```
HDC GUIAPI CreatePrivateDC (HWND hwnd);
HDC GUIAPI CreatePrivateClientDC (HWND hwnd);
HDC GUIAPI GetPrivateClientDC (HWND hwnd);
void GUIAPI DeletePrivateDC (HDC hdc);
```

When creating a main window, if `WS_EX_USEPRIVATEDC` style is specified in the extended style of the main window, `CreateMainWindow` function will automatically set up a private device context for the window client area. You can get a device context through `GetPrivateClientDC` function. For a control,

if the control class has `CS_OWNDC` property, all the controls belonging to this control class will automatically set up a private device context.

`DeletePrivateDC` function is used to delete the private device context. For the two cases above, the system will automatically call `DeletePrivateDC` function when destroy the window.

### 13.3.3 Saving and Restoring of Device Context

The device context can be saved and restored through `SaveDC` and `RestoreDC` function. The prototypes of these two functions are as follow

(`minigui/gdi.h`):

```
int GUIAPI SaveDC (HDC hdc);  
BOOL GUIAPI RestoreDC (HDC hdc, int saved_dc);
```

### 13.3.4 Device Context in Memory

MiniGUI also provides the creating and destroying function of the device context in memory. Using the memory device context, you can set up a region similar to the video memory in the system memory, perform painting operations in this region, and copy to the video memory when finishing painting. There are many advantages using this painting method, e.g. fast speed, reducing the blinking phenomenon caused by direct operation on the video memory, etc. The prototypes of the function used to create and destroy the memory device context are as follow (`minigui/gdi.h`):

```
HDC GUIAPI CreateCompatibleDC (HDC hdc);  
void GUIAPI DeleteCompatibleDC (HDC hdc);
```

### 13.3.5 Screen Device Context

MiniGUI sets up a global screen device context after started up. This DC is for the whole screen, and has no predefined clipping region. In some applications, you can use directly this device context to paint, which may increase the paint efficiency remarkably. In MiniGUI, the screen device context is identified by `HDC_SCREEN`, and need no getting and releasing operations for this DC.

## 13.4 Mapping Mode and Coordinate Space

### 13.4.1 Mapping Mode

Once the Device Context (DC) has been initialized, the origin of the coordinates is usually the upper-left corner of the output rectangle, while the x coordinate axis is horizontal right and the y coordinate axis is vertical downward, with both using pixel as unit. Usually, in MiniGUI, the default unit used to draw graphics is pixel, however, we can choose other ways by changing GDI mapping mode. Mapping mode offers the measurement unit that can be used to convert page space (logical coordinate) into device space (device coordinate).

The mapping mode of GDI is a device context property that almost influences the graphics result in any client area. There are four other device context properties that are closely related to the mapping mode: window origin, window scope, viewport origin, and viewport scope.

Most GDI functions use coordinate value as arguments, which are called "logical coordinates". Before drawing something, MiniGUI firstly converts "logical coordinates" into "device coordinates", that is, pixel. The mapping mode, window and viewport origin, and window and viewport scope control such conversion. In addition, mapping mode also provides the direction of both x and y coordinate axis; in other words, it helps to confirm whether the x value is increasing or decreasing while you move to the left or right of the screen, so is the y value while the screen is moved up and down.

At present MiniGUI only supports two types of mapping modes:

- **MM\_TEXT**

Each logical unit is mapped as a device pixel. X coordinate increases progressively from left to right, while y coordinates increases progressively from top to bottom.

- **MM\_ANISOTROPIC**

Logical unit is mapped as arbitrary device space unit; the proportion of

the coordinate scale is also arbitrary. Using `SetWindowExt` and `SetViewportExt` to define unit, direction and scale.

The default mapping mode is `MM_TEXT`. Under this mapping mode, the logical coordinate is equal to the device coordinate. That is, the default unit of drawing graphics is pixel.

Changing mapping mode helps us to avoid scaling by ourselves; it is very convenient in some conditions. You can use `SetMapMode` function to set your mapping mode:

```
SetMapMode (hdc, mapmode);
```

The argument `mapmode` is one of the two mapping modes above. You can also use `GetMapMode` function to get current mapping mode:

```
mapmode = GetMapMode (hdc);
```

### 13.4.2 Viewport and Window

Mapping modes are used to define the mapping from “window” (logical coordinates) to “viewport” (device coordinates). “Window” is a rectangular area in the page coordinate space, while viewport is a rectangular area of the device coordinate space. It is “window” that determines which part of the geometric model of the page space should be displayed, while “viewport” determines where to draw. The scale between them determines the zoom of the coordinates. Viewport is pixel-based (device coordinates), while window is logical-based.

The following formulas can be used to convert between page space (window) coordinates and device space (viewport) coordinates:

```
xViewport = ((xWindow - xWinOrg) * xViewExt / xWinExt) + xViewOrg
yViewport = ((yWindow - yWinOrg) * yViewExt / yWinExt) + yViewOrg
```

- `xViewport, yViewport` the x value, y value in device unit
- `xWindow, yWindow` the x value, y value in logical unit (page

space unit)

- **xWinOrg, yWinOrg**      window x origin, window y origin
- **xViewOrg, yViewOrg**    viewport x origin, viewport y origin
- **xWinExt, yWinExt**      window x extent, window y extent
- **xViewExt, yViewExt**    viewport x extent, viewport y extent

The transfer principal of above formulas is: the scale of certain distance value in device space and extent value of coordinates should be equal to the scale of page space, in other words, the logical origin (**xWinOrg, yWinOrg**) is always mapped as device origin (**xViewOrg, yViewOrg**).

These two formulas use the origin and extent of both window and viewport. We can see from this that the scale between the extent of viewport and the extent of window is the conversion factor.

MiniGUI provides two functions to realize the conversion between device coordinates and logical coordinates. **LPtoDP** is used to convert from logical coordinates to device coordinates, while **DPtoLP** is used to convert from device coordinates to logical coordinates:

```
void GUIAPI DPtoLP (HDC hdc, POINT* pPt);
void GUIAPI LPtoDP (HDC hdc, POINT* pPt);
```

This conversion relies on the mapping mode of device context **hdc** as well as the origin and extent of the window and the viewport. Those x and y coordinates included in the structure **POINT pPt** will be converted into other coordinates in another coordinate system.

In the MiniGUI's source codes (**src/newgdi/coord.c**), the conversion between **LPtoDP** and **DPtoLP** are implemented as follow. It can be seen that the coordinate conversion between them is based on the formulas mentioned above.

```
void GUIAPI LPtoDP(HDC hdc, POINT* pPt)
{
    PDC pdc;

    pdc = dc_HDC2PDC(hdc);
```

```

    if (pdc->mapmode != MM_TEXT) {
        pPt->x = (pPt->x - pdc->WindowOrig.x)
            * pdc->ViewExtent.x / pdc->WindowExtent.x
            + pdc->ViewOrig.x;

        pPt->y = (pPt->y - pdc->WindowOrig.y)
            * pdc->ViewExtent.y / pdc->WindowExtent.y
            + pdc->ViewOrig.y;
    }
}

void GUIAPI DPtoLP (HDC hdc, POINT* pPt)
{
    PDC pdc;

    pdc = dc_HDC2PDC (hdc);

    if (pdc->mapmode != MM_TEXT) {
        pPt->x = (pPt->x - pdc->ViewOrig.x)
            * pdc->WindowExtent.x / pdc->ViewExtent.x
            + pdc->WindowOrig.x;

        pPt->y = (pPt->y - pdc->ViewOrig.y)
            * pdc->WindowExtent.y / pdc->ViewExtent.y
            + pdc->WindowOrig.y;
    }
}

```

In addition, the function **LPtoSP** and function **SPtoLP** can be used to convert between logical coordinates and screen coordinates:

```

void GUIAPI SPtoLP(HDC hdc, POINT* pPt);
void GUIAPI LPtoSP(HDC hdc, POINT* pPt);

```

### 13.4.3 Conversion of Device Coordinates

The mapping mode determines how MiniGUI maps logical coordinates into device coordinates. Device coordinates use pixel as unit, the value of x coordinate progressively increase from left to right, while the value of coordinate progressively increase from top to bottom.. There are three types of device coordinates in MiniGUI: screen coordinates, window coordinates, and client area coordinates. Usually device coordinates rely on the type of chosen device context to choose.

The (0, 0) point in screen coordinates is on the upper-left corner of the whole screen. When we need to use the entire screen, we can do it according to the screen coordinates. Screen coordinates are usually used in the functions that are irrelevant to window or functions that are tightly related to the screen, such as **GetCursorPos** and **SetCursorPos**. If the device context used by GDI

functions is `HDC_SCREEN`, the logical coordinates will be mapped as screen coordinates.

The coordinates in the window coordinates are based on entire window, including window border, caption bar, menu bar and scroll bar, in which the origin of window coordinates is the upper-left corner of the window. While using the device context handle returned by `GetDC`, the logical coordinates passed to GDI functions will be converted into window coordinates.

The point (0, 0) of the client area coordinates is the upper-left corner of this area. When we use the device context handle returned `GetClientDC` or `BeginPaint`, the logical coordinates passed to GDI functions will be converted to the client area coordinates.

When programming we need to know on which coordinate system the coordinates or position is based, as the meaning of position may be different under different situation. Some time we need get the coordinates in another coordinate system. MiniGUI provides functions that realize the conversion among those three device coordinate systems:

```
void GUIAPI WindowToScreen (HWND hWnd, int* x, int* y);
void GUIAPI ScreenToWindow (HWND hWnd, int* x, int* y);
void GUIAPI ClientToScreen (HWND hWnd, int* x, int* y);
void GUIAPI ScreenToClient (HWND hWnd, int* x, int* y);
```

`WindowToScreen` converts window coordinates into screen coordinates, while `ScreenToWindow` converts screen coordinates to window coordinates. The converted value is stored in the original place. `ClientToScreen` converts client coordinates into screen coordinates, while `ScreenToClient` converts screen coordinates to client coordinates.

#### 13.4.4 The Deviation and Zoom of Coordinate System

MiniGUI provides a set of functions that can be used to realize the deviation, zoom of the coordinate system. The prototypes of these functions are as follow:



```

void GUIAPI GetViewportExt(HDC hdc, POINT* pPt);
void GUIAPI GetViewportOrg(HDC hdc, POINT* pPt);
void GUIAPI GetWindowExt(HDC hdc, POINT* pPt);
void GUIAPI GetWindowOrg(HDC hdc, POINT* pPt);
void GUIAPI SetViewportExt(HDC hdc, POINT* pPt);
void GUIAPI SetViewportOrg(HDC hdc, POINT* pPt);
void GUIAPI SetWindowExt(HDC hdc, POINT* pPt);
void GUIAPI SetWindowOrg(HDC hdc, POINT* pPt);

```

Get-functions are used to get the origin and extent of the window and/or the viewport, the value is stored in `POINT` structure `pPt`; Set-functions use the value of `pPt` to set the origin and the extent of the window and/or the viewport.

## 13.5 Rectangle and Region Operations

### 13.5.1 Rectangle Operations

Rectangle usually refers to a rectangular region on the screen. It is defined in MiniGUI as follows:

```

typedef struct _RECT
{
    int left;
    int top;
    int right;
    int bottom;
} RECT;

```

In short, rectangle is a data structure used to represent a rectangular region on the screen. It defines the x coordinate and y coordinate of the upper-left corner of the rectangle (left and top), as well as the x coordinate and y coordinate of the lower-bottom corner of the rectangle. It is necessary to notice that the right and bottom borders are not included by MiniGUI's rectangle. For example, if we want to figure a scan line on the screen, we should use

```
RECT rc = {x, y, x + w, y + 1};
```

to represent it. In that `x` is the jumping-off point while `y` is the vertical place of that scan line, and `w` is the width of that scan line.

MiniGUI provides a group of functions, which can operate on `RECT` objects:

- `SetRect` assigns each parameter of a `RECT` object.
- `SetRectEmpty` sets a `RECT` object to be empty. In MiniGUI, the empty rectangle is defined as a rectangle with its width or height as zero.
- `IsRectEmpty` determines if the given `RECT` object is empty.
- `NormalizeRect` normalizes a given rectangle. The rectangle should meet the requirement of `right > left` and `bottom > top`. Those rectangles that meet the above requirements are called normalized rectangles. This function can normalize any rectangle.
- `CopyRect` copies between two rectangles.
- `EqualRect` determines if two `RECT` objects are equal, that is, if the all parameters are equal.
- `IntersectRect` gets the intersection of two `RECT` objects. If there is no intersection between those two rectangles, the function will return to `FALSE`.
- `DoesIntersect` determines if the two rectangles are intersected.
- `IsCovered` determines if `RECT A` completely overlay `RECT B`, that is, if `RECT B` is the true subset of `RECT A`.
- `UnionRect` gets the union of two rectangles. If there is no union, the function will return `FALSE`; any point included in the union should also belong to either of the rectangles.
- `GetBoundRect` gets the bound of the two rectangles, that is, the minimum rectangle bounding the two rectangles.
- `SubtractRect` subtracts one rectangle from another one. Such subtraction may result in four non-intersected rectangles. This function will return the number of the result rectangles.
- `OffsetRect` offsets the given `RECT` object.
- `InflateRect` inflates the given `RECT` object. The width and height of the inflated rectangle will be twice of the given inflation value.
- `InflateRectToPt` inflates the given `RECT` object to a given point.
- `PtInRect` determines if the given point lies in the given `RECT` object.

### 13.5.2 Region Operations

Region is a scope on the screen, which is defined as a collection of

non-intersected rectangles and represented as a linked list. Region can be used to represent the clipped region, invalid region, and visible region. In MiniGUI, the definition of region equals to the definition of clipped region, which is defined as follows (`minigui/gdi.h`):

```
typedef struct _CLIPRECT
{
    RECT rc;
    struct _CLIPRECT* next;
#ifdef _USE_NEWGAL
    struct _CLIPRECT* prev;
#endif
} CLIPRECT;
typedef CLIPRECT* PCLIPRECT;

typedef struct _CLIPRGN
{
#ifdef _USE_NEWGAL
    BYTE type; /* type of region */
    BYTE reserved[3];
#endif
    RECT rcBound;
    PCLIPRECT head;
    PCLIPRECT tail;
    PBLOCKHEAP heap;
} CLIPRGN;
```

Each clipped region has one **BLOCKHEAP** member, which is the private heap of **RECT** objects used by the clipped region. Before using a region object, we should firstly build up a **BLOCKHEAP** object, and then initialize the region object. Shown as follows:

```
static BLOCKHEAP sg_MyFreeClipRectList;

...

CLIPRGN my_region

InitFreeClipRectList (&sg_MyFreeClipRectList, 20);
InitClipRgn (&my_regioni, &sg_MyFreeClipRectList);
```

When being actually used, multiple regions can share one **BLOCKHEAP** object.

Following operations can be done after initializing the region object:

- **SetClipRgn** sets only one rectangle in the region;
- **ClipRgnCopy** copies one region to another;
- **ClipRgnIntersect** gets the intersection of two regions;
- **GetClipRgnBoundRect** gets the bounding box of the region;
- **IsEmptyClipRgn** determines if the region is empty, that is, if the region

includes any rectangle;

- **EmptyClipRgn** releases the rectangles in the region and empty the region;
- **AddClipRect** adds a rectangle to the region, but it does not determine if the region intersects with the rectangle;
- **IntersectClipRect** gets the intersection of region and given rectangle;
- **SubtractClipRect** subtracts the given rectangle from the region.
- **CreateClipRgn** creates an empty region.
- **DestroyClipRgn** clears and destroys a region.

The operations of rectangles and regions form the main algorithms of window management. It is very important in GUI programming, as it is also one of the basic algorithms of advanced GDI function.

## 13.6 Basic Graphics Drawing

### 13.6.1 Basic Drawing Attributes

Before understanding basic drawing functions, we need to know basic drawing attributes. In the current MiniGUI version, the drawing attributes include pen color, brush color, text background mode, text color, TAB width, and so on. The operation functions for these attributes are listed in Table 13.1.

Table 13.1 Basic drawing attributes and operation function

Drawing Attributes	Operations	Effectuated GDI Functions
Pen color	GetPenColor/SetPenColor	LineTo, Circle, Rectangle
Brush color	GetBrushColor/SetBrushColor	FillBox
Text background mode	GetBkMode/SetBkMode	TextOut, DrawText
Text color	GetTextColor/SetTextColor	TextOut, DrawText
TAB width	GetTabStop/SetTabStop	TextOut, DrawText

The current MiniGUI version also defines some functions for brush and pen. We will discuss the functions in Chapter 15.

### 13.6.2 Basic Drawing Functions

In MiniGUI, basic drawing functions include such basic functions such as **SetPixel**, **LineTo**, **Circle**, and so on. The prototypes are defined as follow:

```
void GUIAPI SetPixel (HDC hdc, int x, int y, gal_pixel c);
void GUIAPI SetPixelRGB (HDC hdc, int x, int y, int r, int g, int b);
gal_pixel GUIAPI GetPixel (HDC hdc, int x, int y);
void GUIAPI GetPixelRGB (HDC hdc, int x, int y, int* r, int* g, int* b);
gal_pixel GUIAPI RGB2Pixel (HDC hdc, int r, int g, int b);

void GUIAPI LineTo (HDC hdc, int x, int y);
void GUIAPI MoveTo (HDC hdc, int x, int y);

void GUIAPI Circle (HDC hdc, int x, int y, int r);
void GUIAPI Rectangle (HDC hdc, int x0, int y0, int x1, int y1);
```

We need to differentiate two basic conceptions: pixel value and RGB value. RGB is a way to represent color according to the different proportion of tricolor. Usually, the red, blue and green can get any value between 0 and 255, so there are 256x256x256 different colors. However, in video memory, the color displayed on the screen is not represented by RGB; it is represented by pixel value. The scope of pixel value varies according to the difference of video mode. In 16-color mode, the scope is in [0, 15]; while in 256-color mode, the scope is [0, 255]; in 16bit-color mode, the scope is [0, 2<sup>16</sup> - 1]. Here the number of bits of one mode refers to the number of bits per pixel.

When setting the color of a pixel in MiniGUI, you can directly use pixel value (**SetPixel**) or **SetPixelRGB**. The function **RGB2Pixel** can convert RGB value into pixel value.

### 13.6.3 Clipping Region Operations

Clipping can be done when using device context to draw. MiniGUI provides following functions to clip the given device context (**minigui/gdi.h**):

```
// Clipping support
void GUIAPI ExcludeClipRect (HDC hdc, int left, int top,
                             int right, int bottom);
void GUIAPI IncludeClipRect (HDC hdc, int left, int top,
                             int right, int bottom);
void GUIAPI ClipRectIntersect (HDC hdc, const RECT* prc);
void GUIAPI SelectClipRect (HDC hdc, const RECT* prc);
void GUIAPI SelectClipRegion (HDC hdc, const CLIPRGN* pRgn);
void GUIAPI GetBoundsRect (HDC hdc, RECT* pRect);
```

```
BOOL GUIAPI PtVisible (HDC hdc, const POINT* pPt);  
BOOL GUIAPI RectVisible (HDC hdc, const RECT* pRect);
```

**ExcludeClipRect** is used to exclude the given rectangle region from current visible region, then the visible region will be reduced; **IncludeClipRect** adds a rectangle region into the visible region of device context, then the visible region will be extended; **ClipRectIntersect** sets the visible region of device context as the intersection of the existed region and the given rectangle; **SelectClipRect** resets the visible region of device context as the given rectangle region; **SelectClipRegion** sets the visible region of device context as the given region; **GetBoundsRect** is used to get the minimum bounding rectangle of the visible region; **PtVisible** and **RectVisible** determine if the given point or rectangle is visible, that is, if they are included or partly included in the visible region.

## 13.7 Text and Font

It is necessary for any GUI system to provide the support for font and charset. However, different GUI has its different way to implement the multi-font and multi-charset. For example, QT/Embedded uses UNICODE, which is a popular solution for most general operating systems. However, it is not acceptable for some embedded systems as the conversion between UNICODE and other charsets will increase the size of GUI system.

The MiniGUI does not use UNICODE to support multiple charsets; instead, it uses a different policy to handle multiple charsets. For a certain charset, MiniGUI uses the same internal encoding presentation as the charset standard. After using a series of abstract interfaces, MiniGUI provides a consistent analysis interface to multiple charsets. This interface can be used in font module; also can be used to analysis multi-bytes string. When adding support for a new charset (encoding), the only thing need to do is to provide an interface to the charset (encoding). So far MiniGUI has been able to support ISO8859-x single byte charsets, and some multi-bytes charsets, including GB2312, GBK, GB18030, BIG5, EUCKR, Shift-JIS, EUCJP, Unicode and so on.

Similar to charset, MiniGUI also defines a series of abstract interfaces to font. When adding support for a new font type, we just need to realize the interface of such type of font. So far MiniGUI has got the support of RBF and VBF, QPF, TrueType and Adobe Type1.

Based on the abstract interface of multi-font and multi-charset, MiniGUI provides a consistent interface to applications through logical font.

We will discuss the interfaces about text and font in Chapter 14 of this guide.

## 13.8 Bitmap Operations

Bitmap operation function is very important in GDI function of MiniGUI. In fact, most advanced drawing operation functions are based on the bitmap operations, for example, the text output functions.

The main bitmap operations of MiniGUI are listed below (`minigui/gdi.h`):

```
void GUIAPI FillBox (HDC hdc, int x, int y, int w, int h);
void GUIAPI FillBoxWithBitmap (HDC hdc, int x, int y, int w, int h,
                               PBITMAP pBitmap);
void GUIAPI FillBoxWithBitmapPart (HDC hdc, int x, int y, int w, int h,
                                   int bw, int bh, PBITMAP pBitmap, int xo, int yo);

void GUIAPI BitBlt (HDC hsrc, int sx, int sy, int sw, int sh,
                   HDC hdst, int dx, int dy, DWORD dwRop);
void GUIAPI StretchBlt (HDC hsrc, int sx, int sy, int sw, int sh,
                       HDC hdst, int dx, int dy, int dw, int dh, DWORD dwRop);
```

### 13.8.1 Concept of Bitmap

Most graphical output devices are raster operation devices, such as printer and video display. Raster operation devices use dispersed pixel point to indicate the image being output. Bitmap is a two-dimension array, which records the pixel value of every pixel point in that image. In bitmap, each pixel value points out the color of that point. For monochrome bitmap, only one bit is needed for each pixel; gray bitmap and multicolor bitmap need multiple bits to present the value of color for the pixel. Bitmap is always used to indicate complicated image of the real world.

Bitmap has two main disadvantages. First, bitmap is easy to be influenced by the device independence, for example, resolution and color. Bitmap always suggests certain display resolution and image aspect ratio. Bitmap can be zoomed in and zoomed out, but during this process certain rows and columns are copied or deleted, which will result in image distortion. The second disadvantage of bitmap is that it needs huge storage space. The storage space of bitmap is determined by the size of bitmap and the number of the color. For instance, to indicate 320x240 needs at least  $320 \times 240 \times 2 = 150\text{KB}$  storage space on a 16-bit color screen, while to store 1024x768 needs more than 2MB on a 24 bit-color screen.

Bitmap is rectangular, the height and width of the image use pixel as unit. Bitmap is always stored in memory and ranked by rows. In each row, the pixel starts from left to right, in turn be stored.

### **13.8.2 Bitmap Color**

The color of bitmap usually uses bit-count of pixel value to measure. This value is called color depth of the bitmap, or bit-count, or bits per pixel (bpp). Each pixel in the bitmap has same color bit-count.

The so-call monochrome bitmap is the one that the color value of each pixel is stored in one bit. The color value of each pixel in monochrome bitmap is 0 or 1, respectively represents black and white. The color value of each pixel stored by four bits can demonstrate 16 kinds of color, the one stored by eight can demonstrate 256 while the one saved by 16 can demonstrate 65536 kinds of color.

Two of the important display hardware in PC is video adapter and monitor. The video adapter is a circuitry board inserted in the main board, which consists of registers, memory (RAM, ROM and BIOS), and control circuitry. Most graphics video adapters are based on VGA model. For most embedded devices, the display hardware is always LCD and its LCD controller.



Both PC display adapter and LCD controller have a video RAM (VRAM) to represent image on the screen. VRAM have to be big enough to manage all pixels on the screen. The programrs change the screen display by directly or indirectly fetch the data stored in VRAM. Most video hardware provides the ability of visiting VRAM from CPU address and data BUS. It equals to map VRAM to CPU address space, and increase the visiting speed.

PC monitor and LCD are all raster operation devices. Each point on the screen is a pixel and thus the display screen looks like a pixel matrix. VRAM stores data according to video mode. It records the color value of each pixel on the display screen. As we know, the computer uses binary ways to store data, in which 0 and 1 are used to represent each bit. As for monochrome video mode, the color value of one pixel point only needs one bit of VRAM to represent, if this bit is 1, it means the pixel is light. As for multicolor video mode, the color information of the pixel needs more bytes or bits to represent. 16-color video mode needs four bits to store one color value; 256-color mode needs 8 bits (1 byte) while 16-bit true color video mode needs two bytes to store the color value for one pixel.

When using 16-color and 256-color video mode, a color table is needed to translate the RGB color data into pixel value of video device. The so-called color table is also called palette. When displaying a pixel on the screen, the video adapter will first read the data stored in video memory and get a group of RGB color information, then, adjust radiation tube of the display, then, a point will be showed on the corresponding place of the screen. When all points in the display memory have been showed on the screen, the image is formed. You can also change the correspondent RGB value of the color table to get the self-defined color according to your needs. When video mode attains a true-color level, the palette becomes meaningless, as the information stored in video memory is already RGB information of pixel. So the pallet is no longer needed in true-color mode.

The color used for screen display usually uses RGB color system, in which one color is determined by the value of red, green, and blue. Different display

device has different color scope; so a certain color may not be displayed on all devices. Most graphic systems define their own color standard that is irrelevant to the device.

The display of color is very complicated and always depends on the actual display ability of display device and the requirement of applications.

Application may use monochrome and fixed palette, adjustable palette or true color, while display system will try to display the closest color in order to meet the requirement of application. True color display device can simulate a palette by mapping all color during the drawing process. The palette device also can simulate true color by setting palette. The palette device provides a color table to disperse color scope, and then maps the needed color to the closest color. On a small palette display device, a way named dithering can be used to increase the displayed color scope. The palette that can be modified needs support of hardware.

The video adapter of true color uses 16-bits or 24-bits per pixel. When using 16-bit, 6 bits will be assigned to green, red and blue get 5 bits each, it is totally 65536 kinds of color; when only using 15 bits, red, green and blue get 5 bits each, it is 32768 kinds of color. Usually 16-bit color is called high color, sometime also called true color. 24-bit is called true color as it can indicate millions of color and has reached the limitation that human eyes are able to discern.

### 13.8.3 Device-Dependent Bitmap and Device-Independent Bitmap

Device-dependent bitmap means the one that includes pixel matching the video mode of a given device context, not the bitmap that is independent to video device. In MiniGUI, these two bitmap types are represented respectively by **BITMAP** and **MYBITMAP** data structures, showed as follow (**minigui/gdi.h**):

```
#ifndef _USE_NEWGAL
#define BMP_TYPE_NORMAL      0x00
#define BMP_TYPE_RLE        0x01
#define BMP_TYPE_ALPHA      0x02
#define BMP_TYPE_ALPHACHANNEL 0x04
#define BMP_TYPE_COLORKEY   0x10
```

```

#define BMP_TYPE_PRIV_PIXEL    0x20

/** Expanded device-dependent bitmap structure. */
struct _BITMAP
{
    /**
     * Bitmap types, can be OR'ed by the following values:
     * - BMP_TYPE_NORMAL\n
     *   A normal bitmap, without alpha and color key.
     * - BMP_TYPE_RLE\n
     *   A RLE encoded bitmap, not used so far.
     * - BMP_TYPE_ALPHA\n
     *   Per-pixel alpha in the bitmap.
     * - BMP_TYPE_ALPHACHANNEL\n
     *   The \a bmAlpha is a valid alpha channel value.
     * - BMP_TYPE_COLORKEY\n
     *   The \a bmColorKey is a valid color key value.
     * - BMP_TYPE_PRIV_PIXEL\n
     *   The bitmap have a private pixel format.
     */
    UInt8    bmType;
    /** The bits per pixel. */
    UInt8    bmBitsPerPixel;
    /** The bytes per pixel. */
    UInt8    bmBytesPerPixel;
    /** The alpha channel value. */
    UInt8    bmAlpha;
    /** The color key value. */
    UInt32    bmColorKey;

    /** The width of the bitmap */
    UInt32    bmWidth;
    /** The height of the bitmap */
    UInt32    bmHeight;
    /** The pitch of the bitmap */
    UInt32    bmPitch;
    /** The bits of the bitmap */
    UInt8*    bmBits;

    /** The private pixel format */
    void*     bmAlphaPixelFormat;
};

#else

/* expanded bitmap struct */
struct _BITMAP
{
    UInt8    bmType;
    UInt8    bmBitsPerPixel;
    UInt8    bmBytesPerPixel;
    UInt8    bmReserved;

    UInt32    bmColorKey;

    UInt32    bmWidth;
    UInt32    bmHeight;
    UInt32    bmPitch;

    void*     bmBits;
};

#endif /* _USE_NEWGAL */

#define MYBMP_TYPE_NORMAL        0x00000000
#define MYBMP_TYPE_RLE4         0x00000001
#define MYBMP_TYPE_RLE8         0x00000002
#define MYBMP_TYPE_RGB          0x00000003
#define MYBMP_TYPE_BGR          0x00000004
#define MYBMP_TYPE_RGBA         0x00000005
#define MYBMP_TYPE_MASK         0x0000000F

#define MYBMP_FLOW_DOWN         0x00000010
#define MYBMP_FLOW_UP           0x00000020
#define MYBMP_FLOW_MASK         0x000000F0

```

```
#define MYBMP_TRANSPARENT      0x00000100
#define MYBMP_ALPHACHANNEL    0x00000200
#define MYBMP_ALPHA           0x00000400

#define MYBMP_RGBSIZE_3       0x00001000
#define MYBMP_RGBSIZE_4       0x00002000

#define MYBMP_LOAD_GRAYSCALE  0x00010000
#define MYBMP_LOAD_NONE       0x00000000

/** Device-independent bitmap structure. */
struct _MYBITMAP
{
    /**
     * Flags of the bitmap, can be OR'ed by the following values:
     * - MYBMP_TYPE_NORMAL\n
     *   A normal palette bitmap.
     * - MYBMP_TYPE_RGB\n
     *   A RGB bitmap.
     * - MYBMP_TYPE_BGR\n
     *   A BGR bitmap.
     * - MYBMP_TYPE_RGBA\n
     *   A RGBA bitmap.
     * - MYBMP_FLOW_DOWN\n
     *   The scanline flows from top to bottom.
     * - MYBMP_FLOW_UP\n
     *   The scanline flows from bottom to top.
     * - MYBMP_TRANSPARENT\n
     *   Have a transparent value.
     * - MYBMP_ALPHACHANNEL\n
     *   Have a alpha channel.
     * - MYBMP_ALPHA\n
     *   Have a per-pixel alpha value.
     * - MYBMP_RGBSIZE_3\n
     *   Size of each RGB triple is 3 bytes.
     * - MYBMP_RGBSIZE_4\n
     *   Size of each RGB triple is 4 bytes.
     * - MYBMP_LOAD_GRAYSCALE\n
     *   Tell bitmap loader to load a grayscale bitmap.
     */
    DWORD flags;
    /** The number of the frames. */
    int frames;
    /** The pixel depth. */
    UInt8 depth;
    /** The alpha channel value. */
    UInt8 alpha;
    UInt8 reserved [2];
    /** The transparent pixel. */
    UInt32 transparent;

    /** The width of the bitmap. */
    UInt32 w;
    /** The height of the bitmap. */
    UInt32 h;
    /** The pitch of the bitmap. */
    UInt32 pitch;
    /** The size of the bits of the bitmap. */
    UInt32 size;

    /** The pointer to the bits of the bitmap. */
    BYTE* bits;
};
```

### 13.8.4 Loading a Bitmap from File

The function group `LoadBitmap` of MiniGUI can load certain bitmap file as device dependent bitmap object, that is, `BITMAP` object. Currently MiniGUI can be used to load different format of bitmap file, including Windows BMP file,

JPEG file, GIF file, PCX file, and TGA file. **LoadMyBitmap** function group can be used to load bitmap file as device-independent bitmap objects. The related function prototypes are as follow (**minigui/gdi.h**):

```
int GUIAPI LoadBitmapEx (HDC hdc, PBITMAP pBitmap, MG_WOps* area, const char* ext);
int GUIAPI LoadBitmapFromFile (HDC hdc, PBITMAP pBitmap, const char* spFileName);
int GUIAPI LoadBitmapFromMemory (HDC hdc, PBITMAP pBitmap,
    void* mem, int size, const char* ext);

#define LoadBitmap LoadBitmapFromFile

void GUIAPI UnloadBitmap (PBITMAP pBitmap);

int GUIAPI LoadMyBitmapEx (PMYBITMAP my_bmp, RGB* pal, MG_WOps* area, const char* ext);
int GUIAPI LoadMyBitmapFromFile (PMYBITMAP my_bmp, RGB* pal, const char* file_name);
int GUIAPI LoadMyBitmapFromMemory (PMYBITMAP my_bmp, RGB* pal,
    void* mem, int size, const char* ext);

void* GUIAPI InitMyBitmapSL (MG_WOps* area, const char* ext,
    MYBITMAP* my_bmp, RGB* pal);
int GUIAPI LoadMyBitmapSL (MG_WOps* area, void* load_info,
    MYBITMAP* my_bmp, CB_ONE_SCANLINE cb, void* context);
int GUIAPI CleanupMyBitmapSL (MYBITMAP* my_bmp, void* load_info);

BOOL GUIAPI PaintImageEx (HDC hdc, int x, int y, MG_WOps* area, const char *ext);
int GUIAPI PaintImageFromFile (HDC hdc, int x, int y, const char *file_name);
int GUIAPI PaintImageFromMem (HDC hdc, int x, int y, const void* mem,
    int size, const char *ext);

void GUIAPI UnloadMyBitmap (PMYBITMAP my_bmp);

int GUIAPI ExpandMyBitmap (HDC hdc, PBITMAP bmp, const MYBITMAP* my_bmp,
    const RGB* pal, int frame);
```

In order to decrease the memory usage, **LoadBitmapEx** can load the scan line of the bitmap object one by one into a bitmap object independent to device. In this process, **InitMyBitmapSL** initializes for the loading of the **LoadMyBitmapSL**; after loading every scan line, **LoadMyBitmapSL** calls the user defined callback function **cb**. In this way, application can deal with the loaded scan line, such as transforming to a scan line of the **BITMAP** structure, or output to the window client region. Finally, after **LoadMyBitmapSL** returns, user should call **CleanupMyBitmapSL** function to release the resource.

The design idea of **LoadMyBitmapSL** function is similar to MiniGUI curve generators. **LoadBitmapEx** function group and **PaintImageEx** function group mentioned below are all implemented based on **LoadMyBitmapSL** function group. For more information about MiniGUI curve generators, please refer to segment 15.6.

A group of functions, such as **PaintImageEx**, **PaintImageFromFile** and

**PaintImageFromMem** are added for NEWGAL. This group of functions can draw the image specified by the parameters on the DC directly without loading into a **BITMAP** object to decrease the memory usage. And it needs to be noted that this group of functions cannot scale the image.

**ExpandMyBitmap** can convert **MYBITMAP** into bitmap object dependent to a certain device context. After the applications get **BITMAP**, they can call some functions (that will be mentioned in the next section) to fill bitmap in some place of DC.

It is worth noticing that when loading bitmaps from files, MiniGUI uses the suffix name of the file to determine the type of one bitmap. The support for Windows BMP and GIF file are built-in MiniGUI, while the support for JPEG and PNG files are based on libjpeg and libpng libraries.

### 13.8.5 Filling Block

The function used to fill block in MiniGUI is **FillBoxWithBitmap** and **FillBoxWithBitmapPart**. **FillBoxWithBitmap** uses a device-dependent bitmap object to fill a rectangle box, while **FillBoxWithBitmapPart** uses a part of device-dependent bitmap object to fill a rectangle box. Both **FillBoxWithBitmap** and **FillBoxWithBitmapPart** can be used to scale the bitmap.

```
void GUIAPI FillBoxWithBitmap (HDC hdc, int x, int y, int w, int h,
                              PBITMAP pBitmap);
void GUIAPI FillBoxWithBitmapPart (HDC hdc, int x, int y, int w, int h,
                                   int bw, int bh, PBITMAP pBitmap, int xo, int yo);
```

The program in List 13.1 loads a bitmap from a file and displays it on the screen (please refers to Fig. 13.1). The complete code of this program can be seen from **loadbmp.c** included in **mg-samples** program package for this guide.

List 13.1 Loading and showing a bitmap

```
case MSG_CREATE:
    if (LoadBitmap (HDC_SCREEN, &bmp, "bkgnd.jpg"))
        return -1;
```

```

return 0;

case MSG_PAINT:
    hdc = BeginPaint (hWnd);

    /* Show the bitmap scaled on the position of (0,0,100,100) in the window*/
    FillBoxWithBitmap (hdc, 0, 0, 100, 100, &bmp);
    Rectangle (hdc, 0, 0, 100, 100);

    /*
     * Show the bitmap scaled on the position of (100,0,200,200) in the window.
     * The bitmap displayed is twice the above bitmap in size.
     */
    FillBoxWithBitmap (hdc, 100, 0, 200, 200, &bmp);
    Rectangle (hdc, 100, 0, 300, 200);

    /*
     * Display the bitmap by actual size, but take a part of bitmap located
     * in (10, 10, 410, 210) and
     * display it on the position of (0, 200, 400, 200) in screen
     */
    FillBoxWithBitmapPart (hdc, 0, 200, 400, 200, 0, 0, &bmp, 10, 10);
    Rectangle (hdc, 0, 200, 400, 400);

    EndPaint (hWnd, hdc);
    return 0;

case MSG_CLOSE:
    UnloadBitmap (&bmp);
    DestroyMainWindow (hWnd);
    PostQuitMessage (hWnd);
    return 0;

```

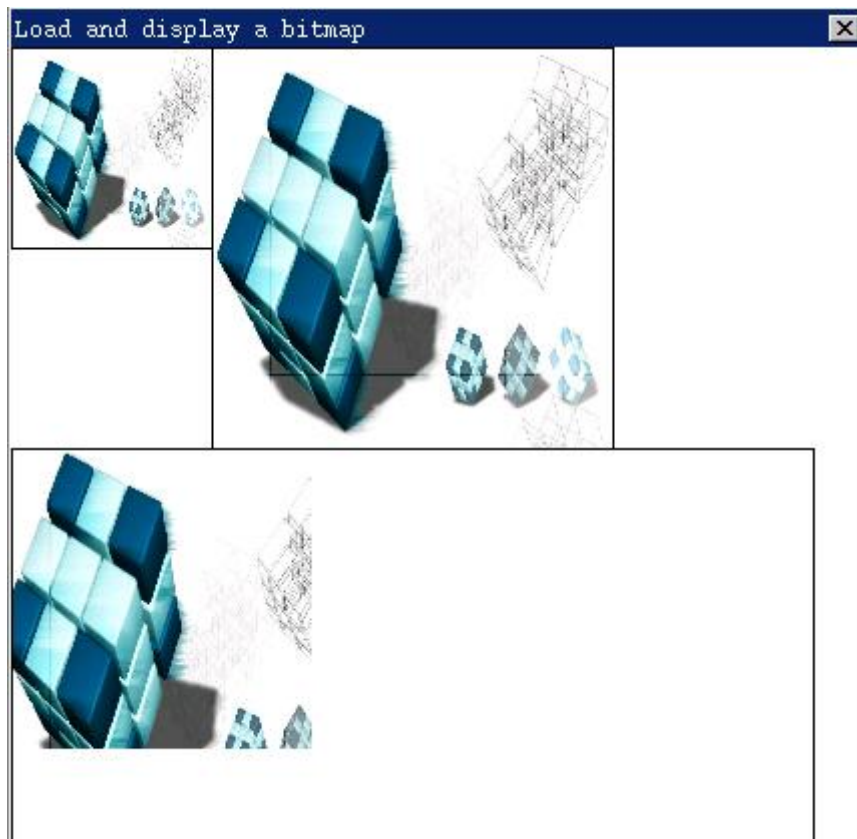


Fig 13.1 Loading and showing a bitmap

### 13.8.6 Bit Blitting

Bit blitting means that copy the pixel data of certain rectangle in memory or video RAM to another memory or display region. Bit blitting usually is a high-speed image transfer process.

The function to perform this operation (bit blitting) is `BitBlt` and `StretchBlt`. `BitBlt` is used to copy the display memory between two device contexts, while `StretchBlt` performs stretch operation based on `BitBlt`.

The prototype of `BitBlt` function is as follows:

```
void GUIAPI BitBlt (HDC hsrc, int sx, int sy, int sw, int sh,
                  HDC hdst, int dx, int dy, DWORD dwRop);
```

`BitBlt` is used to transfer the image (pixel data) of a certain rectangle in the source device context to a same-size rectangle in the destination device context. In the GDI interface based on the original GAL, two device contexts operated by the function must be compatible, that is, the two device contexts have same color format (the GDI interface based on NEWGAL is without such limitation). Source device context can equal to target device context. The meaning of arguments of `BitBlt` function is illustrated as follow:

- `hsrc`: the source device context;
- `sx`, `sy`: the upper-left coordinates of the rectangle in the source device context;
- `sw`, `sh`: the width and height of the source rectangle
- `hdst`: the destination device context
- `dx`, `dy`: the upper-left coordinates of the rectangle in the destination device context
- `dwRop`: raster operation, currently ignored

The program in List 13.2 first fills in a round shape then uses `BitBlt` to copy it and fill out whole client area. The complete code of this program can be seen from program `bitb1t.c` included in `mg-samples` program package.



## List 13.2 Using BitBlt function

```

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>

static int BitbltWinProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    int x, y;

    switch (message) {
    case MSG_PAINT:
        hdc = BeginPaint (hWnd);
        SetBrushColor (hdc, PIXEL_blue);
        /* Draw a circle in client area of window */
        FillCircle (hdc, 10, 10, 8);
        for (y = 0; y < 240; y += 20) {
            for (x = 0; x < 320; x += 20) {
                /* Copy the circle in other position of client area by BitBlt function */
                BitBlt (hdc, 0, 0, 20, 20, hdc, x, y, 0);
            }
        }
        EndPaint (hWnd, hdc);
        return 0;

    case MSG_CLOSE:
        DestroyMainWindow (hWnd);
        PostQuitMessage (hWnd);
        return 0;
    }

    return DefaultMainWinProc (hWnd, message, wParam, lParam);
}

/* /* Following codes to create the main window are omitted */ */

```

The output of the above code can be seen from Fig. 13.2.

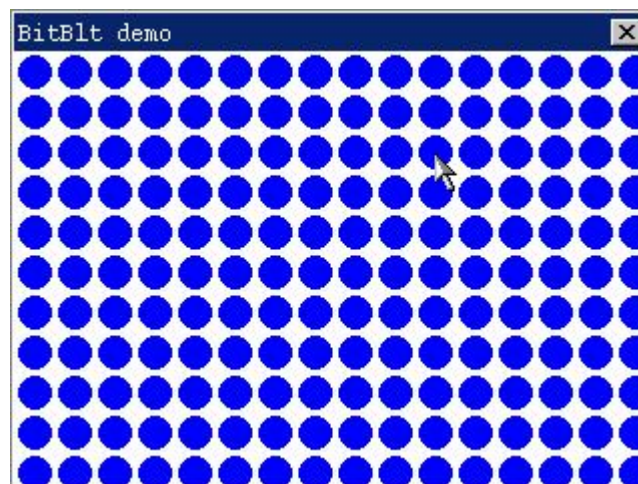


Fig. 13.2 Presentation of BitBlt operation

In program `bitblt.c`, the source device context and target device context of `BitBlt` operation are client area of window; the handle of device context is

obtained from function **BeginPaint**.

This program first draws a filled circle on the upper-left corner of the client area of the window. The coordinates of the circle center are (10, 10). The upper-left corner of the bounding box of the circle is (0, 0), both of the width and height of the box are 20. Then the program goes into a loop and uses **BitBlt** to copy the image located in the box to other places of window client area.

In this program, **BitBlt** copies certain data in the video memory to another place of the video memory.

Another bit blitting function is **StretchBlt**, which is different from **BitBlt** as it can stretch the image while copying. The prototype of **StretchBlt** is as follows:

```
void WINAPI StretchBlt (HDC hsrc, int sx, int sy, int sw, int sh,
                      HDC hdst, int dx, int dy, int dw, int dh, DWORD dwRop);
```

Compared with **BitBlt**, function **StretchBlt** adds two more arguments, which point out the width and height of the destination rectangle. The program in List 13.3 shows the usage of function **StretchBlt**.

List 13.3 Using StretchBlt function

```
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

static int StretchbltWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;

    switch (message) {
    case MSG_PAINT:
        hdc = BeginPaint(hWnd);
        SetBrushColor(hdc, PIXEL_blue);
        /* Draw a circle in client area of window */
        FillCircle(hdc, 10, 10, 8);
        /* Enlarge and Copy the circle above another position by StretchBlt function*/
        StretchBlt(hdc, 0, 0, 20, 20, hdc, 20, 20, 180, 180, 0);
        EndPaint(hWnd, hdc);
        return 0;

    case MSG_CLOSE:
```

```
    DestroyMainWindow (hWnd);  
    PostQuitMessage (hWnd);  
    return 0;  
}  
  
return DefaultMainWinProc(hWnd, message, wParam, lParam);  
}  
  
/* /* Following codes to create the main window are omitted */  
*/
```

The output of program is as shown in Fig. 13.3.

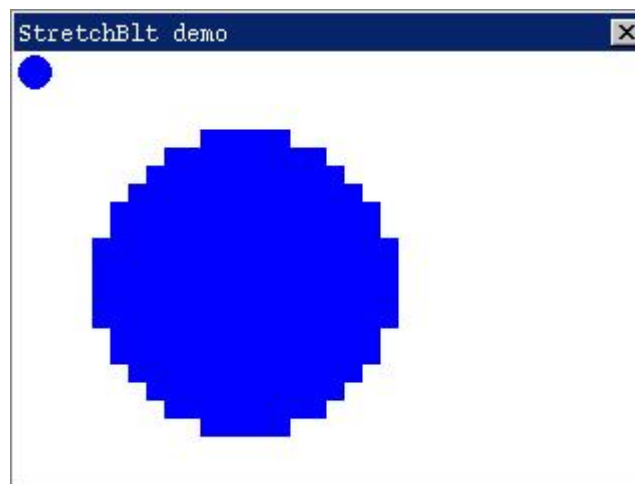


Fig. 13.3 The presentation of StretchBlt operation

**StretchBlt** operation involves the copy or combination of pixel, so the image may look abnormal, such as distortion.

## 13.9 Palette

Palette is the tool of video hardware used to map the color index value to RGB color value.

### 13.9.1 Why Is the Palette Needed?

Why is the palette needed? Let's first see how 16-color (4 bits per pixel) and 256-color (8 bits per pixel) model work. Let's start from hardware layer and then software interface. The cathode ray tube has 3 electronic guns, each of which is respectively responsible for red, green and blue. Each electronic gun can be adjusted to different degree of light. The combination of tricolor with

different light degree forms kinds of color variations on the screen. The physical memory (video RAM) on the video card is usually called FrameBuffer. All display screen operations use read or write the frame buffer to plot. Such block of video memory may have different organizing format under different color mode. For example, under monochrome mode each bit represents one pixel, in other words, each byte represents eight pixels. We call the value used to represent pixel as pixel value, for instance, in 16-color mode, the possible pixel value is the integer between 0 and 15.

Under 256-color mode, the computer uses palette to determine the actual RGB value corresponding to each pixel value. Generally, palette is a structure of linear list, and in which each entry represents a RGB value of corresponding pixel. For example, in 4-bit pattern (each pixel is represented by 2 bits), palette can be set as:

```
struct palette {
    unsigned char r, g, b;
} [4] =
{
    {0, 0, 0},
    {128, 128, 128},
    {192, 192, 192},
    {255, 255, 255}
};
```

Now, the four possible pixel value (0, 1, 2, 3) may be corresponding to black, deep gray, gray, and white respectively; the following palette can adjust the four possible pixel value to red, green, blue, and white.

```
struct palette {
    unsigned char r, g, b;
} [4] =
{
    {255, 0, 0},
    {0, 255, 0},
    {0, 0, 255},
    {255, 255, 255}
};
```

For other display modes lower than 256-color, the structure of palette is basically consistent.

### 13.9.2 Using Palette

As we know, palette is a linear table used to build the correspondence relationship between limited pixel value and RGB value under low color bit-count modes (such as 256-color mode or those modes lower than 256-color).

In MiniGUI, we can use `SetPalette` and `GetPalette` to operate palette while `SetColorfulPalette` set the palette as default palette that includes the maximum scope of color.

New interfaces are added to the new GDI for the manipulation of palette:

```
HPALETTE GUIAPI CreatePalette (GAL_Palette* pal);
HPALETTE GUIAPI GetDefaultPalette (void);
int GUIAPI GetPaletteEntries (HPALETTE hpal, int start, int len, GAL_Color* cmap);
int GUIAPI SetPaletteEntries (HPALETTE hpal, int start, int len, GAL_Color* cmap);
BOOL GUIAPI ResizePalette (HPALETTE hpal, int len);
UINT GUIAPI GetNearestPaletteIndex(HPALETTE hpal, Uint8 red, Uint8 green, Uint8 blue);
RGBCOLOR GUIAPI GetNearestColor (HDC hdc, Uint8 red, Uint8 green, Uint8 blue);
```

`CreatePalette` function creates a new palette and `GetDefaultPalette` gets the default palette. `SetPaletteEntries` function and `GetPaletteEntries` function can be used to set or get the entry of the palette. `ResizePalette` function can be used to resize the size of palette. `GetNearestPaletteIndex` and `GetNearestColor` function can get the nearest index value and color of the palette.

Generally speaking, higher color bit-count (such as 15 bit or more) no longer uses palette to set the correspondence relationship between pixel value and RGB value, but uses simpler way to set such relationship. For example, 16-color mode uses the upper 5 bits to represent red, the middle 6 bits to represent green, and the low 5 bits to represent blue. Under this mode the relationship between pixel value and RGB value is directly correspondence relationship, which no longer involves palette, so this mode is also called direct color mode.

In MiniGUI, we can call function `RGB2Pixel` or `Pixel2RGB` function to perform

the transformation between a RGB value and a pixel value.

## 14 Handling and Rendering of Text

As discussed before, the handling and rendering of text has some of their own features in MiniGUI. We will elaborate the concepts related to text handling and introduce the relevant APIs in this chapter.

### 14.1 Charset and Encoding

Charset (character set) is a collection of characters defined to represent certain language; Encoding is the coding rules set to represent characters in certain charset. Encoding usually ranks the character by fixed order and uses them as the internal features of recording, storing, transfer and exchange. People who have conducted computer know the ASCII code defined by the US National Standard Authorization. ASCII code can be understood as an encoding format of American English charset; this coding format uses one 7-bit byte to represent one character scoped from 0x00 to 0x7F.

**[Note] Type `man ascii` to get the definition of ASCII on Linux command line.**

ASCII cannot meet the requirement of non-English speakers as the use of computer has spreads to the entire world. Therefore, almost all countries define the standard of charset and encoding based on their own official languages. The well-known standard GB2312-80 is the simplified Chinese charset standard defined by China government. GB2312-80 includes 682 symbols and 6763 Chinese words. It has 87 divisions, each of which has 94 characters. There are other standards, such as ISO8859 used for the single-byte charsets, JISX0201, JISX0208 charset defined by Japan, BIG5 traditional Chinese charset, and so on.

One charset can have different encoding format. Usually we use EUC encode (extended UNIX code) for GB2312 charset. EUC encodes each GB2312 charset as 2 bytes scoped in 0xA1~0xFE. The higher byte represents GB2312 area

code while the lower one represents GB2312 position code. Another popular GB2312 code is HZ code, which removes the highest bit of EUC code, thus using ASCII to represent Chinese characters. For example, the code of Chinese word “啊” is 0xB1A1 in EUC encoding, while in HZ encoding the character is ~{1!~}.

With the publication and update of the charset of each country and area, the problem of compatibility rises. For example, a text file that uses GB2312 EUC code cannot be properly displayed on BIG5 system. Therefore, some international organizations begin to develop the globally universal charset standard, that is, the well-known UNICODE charset.

The international standard organization established ISO/IEC JTC1/SC2/WG2 work group in April 1984. This group is responsible for integrating different characters and symbols of different countries. In 1991, some American companies established Unicode Consortium and got agreement with WG2 to use the same code-set in October 1991. At present, UNICODE 2.0 version includes 6811 characters, 20902 Chinese characters, 11172 Korean characters, 6400 make-up divisions and 20249 reserved characters, totally 65534 characters. UNICODE charset has multiple encoding formats, the most popular one is using 16-bit double bytes to express one character; it is also called USC2; another is UTF8 encoding format, which can be compatible with ASCII and ISO8859-1 charset. The byte-count used to represent a character is variable.

**[Hint] Type `man unicode` and `man utf-8` on Linux command line can get the information of UNICODE charset and UTF8 encoding.**

UNICODE can solve the compatibility problem of charsets. However, most countries and regions do not recognize UNICODE charset. For example, China government asks all OS software products must support GB18030 charset, not the UNICODE charset. The reason is that GB18030 is compatible with GB2312 and GBK charset popularly used in China main land areas, but not compatible with UNICODE.



UNICODE provides a way to solve charset compatibility problem for general-purpose operating systems. However, it is not the best way for the embedded systems. MiniGUI uses the internal code that is completely consistent with the default code of that charset to represent. Then, the abstract interfaces provide an universal analysis interface to text in any charset. This interface can be used to analyze both of the font module and the multi-byte character string. So far MiniGUI can support ISO8859-x (single-byte charsets), GB2312, GBK, GB18030, BIG5, EUCKR, Shift-JIS, and EUCJP (multi-byte charsets). MiniGUI also support UTF-8 and UTF-16 encode of UNICODE charset through the abstract interface of charset.

**[Hint] The charset support of MiniGUI can be also understood as the support of certain encoding format of that charset.**

MiniGUI uses logical font interface to support multi-byte charset. When application displays text, it usually needs to set logical font and assign the encoding name of chosen charset. After creating logical font, application can use it to display text or analyze text string.

## 14.2 Device Font

To correctly display text needs to get the shape data corresponding to each character. These shape data is called glyph and is saved in a file of certain type, which is usually called a font file. The most popular type of font file is dot-matrix (bitmap) font, which uses bitmap to store the information of dot-matrix glyph of each character. Another popular type is vector font, which stores the frame information of each character and can be zoomed by certain algorithms. The popular types of vector font are TrueType and Adobe Type1.

Similar to charset, MiniGUI defines a series of abstract interfaces for font. Now MiniGUI can support RBF, VBF (two MiniGUI-defined dot-matrix font formats), TrueType and Adobe Type1 fonts.

When initializing MiniGUI, it is needed to read MiniGUI.cfg and load certain

font files. The loaded font is internally called device font. Device font defines the format name, type, size and its supported charset. According to the loaded device font and the font type, name, size and character information assigned by application program, MiniGUI searches the proper device font to display text.

The next functions provide the support for loading and destroying dynamically font (ttf or qpf) since MiniGUI version 2.0.4/1.6.10.

```
BOOL GUIAPI LoadDevFontFromFile(const char *devfont_name,  
                                const char *file_name, DEVFONT** devfont);  
  
void GUIAPI DestroyDynamicDevFont (DEVFONT **devfont);
```

**[Hint] Information of the definition, name and format of device font is included in Chapter 4 of *MiniGUI User Manual*.**

MiniGUI-Processes does not load vector device font (TrueType and Type1 font) while initializing. If a MiniGUI-Processes application wants to use vector font, it should call `InitVectorialFonts`, and call `TermVectorialFonts` when done.

### 14.3 Logical font

The logical font of MiniGUI has strong functions, including abundant information such as charset, font type, and style. It can be used not only to render text, but also to analyze the text string. This is very useful in most text edition applications. Before using its logical font, you need firstly build it and choose it to the device context, which will use this logical font to output text. The default logical font of each device context is the default system-defined font in `MiniGUI.cfg`. You can establish the logical font by calling `CreateLogFont`, `CreateLogFontByName`, and `CreateLogFontIndirect`. You can also use function `SelectFont` to select a logical font to a device context. It is `DestroyLogFont` that is used to destroy logical font. However, you cannot destroy the selected logical font. The prototypes of these functions are as follow (`minigui/gdi.h`):

```

PLOGFONT GUIAPI CreateLogFont (const char* tvpe, const char* familv,
    const char* charset, char weight, char slant, char flip,
    char other, char underline, char struckout,
    int size, int rotation);
PLOGFONT GUIAPI CreateLogFontByName (const char* font_name);
PLOGFONT GUIAPI CreateLogFontIndirect (LOGFONT* logfont);
void GUIAPI DestroyLogFont (PLOGFONT log_font);

void GUIAPI GetLogFontInfo (HDC hdc, LOGFONT* log_font);

PLOGFONT GUIAPI GetSystemFont (int font_id);

PLOGFONT GUIAPI GetCurFont (HDC hdc);
PLOGFONT GUIAPI SelectFont (HDC hdc, PLOGFONT log_font);

```

The following code fragment creates multiple logical fonts:

```

static LOGFONT *logfont, *logfontgb12, *logfontbig24;

logfont = CreateLogFont (NULL, "SansSerif", "ISO8859-1",
    FONT_WEIGHT_REGULAR, FONT_SLANT_ITALIC, FONT_FLIP_NIL,
    FONT_OTHER_NIL, FONT_UNDERLINE_NONE, FONT_STRUCKOUT_LINE,
    16, 0);
logfontgb12 = CreateLogFont (NULL, "song", "GB2312",
    FONT_WEIGHT_REGULAR, FONT_SLANT_ROMAN, FONT_FLIP_NIL,
    FONT_OTHER_NIL, FONT_UNDERLINE_LINE, FONT_STRUCKOUT_LINE,
    12, 0);
logfontbig24 = CreateLogFont (NULL, "ming", "BIG5",
    FONT_WEIGHT_REGULAR, FONT_SLANT_ROMAN, FONT_FLIP_NIL,
    FONT_OTHER_AUTOSCALE, FONT_UNDERLINE_LINE, FONT_STRUCKOUT_NONE,
    24, 0);

```

The first font, `logfont`, belongs to ISO8859-1 charset and uses SansSerif with the height of 16 pixels; `logfontgb12` belongs to GB2312 charset and uses Song with the height of 12 pixels; `logfontbig24` belongs to BIG5 charset and uses Ming. The desired size of the last font is 24 pixels, and we use `FONT_OTHER_AUTOSCALE` style to create the logical font. The style `FONT_OTHER_AUTOSCALE` tells MiniGUI to auto-scale the font glyph to meet the desired font size.

We can also call `GetSystemFont` function to return a system logical font, the argument `font_id` in that can be one of the following values:

- **SYSLOGFONT\_DEFAULT**: System default font, it has to be a single-byte charset logical font and must be formed by RBF device font.
- **SYSLOGFONT\_WCHAR\_DEF**: System default multi-byte charset font. It is usually formed by RBF device font. Its width is twice of the **SYSLOGFONT\_DEFAULT** logical font.
- **SYSLOGFONT\_FIXED**: System font with fixed width.

- **SYSLOGFONT\_CAPTION**: The logical font used to display text on caption bar.
- **SYSLOGFONT\_MENU**: The logical font used to display menu text.
- **SYSLOGFONT\_CONTROL**: The default logical font used by controls.

The system logical fonts above are created corresponding to definition of MiniGUI.cfg when MiniGUI is initialized.

**[Hint] The information of definition, name and format of system logical font is described in the Chapter 4 of *MiniGUI User Manual*.**

**GetCurFont** function returns current logical font in a device context. You cannot call **DestroyLogFont** to destroy a system logical font.

## 14.4 Text Analysis

After establishing logical font, the application program can use logical font to analyze multi-language-mixed text. Here the multi-language-mixed text means the character string formed by two non-intersected charset texts, such as GB2312 and ISO8859-1, or BIG5 and ISO8859-2. You can use the following functions to analyze the text constitutes of multi-language-mixed text (**minigui/gdi.h**):

```
// Text parse support
int GUIAPI GetTextMCharInfo (PLOGFONT log_font, const char* mstr, int len,
                             int* pos_chars);
int GUIAPI GetTextWordInfo (PLOGFONT log_font, const char* mstr, int len,
                             int* pos_words, WORDINFO* info_words);
int GUIAPI GetFirstMCharLen (PLOGFONT log_font, const char* mstr, int len);
int GUIAPI GetFirstWord (PLOGFONT log_font, const char* mstr, int len,
                         WORDINFO* word_info);
```

**GetTextMCharInfo** returns the byte address of each character of the multi-language-mixed text. For example, for the string "ABC 汉语", this function will return {0, 1, 2, 3, 5} five values in **pos\_chars**. **GetTextWordInfo** will analyze the place of each word of the multi-language-mixed text. As for single-byte charset text, we use blank and TAB key as the delimiter; as for multi-byte charset text, the word uses single-byte character as the delimiter.

**GetFirstMCharLen** returns the byte length of the first character. **GetFirstWord** returns the word information of the first word.

## 14.5 Text Transformation

MiniGUI provides the functions of converting multibyte charset text to wide charset text in UCS or wide charset text in UCS to multibyte charset text. (minigui/gdi.h)

```
typedef unsigned short  UChar16;
typedef signed int      UChar32;

int GUIAPI MB2WCEX (PLOGFONT log_font, void* dest, BOOL wc32,
const unsigned char* mstr, int n);
#define MB2WC(log_font, dest, mstr, n) \
    MB2WCEX (log_font, dest, sizeof(wchar_t) == 4, mstr, n)

int GUIAPI WC2MBEx (PLOGFONT log_font, unsigned char *s, UChar32 wc);
#define WC2MB(log_font, s, wc) \
    WC2MBEx (log_font, s, (UChar32)wc);

int GUIAPI MBS2WCSEx (PLOGFONT log_font, void* dest, BOOL wc32,
const unsigned char* mstr, int mstr_len, int n,
int* conved_mstr_len);

#define MBS2WCS(log_font, dest, mstr, mstr_len, n) \
    MBS2WCSEx(log_font, dest, sizeof (wchar_t) == 4, mstr, \
    mstr_len, n, NULL)

int GUIAPI WCS2MBSEx (PLOGFONT log_font, unsigned char* dest,
const void *wcs, int wcs_len, BOOL wc32, int n,
int* conved_wcs_len);

#define WCS2MBS(log_font, dest, wcs, wcs_len, n) \
    WCS2MBSEx (log_font, dest, wcs, wcs_len, sizeof (wchar_t) == 4, \
    n, NULL)
```

**MB2WCEX** is used to convert a multibyte character (GB2312, ISO8859, UTF-8, GBK, BIG5, etc) to a wide character in UCS according to the charset/encoding of the logical font. And **MBS2WCSEx** is used to convert a multibyte string (GB2312, ISO8859, UTF-8, GBK, BIG5, etc) to a wide string in UCS according to the charset/encoding of the logical font.

**WC2MBEx** is used to convert a wide character in UCS to a multibyte character (GB2312, ISO8859, UTF-8, GBK, BIG5, etc) according to the charset/encoding of the logical font. And **WCS2MBSEx** is used to convert a wide string in UCS to a multibyte string (GB2312, ISO8859, UTF-8, GBK, BIG5, etc) according to the charset/encoding of the logical font.

## 14.6 Text Output

The following functions can be used to calculate the output length and width of text (`minigui/gdi.h`):

```
int GUIAPI GetTextExtentPoint (HDC hdc, const char* text, int len, int max_extent,
                              int* fit_chars, int* pos_chars, int* dx_chars, SIZE* size);

// Text output support
int GUIAPI GetFontHeight (HDC hdc);
int GUIAPI GetMaxFontWidth (HDC hdc);
void GUIAPI GetTextExtent (HDC hdc, const char* spText, int len, SIZE* pSize);
void GUIAPI GetTabbedTextExtent (HDC hdc, const char* spText, int len, SIZE* pSize);
```

**GetTextExtentPoint** is used to calculate the maximal number of the characters can be output, the byte place of each character, the output place of each character, and the actual output width and height of multi-byte text in a given output width (that is, the width of the output character is limited in a certain extent). **GetTextExtentPoint** is an integrated function, which is very useful for editor-type application. For example, in the single-line and multi-line edit box control; MiniGUI uses this function to calculate the position of the caret.

**GetFontHeight** and **GetMaxFontWidth** return the height and maximum width of a font. **GetTextExtent** calculates the output width and height of text.

**GetTabbedTextExtent** returns the output width and height of formatted text string.

The following function is used to output text (`minigui/gdi.h`):

```
int GUIAPI TextOutLen (HDC hdc, int x, int y, const char* spText, int len);
int GUIAPI TabbedTextOutLen (HDC hdc, int x, int y, const char* spText, int len);
int GUIAPI TabbedTextOutEx (HDC hdc, int x, int y, const char* spText, int nCount,
                           int nTabPositions, int *pTabPositions, int nTabOrigin);
void GUIAPI GetLastTextOutPos (HDC hdc, POINT* pt);

// Compatiblity definitions
#define TextOut(hdc, x, y, text)    TextOutLen (hdc, x, y, text, -1)
#define TabbedTextOut(hdc, x, y, text)  TabbedTextOutLen (hdc, x, y, text, -1)
...

int GUIAPI DrawTextEx (HDC hdc, const char* pText, int nCount,
                      RECT* pRect, int nIndent, UINT nFormat);
```

**TextOutLen** is used to output a certain text with appropriate length at given

position. If `length` is -1, the character string must terminate with `'\0'`.

`TabbedTextOutLen` is used to output formatted text string. `TabbedTextOutEx` is used to output formatted character string, but also can specify the position of each TAB character in the text string.

Fig. 14.1 is the output of `TextOut`, `TabbedTextOut`, and `TabbedTextOutEx` functions.

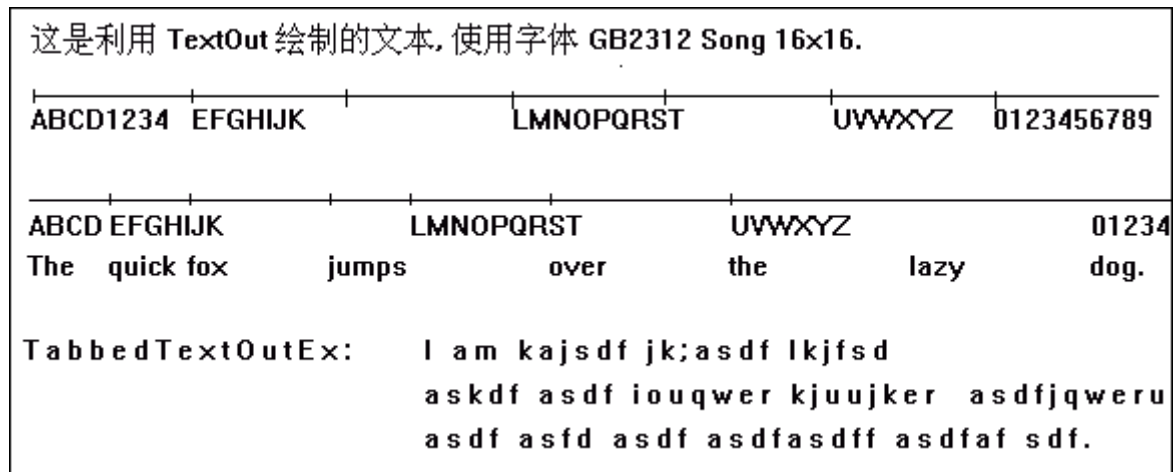


Fig. 14.1 Output of `TextOut`, `TabbedTextOut`, and `TabbedTextOutEx` functions

`DrawText` is the most complicated text output function, which can use different ways to output text in a given rectangle. Now, `DrawText` doesn't support UTF-16 text. Table 14.1 lists the formats supported by `DrawText`.

Table 14.1 Output formats of `DrawText` function

Format identifier	Meaning	Note
<code>DT_TOP</code>	Top-justifies the text.	single line only ( <code>DT_SINGLELINE</code> )
<code>DT_VCENTER</code>	Centers text vertically.	
<code>DT_BOTTOM</code>	Justifies the text to the bottom of the rectangle.	
<code>DT_LEFT</code>	Aligns text to the left.	
<code>DT_CENTER</code>	Aligns text in the center.	
<code>DT_RIGHT</code>	Aligns text to the right.	
<code>DT_WORDBREAK</code>	Lines are automatically broken between words if a word would extend past the edge of the rectangle specified by the <code>pRect</code> parameter.	
<code>DT_CHARBREAK</code>	Lines are automatically broken between characters if a character would extend past the edge of the rectangle specified by the <code>pRect</code> parameter.	

DT_SINGLELINE	Displays text on the single line only. Carriage returns and linefeeds do not break the line.	The vertical align flag will be ignored when there is not this flag
DT_EXPANDTABS	Expands TAB characters.	
DT_TABSTOP	Sets tab stops. Bits 15-8 (high-order byte of the low-order word) of the <b>uFormat</b> parameter specify the number of characters for each TAB.	
DT_NOCLIP	Draws without clipping. Output will be clipped to the specified rectangle by default.	
DT_CALCRECT	Do not output actually, only calculate the size of output rectangle.	

Code in List 14.1 calls **DrawText** function to perform aligned text output, according to the description of character to be output. Please refer to **fontdemo.c** program in MDE for complete code of the program. Fig. 14.2 shows the output effect of the program.

List 14.1 Using DrawText function

```
void OnModeDrawText (HDC hdc)
{
    RECT rc1, rc2, rc3, rc4;
    const char* szBuff1 = "This is a good day. \n"
        "这是利用 DrawText 绘制的文本, 使用字体 GB2312 Song 12. "
        "文本垂直靠上, 水平居中";
    const char* szBuff2 = "This is a good day. \n"
        "这是利用 DrawText 绘制的文本, 使用字体 GB2312 Song 16. "
        "文本垂直靠上, 水平靠右";
    const char* szBuff3 = "单行文本垂直居中, 水平居中";
    const char* szBuff4 =
        "这是利用 DrawTextEx 绘制的文本, 使用字体 GB2312 Song 16. "
        "首行缩进值为 32. 文本垂直靠上, 水平靠左";

    rc1.left = 1; rc1.top = 1; rc1.right = 401; rc1.bottom = 101;
    rc2.left = 0; rc2.top = 110; rc2.right = 401; rc2.bottom = 351;
    rc3.left = 0; rc3.top = 361; rc3.right = 401; rc3.bottom = 451;
    rc4.left = 0; rc4.top = 461; rc4.right = 401; rc4.bottom = 551;

    SetBkColor (hdc, COLOR_lightwhite);

    Rectangle (hdc, rc1.left, rc1.top, rc1.right, rc1.bottom);
    Rectangle (hdc, rc2.left, rc2.top, rc2.right, rc2.bottom);
    Rectangle (hdc, rc3.left, rc3.top, rc3.right, rc3.bottom);
    Rectangle (hdc, rc4.left, rc4.top, rc4.right, rc4.bottom);

    InflateRect (&rc1, -1, -1);
    InflateRect (&rc2, -1, -1);
    InflateRect (&rc3, -1, -1);
    InflateRect (&rc4, -1, -1);

    SelectFont (hdc, logfontgb12);
    DrawText (hdc, szBuff1, -1, &rc1, DT_NOCLIP | DT_CENTER | DT_WORDBREAK);

    SelectFont (hdc, logfontgb16);
    DrawText (hdc, szBuff2, -1, &rc2, DT_NOCLIP | DT_RIGHT | DT_WORDBREAK);

    SelectFont (hdc, logfontgb24);
    DrawText (hdc, szBuff3, -1, &rc3, DT_NOCLIP | DT_SINGLELINE | DT_CENTER | DT_VCENTER);
;

    SelectFont (hdc, logfontgb16);
```



```

} DrawTextEx (hdc, szBuff4, -1, &rc4, 32, DT NOCLIP | DT LEFT | DT WORDBREAK);

```

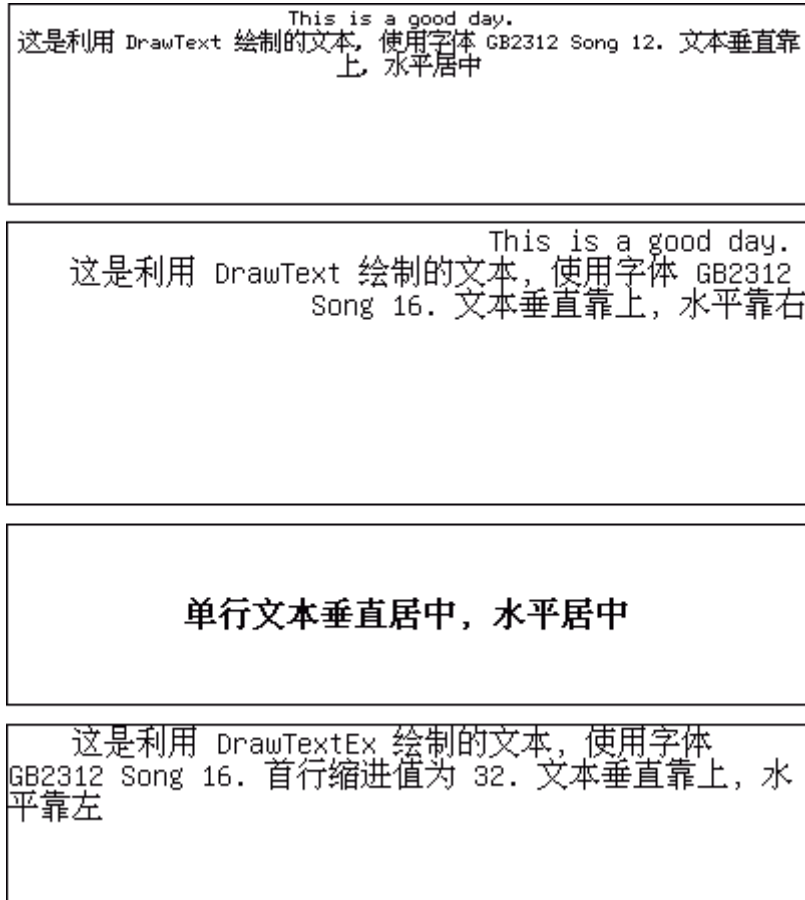


Fig. 14.2 The output of DrawText function

Except the above output functions, MiniGUI also provides functions listed in Table 14.2, which can be used to set or get the extra space between characters and lines.

Table 14.2 Functions to set/get extra space between characters and lines

Function	Meaning
GetTextCharacterExtra	Get the extra space between characters
SetTextCharacterExtra	Set the extra space between characters.
GetTextAboveLineExtra	Get the extra space above line.
SetTextAboveLineExtra	Set the extra space above line
GetTextBellowLineExtra	Get the extra space bellow line
SetTextBellowLineExtra	Set the extra space bellow line

The more usage of logical font and text output functions is illustrated in `fontdemo.c` file of MDE.

## 14.7 Special Render Effects of Font Glyph

MiniGUI provides many special effects to render the font glyphs. For example, you can specify the logical font style to render the glyph with bold, italic, underline, and/or struckout line. In version 2.0.3/1.6.9, MiniGUI provides anti-aliased effect by using low-pass filter, to flip the glyph vertically and/or horizontally, and to scale the glyph to meet the desired logical font size when using bitmap font. In version 2.0.4/1.6.10, MiniGUI provides FreeType2 support, and it makes that user can choose sub-pixel render using FreeType2 or MiniGUI; Before using sub-pixel render, please confirm that you have opened `FT_CONFIG_OPTION_SUBPIXEL_RENDERING` macro in FreeType2.

List 14.2 Using sub-pixel render

```
LOGFONT* mg_font;
mg_font = CreateLogFont (FONT_TYPE_NAME_SCALE_TTF, "times", "ISO8859-1",
                        FONT_WEIGHT_SUBPIXEL, FONT_SLANT_ROMAN, FONT_FLIP_NIL,
                        FONT_OTHER_NIL, FONT_UNDERLINE_NONE, FONT_STRUCKOUT_NONE,
                        15, 0);
.....
SelectFont(hdc, mg_font);
ft2SetLcdFilter (mg_font, MG_SMOOTH_NONE);
TextOut(hdc, 0,0, "text with MiniGUI sub-pixels smooth");

.....
ft2SetLcdFilter (mg_font, MG_SMOOTH_DEFAULT);
TextOut(hdc, 0,0, "text with FreeType2 sub-pixels smooth");
```

The render effects are specified through the logical font style when you create the logical font. For example, if you specify `FS_WEIGHT_BOOK` style, MiniGUI will use low-pass filter to handle the edge of the vectorial font glyph in order to anti-alias. When MiniGUI scales the font glyph of bitmap font, this style also can be used to anti-alias. Table 14.3 illustrates the render effects supported by MiniGUI.

Table 14.3 The font glyph render effects of MiniGUI

Style of logical font	Style character in logical font name	Style value of logical font	Meaning
weight: FONT_WEIGHT_REGULAR	First char is 'r'	FS_WEIGHT_REGULAR	No any weight effect
weight: FONT_WEIGHT_BOLD	First char is 'b'	FS_WEIGHT_BOLD	Bold
weight: FONT_WEIGHT_LIGHT	First char is 'l'	FS_WEIGHT_LIGHT	To render glyph edge with the

weight: FONT_WEIGHT_BOOK	First char is 'b'	FS_WEIGHT_BOOK	background color To handle to glyph edge by using the low-pass filter to anti-alias
weight: FONT_WEIGHT_DEMIBOLD	Fist char is 'd'	FS_WEIGHT_DEMIBOLD	Both FS_WEIGHT_BOLD and FS_WEIGHT_BOOK
weight: FONT_WEIGHT_SUBPIXEL	Fist char is 's'	FS_WEIGHT_SUBPIXEL	To handle to glyph by using sub-pixel render
slant: FONT_SLANT_ROMAN	Second char is 'r'	FONT_SLANT_ROMAN	No any slant effect
slant: FONT_SLANT_ITALIC	Second char is 'i'	FONT_SLANT_ITALIC	Italic
flip: FONT_OTHER_NIL	Third char is any letter but H/V/T	N/A	No any flipping effect
flip: FONT_FLIP_HORZ	Third char is 'H'	FS_FLIP_HORZ	To flip the glyph horizontally
flip: FONT_FLIP_VERT	Third char is 'V'	FS_FLIP_VERT	To flip the glyph vertically
flip: FONT_FLIP_HORZVERT	Third char is 'T'	FS_FLIP_HORZVERT	Both FS_FLIP_HORZ and FS_FLIP_VERT
other: FONT_OTHER_NIL	Forth char is any letter but S/N	N/A	No any other effect
other: FONT_OTHER_AUTOSCALE	Forth char is 'S'	FS_OTHER_AUTOSCALE	To scale the glyph to meet the desired logical font size, only good for bitmap font.
other: FONT_OTHER_TTFNOCACHE	Forth char is 'N'	FS_OTHER_TTFNOCACHE	To close the cache when render glyph by using TrueType font.
other: FONT_OTHER_LCDPORTRAIT	Forth char is 'P'	FS_OTHER_LCDPORTRAIT	Using portrait LCD when render glyph by using TrueType font.
underline: FONT_UNDERLINE_NONE	Fifth char is 'n'	FS_UNDERLINE_NONE	No underline
underline: FONT_UNDERLINE_LINE	Fifth char is 'u'	FS_UNDERLINE_LINE	Underline
struckout: FONT_STRUCKOUT_NONE	Sixth char is 'n'	FS_STRUCKOUT_NONE	No struckout line
struckout: FONT_STRUCKOUT_LINE	Sixth char is 's'	FS_STRUCKOUT_LINE	Struckout line

The style character given by Table 14.3 can be used to define the logical font name.



## 15 Advanced GDI Functions Based on NEWGAL

It is mentioned in Chapter 13 that MiniGUI 1.1.0 has greatly improved GAL and GDI by rewriting almost all code. Those new interfaces and functions strongly enhance the graphics capability of MiniGUI. In this chapter we will introduce the related concepts and interfaces of new GDI interface in detail.

### 15.1 New Region Implementation

New GDI uses new region algorithm, which is popularly used in X Window and other GUI systems. This region is called x-y-banned region, featured as follow:

- Region is constituted by non-null rectangles that is not intersected each other.
- Region can be divided into several non-intersected horizontal strip, each of which has the rectangles with same width and top-aligned; In other words, all rectangles have the same y coordinates on the upper-left corner.
- The rectangles rank from left to right at the x direction, and then form top to bottom at y direction.

GDI can use the special property of x-y-banned region to optimize the drawing. Those drawing functions that will be added into the future version will use this property to optimize plotting output.

New GDI adds the following interfaces, which can be used to do operations between regions (`minigui/gdi.h`):

```

BOOL GUIAPI PtInRegion (PCLIPRGN region, int x, int y);
BOOL GUIAPI RectInRegion (PCLIPRGN region, const RECT* rect);

BOOL GUIAPI IntersectRegion (CLIPRGN *dst, const CLIPRGN *src1, const CLIPRGN *src2);
BOOL GUIAPI UnionRegion (PCLIPRGN dst, const CLIPRGN* src1, const CLIPRGN* src2);
BOOL GUIAPI SubtractRegion (CLIPRGN* rgnD, const CLIPRGN* rgnM, const CLIPRGN* rgnS);
BOOL GUIAPI XorRegion (CLIPRGN *dst, const CLIPRGN *src1, const CLIPRGN *src2);

```

- **PtInRegion** can be used to check if the given point locates in the given region.

- **RectInRegion** can be used to check if given rectangle intersects with the given region.
- **IntersectRegion** can be used to calculate the intersection of two given region.
- **UnionRegion** can merge two different regions. The merged region is still a region of x-y-banned.
- **SubtractRegion** subtracts one region from another region.
- **XorRegion** is used to perform the XOR (Exclusive OR) operation between two regions. The result equals to the intersection between the result A that **src1** subtracts **src2** and the result B that **src2** subtracts **src1**.

Apart from the region operation functions above, MiniGUI also provides those GDI functions that created from the closed curve such as polygon and ellipse. These functions can limit GDI output within the specially closed curve. These functions will be discussed in the following sections.

## 15.2 Raster Operations

Raster operation specifies the operation between the target pixel and the pixel existed on the screen while drawing. The most popular one is alpha blending. Here the raster operation refers to binary bit operation, including AND, OR, XOR, and directly overwrite (SET). Application can use **SetRasterOperation** function and **GetRasterOperation** function to set or get current raster operation. The prototypes of these two functions are as follow

(minigui/gdi.h):

```
#define ROP_SET      0
#define ROP_AND      1
#define ROP_OR       2
#define ROP_XOR      3

int GUIAPI GetRasterOperation (HDC hdc);
int GUIAPI SetRasterOperation (HDC hdc, int rop);
```

In the function, **rop** is the raster operation mode. The optional parameters are **ROP\_SET** (directly set), **ROP\_AND** (do bit-AND with the pixels on the screen), **ROP\_OR** (do bit-OR with the pixels on the screen) and **ROP\_XOR** (do exclusive-OR

with the pixels on the screen).

After setting new raster operation, the subsequent graphics output will be affected by it. Those drawing functions include `SetPixel`, `LineTo`, `Circle`, `Rectangle`, `FillCircle`, and `FillBox`.

### 15.3 Memory DC and BitBlt

New GDI function boosts up the memory DC operation function. GDI function will call GAL-related interface while setting memory DC as it will fully use the hardware acceleration of video adapter to quickly move or copy the bit blocks between different regions, including transparency processing and alpha blending. Application can create a memory DC with the property that each point has different alpha value. Also, application can create the alpha value (alpha channel of the DC) of all pixels of memory DC by using `SetMemDCAlpha`, then use `BitBlt` and `StretchBlt` to transfer the bits between DCs. Application also can set the transparent pixel of a source DC, so as to leap over such pixel while performing `BitBlt`.

GDI functions regarding memory DC are as follow (`minigui/gdi.h`):

```
#define MEMDC_FLAG_NONE          0x00000000          /* None. */
#define MEMDC_FLAG_SWSURFACE     0x00000000          /* DC is in system memory */
#define MEMDC_FLAG_HWSURFACE     0x00000001          /* DC is in video memory */
#define MEMDC_FLAG_SRCCOLORKEY   0x00001000          /* Blit uses a source color key */
#define MEMDC_FLAG_SRCALPHA      0x00010000          /* Blit uses source alpha blending */
#define MEMDC_FLAG_RLEACCEL      0x00004000          /* Surface is RLE encoded */

HDC GUIAPI CreateCompatibleDC (HDC hdc);
HDC GUIAPI CreateMemDC (int width, int height, int depth, DWORD flags,
    Uint32 Rmask, Uint32 Gmask, Uint32 Bmask, Uint32 Amask);
BOOL GUIAPI ConvertMemDC (HDC mem_dc, HDC ref_dc, DWORD flags);
BOOL GUIAPI SetMemDCAlpha (HDC mem_dc, DWORD flags, Uint8 alpha);
BOOL GUIAPI SetMemDCColorKey (HDC mem_dc, DWORD flags, Uint32 color_key);
void GUIAPI DeleteMemDC (HDC mem_dc);
```

`CreateCompatibleDC` creates a memory DC compatible with given DC.

Compatibility means that the pixel format, width and height of new created memory DC are same with the given DC. Memory DC based on this format can be quickly blit to the compatible DC.

Now we further explain the pixel format. Pixel format includes the depth of color (that is, the number of bits per pixel), palette, or the format of four components of RGBA (red, green, blue, and alpha). The parameter alpha can be seen as the transparency degree of one pixel point, 0 means 100% transparent, 255 means completely opaque. If the degree of color below 8, GAL will create a default palette and you can call the function `SetPalette` to modify the palette. If the degree is higher than 8, MiniGUI will respectively appoint the bits taken up by the RGBA parameters of the given pixel. Compatible memory DC has the same color degree with the given DC, and has the same palette or RGBA parameter combination format.

Calling function `CreateMemDC` can specify the height, width, color depth, and necessary RGBA components combination format of the new created memory DC. MiniGUI use bit mask taken up by each pixel to represent the combination format of RGBA components. For example, to create a 16-bit memory DC including per-pixel alpha value, you can use four 4-bit to assign 16-bit pixel, thus the mask of the four components of RBGA will respectively be: 0x0000F000, 0x00000F00, 0x000000F0, and 0x0000000F.

`ConvertMemDC` is used to convert a specified memory DC according to the pixel format of given referencing DC. This conversion makes the result DC has the same pixel format with referencing DC. Thus, the converted DC can be quickly blitted to the compatible DC.

`SetMemDCAlpha` is used to set or delete the alpha channel value of whole memory DC. We can use `MEMDC_FLAG_RLEACCEL` to specify the memory DC adopt or delete the RLE encoding format. Alpha channel value will function on all pixels of the DC.

`SetMemDCColorKey` is used to set or delete the color key of whole memory DC object, that is, the transparent pixel value. We can also use `MEMDC_FLAG_RLEACCEL` to specify memory DC adopt or delete the RLE encoding format.



Similar to other DCs, you can also call GDI drawing function to perform any drawing output operation in memory DC and then blit it to other DC. Code in List 15.1 comes from `gdidemo.c` program of MDE, which demonstrates how to use memory DC to realize transparent and alpha blending blitting operation between DCs.

List 15.1 The enhanced memory DC operations

```

/* Alpha operation point by point */
mem_dc = CreateMemDC (400, 100, 16, MEMDC_FLAG_HWSURFACE | MEMDC_FLAG_SRCALPHA,
                     0x0000F000, 0x0000F00, 0x000000F0, 0x0000000F);

/* Set an opaque brush and fill a rectangle */
SetBrushColor (mem_dc, RGBARGB2Pixel (mem_dc, 0xFF, 0xFF, 0x00, 0xFF));
FillBox (mem_dc, 0, 0, 200, 50);

/* Set a brush transparent by 25% and fill a rectangle */
SetBrushColor (mem_dc, RGBARGB2Pixel (mem_dc, 0xFF, 0xFF, 0x00, 0x40));
FillBox (mem_dc, 200, 0, 200, 50);

/* Set a brush half transparent and fill a rectangle */
SetBrushColor (mem_dc, RGBARGB2Pixel (mem_dc, 0xFF, 0xFF, 0x00, 0x80));
FillBox (mem_dc, 0, 50, 200, 50);

/* Set a brush transparent by 75% and fill a rectangle */
SetBrushColor (mem_dc, RGBARGB2Pixel (mem_dc, 0xFF, 0xFF, 0x00, 0xC0));
FillBox (mem_dc, 200, 50, 200, 50);
SetBkMode (mem_dc, BM_TRANSPARENT);

/* Output text with half transparent pixel point */
SetTextColor (mem_dc, RGBARGB2Pixel (mem_dc, 0x00, 0x00, 0x00, 0x80));
TabbedTextOut (mem_dc, 0, 0, "Memory DC with alpha.\n"
               "The source DC have alpha per-pixel.");

/* Blit into window DC */
start_tick = GetTickCount ();
count = 100;
while (count--) {
    BitBlt (mem_dc, 0, 0, 400, 100, hdc, rand () % 800, rand () % 800);
}
end_tick = GetTickCount ();
TellSpeed (hwnd, start_tick, end_tick, "Alpha Blit", 100);

/* Delete memory DC */
DeleteMemDC (mem_dc);

/* A memory DC with Alpha channel: 32 bits, RGB takes 8 respectively, no Alpha 分量 */
mem_dc = CreateMemDC (400, 100, 32,
                     MEMDC_FLAG_HWSURFACE | MEMDC_FLAG_SRCALPHA | MEMDC_FLAG_SRCCOLORKEY,
                     0x00FF0000, 0x0000FF00, 0x000000FF, 0x00000000);

/* Output the filled rectangle and text to memory DC */
SetBrushColor (mem_dc, RGBARGB2Pixel (mem_dc, 0xFF, 0xFF, 0x00, 0x00));
FillBox (mem_dc, 0, 0, 400, 100);
SetBkMode (mem_dc, BM_TRANSPARENT);
SetTextColor (mem_dc, RGBARGB2Pixel (mem_dc, 0x00, 0x00, 0xFF, 0xFF));
TabbedTextOut (mem_dc, 0, 0, "Memory DC with alpha.\n"
               "The source DC have alpha per-surface.");

/* Blit into window DC */
start_tick = GetTickCount ();
count = 100;
while (count--) {
    /* Set the Alpha channel of memory DC */
    SetMemDCAlpha (mem_dc, MEMDC_FLAG_SRCALPHA | MEMDC_FLAG_RLEACCEL, rand () % 256);
    BitBlt (mem_dc, 0, 0, 400, 100, hdc, rand () % 800, rand () % 800);
}

```

```

}
end_tick = GetTickCount ();
TellSpeed (hwnd, start_tick, end_tick, "Alpha Blit", 100);

/* Fill rectangle area, and output text */
FillBox (mem_dc, 0, 0, 400, 100);
SetBrushColor (mem_dc, RGB2Pixel (mem_dc, 0xFF, 0x00, 0xFF));
TabbedTextOut (mem_dc, 0, 0, "Memory DC with alpha and colorkey.\n"
                    "The source DC have alpha per-surface.\n"
                    "And the source DC have a colorkey, \n"
                    "and RLE accelerated.");

/* Set the transparent pixel value of memory DC */
SetMemDCColorKey (mem_dc, MEMDC_FLAG_SRCCOLORKEY | MEMDC_FLAG_RLEACCEL,
                    RGB2Pixel (mem_dc, 0xFF, 0xFF, 0x00));
/* Blit into window DC */
start_tick = GetTickCount ();
count = 100;
while (count--) {
    BitBlt (mem_dc, 0, 0, 400, 100, hdc, rand () % 800, rand () % 800);
    CHECK_MSG;
}
end_tick = GetTickCount ();
TellSpeed (hwnd, start_tick, end_tick, "Alpha and colorkey Blit", 100);

/* Delete the object of memory DC */
DeleteMemDC (mem_dc);

```

## 15.4 Enhanced BITMAP Operations

New GDI function enhances the **BITMAP** structure and adds support to transparent and alpha channel as well as. By setting such memberships as **bmType**, **bmAlpha**, and **bmColorkey** of a **BITMAP** object, you can get certain properties. Then you can use function **FillBoxWithBitmap/Part** to draw the **BITMAP** object to certain DC. You can regard the **BITMAP** object as the memory DC created in system memory, but you cannot perform drawing output to the **BITMAP** object. Code in List 15.2 loads a bitmap object from an image file, set transparency and alpha channel, and finally use function **FillBoxWithBitmap** to output to a DC. This program comes from the demo program **gdidemo.c** of MDE.

List 15.2 The enhanced BITMAP operations

```

int tox = 800, toy = 800;
int count;
BITMAP bitmap;
unsigned int start_tick, end_tick;

if (LoadBitmap (hdc, &bitmap, "res/icon.bmp"))
    return;

bitmap.bmType = BMP_TYPE_ALPHACHANNEL;

/* Alpha mixing of the bitmap */

start_tick = GetTickCount ();

```

```

count = 1000;
while (count--) {
    tox = rand() % 800;
    toy = rand() % 800;

    /* Set random Alpha channel value */
    bitmap.bmAlpha = rand() % 256;
    /* Display into window DC */
    FillBoxWithBitmap (hdc, tox, toy, 0, 0, &bitmap);
}
end_tick = GetTickCount ();
TellSpeed (hwnd, start_tick, end_tick, "Alpha Blended Bitmap", 1000);

bitmap.bmType = BMP_TYPE_ALPHACHANNEL | BMP_TYPE_COLORKEY;
/* Get the first pixel value, and set it as transparent pixel value */
bitmap.bmColorKey = GetPixelInBitmap (&bitmap, 0, 0);

/* Transparent and Alpha mixing */
start_tick = GetTickCount ();
count = 1000;
while (count--) {
    tox = rand() % 800;
    toy = rand() % 800;

    /* Set random Alpha channel value */
    bitmap.bmAlpha = rand() % 256;
    /* Display into window DC */
    FillBoxWithBitmap (hdc, tox, toy, 0, 0, &bitmap);
}
end_tick = GetTickCount ();
TellSpeed (hwnd, start_tick, end_tick, "Alpha Blended Transparent Bitmap", 1000);

UnloadBitmap (&bitmap);

```

You can also convert a certain **BITMAP** object to memory DC object by using function **CreateMemDCFromBitmap**. The prototype of this function is as follows (**minigui/gdi.h**):

```
HDC GUIAPI CreateMemDCFromBitmap (HDC hdc, BITMAP* bmp);
```

The memory DC created from a **BITMAP** object directly uses the memory to which **bmBits** of the **BITMAP** object points. The memory lies in system memory, but not video memory.

New GDI also enhances the **BITMAP**-related **MYBITMAP** structure. **MYBITMAP** can be seen as a bitmap structure that is not dependent to the device. You can also convert one **MYBITMAP** object to a memory DC by using function **CreateMemDCFromMyBitmap**. The prototype of this function is as follows (**minigui/gdi.h**):

```
HDC GUIAPI CreateMemDCFromMyBitmap (HDC hdc, MYBITMAP* mybmp);
```

Most GAL engines are unable to provide hardware acceleration for blitting operation which transfers pixels from system memory to video memory, therefore, the **FillBoxWithBitmap** function, the **BITMAP** object or the memory DC created by **MYBITMAP** object are unable to be quickly blitted to other DC through hardware acceleration. The support for such acceleration can be achieved by pre-created memory DC in video memory.

## 15.5 New GDI functions

Apart from raster operation, MiniGUI also adds some useful GDI functions, including **FillBox**, **FillCircle**, and so on. The new GDI functions are:

```
void GUIAPI FillBox (HDC hdc, int x, int y, int w, int h);
void GUIAPI FillCircle (HDC hdc, int sx, int sy, int r);

BOOL GUIAPI ScaleBitmap (BITMAP* dst, const BITMAP* src);

BOOL GUIAPI GetBitmapFromDC (HDC hdc, int x, int y, int w, int h, BITMAP* bmp);

gal_pixel GUIAPI GetPixelInBitmap (const BITMAP* bmp, int x, int y);
BOOL GUIAPI SetPixelInBitmap (const BITMAP* bmp, int x, int y, gal_pixel pixel);
```

- **FillBox** fills specified rectangle.
- **FillCircle** fills specified circle, affected by raster operation.
- **ScaleBitmap** scales a **BITMAP** object
- **GetBitmapFromDC** copies pixels which the scope is in a specified rectangle to a **BITMAP** object.
- **GetPixelInBitmap** gets pixel value of a given position in a **BITMAP** object.
- **SetPixelInBitmap** sets pixel value of a given position in a **BITMAP** object.

## 15.6 Advanced GDI functions

### 15.6.1 Image Scaling Functions

The following functions can be used to paint and scale an image simultaneously.

```
int GUIAPI StretchPaintImageFromFile (HDC hdc, int x, int y, int w, int h,
```

```

        const char* spFileName);
int GUIAPI StretchPaintImageFromMem (HDC hdc, int x, int y, int w, int h,
        const void* mem, int size, const char* ext);
int GUIAPI StretchPaintImageEx (HDC hdc, int x, int y, int w, int h, MG_RWops* area,
        const char* ext);

```

**StretchPaintImageFromFile** stretches and paints an image from file.

Parameter **hdc** is device context. Parameter **x** and **y** are painting position on device. Parameter **w** is the width of the stretched bitmap. Parameter **h** is the height of the stretched bitmap. Parameter **spFileName** is the file name.

**StretchPaintImageFromMem** stretches an image from memory. Parameter **hdc** is device context. Parameter **x** and **y** are painting position on device.

Parameter **w** is the width of the stretched bitmap. Parameter **h** is the height of the stretched bitmap. Parameter **mem** points to memory containing image data and parameter **size** is the size of the image data. Parameter **ext** is the type of bitmap.

**StretchPaintImageEx** paints and stretches an image from data source onto device directly. Parameter **hdc** is device context. Parameter **x** and **y** are painting position on device. Parameter **w** is the width of the stretched bitmap. Parameter **h** is the height of the stretched bitmap. Parameter **area** points to data source and parameter **ext** is the extension the type of bitmap.

### 15.6.2 Image Rotation Functions

The following functions can be used to paint an BITMAP object and rotate it.

```

void GUIAPI PivotScaledBitmapFlip (HDC hdc, const BITMAP *bmp, fixed x, fixed y,
        fixed cx, fixed cy, int angle, fixed scale_x,
        fixed scale_y, BOOL h_flip, BOOL v_flip);
void GUIAPI RotateBitmap (HDC hdc, const BITMAP* bmp, int lx, int ty, int angle);
void GUIAPI PivotBitmap (HDC hdc, const BITMAP *bmp, int x, int y,
        int cx, int cy, int angle);
void GUIAPI RotateScaledBitmap (HDC hdc, const BITMAP *bmp, int lx, int ty,
        int angle, int w, int h);
void GUIAPI RotateBitmapVFlip (HDC hdc, const BITMAP *bmp, int lx, int ty, int angle);
void GUIAPI RotateBitmapHFlip (HDC hdc, const BITMAP *bmp, int lx, int ty, int angle);
void GUIAPI RotateScaledBitmapVFlip (HDC hdc, const BITMAP *bmp, int lx, int ty,
        int angle, int w, int h);
void GUIAPI RotateScaledBitmapHFlip (HDC hdc, const BITMAP *bmp, int lx, int ty,
        int angle, int w, int h);

```

**PivotScaledBitmapFlip** rotates, stretches, shrinks, or flips a bitmap object.

The unit of rotation is 1/64 degree in this function. Parameter `hdc` is device context. Parameter `bmp` is a BITMAP object pointer. Parameter `x` and `y` are rotation center on DC. Parameter `(cx, cy)` is the rotation center on the bitmap. The data type of `x`, `y`, `cx`, `cy` are fixed-point values. Parameter `angle` is the degree of rotation. Parameter `scale_x` and `scale_y` are proportion of scaling. Parameter `h_flip` and `v_flip` are flags for flipping horizontally or flipping vertically.

**RotateBitmap** rotates a bitmap object surrounded its center. Parameter `hdc` is the graphic device context handle. Parameter `bmp` is the pointer to a BITMAP object. Parameter `lx` and `ly` are upper-left coordinate of bitmap on DC. Parameter `angle` is the degree of rotation. The unit of rotation is 1/64 degree.

**PivotBitmap** aligns the point in the bitmap given by parameter `(cx, cy)` to parameter `(x, y)` in device context. Parameter `hdc` is the device context. Parameter `bmp` is the pointer to a BITMAP object. Parameter `x` and `y` are rotation center on DC. Parameter `(cx, cy)` is the rotation center on the bitmap. Parameter `angle` is the degree of rotation. The unit of rotation is 1/64 degree.

**RotateScaledBitmap** stretches or shrinks a bitmap object at the same as rotating it. Parameter `hdc` is the device context. Parameter `bmp` is the pointer to a BITMAP object. Parameter `lx` and `ly` are upper-left coordinates of bitmap on DC. Parameter `angle` is the degree of rotation. The unit of rotation is 1/64 degree. Parameter `w` and `h` are width and height of the stretched bitmap.

**RotateBitmapVFlip** flips vertically and rotates a bitmap object. See also **RotateBitmap**.

**RotateBitmapHFlip** flips horizontally and rotates a bitmap object. See also **RotateBitmap**.

**RotateScaledBitmapVFlip** flips vertically, rotates, stretch or shrinks a bitmap object. This function is similar to **RotateScaledBitmap** expect that it flips the

bitmap vertically first.

**RotateScaledBitmapHFlip** flips horizontally, rotates, stretches or shrinks a bitmap object. For the means of parameters, please refer to **RotateScaledBitmap**.

### 15.6.3 Rounded Corners Rectangle

The follow function draws a rounded corners rectangle

```
BOOL GUIAPI RoundRect (HDC hdc, int x0, int y0, int x1, int y1, int rw, int rh);
```

Here, **hdc** is the graphic device context handle. **x0** and **y0** are coordinates of upper-left corner of the rectangle; **x1** and **y1** are coordinates of lower-right of the rectangle; **rw** and **rh** are the x-radius and y-radius of the rounded corners respectively.

## 15.7 Curve Generators

A general graphics system usually provides the user with the drawing functions to plot line and complicated curve, such as ellipse, and spline. Users can use this function to plot, but unable to do other jobs by using the existed curve-generating algorithms. In the development of new GDI interface, we use a special design pattern to implement drawing of curve and filling of closed curve. This way is very flexible and provides you a chance of directly using system internal algorithm:

- System defines several functions used to create line and curve. We call these functions curve generators.
- You need to define a callback and pass this function address to curve generator before calling a generator. The curve generator will call this callback while generating a point on the curve or a horizontal line of closed curve.
- You can finish some operations based on new point or new horizontal

line. For the MiniGUI drawing function, that means drawing the point or the line.

- As callback is frequently called on during the generator operating process, system allows you to pass a pointer to represent context while calling the curve generator. The generator will pass this pointer to your callback.

We will describe curve and filling generator provided by MiniGUI in the following sections.

### 15.7.1 Line Clipper and Line Generator

The prototype of the line clipper and generator is as follows:

```
/* Line clipper */
BOOL GUIAPI LineClipper (const RECT* cliprc, int *_x0, int *_y0, int *_x1, int *_y1);

/* Line generators */
typedef void (* CB_LINE) (void* context, int stepx, int stepy);
void GUIAPI LineGenerator (void* context, int x1, int y1, int x2, int y2, CB_LINE cb);
```

**LineClipper** is not a generator, which is used to clip the given line into a rectangle. **cliprc** is the given rectangle, while **\_x0**, **\_y0**, **\_x1**, and **\_y1** present the two end-points of the line that will be clipped into **cliprc** and then return to the end-points of the clipped line. MiniGUI internally uses Cohen-Sutherland algorithm to clip a line.

**LineGenerator** is the line generator using Bresenham algorithm. The generator starts from the beginning point of given line, every time when it generates a point it will call the callback function and pass context as well as the step forward value or amount of difference between new point and the previous one. For example, passing **stepx = 1**, **stepy = 0** indicate that the new point is one step forward then the previous point at x coordinate, while y coordinate keeps the same. The callback function can realize the optimization at certain degree on the basis of step forward value.



### 15.7.2 Circle Generator

The prototypes of MiniGUI-defined circle generator are as follow:

```
/* Circle generator */
typedef void (* CB_CIRCLE) (void* context, int x1, int x2, int y);
void GUIAPI CircleGenerator (void* context, int sx, int sy, int r, CB_CIRCLE cb);
```

You should firstly specify the coordinates of center of the circle and the radius, and then pass information of context and the callback function. Every time when it generates a point, the generator will call the callback function once, at the same time it passes three values: **x1**, **x2**, and **y**. These three values actually indicate two points on the circle: (**x1**, **y**) and (**x2**, **y**). Because of the symmetry property of circle, the generator can get all points on the circle by calculating only 1/4 of points in the circle.

### 15.7.3 Ellipse Generator

Similar to circle generator, the prototypes of ellipse generator are as follow:

```
/* Ellipse generator */
typedef void (* CB_ELLIPSE) (void* context, int x1, int x2, int y);
void GUIAPI EllipseGenerator (void* context, int sx, int sy, int rx, int ry,
                             CB_ELLIPSE cb);
```

You should firstly specify the coordinates of center of ellipse and the radius of x coordinate and y coordinate, and then pass the context and the callback function. Every time when it generates a point, the generator will call the callback function once, at the same time it passes three values: **x1**, **x2**, and **y**. These three values actually indicate two points in the ellipse: (**x1**, **y**) and (**x2**, **y**). Because of the symmetry property of ellipse, the generator can get all points in the ellipse by calculating only half of points of the ellipse.

### 15.7.4 Arc Generator

The MiniGUI-defined arc generator can be seen as follows:

```
/* Arc generator */
typedef void (* CB_ARC) (void* context, int x, int y);
```

```
void GUIAPI CircleArcGenerator (void* context, int sx, int sv, int r,
                                int angl, int ang2, CB_ARC cb);
```

You should firstly specify the center of the arc, the radius, the starting angle and the end angle. It is necessary to note that the unit of the starting angle and end angle is integers in 1/64 degree, not float-point numbers. Then you pass the callback function. Every time when it generates a point on the arc, the function will call the callback function and pass the coordinates value ( $x$ ,  $y$ ) of a new point on the arc.

### 15.7.5 Vertical Monotonous Polygon Generator

Generally speaking, polygons include protruding polygons and concave polygons. Here the vertical monotonous polygon is a special polygon produced for optimizing polygon algorithm in the computer graphics. The definition of this kind of polygon is: all the horizontal lines of the computer screen and the side of polygon can only have one or two points of intersection. Fig. 15.1 gives examples of protruding polygon, concave polygon, and vertical monotonous polygon.

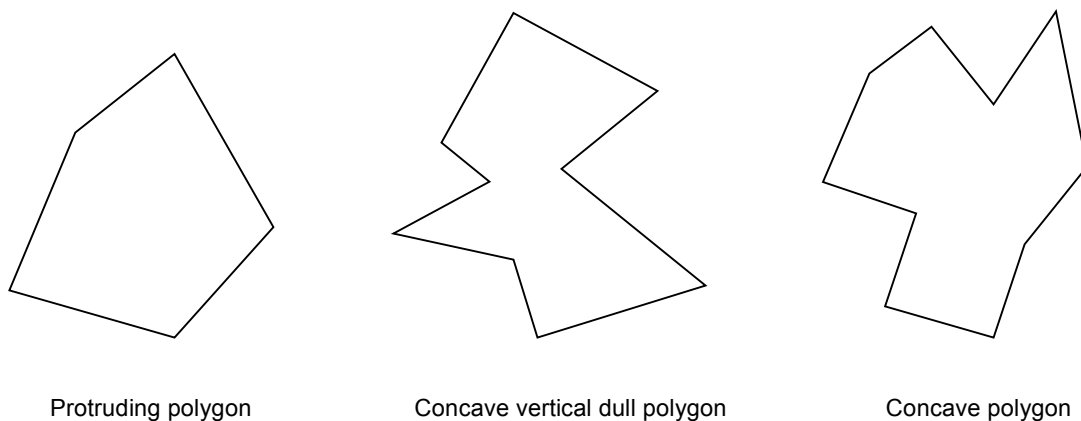


Fig. 15.1 Polygons

It is necessary to note that a protruding polygon must be vertical monotonous polygon, but a vertical monotonous polygon can be concave polygon. Obviously, normal polygon filling algorithms need to determine the number of intersection point between polygon and the screen scan line while vertical monotonous polygons not. Therefore, the speed of polygon filling can be greatly increased.

The prototypes of MiniGUI-defined vertical monotonous polygon related functions are as follow:

```
/* To determine whether the specified Polygon is Monotone Vertical Polygon */
BOOL GUIAPI PolygonIsMonotoneVertical (const POINT* pts, int vertices);

/* Monotone vertical polygon generator */
typedef void (* CB_POLYGON) (void* context, int x1, int x2, int y);
BOOL GUIAPI MonotoneVerticalPolygonGenerator (void* context, const POINT* pts,
                                              int vertices, CB_POLYGON cb);
```

**PolygonIsMonotoneVertical** is used to determine if a given polygon is vertical monotonous polygon while **MonotoneVerticalPolygonGenerator** is the generator of vertical monotonous polygon. In MiniGUI, the vertices of the polygon represent a polygon. **pts** represents the vertices array while **vertices** represents the number of the vertices. What generator generates actually are the end-points (**x1**, **y**) and (**x2**, **y**) of each horizontal line to filling the polygon.

### 15.7.6 General Polygon Generator

MiniGUI also provides the general polygon generator, which can be used to process protruding polygons and concave polygons. The prototypes are as follow:

```
/* General polygon generator */
typedef void (* CB_POLYGON) (void* context, int x1, int x2, int y);
BOOL GUIAPI PolygonGenerator (void* context, const POINT* pts,
                              int vertices, CB_POLYGON cb);
```

Similar to vertical monotonous polygon generator, what this function generates are horizontal scan lines for filling the polygon: **x1** is the beginning x coordinates of the horizontal line; **x2** is the end x coordinate; **y** is the y coordinate of horizontal line.

### 15.7.7 Flood Filling Generator

The flood-filling generator is relatively complex. It is used by **FloodFill** function by MiniGUI internally. As we know, **FloodFill** function starts from a given position, and ends until getting the point that has different pixel value

with the one at the start position (Like water spreading, so called flood filling). Between the starting point and the ending point, it fills certain region according to a given color. During this process, we need two callback functions, one of which is used to determine if the encountered point is the same with the starting one; another callback function is used to generate the horizontal scan line of the filling region. When plotting what this function compares is pixel, but actually it can be used to compare any other values so as to finish the special spreading actions. MiniGUI defines flood-filling generator as follows:

```
/* General Flood Filling generator */
typedef BOOL (* CB_EQUAL_PIXEL) (void* context, int x, int y);
typedef void (* CB_FLOOD_FILL) (void* context, int x1, int x2, int y);
BOOL WINAPI FloodFillGenerator (void* context, const RECT* src_rc, int x, int y,
                                CB_EQUAL_PIXEL cb_equal_pixel, CB_FLOOD_FILL cb_flood_fill);
```

The function `cb_equal_pixel` is called to determine if the target point is same with the starting point. The pixel of starting point can be transferred by context. The function `cb_flood_fill` is used to fill in a scan line. It passes the end-points of the scan line, that is, `(x1, y)` and `(x2, y)`.

### 15.7.8 Using Curve Generator

The way of using curve and filling generator is very simple. We firstly see how MiniGUI uses the curve and filling generator in its internal in order to deeply understand them.

The program paragraph comes from `FloodFill` function of MiniGUI (`src/newgdi/flood.c`):

```
static void _flood_fill_draw_hline (void* context, int x1, int x2, int y)
{
    PDC pdc = (PDC)context;
    RECT rcOutput = {MIN (x1, x2), y, MAX (x1, x2) + 1, y + 1};

    ENTER_DRAWING (pdc, rcOutput);
    _dc_draw_hline_clip (context, x1, x2, y);
    LEAVE_DRAWING (pdc, rcOutput);
}

static BOOL equal_pixel (void* context, int x, int y)
{
    gal_pixel pixel = _dc_get_pixel_cursor ((PDC)context, x, y);
```

```

    return ((PDC)context)->skip_pixel == pixel;
}

/* FloodFill
 * Fills an enclosed area (starting at point x, y).
 */
BOOL GUIAPI FloodFill (HDC hdc, int x, int y)
{
    PDC pdc;
    BOOL ret = TRUE;

    if (!(pdc = check_ecrgrn (hdc)))
        return TRUE;

    /* hide cursor temporarily */
    ShowCursor (FALSE);

    coor_LP2SP (pdc, &x, &y);

    pdc->cur_pixel = pdc->brushcolor;
    pdc->cur_ban = NULL;

    pdc->skip_pixel = _dc_get_pixel_cursor (pdc, x, y);

    /* does the start point have a equal value? */
    if (pdc->skip_pixel == pdc->brushcolor)
        goto equal_pixel;

    ret = FloodFillGenerator (pdc, &pdc->DevRC, x, y, equal_pixel, _flood_fill_draw_hline);
equal_pixel:
    UNLOCK_GCRINFO (pdc);

    /* Show cursor */
    ShowCursor (TRUE);

    return ret;
}

```

This function calls **FloodFillGenerator** after some necessary initialization and passes context **pdc** (**pdc** is the data structure in MiniGUI internal to represent DC) and two callback functions: **equal\_pixel** and **\_flood\_fill\_draw\_hline**. Before this, the function gets the pixel value of starting point and stored it in **pdc->skip\_pixel**. Function **equal\_pixel** gets the pixel value of the given point and returns the value after comparing to **pdc->skip\_pixel**; **\_flood\_fill\_draw\_hline** calls internal functions to plot horizontal line.

This way can greatly decrease the complexity of codes while increase their reuse capability. Readers who are interested in this can compare the function **LineTo** between old and new GDI interfaces.

The main aim of designing generator is to facilitate the users. For example, you can use MiniGUI's curve generator to finish your job. The example below assumes that you use a circle generator to plot a circle with 4-pixel wide:

```
static void draw_circle_pixel (void* context, int x1, int x2, int y)
{
    HDC hdc = (HDC) context;

    /*With each point on the circle as the center, fill the circle by radius 2*/
    FillCircle (hdc, x1, y, 2);
    FillCircle (hdc, x2, y, 2);
}

void DrawMyCircle (HDC hdc, int x, int y, int r, gal_pixel pixel)
{
    gal_pixel old_brush;

    old_brush = SetBrushColor (hdc, pixel);

    /* Call circle generator */
    CircleGenerator ((void*)hdc, x, y, r, draw_circle_pixel);

    /* Recover to the old brush color */
    SetBrushColor (hdc, old_brush);
}
```

The use of curve generator is very simple. Its structure is also very clear. You can learn this way during your own application developing.

## 15.8 Plotting Complex Curve

Based on the curve generator described in 15.7, MiniGUI provides the following basic curve plotting functions:

```
void GUIAPI MoveTo (HDC hdc, int x, int y);
void GUIAPI LineTo (HDC hdc, int x, int y);
void GUIAPI Rectangle (HDC hdc, int x0, int y0, int x1, int y1);
void GUIAPI PolyLineTo (HDC hdc, const POINT* pts, int vertices);
void GUIAPI SplineTo (HDC hdc, const POINT* pts);
void GUIAPI Circle (HDC hdc, int sx, int sy, int r);
void GUIAPI Ellipse (HDC hdc, int sx, int sy, int rx, int ry);
void GUIAPI CircleArc (HDC hdc, int sx, int sy, int r, int ang1, int ang2);
```

- **MoveTo** moves current starting point of pen to given point (**x**, **y**), specified in logical coordinates.
- **LineTo** draws line from current starting point to a given point (**x**, **y**), specified in logical coordinates.
- **Rectangle** draws a rectangle with the vertices as (**x0**, **y0**) and (**x1**, **y1**).
- **PolyLineTo** uses **LineTo** to draw a poly-line. **Pts** specify each of the vertices of poly-line while **vertices** specify the number of vertices.
- **SplineTo** uses **LineTo** function to draw spline. Four points can determine the only one spline, that is, **pts** is a pointer to a 4-point

structure array.

- **Circle** draws a circle with the center of (**sx**, **sy**) and radius of **r**, specified in logical coordinates.
- **Ellipse** draws an ellipse with the center of (**sx**, **sy**), x coordinate radius of **rx**, and y coordinate radius of **ry**.
- **CircleArc** draws a circle arc. The center of the circle is (**sx**, **sy**), radius is **r**, and the starting angel and ending angle of the arc is **ang1** and **ang2** respectively. **Ang1** and **ang2** use 1/64 degree as unit.

Let's see the use of **Circle** and **Ellipse** functions. Assuming there are two given points, **pts[0]** and **pts[1]**. **pts[0]** is the center of circle or ellipse, while **pts[1]** is a vertex of the bounding rectangle of them. Following code fragment plots the circle or ellipse specified by the two points:

```
int rx = ABS (pts[1].x - pts[0].x);
int ry = ABS (pts[1].y - pts[0].y);

if (rx == ry)
    Circle (hdc, pts[0].x, pts[0].y, rx);
else
    Ellipse (hdc, pts[0].x, pts[0].y, rx, ry);
```

## 15.9 Filling Enclosed Curve

MiniGUI provides following enclosed curve filling functions:

```
void GUIAPI FillBox (HDC hdc, int x, int y, int w, int h);
void GUIAPI FillCircle (HDC hdc, int sx, int sy, int r);
void GUIAPI FillEllipse (HDC hdc, int sx, int sy, int rx, int ry);
BOOL GUIAPI FillPolygon (HDC hdc, const POINT* pts, int vertices);
BOOL GUIAPI FloodFill (HDC hdc, int x, int y);
```

- **FillBox** fills a specified rectangle, the upper-left corner of which is (**x**, **y**), the width is **w** and the height is **h**. All are specified in logical coordinates.
- **FillCircle** fills a given circle with center of (**sx**, **sy**), radius of **r** (in logical coordinates).
- **FillEllipse** fills a given ellipse with the center of (**sx**, **sy**), x coordinate radius of **rx**, and y coordinate of **ry** (in logical coordinates).
- **FillPolygon** fills polygons. **Pts** means each vertex of the polygon, **vertices** means the number of those vertices.

- **FloodFill** does the flood filling from given point (**x**, **y**).

Note that all filling functions use the property of current brush (color) and affected by raster operation.

The following example illustrates how to use **FillCircle** and **FillEllipse** to fill a circle or an ellipse. Assuming two given points: **pts [0]** and **pts [1]**, **pts [0]** is the center of circle or ellipse while **pts [1]** is a vertex of the bounding rectangle of the circle or the ellipse.

```
int rx = ABS (pts[1].x - pts[0].x);
int ry = ABS (pts[1].y - pts[0].y);

if (rx == ry)
    FillCircle (hdc, pts[0].x, pts[0].y, rx);
else
    FillEllipse (hdc, pts[0].x, pts[0].y, rx, ry);
```

## 15.10 Building Complex Region

Apart from using curve generator and filling generator to plot a curve or fill a closed curve, we can also use the generators to build a complex region enclosed by closed curve. As we know, the region in MiniGUI is formed by non-intersected rectangles and meets the x-y-banned rule. Using above polygon or enclosed curve generator can regard each scan line as a rectangle with the height of one. Thus, we can use these generators to build complex region. MiniGUI uses existed enclosed curve generator to implement the following complex region generating functions:

```
BOOL GUIAPI InitCircleRegion (PCLIPRGN dst, int x, int y, int r);
BOOL GUIAPI InitEllipseRegion (PCLIPRGN dst, int x, int y, int rx, int ry);
BOOL GUIAPI InitPolygonRegion (PCLIPRGN dst, const POINT* pts, int vertices);
```

Using such functions we can initialize certain region as circle, ellipse, or polygon region. Then, we can use this region to perform hit-test (**PtInRegion** and **RectInRegion**), or select into a DC as the clipped region to get special painting effect. Fig. 15.2 is a special region effect given by **gdidemo.c** of MDE. The code building the special regions showed by Fig. 15.2 is listed in List 15.3.



## List 15.3 Creating special regions

```

static BLOCKHEAP my_cliprc_heap;
static void GDIDemo_Region (HWND hWnd, HDC hdc)
{
    CLIPRGN my_cliprgn1;
    CLIPRGN my_cliprgn2;

    /* Creat private heap of clipped rectangle for the region */
    InitFreeClipRectList (&my_cliprc_heap, 100);

    /* Initialize the region, and specify that the region uses the created private heap */
    /
    InitClipRgn (&my_cliprgn1, &my_cliprc_heap);
    InitClipRgn (&my_cliprgn2, &my_cliprc_heap);

    /* Initiaize two region with circle region and ellipse region respectively */
    InitCircleRegion (&my_cliprgn1, 100, 100, 60);
    InitEllipseRegion (&my_cliprgn2, 100, 100, 50, 70);

    /* Render the background with a blue brush */
    SetBrushColor (hdc, PIXEL_blue);
    FillBox (hdc, 0, 0, DEFAULT_WIDTH, 200);

    /* Substract region 2 from region1, and select the result into DC */
    SubtractRegion (&my_cliprgn1, &my_cliprgn1, &my_cliprgn2);
    SelectClipRegion (hdc, &my_cliprgn1);

    /* Fill region 1 with red brush */
    SetBrushColor (hdc, PIXEL_red);
    FillBox (hdc, 0, 0, 180, 200);

    /* Reinitialize region 1 to be a circle region, and shift region 2 right by 200 pixels */
    /*
    InitCircleRegion (&my_cliprgn1, 300, 100, 60);
    OffsetRegion (&my_cliprgn2, 200, 0);

    /* Perform the XOR Exclusive (OR) operation between two regions, and select the result into DC */
    XorRegion (&my_cliprgn1, &my_cliprgn1, &my_cliprgn2);
    SelectClipRegion (hdc, &my_cliprgn1);

    /* Fill the region with a red brush */
    FillBox (hdc, 200, 0, 180, 200);

    /* Reinitialize region 1 to be circle region, and shift region 2 right by 200 pixels */
    /*
    InitCircleRegion (&my_cliprgn1, 500, 100, 60);
    OffsetRegion (&my_cliprgn2, 200, 0);

    /* Perform the intersection operation of two given region, and select the result into DC */
    /*
    IntersectRegion (&my_cliprgn1, &my_cliprgn1, &my_cliprgn2);
    SelectClipRegion (hdc, &my_cliprgn1);

    /* Fill the region with a red brush */
    FillBox (hdc, 400, 0, 180, 200);

    /* Empty regions, and release the special clipped rectangle */
    EmptyClipRgn (&my_cliprgn1);
    EmptyClipRgn (&my_cliprgn2);

    /* Destroy the privite leap of clipped rectangle */
    DestroyFreeClipRectList (&my_cliprc_heap);
}

```



Fig. 15.2 The output effect of special regions

## 15.11 Visiting Frame Buffer Directly

In new GDI interfaces we add the function used to directly visit the video frame buffer. The prototype is as follows:

```
UInt8* GUIAPI LockDC (HDC hdc, const RECT* rw_rc, int* width, int* height, int* pitch);
void GUIAPI UnlockDC (HDC hdc);
```

- **LockDC** locks the specified rectangle region of given DC, then returns a pointer which points to the video frame buffer. When **width**, **height** and **pitch** are not NULL, the function **LockDC** will return the valid width, height and pitch of the locked rectangle.
- **UnlockDC** function unlocks a locked DC.

To lock a DC means that MiniGUI has entered a state of using exclusive way to visit video frame buffer. If the locked DC is a screen DC, the function will hide the mouse cursor when necessary and lock global clipping region. After locking a DC, program can visit locked frame buffer by using the returned pointer of this function. DC cannot be locked for a long time, neither calling other system calls while locking a DC.

Assuming we use the upper-left corner of locked rectangle as the origin to build coordinate system, x coordinate is horizontally rightward while y coordinate is vertically downward. We can use following formula to calculate the address corresponding to (x, y) point (assume the returned pointer value is **frame\_buffer**):

```
UInt8* pixel_add = frame_buffer + y * (*pitch) + x * GetGDCapability (hdc, GDCAP_BPP);
```

According to the color depth of the DC, we can perform read and write operation to the pixel. The following code fragment randomly fills a locked region:

```
int i, width, height, pitch;
RECT rc = {0, 0, 200, 200};
int bpp = GetGDCapability (hdc, GDCAP_BPP);
Uint8* frame_buffer = LockDC (hdc, &rc, &width, &height, &pitch);
Uint8* row = frame_buffer;

for (i = 0; i < *height; i++) {
    memset (row, rand ()%0x100, *width * bpp);
    row += *pitch;
}

UnlockDC (hdc);
```

## 15.12 YUV Overlay and Gamma Correction

In order to enhance the support for multi-media applications, we add support for YUV overlay and gamma correction.

### 15.12.1 YUV Overlay

In the multi-media area, we usually use YUV color space to represent color. If we need to display a MPEG-uncompressed image on computer's monitor, we need to convert YUV color space to RGB color space. YUV overlay comes from the acceleration function of some video chipsets. These video chipsets can do conversion from YUV to RGB on the basis of hardware so as to avoid the loss of performance. YUV overlay can directly write YUV information into a buffer, hardware can automatically finish the conversion from YUV to RGB, and display on RGB monitor. MiniGUI can also realize YUV overlay on those display chipsets that do not support YUV overlay. Here we call function `DisplayYUVOverlay` to convert YUV information and stretch to the DC device.

The prototype of YUV overlay operation function provided by MiniGUI is as follows:

```
/* ***** YUV overlay support ***** */
/* The most frequent vision overlay format.
 */
#define GAL_YV12_OVERLAY 0x32315659 /* Planar mode: Y + V + U (3 planes) */
#define GAL_IYUV_OVERLAY 0x56555949 /* Planar mode: Y + U + V (3 planes) */
```

```
#define GAL_YUV2_OVERLAY 0x32595559 /* Packed mode: Y0+U0+Y1+V0 (1 plane) */
#define GAL_UYVY_OVERLAY 0x59565955 /* Packed mode: U0+Y0+V0+Y1 (1 plane) */
#define GAL_VYU_OVERLAY 0x55595659 /* Packed mode: Y0+V0+Y1+U0 (1 plane) */

/* This function creates a vision output overlay
*/
GAL_Overlay* GUIAPI CreateYUVOverlay (int width, int height,
    Uint32 format, HDC hdc);

/* Lock the overlay to access buffer directly, and when it is over unlock it. */
int GAL_LockYUVOverlay (GAL_Overlay *overlay);
void GAL_UnlockYUVOverlay (GAL_Overlay *overlay);

#define LockYUVOverlay GAL_LockYUVOverlay
#define UnlockYUVOverlay GAL_UnlockYUVOverlay

/* Release the vision overlay */
void GAL_FreeYUVOverlay (GAL_Overlay *overlay);
#define FreeYUVOverlay GAL_FreeYUVOverlay

/* Transfer the the vision overlay onto the specified DC device. This function supports 2
dimensional scaling
*/
void GUIAPI DisplayYUVOverlay (GAL_Overlay* overlay, const RECT* dstrect);
```

Regarding information about video format can be referenced in:

<http://www.webartz.com/fourcc/indexyuv.htm>

Regarding information about color space relationship can be referenced in:

<http://www.neuro.sfc.keio.ac.jp/~aly/polygon/info/color-space-faq.html>

Code in List 15.4 comes from the gdidemo.c program of MDE. This program creates a YUV overlay object and converts a bitmap object into YUV data, finally fills the YUV overlay with the converted YUV data. In actual situations, YUV data usually comes from bottom hardware device, such as a video camera.

List 15.4 Creating YUV overlay

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>

static HDC pic;
static BITMAP logo;
static GAL_Overlay *overlay;

/* Transform RGB value to YUV value */
void RGBtoYUV(Uint8 r, Uint8 g, Uint8 b, int *yuv)
```

```

{
    yuv[0] = 0.299*r + 0.587*g + 0.114*b;
    yuv[1] = (b-yuv[0])*0.565 + 128;
    yuv[2] = (r-yuv[0])*0.713 + 128;
}

void ConvertRGBtoYV12(HDC pic, GAL_Overlay *o)
{
    int x,y;
    int yuv[3];
    Uint8 *op[3];
    int width = logo.bmWidth, height = logo.bmHeight;
    Uint8 r, g, b;

    /* Lock YUV overlay to access YUV data directly */
    LockYUVOverlay(o);

    /* Convert */
    for(y=0; y<height && y<=>h; y++)
    {
        op[0] = o->pixels[0] + o->pitches[0]*y;
        op[1] = o->pixels[1] + o->pitches[1]*(y/2);
        op[2] = o->pixels[2] + o->pitches[2]*(y/2);

        for ( x=0; x<width && x<=>w; x++)
        {
            /* Get RGB value of a pixel on a specified position of memory DC */
            GetPixelRGB (pic, x, y, &r, &g, &b);
            /* Transform RGB value of pixel to YUV value */
            RGBtoYUV (r, g, b, yuv);
            /* Assign data to YUV overlay */
            *(op[0]++) = yuv[0];
            if(x%2==0 && y%2==0)
            {
                *(op[1]++) = yuv[2];
                *(op[2]++) = yuv[1];
            }
        }
    }

    /* Unlock YUV overlay */
    UnlockYUVOverlay (o);
}

void Draw(void)
{
    RECT rect;
    int i;
    for(i=30; i<200; i++)
    {
        rect.left = i;
        rect.top=i;
        rect.right=rect.left + overlay->w;
        rect.bottom=rect.top + overlay->h;

        /* Display YUV overlay to the specified rectangle region of DC object */
        DisplayYUVOverlay(overlay, &rect);
    }
    printf("Displayed %d times.\n",i);
}

void test_yuv(HWND hwnd, HDC hdc)
{
    Uint32 overlay_format;
    int i;

    /* Create memory DC object with color depth of 32_bit, which is used to load bitmap */
    pic = CreateMemDC (400, 300, 32, MEMDC_FLAG_HWSURFACE, 0x00FF0000, 0x0000FF00, 0x000000FF, 0x00000000);

    /* Load a bitmap, and fill memory DC object with this bitmap */
    LoadBitmapFromFile (pic, &logo, "./res/sample.bmp");
    FillBoxWithBitmap (pic, 0, 0, 0, 0, &logo);

    /* Specify the format of YUV overlay as plane model: Y + V + U (three planes overall) */
}

```

```

overlaid format = GAL_YV12_OVERLAY:

/* Create YUV overlay of the same size as bitmap */
overlay = CreateYUVOverlay(logo.bmWidth, logo.bmHeight, overlaid_format, hdc);
if ( overlay == NULL ) {
    fprintf(stderr, "Couldn't create overlay!\n");
    exit(1);
}

/* transform RGB data of bitmap to YUV data, and assign it to YUV overlay */
ConvertRGBtoYV12(pic, overlay);

/* Display YUV overlay in DC */
Draw();

/* Destroy YUV overlay object */
FreeYUVOverlay (overlay);
/* Delete memory DC object */
DeleteMemDC (pic);
/* Destroy bitmap object */
UnloadBitmap (&logo);
}

```

### 15.12.2 Gamma Correction

Gamma correction uses each color channel of RGB color space to set gamma factor, and dynamically adjust the actual RGB effect on the RGB monitor. Gamma correction needs the video chipset's hardware support.

Application can set the gamma correction value of RGB's three-color channels by using function **SetGamma**. The prototypes of **SetGamma** are as follow:

```

int GAL_SetGamma (float red, float green, float blue);
#define SetGamma GAL_SetGamma

```

The scope of the linear gamma correction value is between 0.1 and 10.0. If the hardware does not support gamma correction, the function will return -1.

Application can use **SetGammaRamp** function to set the non-linear gamma correction value of three RGB color channels:

```

int GAL_SetGammaRamp (Uint16 *red, Uint16 *green, Uint16 *blue);
#define SetGammaRamp GAL_SetGammaRamp

int GAL_GetGammaRamp (Uint16 *red, Uint16 *green, Uint16 *blue);
#define GetGammaRamp GAL_GetGammaRamp

```

What function **SetGammaRamp** actually set is the gamma conversion table of each color channel. Each table is formed by 256 values, indicating the

inter-relationship between the setting value and actual value. When the RGB of certain pixel on the screen is R, G, B, the actual pixel RGB actually obtained on the display is `red[R]`, `green[G]`, and `blue[B]`. If hardware does not support gamma correction, the function will return -1.

Function `GetGammaRamp` gets the current Gamma conversion table.

The initial aim of gamma correction is to exactly restore an image on the display. Gamma value indicates the variation of contrast ratio of certain color channel. However, gamma correction has some special function in multi-media and game programs - Gamma correction can get the gradual effect of contrast ratio.

## 15.13 Advanced Two-Dimension GDI Functions

We added advanced two-dimension GDI functions in MiniGUI 1.5.x to support the development of advanced graphics applications. We can use these advanced two-dimension GDI function to set the advanced GDI properties such as pen width, pen type, and/or filling model. We will introduce the use of such advanced two-dimension GDI functions in this section.

### 15.13.1 Pen and Its Properties

Pen is a logical object used by MiniGUI to describe line drawing. We can determine the drawing activity by setting properties of a pen. Those properties include:

- Pen types. A pen has following types (Fig. 15.3 gives several kinds of effects of pen):
  - `PT_SOLID`: The Solid pen.
  - `PT_ON_OFF_DASH`: The on/off dash pen, even segments are drawn; odd segments are not drawn. The length of every dash/solid line segment is specified by `SetPenDashes`.
  - `PT_DOUBLE_DASH`: The double dash pen, even segments are

normally. Odd segments are drawn in the brush color if the brush type is **BT\_SOLID**, or in the brush color masked by the stipple if the brush type is **BT\_STIPPLED**.



Fig. 15.3 Pen types

- Width of pen. The width of pen controls the width of the plotted line segment. The width of painting pen uses pixel as unit. The default width is zero, we usually call the solid pen with the width of 0 as zero pen. General two-dimension GDI function of MiniGUI uses zero pen.
- The cap style of pen. The cap style determines the shape of end point of the line segment (see Fig. 15.4). It can be divided into the following styles:
  - **PT\_CAP\_BUTT**: The ends of the lines are drawn squared off and extending to the coordinates of the end point.
  - **PT\_CAP\_ROUND**: the ends of the lines are drawn as semicircles with the diameter equal to the line width and centered at the end point.
  - **PT\_CAP\_PROJECTING**: The ends of the lines are drawn squared off and extending half the width of the line beyond the end point.



Fig. 15.4 Cap styles of pen

- The join style of pen. The joining style of pen determines the way of the connection between two line segments (see Fig. 15.5). It is divided into the following styles:
  - **PT\_JOIN\_MITER**: The sides of each line are extended to meet at an angle.
  - **PT\_JOIN\_ROUND**: a circular arc joins the sides of the two lines.
  - **PT\_JOIN\_BEVEL**: the sides of the two lines are joined by a straight line, which makes an equal angle with each line.





Fig. 15.5 The join styles of pen

Table 15.1 gives the operations on pen properties.

Table 15.1 Operating functions on pen properties

Function	Purpose
<code>GetPenType/SetPenType</code>	Get/Set pen type
<code>GetPenWidth/SetPenWidth</code>	Get/Set pen width
<code>GetPenCapStyle/SetPenCapStyle</code>	Get/Set pen cap style
<code>GetPenJoinStyle/SetPenJoinStyle</code>	Get/Set pen join style

Before using dash pen, we need use `SetPenDashes` function to set the dash/solid means of pen. The prototype of this function is as follows:

```
void WINAPI SetPenDashes (HDC hdc, int dash_offset,
                        const unsigned char *dash_list, int n);
```

This function uses value in non-signed byte array to in turn represent the length of dash/solid line segments. For example, when `dash_list = "\1\1"`, the length of solid line segment is 1 pixel wide while length of dash line segment is also 1 pixel wide, it is so-called dot-line; if `dash_list="\4\1"`, it is lineation; if `dash_list="\4\1\1\1"`, it is lineation-dotline; if `dash_list="\4\1\1\1\1\1\1"`, it is lineation-dot-dotline. The above dash model can be seen in Fig. 15.6. It should be noted that draw solid line segment itself is affected by other properties of painting pen, such as cap style of pen.

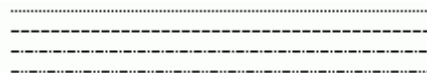


Fig. 15.6 Dash lines

The argument `dash_offset` of function `SetPenDashes` represents the starting place of dash/solid line segment in actual lines, usually is 0; the argument `n` means the length of `dash_list` non-signed byte array.

### 15.13.2 Brush and Its Properties

Brush is a logical object used to describe filling ways. The properties of the brush are simpler than pen. MiniGUI provides the following brush types:

- **BT\_SOLID**: The solid brush drawing with the current brush color.
- **BT\_TILED**: The tiled bitmap brush drawing with a tiled bitmap.
- **BT\_STIPPLED**: The transparent stippled bitmap brush drawing by using the stipple bitmap. Pixels corresponding to bits in the stipple bitmap that are set will be drawn in the brush color; pixels corresponding to bits that are not set will be left untouched.
- **BT\_OPAQUE\_STIPPLED**: The opaque stipple bitmap brush drawing by using the stipple bitmap. Pixels corresponding to bits in the stipple bitmap that are set will be drawn in the brush color; pixels corresponding to bits that are not set will be drawn with the background color.

Using function `GetBrushType/SetBrushType` can get and set the type of brush.

If the brush type is not solid, we need use `SetBrushInfo` to set the bitmap or stippled bitmap used by the brush. The prototype of this function is as follows:

```
void GUIAPI SetBrushInfo (HDC hdc, const BITMAP *tile, const STIPPLE *stipple);
```

The tiled bitmap is object of **BITMAP** in MiniGUI, while stippled bitmap is represented by **STIPPLE** structure.

We can treat stippled bitmap as monochrome bitmap, each bit of which represents a pixel. When the bit gets 1, it means that the drawing is based on brush color; when the bit gets 0, it means that the drawing is based on background color (brush type is **BT\_OPAQUE\_STIPPLED**) or reservation (brush type is **BT\_STIPPLED**). The **STIPPLE** structure is defined as follows:

```
typedef struct _STIPPLE
{
    int width;
    int height;
    int pitch;
    size_t size;
}
```

```
const unsigned char* bits;
} STIPPLE;
```

The following stipple bitmap can be used to represent slanting squares:

```
const unsigned char stipple_bits [] = "\x81\x42\x24\x18\x18\x24\x42\x81";
static STIPPLE my_stipple =
{
    8, 8, 1, 8,
    stipple_bits
};
```

There is an important concept when using brush, that is, the origin point of brush. The origin point of a brush determines the starting filling position of the brush bitmap. The upper-left corner of tiled bitmap or stippled bitmap will be aligned to the brush origin. By default, the brush origin is the origin of DC. Sometimes application need to reset the brush origin, it can call function **SetBrushOrigin**.

### 15.13.3 Advanced Two-Dimension Drawing Functions

When configuring MiniGUI, we can enable the advanced two-dimension GDI functions by using option `--enable-adv2dapi`. When MiniGUI includes the interface of advanced two-dimension GDI functions, all of the filling-typed functions mentioned before will be affected by the properties of current brush. These functions include **FillBox**, **FillCircle**, **FillEllipse**, **FillPolygon**, **FloodFill**, and so on. However, basic line/curve drawing functions will not be affected by the properties of pen. Those functions include **MoveTo/LineTo**, **Rectangle**, **PolyLineTo**, **SplineTo**, **Circle**, **Ellipse**, and **CircleArc**. These basic line/curve drawing functions still use zero pen to draw.

We introduced some advanced two-dimension drawing functions, the activity of which will be affected by pen and brush:

```
void GUIAPI LineEx (HDC hdc, int x1, int y1, int x2, int y2);
void GUIAPI ArcEx (HDC hdc, int sx, int sy, int width, int height, int angl, int ang2);
void GUIAPI FillArcEx (HDC hdc, int x, int y, int width, int height, int angl, int ang2);
void GUIAPI PolyLineEx (HDC hdc, const POINT *pts, int nr_pts);
void GUIAPI PolyArcEx (HDC hdc, const ARC *arcs, int nr_arcs);
void GUIAPI PolyFillArcEx (HDC hdc, const ARC *arcs, int nr_arcs);
```

- **LineEx** function plots a beeline according to the properties of current pen/brush. The line is from (x1, y1) to (x2, y2).
- **ArcEx** function plots an arc line according to the properties of current pen/brush. The center of the arc is (x, y), the width of the bounding box of the arc is width; the height of the bounding box of the arc is height; the starting angel the arc is ang1, in 1/64ths of a degree; ang2 is the end angel relative to starting angel, in 1/64ths of a degree. If ang2 is positive, representing anti-clockwise; ang2 is negative, representing clockwise. When ang2 is more than or equals to 360x64, it means an entire circle or ellipse, but not an arc.
- **FillArcEx** function fills an arc-shaped sector. The meaning of parameter is same as **ArcEx**.
- **PolyLinEx** function plots multiple lines according to the properties of current pen/brush. If having to joint lines, it will perform the joint according to the properties of current pen.
- **PolyArcEx** function plots multiple arcs according to the properties of current painting brush. If having to joint arcs, it will perform the joint according to the properties of pen. This function uses **ARC** structure to describe the parameter of each arc. The structure is defined as follows:

```
typedef struct _ARC
{
    /** the x coordinate of the left edge of the bounding rectangle. */
    int x;
    /** the y coordinate of the left edge of the bounding rectangle. */
    int y;
    /** the width of the bounding box of the arc. */
    int width;
    /** the height of the bounding box of the arc. */
    int height;
    /**
     * The start angle of the arc, relative to the 3 o'clock position,
     * counter-clockwise, in 1/64ths of a degree.
     */
    int angle1;
    /**
     * The end angle of the arc, relative to angle1, in 1/64ths of a degree.
     */
    int angle2;
} ARC;
```

- **PolyFillArcEx** function fills multiple arcs. This function uses **ARC** structure to describe the parameter of each arc.

#### 15.13.4 Using Advanced Two-Dimension GDI Functions

The code in List 15.5 uses the above advanced two-dimension GDI function to

draw some graphics objects. Fig. 15.7 gives the output effect of this code.

List 15.5 Using advanced two-dimension GDI functions

```
#ifndef _ADV_2DAPI

/* Define grid stripple bitmap */
const unsigned char stipple_bits [] = "\x81\x42\x24\x18\x18\x24\x42\x81";
static STIPPLE my_stipple =
{
    8, 8, 1, 8,
    stipple_bits
};

/* Example code of advanced 2D GDI functions */
void GDIDemo_Adv2DAPI (HWND hWnd, HDC hdc)
{
    POINT pt [10];
    BITMAP bitmap;

    /* Load a bitmap as brush tile bitmap */
    if (LoadBitmap (hdc, &bitmap, "res/sample.bmp"))
        return;

    /* Set pen type, dash style and width */
    SetPenType (hdc, PT_SOLID);
    SetPenDashes (hdc, 0, "\1\1", 2);
    SetPenWidth (hdc, 5);

    /* Set the end point style of pen is "round end point" */
    SetPenCapStyle (hdc, PT_CAP_ROUND);

    /* Plot a line point */
    LineEx (hdc, 10, 10, 50, 50);

    /* Set pen jointing style as "bevel joining" */
    SetPenJoinStyle (hdc, PT_JOIN_BEVEL);

    /* Plot poly line segment */
    pt [0].x = 20;    pt [0].y = 20;
    pt [1].x = 80;    pt [1].y = 20;
    pt [2].x = 80;    pt [2].y = 80;
    pt [3].x = 20;    pt [3].y = 80;
    pt [4].x = 20;    pt [4].y = 20;
    PolyLineEx (hdc, pt, 5);

    /* Set pen width to be 20, pen color to be red, pen end point style to
    /*be "round end point"
    */
    SetPenWidth (hdc, 20);
    SetPenColor (hdc, PIXEL_red);
    SetPenCapStyle (hdc, PT_CAP_ROUND);

    /* Plot a line segment */
    LineEx (hdc, 80, 80, 400, 300);

    /* Set pen color to be blue */
    SetPenColor (hdc, PIXEL_blue);

    /* Plot an arc across thid quadrant and forth quadrant */
    ArcEx (hdc, 100, 100, 200, 300, 180*64, 180*64);

    /* Set brush type as solid */
    SetBrushType (hdc, BT_SOLID);
    /* Set brush color as green */
    SetBrushColor (hdc, PIXEL_green);

    /* Fill arc from 0 degree to 120 degree */
    FillArcEx (hdc, 100, 0, 200, 100, 0, 120*64);

    /* Set brush type as tile */
    SetBrushType (hdc, BT_TILED);
}
```

```
SetBrushInfo (hdc, &bitmap, &my_stipple);

/* Set the origin of brush */
SetBrushOrigin (hdc, 100, 100);
/* Fill arc from 0 to 270 degree with tile brush */
FillArcEx (hdc, 100, 100, 200, 100, 0, 270*64);

/* Set brush type as transparent stipple brush, to be filled with grid */
SetBrushType (hdc, BT_STIPPLED);
/* Fill arc from 0 to 360 degree with the transparent stipple brush */
FillArcEx (hdc, 100, 300, 200, 100, 0, 360*64);

/* Set pen type as double dash line, fill the dash line segemnt with stipple brush */
SetPenType (hdc, PT_DOUBLE_DASH);
/* Set the dash and solid length of pen */
SetPenDashes (hdc, 0, "\20\40", 2);
/* Set the pen end point style as BUTT end point */
SetPenCapStyle (hdc, PT_CAP_BUTT);
/* Set the dash width as 20 */
SetPenWidth (hdc, 20);

/* Plot a line segment */
LineEx (hdc, 500, 0, 20, 100);

/* Set the pen type as opaque stipple bitmap */
SetBrushType (hdc, BT_OPAQUE_STIPPLED);
/* Fill an arc across thid quadrant and forth quadrant using opaque stipple bitmap */
ArcEx (hdc, 400, 100, 200, 300, 180*64, 180*64);

/* Unload bitmap */
UnloadBitmap (&bitmap);
}
#endif /* _ADV_2DAPI */
```

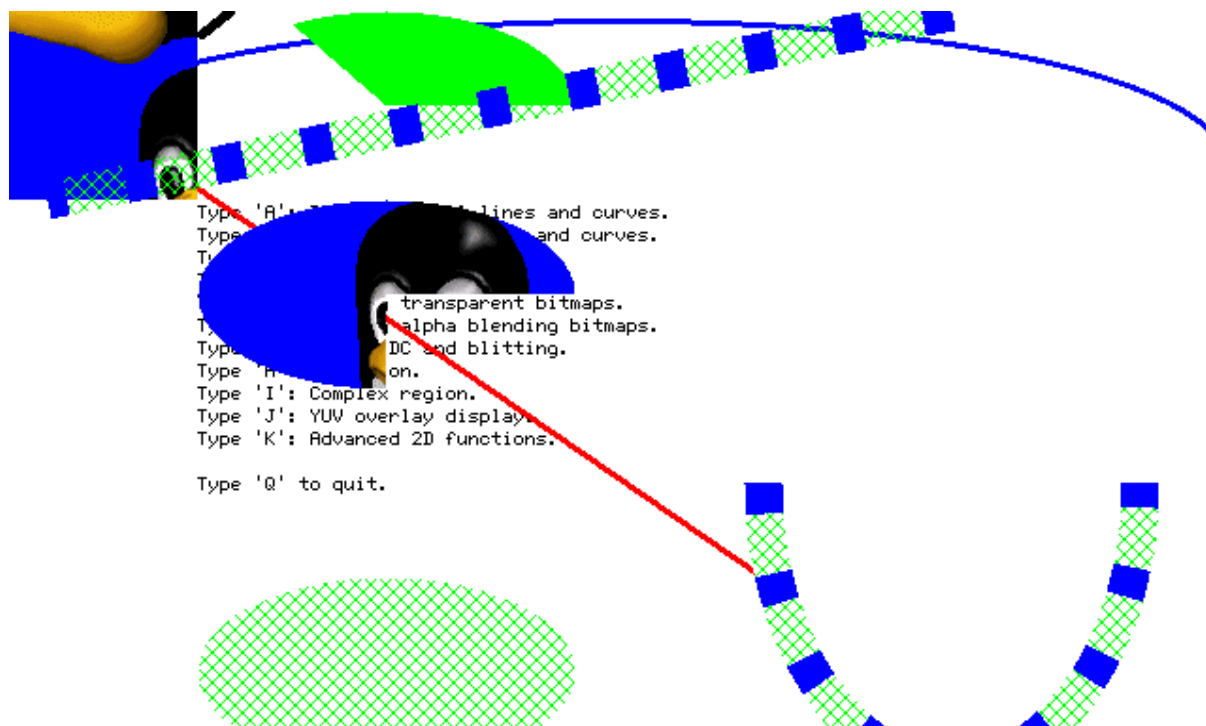


Fig. 15.7 The output of advanced two-dimension GDI functions

## 15.14 Support for Slave Screens

MiniGUI supports the slave screens if there are multiple video devices installed. The slave screen only can display information and cannot receive any input at all. The slave screen is abstracted as a DC object. So GDI functions can be called for the slave screen.

### 15.14.1 Creating Slave Screen

The following function can be used to open the slave screen:

```
HDC GUIAPI InitSlaveScreen (const char *name, const char *mode);
```

**InitSlaveScreen** returns a graphics device context handle by a special device name and display mode.

### 15.14.2 Destroying Slave Screen

The following function can be used to destroy the slave screen; you should use this function to release the internal resource associated with the slave screen:

```
HDC GUIAPI TerminateSlaveScreen(HDC hdc);
```

Here, **hdc** is a graphic device context handle returned by **InitSlaveScreen**.





## **III MiniGUI Advanced Programming Topics**

- Inter-Process Communication and Asynchronous Event Process
- Developing a Customized MiniGUI-Processes Server Program
- Developing a Customized IAL Engine



## 16 Inter-Process Communication and Asynchronous Event Process

In this chapter, we introduce how application processes asynchronous event, and how it implements the inter-process communication by using APIs provided by MiniGUI.

### 16.1 Asynchronous Event Process

Usually the programming interfaces provided by a GUI system mainly focus on windowing, messaging, and graphics device. However, a GUI usually provides its own mechanism while processing system events and such mechanism is always incompatible with the one provided by operating system itself. The structure of an application is usually message-drive corresponding to the message loop mechanism of MiniGUI. In other words, application works through passively receiving messages. If application needs to actively monitor certain system events, for example, in UNIX operating system, it can call `select` system call to monitor if a certain file descriptor has readable data. We need combine the message queue mechanism of MiniGUI and other existed mechanisms of OS, in order to provide a consistent mechanism for application. We will illustrate several methods to resolve this problem in the chapter.

As we know, there is only one message queue in an application running on MiniGUI-Processes. Application will create a message loop after being initialized, and then continuously get messages from this queue until receiving message `MSG_QUIT`. When the window procedure of the application handles a message, it should immediately returns after having handled the message in order to have a chance to get other messages and handle them. If an application calls `select` to listen in a certain file descriptor, it is possible to encounter a problem: as `select` system call may cause long-time block, and those events sent to the application by the MiniGUI-Processes serve will not be processed in time. Therefore, the way of message driving and `select` system call is difficult to be well integrated. In MiniGUI-Threads, each thread has its

own corresponding message queue, while system message queue is managed by solely performed desktop thread. Therefore, the threads created by any application can be long-time blocked and can call select like system calls. However, in MiniGUI-Processes, if you need to listen in a file descriptor event in one application, you must handle it correctly to avoid the block.

Under MiniGUI-Processes, we have some ways of resolving this problem:

- When calling `select` system call, pass the value of timeout to ensure `select` system call will not be long-time blocked.
- Using timer. When the timer is expired, you can use `select` system call to check the listened file descriptor. If there is not any event occurring, it will return immediately, otherwise perform read or write operation.
- Using function `RegisterListenFD` provided by MiniGUI-Processes to register a listened file descriptor. When a desired event occurs, MiniGUI-Processes will send message `MSG_FDEVENT` to a certain window.

As the former two solutions are comparatively simple, here we will focus on the third solution. MiniGUI-Processes provides the following functions and one macro to application:

```
#define MAX_NR_LISTEN_FD 5

/* Return TRUE if all OK, and FALSE on error. */
BOOL GUIAPI RegisterListenFD (int fd, int type, HWND hwnd, void* context);

/* Return TRUE if all OK, and FALSE on error. */
BOOL GUIAPI UnregisterListenFD (int fd);
```

- Macro `MAX_NR_LISTEN_FD` defines the maximal number of file descriptors that can be listened by MiniGUI. The default value is 5.
- `RegisterListenFD` registers a file descriptor that needs to be listened. You should specify the event type (the argument of type can be one of `POLLIN`, `POLLOUT`, or `POLLERR`), the context information, and the window handle receiving the message `MSG_FDEVENT`.
- `UnregisterListenFD` unregisters a registered listening file descriptor.

After application uses `RegisterListenFD` function to register a listening file descriptor, MiniGUI will send a message of `MSG_FDEVENT` to the specified window when the specified event occurs on the descriptor. Application can

handle this message in the window procedure. Libvcongui of MiniGUI uses the above function to listen to the readable event coming from the master pseudo terminal, seen as follows (`vcongui/vcongui.c`):

```
...

/* Registers a listened file descriptor for bogus master control terminal */
RegisterListenFD (pConInfo->masterPty, POLLIN, hMainWnd, 0);

/* Go into message loop */
while (!pConInfo->terminate && GetMessage (&Msg, hMainWnd)) {
    DispatchMessage (&Msg);
}
/* Unregisters a listened file descriptor */
UnregisterListenFD (pConInfo->masterPty);

...

/* Window process of virtual console */
static int VConGUIMainWinProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    PCONINFO pConInfo;

    pConInfo = (PCONINFO)GetWindowAdditionalData (hWnd);
    switch (message) {

        ...

        /* If MSG_FDEVENT is received, handle the input data on bogus master control terminal */
        case MSG_FDEVENT:
            ReadMasterPty (pConInfo);
            break;

        ...

    }

    /* Call the default window process */
    if (pConInfo->DefWinProc)
        return (*pConInfo->DefWinProc) (hWnd, message, wParam, lParam);
    else
        return DefaultMainWinProc (hWnd, message, wParam, lParam);
}
```

We can see the use of `RegisterListenFD` in the following section. Obviously, a MiniGUI-Processes program can conveniently use the bottom message mechanism to finish the process of asynchronous event by using this simple interface of registering listening file descriptor.

## 16.2 MiniGUI-Processes and Inter-Process Communication

As we know, MiniGUI-Processes uses UNIX domain sockets to realize the interaction between client programs and the server. Application also can use this mechanism to finish its own communicating task – send a request from client, while the server will process the request and send back the response. On

one hand, in the MiniGUI-Processes serve program, you can extend this mechanism to register your own request processing functions to implement your request-response communicating task. On the other hand, MiniGUI-Processes also provides some wrap functions used to create and operate UNIX domain sockets. Any application under MiniGUI-Processes can create UNIX domain sockets and finish the data exchange with other MiniGUI-Processes applications. This chapter will describe how to use the functions provided by MiniGUI-Processes to finish such kind of communication. Before introducing the certain interface, let's first to understand the multi-process communication model and communication between the server and the clients under MiniGUI-Processes.

### 16.2.1 Multi-Process Model under MiniGUI-Processes

MiniGUI-Processes is a multi-process system with C/S architecture. Only one server program is running during the life cycle. The global variable `mgServer` in the server is set to `TRUE`; the other MiniGUI applications are clients, `mgServer` is set to `FALSE`. Each application runs in different process space, seen as Fig. 16.1.

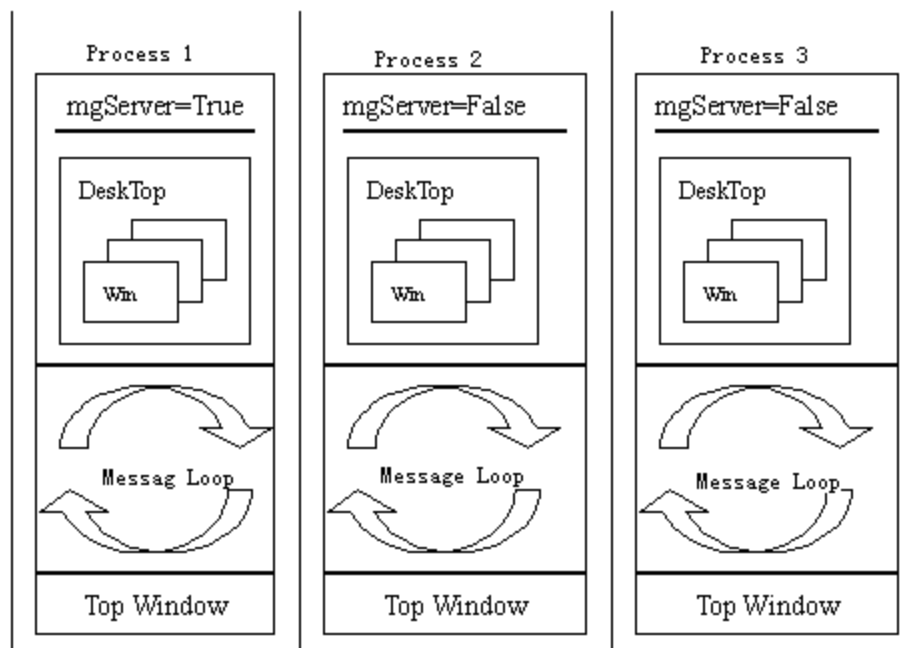


Fig. 16.1 Multi-process model of MiniGUI-Processes

The current program structure allows each process to have its own (virtual) desktop and message queue. The process communication model includes data exchange realized by shared memory and the client/server communication model realized by UNIX domain sockets.

### 16.2.2 Simple Request/Response Processing

MiniGUI-Processes uses UNIX domain socket to communicate between the server and the clients. In order to facilitate such communication, MiniGUI-Processes introduces a simple request/response mechanism. Clients send request to the server through a specific structure; the server handles the request and responds. At the end of clients, a request is defined as follows (`minigui/minigui.h`):

```
typedef struct tagREQUEST {
    int id;
    const void* data;
    size_t len_data;
} REQUEST;
typedef REQUEST* PREQUEST;
```

Here, `id` is an integer used to identify the request type; `data` is the data associated to the request; `len_data` is the length of the data in bytes. After initializing a `REQUEST` structure, clients can call `ClientRequest`<sup>6</sup> to send the request to the server, and wait for the response from the server. The prototype of this function is as follows:

```
/* send a request to server and wait reply */
int ClientRequest (PREQUEST request, void* result, int len_rslt);
```

The server program (`mginit`) can get all client requests during its own message loop; it will process each request and send the final result to the client. The server can call `ServerSendReply`<sup>7</sup> to send the reply to the client:

```
int GUIAPI ServerSendReply (int clifd, const void* reply, int len);
```

<sup>6</sup> Before MiniGUI V2.0.4/V1.6.10, the function's name is `cli_request`.

<sup>7</sup> Before MiniGUI V2.0.4/V1.6.10, the function's name is `send_reply`.

In the above simple C/S communication, the clients and the server must be accord with each other for a certain request type, that is, the clients and the server must understand each type of data and process it properly.

MiniGUI-Processes uses the above way to realize most of system-level communication tasks:

- Mouse cursor management. Mouse cursor is a global resource. When you create or destroy it, change the shape or position of it, or show or hide it, you should send a request to the server. The server will finish the task and reply the result to you.
- Layer management. When a client checks the information of layers, creates new layer, joins a certain existed layer, or delete a layer, it sends a request to the server.
- Window management. When a client creates, destroy, and move a main window, it sends a request to the server.
- Other system-level works. For example, in new GDI interfaces, the server manages the video memory of the video adapter. When clients need create the memory DC in video memory, they will send a request to the server.

In order to allow application to simply implement the communication between clients and the server by using the mechanism above, you can register some customized request process functions in the server, and then clients can send requests to the server. MiniGUI-Processes provides the following interfaces to the server:

```
#define MAX_SYS_REQID      0x0014
#define MAX_REQID         0x0020

/*
 * Register user defined request handlers for server
 * Note that user defined request id should larger than MAX_SYS_REQID
 */
typedef int (* REQ_HANDLER) (int cli, int clifd, void* buff, size_t len);
BOOL GUIAPI RegisterRequestHandler (int req_id, REQ_HANDLER your_handler);
REQ_HANDLER GUIAPI GetRequestHandler (int req_id);
```

The server can register a request handler by calling **RegisterRequestHandler**. Note that the prototype of request handler is defined by **REQ\_HANDLER**. MiniGUI



has also defined two macros: **MAX\_SYS\_REQID** and **MAX\_REQID**. **MAX\_REQID** is the maximum request identifier that can be registered; while **MAX\_SYS\_REQID** does MiniGUI use the maximum request identifier internally. In other words, the request identifier registered by **RegisterRequestHandler** must be larger than **MAX\_SYS\_REQID** and less than or equal to **MAX\_REQID**.

We assume that the server calculates the sum of two integers for clients. Clients send two integers to the server, while the server sends the sum of two integers to the clients. The following program runs in the server program and registers a request handler in the system:

```
typedef struct TEST_REQ
{
    int a, b;
} TEST_REQ;

static int test_request (int cli, int clifd, void* buff, size_t len)
{
    int ret_value = 0;
    TEST_REQ* test_req = (TEST_REQ*)buff;

    ret_value = test_req.a + test_req.b;

    return ServerSendReply (clifd, &ret_value, sizeof (int));
}

...
RegisterRequestHandler (MAX_SYS_REQID + 1, test_request);
...
```

Client can send a request to the server to get the sum of two integers by using the following program:

```
REQUEST req;
TEST_REQ test_req = {5, 10};
int ret_value;

req.id = MAX_SYS_REQID + 1;
req.data = &test_req;
req.len_data = sizeof (TEST_REQ);

ClientRequest (&req, &ret_value, sizeof (int));
printf ("the returned value: %d\n", ret_value);    /* ret_value should be 15 */
```

By using this simple request/response technology, MiniGUI-Processes can create a convenient inter-process communication mechanism between the clients and the server. However, this technology also has some shortcomings, for example, the number of request is limited by the value of **MAX\_REQID**, the communication mechanism is not flexible, and the request can only be sent to

the server (`mginit`) of MiniGUI-Processes from the clients.

## 16.2.4 Wraps for UNIX Domain Socket

In order to solve the problem of above simple request/reply mechanism, MiniGUI-Processes also provides some wrap functions for UNIX domain socket. The prototypes of these functions are as follow (`minigui/minigui.h`):

```
/* Used by server to create a listen socket.
 * Name is the name of listen socket.
 * Please located the socket in /var/tmp directory. */

/* Returns fd if all OK, -1 on error. */
int serv_listen (const char* name);

/* Wait for a client connection to arrive, and accept it.
 * We also obtain the client's pid and user ID from the pathname
 * that it must bind before calling us. */

/* returns new fd if all OK, < 0 on error */
int serv_accept (int listenfd, pid_t *pidptr, uid_t *uidptr);

/* Used by clients to connect to a server.
 * Name is the name of the listen socket.
 * The created socket will located at the directory /var/tmp,
 * and with name of '/var/tmp/xxxxxx-c', where 'xxxxxx' is the pid of client.
 * and 'c' is a character to distinguish diferent projects.
 * MiniGUI use 'a' as the project character.
 */

/* Returns fd if all OK, -1 on error. */
int cli_conn (const char* name, char project);

#define SOCKERR_IO          -1
#define SOCKERR_CLOSED     -2
#define SOCKERR_INVARG     -3
#define SOCKERR_OK         0

/* UNIX domain socket I/O functions. */

/* Returns SOCKERR_OK if all OK, < 0 on error.*/
int sock_write_t (int fd, const void* buff, int count, unsigned int timeout);
int sock_read_t (int fd, void* buff, int count, unsigned int timeout);

#define sock_write(fd, buff, count) sock_write_t(fd, buff, count, 0)
#define sock_read(fd, buff, count) sock_read_t(fd, buff, count, 0)
```

The above functions are used to create UNIX domain socket and perform data transfer. They are the wraps of basic socket function provided by the operating system. The uses of these functions are described as follow:

- **serv\_listen**: The server calls this function to create a listening socket and returns socket file descriptor. It is suggested to create server listening socket under the directory of `/var/tmp/`.
- **serv\_accept**: The server calls this function to accept the connection

request from clients.

- **cli\_conn**: Client calls this function to connect the server. Name is the listening socket of the server. The socket created for the client is stored in the directory of `/var/tmp/`, named with `<pid>-c`, in which `c` is used to differentiate the purpose of different socket communication task, and specified by the argument of project. MiniGUI-Processes internally uses `a`, so the sockets created by application program should use alphabets excluding `a`.
- **sock\_write\_t**: After creating the connection, client and server can use **sock\_write\_t** and **sock\_read\_t** to perform data exchange. Arguments of **sock\_write\_t** are similar to system call `write`, but can pass a timeout value in the unit of 10ms. When the argument is zero, the timeout is disabled. Note that timeout setting is only valid in the program `mginit`.
- **sock\_read\_t**: Arguments of **sock\_read\_t** are similar to system call `read`, but can pass a timeout value in the unit of 10ms. When the argument is zero, the timeout is disabled. Note that timeout setting is only valid in program `mginit`.

The following code illustrates how a server program creates a listening socket by using above functions:

```
#define LISTEN_SOCKET    "/var/tmp/mysocket"

static int listen_fd;

BOOL listen_socket (HWND hwnd)
{
    if ((listen_fd = serv_listen (LISTEN_SOCKET)) < 0)
        return FALSE;
    return RegisterListenFD (fd, POLL_IN, hwnd, NULL);
}
```

When the server receives the client connection request, the window of `hwnd` will receive message **MSG\_FDEVENT**, then the server can accept this connection request:

```
int MyWndProc (HWND hwnd, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        ...

        case MSG_FDEVENT:
```

```

        if (LOWORD (wParam) == listen_fd) { /* From monitor socket */
            pid_t pid;
            uid_t uid;
            int conn_fd;
            conn_fd = serv_accept (listen_fd, &pid, &uid);
            if (conn_fd >= 0) {
                RegisterListenFD (conn_fd, POLL_IN, hwnd, NULL);
            }
        }
        else { /* From the linked monitor socket */
            int fd = LOWORD(wParam);
            /* Handle the data from client */
            sock_read_t (fd, ...);
            sock_write_t (fd, ....);
        }
        break;

        ...
    }
}

```

In the above code, the server registers the connected file descriptor as listening descriptor. Therefore, when handling message **MSG\_FDEVENT**, the server should determine the file descriptor type that causes message **MSG\_FDEVENT**.

Client can use the following code to connect to the server:

```

int conn_fd;

if ((conn_fd = cli_conn (LISTEN_SOCKET, 'b')) >= 0) {
    /* Send a require to the server */
    sock_write_t (fd, ....);
    /* Get the handled result from the server */
    sock_read_t (fd, ....);
}

```

## 17 Developing Customized MiniGUI-Processes Server Program

**Mginit** is the server program of MiniGUI-Processes. This program prepares shared resource for client programs and manages windows created by clients. We explain how to write a customized MiniGUI-Processes server program according to the project requirement in this chapter. We first use program **mginit** of MDE as an example to analyze the basic constitution of **mginit** program.

### 17.1 Mginit of MDE

**Mginit** of MDE is relatively simple, which is mainly responsible for following tasks:

- Initializing itself as the server of MiniGUI-Processes;
- Showing two dialog boxes displaying copyright information;
- Reading **mginit.rc** configuration file and creating a task bar;
- Starting up the default startup program specified by **mginit.rc** file;
- Maintaining information of MiniGUI-Processes layers and switching between layers in the taskbar window procedure.

Now we analyze the implementation of **mginit** of MDE.

#### 17.1.1 Initializing Itself as the Server of MiniGUI-Processes

The following code is located in function **MiniGUIMain**:

```
int MiniGUIMain (int args, const char* arg[])
{
    int pid_desktop;
    struct sigaction siga;
    MSG msg;

    OnNewDelClient = on_new_del_client;
    OnChangeLayer = on_change_layer;

    if (!ServerStartup (0, 0, 0)) {
        fprintf (stderr, "Can not start the server of MiniGUI-Processes: mginit.\n");
        return 1;
    }
}
```

```

}

if (!InitMiniGUIExt ()) {
    fprintf (stderr, "Can not init mgext library.\n");
    return 1;
}

...

```

First, this function initializes variables `OnNewDelClient` and `OnChangeLayer`, which are the global variables owned by the server program. After this, the function calls `ServerStartup` to start the server function of `mginit`.

`ServerStartup` function will create listening socket, prepare to accept the connection requests from clients. So far the server initialization has finished. Let's analyze the above process in details.

### 1. Listen events coming from clients and layers

The global variables `OnNewDelClient` and `OnChangeLayer` are only available for MiniGUI-Processes server program. They are two function pointer variables. When a client connects to `mginit` or drops the socket connection with `mginit`, and if the variable of `OnNewDelClient` has been set, MiniGUI will call the function pointed to by this variable. In `minigui/minigui.h`, the declaration of this function prototype is as follows:

```
typedef void (* ON_NEW_DEL_CLIENT) (int op, int cli);
```

The first argument indicates the operation that will be processed. When the parameter gets `LCO_NEW_CLIENT`, it represents that there is new client being connecting to the server; if parameter is `LCO_DEL_CLIENT`, it means that a client is dropping the connection. The second argument `cli` is the client identifier; it is an integer.

When the layer of MiniGUI-Processes changes, for example, a new client joins to a certain layer, and if the variable of `OnChangeLayer` has been set, MiniGUI will call the function pointed to by this variable. In `minigui/minigui.h`, the declaration of the prototype of this function is as follows:

```
typedef void (* ON_CHANGE_LAYER) (int op, MG_Layer* layer, MG_Client* client);
```

The first argument represents the event type:

- **LCO\_NEW\_LAYER**: A new layer has been created;
- **LCO\_DEL\_LAYER**: A layer has been deleted;
- **LCO\_JOIN\_CLIENT**: A client joined to a certain layer;
- **LCO\_REMOVE\_CLIENT**: A certain client is deleted from the layer it lies;
- **LCO\_TOPMOST\_CHANGED**: The top-most layer changes, that is, the switch of layer occurred;

The second argument is the pointer to the relevant layer. The third argument is the pointer to the relevant client.

The global variable **OnZNodeOperation** only available for MiniGUI-Processes server program since MiniGUI V2.0.4/V1.6.10. It is also function pointer variable. When the znode changes in MiniGUI-Processes and the variable of **OnZNodeOperation** has been set, MiniGUI will call the function pointed to by this variable. In **minigui/minigui.h**, the declaration of this function prototype is as follows:

```
typedef void (* ON_ZNODE_OPERATION) (int op, int cli, int idx_znode);
ON_ZNODE_OPERATION OnZNodeOperation;
```

The first argument represents the event type:

- **ZNOP\_ALLOCATE**: The z-node has been created;
- **ZNOP\_FREE**: The z-node has been destroyed;
- **ZNOP\_MOVE2TOP**: The z-node has been moved to be the topmost one;
- **ZNOP\_SHOW**: The z-node has been shown;
- **ZNOP\_HIDE**: The z-node has been hidden;
- **ZNOP\_MOVEWIN**: The z-node has been moved or its size has changed;
- **ZNOP\_SETACTIVE**: The z-node has been set to be the active one;
- **ZNOP\_ENABLEWINDOW**: The z-node is disabled or enabled;
- **ZNOP\_STARTDRAG**: Start to drag the z-node;
- **ZNOP\_CANCELDRAG**: Cancel to drag the z-node;
- **ZNOP\_CHANGECAPTION**: The caption of the z-node has changed.

The second argument **cli** is the client identifier; it is an integer. The third

argument `idx_znode` is the znode index.

The global variables `OnLockClientReq`, `OnTrylockClientReq`, and `OnUnlockClientReq` are available for MiniGUI-Processes client program. They are three function pointer variables. They can avoid thread deadlock problem in the same process. In `minigui/minigui.h`, the declaration of this function prototype is as follows:

```
typedef int (* ON_LOCK_CLIENT_REQ) (void);
ON_LOCK_CLIENT_REQ OnLockClientReq;

typedef int (* ON_TRYLOCK_CLIENT_REQ) (void);
ON_TRYLOCK_CLIENT_REQ OnTrylockClientReq;

typedef int (* ON_UNLOCK_CLIENT_REQ) (void);
ON_UNLOCK_CLIENT_REQ OnUnlockClientReq;
```

Lock mechanism can depend on applications. For example, we can use POSIX mutex to implement lock and unlock function.

```
int MiniGUIMain (int args, const char* arg[])
{
    #if defined(_LITE_VERSION) && !(_STAND_ALONE)
    ...
    static pthread_mutex_t mutex;

    static void __mg_lock_cli_req(void)
    {
        pthread_mutex_lock (&mutex);
    }

    static int __mg_trylock_cli_req(void)
    {
        return pthread_mutex_trylock (&mutex);
    }

    static void __mg_unlock_cli_req(void)
    {
        pthread_mutex_unlock (&mutex);
    }

    pthread_mutex_init (&mutex, NULL);

    OnLockClientReq = __mg_lock_cli_req;
    OnTrylockClientReq = __mg_trylock_cli_req;
    OnUnlockClientReq = __mg_unlock_cli_req;
    ...
    /*todo: JoinLayer*/
    ...
    #endif
    ...

    pthread_mutex_destroy (&mutex);
}
```

Once having client identifier or layer pointer and client pointer, `mginit` can



easily access the internal data structure of MiniGUI library, and so can get some system information. For doing this, MiniGUI defines the following global variables:

- **mgClients**: An **MG\_client** structure pointer. It points to an **MG\_client** structure array including all clients' information. You can use the client identifier to access it.
- **mgTopmostLayer**: An **MG\_Layer** structure pointer, pointing to the current top-most layer.
- **mgLayers**: An **MG\_Layer** structure pointer, pointing to the head of a linked-list of all layers in the system.

**Mginit** program can visit these data structure at any time to get the current client and the current layer information. Regarding members of **MG\_client** and **MG\_Layer** structures can refer to *MiniGUI API Reference*.

**Mginit** of MDE defines two functions to handle above events, and also sets two global variables mentioned above.

The first function is **on\_new\_del\_client**, which does not have material work, but just simply prints the name of the client coming or dropping the socket connection.

The second function is **on\_change\_layer**, which mainly handles events **LCO\_NEW\_LAYER**, **LCO\_DEL\_LAYER**, and **LCO\_TOPMOST\_CHANGED**. When system creates new layers, this function will create a new button in taskbar and assign the handle value of this button to **dwAddData** member of the current layer; when system deletes a certain layer, the function destroys the button corresponding the layer; when the top-most layer of system changes, the function calls **on\_change\_topmost** to adjust the state of these buttons representing layers. The code of this function is as follows:

```
static void on_change_layer (int op, MG_Layer* layer, MG_Client* client)
{
    static int nr_boxes = 0;
    static int box_width = _MAX_WIDTH_LAYER_BOX;
    int new_width;
```

```

if (op > 0 && op <= LCO_ACTIVE_CHANGED)
    printf (change_layer_info [op], layer?layer->name:"NULL",
            client?client->name:"NULL");

switch (op) {
case LCO_NEW_LAYER:
    nr_boxes ++;
    if (box_width * nr_boxes > _WIDTH_BOXES) {
        new_width = _WIDTH_BOXES / nr_boxes;
        if (new_width < _MIN_WIDTH_LAYER_BOX) {
            new_width = _MIN_WIDTH_LAYER_BOX;
        }

        if (new_width != box_width) {
            adjust_boxes (new_width, layer);
            box_width = new_width;
        }
    }

    layer->dwAddData = (DWORD)CreateWindow (CTRL_BUTTON, layer->name,
        WS_CHILD | WS_VISIBLE | BS_CHECKBOX | BS_PUSHLIKE | BS_CENTER,
        _ID_LAYER_BOX,
        _LEFT_BOXES + box_width * (nr_boxes - 1), _MARGIN,
        box_width, _HEIGHT_CTRL, hTaskBar, (DWORD)layer);

    break;

case LCO_DEL_LAYER:
    DestroyWindow ((HWND)(layer->dwAddData));
    layer->dwAddData = 0;
    nr_boxes --;
    if (box_width * nr_boxes < _WIDTH_BOXES) {
        if (nr_boxes != 0)
            new_width = _WIDTH_BOXES / nr_boxes;
        else
            new_width = _MAX_WIDTH_LAYER_BOX;

        if (new_width > _MAX_WIDTH_LAYER_BOX)
            new_width = _MAX_WIDTH_LAYER_BOX;

        adjust_boxes (new_width, layer);
        box_width = new_width;
    }
    break;

case LCO_JOIN_CLIENT:
    break;
case LCO_REMOVE_CLIENT:
    break;
case LCO_TOPMOST_CHANGED:
    on_change_topmost (layer);
    break;
default:
    printf ("Serious error: incorrect operations.\n");
}
}

```

## 2. ServerStartup function

This function initializes the server, i.e. `mginit`. It creates the shared resource, the listening socket, the default layer, and other internal objects. Your customized `mginit` program should call this function before calling any other function. Note that the default layer created by the server called "`mginit`" (`NAME_DEF_LAYER`). The prototype of this function is:

```

BOOL GUIAPI ServerStartup (int nr_globals,

```

```
int def_nr_topmosts, int def_nr_normals);
```

You can pass some arguments to this function to control the limits of the window management:

- **nr\_globals** The number of the global z-order nodes. All z-order nodes created by `mginit` are global ones.
- **def\_nr\_topmosts** The maximal number of the topmost z-order nodes in the default layer. It is also the default number of topmost z-order nodes of a new layer.
- **def\_nr\_normals** The maximal number of normal z-order nodes in the new layer. It is also the default number of normal z-order nodes of a new layer.

### 17.1.2 Displaying Copyright Information

Then, `mginit` calls two functions to respectively display the copyright information of MiniGUI and MDE:

```
AboutMiniGUI ();
AboutMDE ();
```

### 17.1.3 Creating Taskbar

As mentioned above, `mginit` uses taskbar and buttons on the taskbar to represent the current layers of system. It also provides a simple user interface (see Fig. 17.1):

- If the user clicks an icon on the taskbar, a certain application program will be started.
- If the user clicks the buttons on the taskbar, it will switch the layer to be the top-most layer.

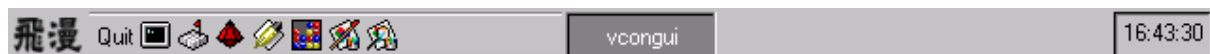


Fig. 17.1 Taskbar created by `mginit` of MDE

This taskbar uses the cool bar control in MiniGUIExt library to create a toolbar

used to startup applications. It reads application configuration information from `mginit.rc` file and initializes the information, including name, description string, and corresponding program icon of application.

Taskbar also creates a timer and a static control, which displays current time and refreshes once per second.

As this code is not the proprietary of `mginit`, so we will not discuss it in detail.

### 17.1.4 Startup the Default Program

`mginit.rc` file defines an initial application that needs to startup by default.

The following code startups this application program:

```
pid_desktop = exec_app (app_info.autostart);

if (pid_desktop == 0 || waitpid (pid_desktop, &status, WNOHANG) > 0) {
    fprintf (stderr, "Desktop already have terminated.\n");
    Usage ();
    return 1;
}
```

Then, `mginit` catches the signal `SIGCHLD`, which helps to avoid the zombie processes if the child exits while there is no process getting its exit state:

```
sig.sa_handler = child_wait;
sig.sa_flags = 0;
memset (&sig.sa_mask, 0, sizeof(sigset_t));
sigaction (SIGCHLD, &sig, NULL);
```

The function `exec_app` used to startup client application program is very simple. It calls `vfork` and `exec1` system calls to startup a client:

```
pid_t exec_app (int app)
{
    pid_t pid = 0;
    char buff [PATH_MAX + NAME_MAX + 1];

    if ((pid = vfork ()) > 0) {
        fprintf (stderr, "new child, pid: %d.\n", pid);
    }
    else if (pid == 0) {
        if (app_info.app_items [app].cdpath) {
            chdir (app_info.app_items [app].path);
        }
        strcpy (buff, app_info.app_items [app].path);
        strcat (buff, app_info.app_items [app].name);
    }
}
```

```
if (app_info.app_items [app].layer [0]) {
    execl (buff, app_info.app_items [app].name,
           "-layer", app_info.app_items [app].layer, NULL);
}
else {
    execl (buff, app_info.app_items [app].name, NULL);
}
perror ("execl");
_exit (1);
}
else {
    perror ("vfork");
}

return pid;
}
```

### 17.1.5 Entering Message Loop

Now, **mginit** program enters message loop:

```
while (GetMessage (&msg, hTaskBar)) {
    DispatchMessage (&msg);
}
```

When the taskbar exits, the message loop will be terminated. Finally, MiniGUI-Processes exits.

## 17.2 A Simple Mginit Program

**Mginit** of MDE is not too complex. It illustrates a basic design method of the MiniGUI-Processes server program. In this section, we will design a simpler **mginit** program, which initializes itself as the server of MiniGUI-Processes and startups **helloworld** program. This program also illustrates the use of the event hook. When the user presses the key **F1**, **F2**, **F3**, or **F4**, some other clients will be started up. If the user has not operated for a long time, **mginit** will startup a screen saver program. When the user closes all clients, **mginit** exits. List 17.1 lists the code of this **mginit** program; you can refer to file **mginit.c** and file **scrnsaver.c** in the program package **mg-sample** of this guide for the whole code.

List 17.1 The code of a simple **mginit** program

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <unistd.h>
#include <signal.h>
#include <time.h>
#include <sys/types.h>
#include <sys/wait.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>

static BOOL quit = FALSE;

static void on_new_del_client (int op, int cli)
{
    static int nr_clients = 0;

    if (op == LCO_NEW_CLIENT) {
        nr_clients ++;
    }
    else if (op == LCO_DEL_CLIENT) {
        nr_clients --;
        if (nr_clients == 0) {
            printf ("There is no any client, I will quit.\n");
            quit = TRUE;
        }
        else if (nr_clients < 0) {
            printf ("Serious error: nr_clients less than zero.\n");
        }
    }
    else
        printf ("Serious error: incorrect operations.\n");
}

static pid_t exec_app (const char* file_name, const char* app_name)
{
    pid_t pid = 0;

    if ((pid = vfork ()) > 0) {
        fprintf (stderr, "new child, pid: %d.\n", pid);
    }
    else if (pid == 0) {
        execl (file_name, app_name, NULL);
        perror ("execl");
        _exit (1);
    }
    else {
        perror ("vfork");
    }

    return pid;
}

static unsigned int old_tick_count;

static pid_t pid_scrnsaver = 0;

static int my_event_hook (PMSG msg)
{
    old_tick_count = GetTickCount ();

    if (pid_scrnsaver) {
        kill (pid_scrnsaver, SIGINT);
        ShowCursor (TRUE);
        pid_scrnsaver = 0;
    }

    if (msg->message == MSG_KEYDOWN) {
        switch (msg->wParam) {
            case SCANCODE_F1:
                exec_app (".\\edit", "edit");
                break;
            case SCANCODE_F2:
                exec_app (".\\timeeditor", "timeeditor");
        }
    }
}
```

```

        break;
    case SCANCODE_F3:
        exec_app ("./propsheet", "propsheet");
        break;
    case SCANCODE_F4:
        exec_app ("./bmpbkgn", "bmpbkgn");
        break;
    }
}

return HOOK_GOON;
}

static void child_wait (int sig)
{
    int pid;
    int status;

    while ((pid = waitpid (-1, &status, WNOHANG)) > 0) {
        if (WIFEXITED (status))
            printf ("--pid=%d--status=%x--rc=%d--\n", pid, status, WEXITSTATUS(status));
        else if (WIFSIGNALED(status))
            printf ("--pid=%d--signal=%d--\n", pid, WTERMSIG (status));
    }
}

int MiniGUIMain (int args, const char* arg[])
{
    MSG msg;
    struct sigaction siga;

    siga.sa_handler = child_wait;
    siga.sa_flags = 0;
    memset (&siga.sa_mask, 0, sizeof(sigset_t));
    sigaction (SIGCHLD, &siga, NULL);

    OnNewDelClient = on_new_del_client;

    if (!ServerStartup (0, 0, 0)) {
        fprintf (stderr, "Can not start the server of MiniGUI-Processes: mginit.\n");
        return 1;
    }

    SetServerEventHook (my_event_hook);

    if (args > 1) {
        if (exec_app (arg[1], arg[1]) == 0)
            return 3;
    }
    else {
        if (exec_app ("./helloworld", "helloworld") == 0)
            return 3;
    }

    old_tick_count = GetTickCount ();

    while (!quit && GetMessage (&msg, HWND_DESKTOP)) {
        if (pid_scrnsaver == 0 && GetTickCount () > old_tick_count + 1000) {
            ShowCursor (FALSE);
            pid_scrnsaver = exec_app ("./scrnsaver", "scrnsaver");
        }
        DispatchMessage (&msg);
    }

    return 0;
}

```

This program sets the `OnNewDelClient` event handling function of `mginit`. It will set the global variable `quit` as `TRUE` when the event handling function encounter `LCO_DEL_CLIENT`, so as to result in the stop of the message loop,

and finally exits the server.

You can compile the program by using the following command:

```
$ gcc -o mginit mginit.c -lminigui
```

As this mginit program need startup `helloworld` program when starting up, so, we have to make sure that there is `helloworld` program in the current directory.

Certainly, we can also add this program into the sample program package in this guide. As `mginit` program can only be compiled under MiniGUI-Processes environment, we need to modify the file `configure.in` and the file `Makefile.am` of the project `mg-samples` in order to add `mginit` into `mg-samples` project. First, we delete the notation of following line in file `configure.in`:

```
AM_CONDITIONAL(MGRM_THREADS, test "x$threads_version" = "xyes")
AM_CONDITIONAL(MGRM_PROCESSES, test "x$procs_version" = "xyes")
AM_CONDITIONAL(MGRM_STANDALONE, test "x$standalone_version" = "xyes")
```

The lines mean that if the MiniGUI is configured as MiniGUI-Processes, MiniGUI-Threads, or MiniGUI-Standalone, and then defines the macro `MGRM_PROCESSES`, `MGRM_THREADS`, or `MGRM_STANDALONE` respectively. These macros will be used in `Makefile.am`.

Then, we modify the file `Makefile.am` that is in directory `src/`:

```
if MGRM_PROCESSES
noinst_PROGRAMS=helloworld mycontrol dialogbox input bmpbkgnb simplekey \
                scrollbar painter capture bitblt stretchblt loadbmp drawicon \
                createicon caret demo cursordemo \
                scrnsaver mginit
else
noinst_PROGRAMS=helloworld mycontrol dialogbox input bmpbkgnb simplekey \
                scrollbar painter capture bitblt stretchblt loadbmp drawicon \
                createicon caret demo cursordemo
endif
...
mginit_SOURCES=mginit.c
scrnsaver_SOURCES=scrnsaver.c
```



The statements above request that `automake` generates the makefile rules used to create `mginit` and `scrnsaver` targets only if `MGRM_PROCESSES` defined. After modifying these scripts, you should run the script `./autogen.sh` to recreate configure file, then run the script `./configure` to create new makefiles for the `mg-samples` project. If MiniGUI is configured as MiniGUI-Threads, the command `make` will not compile `mginit` program and `scrnsaver` program.

### 17.3 Functions Specific to Clients of MiniGUI-Processes

As you know, a client of MiniGUI-Processes should call `JoinLayer` to join itself into a layer before calling other functions of MiniGUI. Besides `JoinLayer`, a client can call other functions to get the information of layers, delete a layer or change the topmost layer.

Clients should call `JoinLayer` before calling any other MiniGUI functions. The prototype of this function is as follows:

```
GHANDLE GUIAPI JoinLayer (const char* layer_name,
                          const char* client_name,
                          int max_nr_topmosts, int max_nr_normals);
```

If the layer to be joined does not exist, the server will try to create a new one. If you passed a NULL pointer or a null string for `layer_name`, the client will be tried to join to the default layer. If the client wants to create a new layer, you should specify the maximal number of topmost frame objects (`max_nr_topmosts`) and the maximal number of normal frame objects (`max_nr_normals`) in the new layer. Passing zero to `max_nr_topmosts` and `max_nr_normals` will use the default values. Note that `ServerStartup` specifies the default values.

You can get the information of a layer through the function `GetLayerInfo`. The prototype of this function is as follows:

```
GHANDLE GUIAPI GetLayerInfo (const char* layer_name,
                             int* nr_clients, BOOL* is_topmost, int* cli_active);
```

The information of the layer will be returned through the pointer arguments if the specific pointer is not NULL. The information are the number of clients in the layer, whether the layer is the topmost one, and the client identifier whose window is the current active one respectively.

Clients can call `SetTopmostLayer` to set the topmost layer and call `DeleteLayer` to delete a layer. Detail information about these functions can be referred to *MiniGUI API Reference*.

## 17.4 Other Functions and Interfaces Specific to Mginit

Apart from `ServerStartup`, `OnNewDelClient`, and `mgClients`, MiniGUI-Processes also defines several interfaces for `mginit` program. These interfaces are specific to the MiniGUI-Processes server programs. Detail information about these interfaces can be referred to *MiniGUI API Reference*.

- **ServerSetTopmostLayer**: This function switches the specified layer to be the topmost layer.
- **ServerCreateLayer**: This function creates a specified layer in the system.
- **ServerDeleteLayer**: This function deletes a specified layer from the system.
- **GetClientByPID**: This function returns the client identifier according to pid of a client.
- **SetTopmostClient**: This function sets the topmost layer by the specified client identifier. It will bring the layer contains the client to be the topmost one.
- **SetServerEventHook**: This function sets a hook of bottom event in `mginit`. When hook function return zero to MiniGUI, MiniGUI will continue the event process, and send the event to the current active client; whereas stops the process of the event.
- **send2Client**: The server can use this function to send the specific message to a certain client.

- **Send2TopMostClients:** The server can send a message to all clients in the topmost layer.
- **Send2ActiveWindow:** The server can send a message to the active window in a layer.
- **ServerGetNextZNode:** Get the next z-node in the specified layer from the server.
- **ServerGetZNodeInfo:** Get the z-node information in the specified layer from the server.
- **ServerDoZNodeOperation:** Does an operation on the z-node in the specified layer from the server.

## 18 GAL and IAL Engines

In the development of MiniGUI version 0.3.xx, we introduce the concepts of Graphics and Input Abstract Layer (GAL and IAL). Abstract layer is similar to Linux kernel virtual file system. It defines a group of abstract interfaces that do not rely on any special hardware. All top-layer graphics operations and input handling are based on abstract interfaces. The code used to realize the abstract interfaces are called graphic engine and input engine, similar to the driver of operating system. Actually it is an object-oriented programming technology. By using this abstract interface, we can easily port MiniGUI to other operating systems.

Generally speaking, Linux-based embedded system kernel will provide FrameBuffer support, thus existed FBCON graphics engine of MiniGUI can run on normal PC, and also can run on special embedded systems. Therefore, we usually do not need to develop a graphics engines for special embedded devices, but use FBCON graphics engine. At the same time, MiniGUI also offers Shadow and CommLCD GAL engines for different situations. There will be simple introduction in the first and second sections of this chapter.

Compared to graphics engine, it is more important to separate the bottom input handling and top input handling of MiniGUI. In Linux-based embedded systems, graphics engine can get from FrameBuffer, while input device does not have the uniformed interface. We usually use keyboard and mouse on PC, but we may only use touch screen and keypad on embedded systems. Considering this situation, we have to say that it is very important for MiniGUI to provide an abstract input layer.

Therefore, the following sections will introduce IAL interface of MiniGUI, and focus on how to develop an input engine aiming at special embedded systems. Although this topic is out of the general MiniGUI programming scope, we include related content in this chapter because of the importance of this topic.

## 18.1 Shadow NEWGAL Engine

The main functions of this engine are:

1. Support for asynchronous update of graphics device, such as YUV output, indirect access to FrameBuffer, etc.
2. Support for display modes of lower than 8-bpp on NEWGAL. At present, it can support all kinds of display modes of QVFB and all packed-pixel modes of FrameBuffer console.

Shadow engine uses the conception of sub-driver. MiniGUI uses the name of target board to determine which sub-driver to be included. Only one sub-driver can be included at one time, which is determined by the configuration option `--with-targetname`.

Some child drivers are implemented in the Shadow engine now, for example:

- `--with-targetname=vfanvil`: The sub-driver for VisualFone Anvil board. This sub-driver runs on ThreadX operation system.
- `--with-targetname=qvfb`: The sub-driver for all kinds of Linux QVFB display modes.
- `--with-targetname=wfcb`: The sub-driver for all kinds of Windows QVFB display modes.
- Without target (`__TARGET_UNKNOWN__`): The default sub-driver. This sub-driver operates similarly to the Dummy GAL engine. You can modify this sub-driver to implement operations on the underlayer graphics device.

**[Note] This engine can only be used under MiniGUI-Threads mode.**

## 18.2 CommLCD NEWGAL Engine

This engine provides the support for direct access to LCD FrameBuffer (video memory) on conditional real-time operation systems like VxWorks, Nucleus, uC/OS-II, and eCos. The LCD's pixel format should be above 8-bpp

(bits-per-pixel), and in packed-pixel mode.

CommLCD engine also uses the conception of sub-driver. The sub-drivers already implemented in CommLCD engine include:

- `--with-targetname= vxi386/vxppc (__TARGET_VXi386__ and __TARGET_VXPPC__)`: The sub-driver for VxWorks i386/PowerPc target.
- `--with-targetname=c33l05(__TARGET_C33L05__)`: The sub-driver for EPSON C33L05 target.
- `--with-targetname=mx21(__TARGET_MX21__)`: The sub-driver for OSE mx21 target.
- Without target (`__TARGET_UNKNOWN__`): The default sub-driver. If it is eCos operation system, MiniGUI will use the standard interfaces to implement the sub-driver. Otherwise, you should define the sub-driver by yourself. For more information, please see `src/newgal/commlcd/unknown.c`. There is a default implementation for uC/OS-II in `rtos/ucos2_startup.c` and it is similar to dummy graphics engine.

In every sub-drive, we should need implement the following interfaces:

```
int __commlcd_drv_init (void);
int __commlcd_drv_getinfo (struct commlcd_info *li);
int __commlcd_drv_release (void);
int __commlcd_drv_setclut (int firstcolor, int ncolors, GAL_Color *colors);
```

- `__commlcd_drv_init` function is used to initialize the LCD device.
- `__commlcd_drv_release` function is used to release the LCD device.
- `__commlcd_drv_getinfo` function is used to get information of the LCD device.
- `__commlcd_drv_setclut` is used to set color palette.

The structure `commlcd_info` is defined as follows:

```
struct commlcd_info {
    short height, width; // Pixels
    short bpp;           // Depth (bits-per-pixel)
    short type;          // Pixel type
    short rlen;          // Length of one raster line in bytes
    void *fb;            // Address of the frame buffer
};
```

The LCD drivers or applications must implement the four functions while porting MiniGUI to a new hardware platform. As follows:

```
.....
struct commlcd_info {
    short height, width; // Pixels
    short bpp;           // Depth (bits-per-pixel)
    short type;          // Pixel type
    short rlen;          // Length of one raster line in bytes
    void *fb;            // Address of the frame buffer
};

int __commlcd_drv_init (void)
{
    if (uglInitialize() == UGL_STATUS_ERROR)
        return 1;

    return 0;
}

int __commlcd_drv_getinfo (struct commlcd_info *li)
{
    UGL_MODE_INFO modeInfo;

    /* Obtain display device identifier */
    devId = (UGL_DEVICE_ID) uglRegistryFind (UGL_DISPLAY_TYPE,
        0, 0, 0)->id;

    /* Create a graphics context */
    gc = uglGcCreate (devId);

    .....
    uglPcBiosInfo (devId, UGL_MODE_INFO_REQ, &modeInfo);
    li->type = COMMLCD_TRUE_RGB565;

    li->height = modeInfo.height;
    li->width = modeInfo.width;
    li->fb = modeInfo.fbAddress;
    li->bpp = modeInfo.colorDepth;
    li->rlen = (li->bpp*li->width + 7) / 8;
    return 0;
}

int __commlcd_drv_release (void)
{
    return 0;
}

int __commlcd_drv_setclut (int firstcolor, int ncolors, GAL_Color *colors)
{
    return 0;
}
.....
```

## 18.3 The IAL Interface of MiniGUI

MiniGUI uses **INPUT** structure to present input engine. See List 18.1.

List 18.1 The input engine structure of MiniGUI (`src/include/ial.h`)

```
typedef struct tagINPUT
{
    char* id;
```

```
// Initialization and termination
BOOL (*init_input) (struct tagINPUT *input, const char* mdev, const char* mtype);
void (*term_input) (void);

// Mouse operations
int (*update_mouse) (void);
int (*get_mouse_xy) (int* x, int* y);
void (*set_mouse_xy) (int x, int y);
int (*get_mouse_button) (void);
void (*set_mouse_range) (int minx, int miny, int maxx, int maxy);

// Keyboard operations
int (*update_keyboard) (void);
char* (*get_keyboard_state) (void);
void (*suspend_keyboard) (void);
void (*resume_keyboard) (void);
void (*set_leds) (unsigned int leds);

// Event
#ifdef _LITE_VERSION
int (*wait_event) (int which, int maxfd, fd_set *in, fd_set *out, fd_set *except,
    struct timeval *timeout);
#else
int (*wait_event) (int which, fd_set *in, fd_set *out, fd_set *except,
    struct timeval *timeout);
#endif

char mdev [MAX_PATH + 1];
} INPUT;

extern INPUT* cur_input;
```

In order to facilitate program writing, we also define the following C language macros:

```
#define IAL_InitInput      (*cur_input->init_input)
#define IAL_TermInput      (*cur_input->term_input)
#define IAL_UpdateMouse    (*cur_input->update_mouse)
#define IAL_GetMouseXY     (*cur_input->get_mouse_xy)
#define IAL_SetMouseXY     (*cur_input->set_mouse_xy)
#define IAL_GetMouseButton (*cur_input->get_mouse_button)
#define IAL_SetMouseRange  (*cur_input->set_mouse_range)

#define IAL_UpdateKeyboard (*cur_input->update_keyboard)
#define IAL_GetKeyboardState (*cur_input->get_keyboard_state)
#define IAL_SuspendKeyboard (*cur_input->suspend_keyboard)
#define IAL_ResumeKeyboard  (*cur_input->resume_keyboard)
#define IAL_SetLeds(leds)   if (cur_input->set_leds) (*cur_input->set_leds) (leds)

#define IAL_WaitEvent      (*cur_input->wait_event)
```

We define all supported input engines in `src/ial/ial.c`:

```
#define LEN_ENGINE_NAME    16
#define LEN_MTYPE_NAME     16

static INPUT inputs [] =
{
#ifdef _SVGALIB
    {"SVGA Lib", InitSVGA LibInput, TermSVGA LibInput},
#endif
#ifdef _LIBGGI
    {"LibGGI", InitLibGGIInput, TermLibGGIInput},
#endif
#ifdef _EP7211_IAL
```



```

    {"EP7211", InitEP7211Input, TermEP7211Input},
#endif
#ifdef _ADS_IAL
    {"ADS", InitADSInput, TermADSInput},
#endif
#ifdef _IPAQ_IAL
    {"iPAQ", InitIPAQInput, TermIPAQInput},
#endif
#ifdef _VR4181_IAL
    {"VR4181", InitVR4181Input, TermVR4181Input},
#endif
#ifdef _HELIO_IAL
    {"Helio", InitHelioInput, TermHelioInput},
#endif
#ifdef _NATIVE_IAL_ENGINE
    {"Console", InitNativeInput, TermNativeInput},
#endif
#ifdef _TFSTB_IAL
    {"TF-STB", InitTFSTBInput, TermTFSTBInput},
#endif
#ifdef _T800_IAL
    {"T800", InitT800Input, TermT800Input},
#endif
#ifdef _DUMMY_IAL
    {"Dummy", InitDummyInput, TermDummyInput},
#endif
#ifdef _QVFB_IAL
    {"QVFB", InitQVFBInput, TermQVFBInput},
#endif
};

INPUT* cur_input;

```

Each input engine is indicated by an **INPUT** structure. All input engines form a structure array called **inputs**. Each input engine defines three members of **INPUT** structure when being initialized:

- **id**: The engine name, used as the identifier of the engine.
- **init\_input**: Initializing function of the input engine. This function is responsible for assigning other members of the **INPUT** structure.
- **term\_input**: Cleanup function of the input engine.

In initializing stage of MiniGUI, system will search special input engine in **inputs** array to be the current input engine, and then call the initializing function of this engine. If successful, system will assign the global variable **cur\_input** to be this engine:

```

int InitIAL (void)
{
    int i;
    char buff [LEN_ENGINE_NAME + 1];
    char mdev [MAX_PATH + 1];
    char mtype[LEN_MTYPE_NAME + 1];

    if (GetValueFromEtcFile (ETCFILEPATH, "system", "ial_engine",
                           buff, LEN_ENGINE_NAME) < 0 )
        return ERR_CONFIG_FILE;

    if (GetValueFromEtcFile (ETCFILEPATH, "system", "mdev",

```

```

        mdev, MAX_PATH) < 0 )
    return ERR_CONFIG_FILE;

    if (GetValueFromEtcFile (ETCFILEPATH, "system", "mtype",
        mtype, LEN_MTYPE_NAME) < 0 )
        return ERR_CONFIG_FILE;

    for (i = 0; i < NR_INPUTS; i++) {
        if (strncasecmp (buff, inputs[i].id, LEN_ENGINE_NAME) == 0) {
            cur_input = inputs + i;
            break;
        }
    }

    if (cur_input == NULL) {
        fprintf (stderr, "IAL: Does not find matched engine.\n");
        return ERR_NO_MATCH;
    }

    strcpy (cur_input->mdev, mdev);

    if (!IAL_InitInput (cur_input, mdev, mtype)) {
        fprintf (stderr, "IAL: Init IAL engine failure.\n");
        return ERR_INPUT_ENGINE;
    }

#ifdef _DEBUG
    fprintf (stderr, "IAL: Use %s engine.\n", cur_input->id);
#endif

    return 0;
}

```

When we need to write input engine for a special embedded device, we should first add three members of this engine into `inputs` structure array, then assign other members in its initializing function. Basically these members are function pointer, called by the upper-layer of MiniGUI and get state and data of bottom-layer input device.

<code>update_mouse</code>	inform bottom-layer engine to update new mouse information
<code>get_mouse_xy</code>	calling this function by upper-layer to get x and y coordinates of mouse
<code>set_mouse_xy</code>	calling function by upper-layer to set the new mouse position. For those engines not supporting this function, the member can be null.
<code>get_mouse_button</code>	get mouse button state. Return value can be <code>IAL_MOUSE_LEFTBUTTON</code> , <code>IAL_MOUSE_MIDDLEBUTTON</code> , <code>IAL_MOUSE_RIGHTBUTTON</code> to respectively represent the pressed states: mouse left key, middle key, and right key
<code>set_mouse_range</code>	set range of mouse movement. For engines not supporting this function, can be set as <code>NULL</code> .
<code>update_keyboard</code>	inform bottom-layer to update keyboard information
<code>get_keyboard_state</code>	get keyboard state, return a byte array, including keyboard pressed state indexed by scan code. Pressed is 1, released is 0.
<code>suspend_keyboard</code>	pause keyboard device read/write, used for switch of virtual console Usually set as <code>NULL</code> for embedded device.
<code>resume_keyboard</code>	resume keyboard device read/write, used for switch of virtual console. Usually set as <code>NULL</code> for embedded device.
<code>set_leds</code>	set keyboard status LEDs, used for CapLock, NumLock,

	and ScrollLock statues.
wait_event	calling this function by upper-layer to wait for an event from input devices. This function has different interfaces for MiniGUI-Threads and MiniGUI-Processes, and must implement with select or poll system calls.

After understanding IAL interface and the data structure, we now need to see how an actual engine is written.

## 18.4 Comm Input Engine

MiniGUI provides comm IAL for conditional real-time operation systems like VxWorks, Nucleus, uC/OS-II, and eCos. Based on this engine, you can easily add the support for input device such as keyboard, mouse, and touch screen.

The comm ial engine needs the OS or low-level device driver to provide five functions as follows:

```
int __comminput_init (void);
void __comminput_deinit (void);
int __comminput_ts_getdata (short *x, short *y, short *button);
int __comminput_kb_getdata (short *key, short *status);
int __comminput_wait_for_input (void);
```

- **\_\_comminput\_init** is used to initialize the input device.
- **\_\_comminput\_deinit** is used to release the input device.
- **\_\_comminput\_ts\_getdata** get the input data of the touch screen. The "x" and "y" returns the position data, and "button" returns the pressed state (if pen is pressed, return a non-zero value).
- **\_\_comminput\_ts\_getdata** returns 0 while getting data successfully, otherwise returns -1.
- **\_\_comminput\_kb\_getdata** gets the input data of the keyboard. "key" returns the key code of corresponding key; "status" returns the key status (1 for key-down, 0 for key-up). **\_\_comminput\_kb\_getdata** returns 0 while getting data successfully, otherwise returns -1. The key code here is the MiniGUI-defined scan code of a keyboard key. The low-level keyboard driver needs to translate a keyboard scan code to a corresponding MiniGUI key code and return this key code.
- **\_\_comminput\_wait\_for\_input** enquires whether there are input data. If no input events comes, this function returns 0; if mouse events or touch screen events comes, the return value's first position is set to 1;

if keyboard events comes, the return value's second position is set to 1.

The control of the external events in MiniGUI is implemented as a single system thread. This event thread sleeps while no external events come. So, `__comminput_wait_for_input` should provide a waiting mechanism, such as using semaphore. `__comminput_wait_for_input` waits on an input semaphore while enquiring about input data, and makes the MiniGUI input task which calls this function go to sleep. When input events come, the low-level drivers (or interrupt routines) should post a semaphore to wake up the MiniGUI event task.

When migrating MiniGUI to new hardware, we need to implement the above five functions interface according to OS or hardware driver.

## 18.5 Developing IAL Engine for a Specific Embedded Device

Actually developing a new IAL engine is not difficult. We use iPAQ as an example to illustrate the design of a customized input engine.

iPAQ produced by COMPAQ is a StrongARM-based high-end hand-held product, which includes touch screen and several control keys. The touch screen is similar to the mouse of PC, but it can only differentiate left button. For the control keys, we can emulate them as some keys in PC keyboard, such as cursor keys, **ENTER** key, and function keys. The source code of this engine is showed in List 18.2.

List 18.2 The customized input engine for iPAQ (`src/ial/ipaq.c`)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/poll.h>
#include <linux/kd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
#include "common.h"
```

```

#ifdef IPAO IAL
#include "misc.h"
#include "ial.h"
#include "ipaq.h"
#include "linux/h3600_ts.h"

/* for data reading from /dev/hs3600_ts */
typedef struct {
    unsigned short b;
    unsigned short x;
    unsigned short y;
    unsigned short pad;
} POS;

static unsigned char state [NR_KEYS];
static int ts = -1;
static int btn_fd = -1;
static unsigned char btn_state=0;
static int mousex = 0;
static int mousey = 0;
static POS pos;

#undef _DEBUG

/***** Low Level Input Operations *****/
/*
 * Mouse operations -- Event
 */
static int mouse_update(void)
{
    return 1;
}

static void mouse_getxy(int *x, int* y)
{
#ifdef _DEBUG
    printf ("mousex = %d, mousey = %d\n", mousex, mousey);
#endif

    if (mousex < 0) mousex = 0;
    if (mousey < 0) mousey = 0;
    if (mousex > 319) mousex = 319;
    if (mousey > 239) mousey = 239;

#ifdef _COORD_TRANS
#ifdef _ROT_DIR_CCW
        *x = mousey;
        *y = 319 - mousex;
#else
        *x = 239 - mousey;
        *y = mousex;
#endif
#else
        *x = mousex;
        *y = mousey;
#endif
}

static int mouse_getbutton(void)
{
    return pos.b;
}

static int keyboard_update(void)
{
    char *statinfo;
    int status;
    int key;

    //Attention!
    statinfo = (btn_state & KEY_RELEASED)? "UP":"DOWN";
    status = (btn_state & KEY_RELEASED)? 0 : 1;

    key = btn_state & KEY_NUM;
#endif _DEBUG

```

```

    fprintf(stderr, "kev %d is %s", kev, statinfo);
#endif
    switch (key)
    {
    case 1:
        //state[H3600_SCANCODE_RECORD] = status;
        state[SCANCODE_LEFTSHIFT] = status;
        break;
    case 2:
        state[H3600_SCANCODE_CALENDAR] = status;
        break;
    case 3:
        state[H3600_SCANCODE_CONTACTS] = status;
        break;
    case 4:
        state[H3600_SCANCODE_Q] = status;
        break;
    case 5:
        state[H3600_SCANCODE_START] = status;
        break;
    case 6:
        state[H3600_SCANCODE_UP] = status;
        break;
    case 7:
        state[H3600_SCANCODE_RIGHT] = status;
        break;
    case 8:
        state[H3600_SCANCODE_LEFT] = status;
        break;
    case 9:
        state[H3600_SCANCODE_DOWN] = status;
        break;
    case 10:
        state[H3600_SCANCODE_ACTION] = status;
        break;
    case 11:
        state[H3600_SCANCODE_SUSPEND] = status;
        break;
    }

    return NR_KEYS;
}

static const char* keyboard_getstate(void)
{
    return (char *)state;
}

#ifdef _LITE_VERSION
static int wait_event (int which, int maxfd, fd_set *in, fd_set *out, fd_set *except,
                      struct timeval *timeout)
#else
static int wait_event (int which, fd_set *in, fd_set *out, fd_set *except,
                      struct timeval *timeout)
#endif
{
    fd_set rfds;
    int retvalue = 0;
    int e;

    if (!in) {
        in = &rfds;
        FD_ZERO (in);
    }

    if ((which & IAL_MOUSEEVENT) && ts >= 0) {
        FD_SET (ts, in);
#ifdef _LITE_VERSION
        if (ts > maxfd) maxfd = ts;
#endif
    }
    if ((which & IAL_KEYEVENT) && btn_fd >= 0) {
        FD_SET (btn_fd, in);
#ifdef _LITE_VERSION
        if (btn_fd > maxfd) maxfd = btn_fd;
#endif
    }

```

```

#endif
}

#ifdef _LITE_VERSION
    e = select (maxfd + 1, in, out, except, timeout) ;
#else
    e = select (FD_SETSIZE, in, out, except, timeout) ;
#endif

    if (e > 0) {
        if (ts >= 0 && FD_ISSET (ts, in))
        {
            FD_CLR (ts, in);
            pos.x=0;
            pos.y=0;
            // FIXME: maybe failed due to the struct alignment.
            read (ts, &pos, sizeof (POS));
            //if (pos.x != -1 && pos.y != -1) {
            if (pos.b > 0) {
                mousex = pos.x;
                mousey = pos.y;
            }
            //}
#ifdef _DEBUG
            if (pos.b > 0) {
                printf ("mouse down: pos.x = %d, pos.y = %d\n", pos.x, pos.y);
            }
#endif
            pos.b = ( pos.b > 0 ? 4:0);
            retvalue |= IAL_MOUSEEVENT;
        }

        if (btn_fd >= 0 && FD_ISSET(btn_fd, in))
        {
            unsigned char key;
            FD_CLR(btn_fd, in);
            read(btn_fd, &key, sizeof(key));
            btn_state = key;
            retvalue |= IAL_KEYEVENT;
        }

        } else if (e < 0) {
            return -1;
        }

        return retvalue;
    }

BOOL InitIPAQInput (INPUT* input, const char* mdev, const char* mtype)
{
    ts = open ("/dev/h3600_ts", O_RDONLY);
    if (ts < 0) {
        fprintf (stderr, "IPAQ: Can not open touch screen!\n");
        return FALSE;
    }

    btn_fd = open ("/dev/h3600_key", O_RDONLY);
    if (btn_fd < 0 ) {
        fprintf (stderr, "IPAQ: Can not open button key!\n");
        return FALSE;
    }

    input->update_mouse = mouse_update;
    input->get_mouse_xy = mouse_getxy;
    input->set_mouse_xy = NULL;
    input->get_mouse_button = mouse_getbutton;
    input->set_mouse_range = NULL;

    input->update_keyboard = keyboard_update;
    input->get_keyboard_state = keyboard_getstate;
    input->set_leds = NULL;

    input->wait_event = wait_event;

    mousex = 0;
    mousey = 0;

```

```

    pos.x = pos.v = pos.b = 0;

    return TRUE;
}

void TermIPAQInput (void)
{
    if (ts >= 0)
        close(ts);
    if (btn_fd >= 0)
        close(btn_fd);
}

#endif /* _IPAQ_IAL */

```

We now analyze how some important interface functions implement:

- The function `InitIPAQInput` is the initializing function of iPAQ input engine defined in `src/ial/ipaq.c`. This function opens two devices: `/dev/h3600_ts` and `/dev/h3600_key`. The former is the device file for touch screen; the latter is the device for control keys. They are similar to device `/dev/psaux` and device `/dev/tty` on PCs. After successfully opening these two device files, the function sets `INPUT` structure and other members, some of which is assigned with `NULL`.
- The function `mouse_update` returns 1, indicating that the mouse state is ready.
- The function `mouse_getxy` returns mouse position data prepared by other functions and performs proper boundary examination.
- The function `mouse_getbutton` returns touch screen state, that is, whether the user have touched the screen. It is similar to whether the user has pressed the left button of mouse.
- The function `keyboard_update` properly fills `state` array according to the keyboard information prepared by other functions.
- The function `keyboard_state` directly returns the address of `state` array.
- The function `wait_event` is the core function of the input engine. This function first combines the two opened device file descriptors within the descriptor set `in`. Then it calls `select` system call. When the value returned by `select` is more than 0, this function will examine if there is any readable data waiting to read in the two file descriptors, if so, it will read touch screen and key stroke data separately from the two file descriptors.



Obviously, the design of input engine of iPAQ is not complicated; the code lines are not much too. It is believed that you can refer to this input engine to develop a new input engine for your special embedded device. After developing your new input engine, we should not forget to add an entry into our input engine array (the `inputs` structure array in `src/ial.c` file). It is needed to properly modify `MiniGUI.cfg` file in order to specify MiniGUI to use your input engine as well as.

## **IV MiniGUI Control Programming**

- Static Control
- Button Control
- List Box Control
- Edit Box Control
- Combo Box Control
- Menu Button Control
- Progress Bar Control
- Track Bar Control
- Toolbar Control
- Property Sheet Control
- Scroll Window Control
- Scroll View Control
- Tree View Control
- List View Control
- Month Calendar Control
- Spin Box Control
- Cool Bar Control
- Animation Control



## 19 Static Control

A static control is used to display information, such as text and digits, in the specified position of a window, and can also be used to display some static image information, such as corporation logos, product brands, etc. As suggested by its name, the behavior of a static box cannot respond dynamically to the user input. The existence of the control is basically for displaying some information, and does not receive any input from the keyboard or the mouse. Fig 19.1 shows the typical use of the static control: to be the label of other controls in a dialog box.

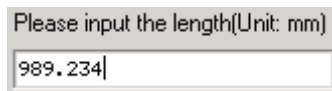


Fig. 19.1 Typical use of the static control

You can create a static control by calling `CreateWindow` function with `CTRL_STATIC` as the control class name.

### 19.1 Types and Styles of Static Control

Styles of a static control comprise the type of the static control and some flags. We can divide the static controls into standard type (display text only), bitmap type (display icon or bitmap), and special type of group box. We will introduce the different types of static control in the following sections.

#### 19.1.1 Standard Type

By setting the style of a static control to be one of `SS_SIMPLE`, `SS_LEFT`, `SS_CENTER`, `SS_RIGHT`, and `SS_LEFTNOWORDWRAP`, you can create a static control displaying only text. The displayed content thereof is specified in the `caption` argument of `CreateWindow` function and can be changed by calling `SetWindowText`.

Static controls with `SS_LEFT`, `SS_CENTER`, or `SS_RIGHT` style can be used to display multiple-line text, and align the text in the control left, center, and right respectively.

The following program creates static controls of the several types described above:

The appearance of the controls described above is shown in Fig. 19.2. For showing the effect clearly, all these static controls have borders.

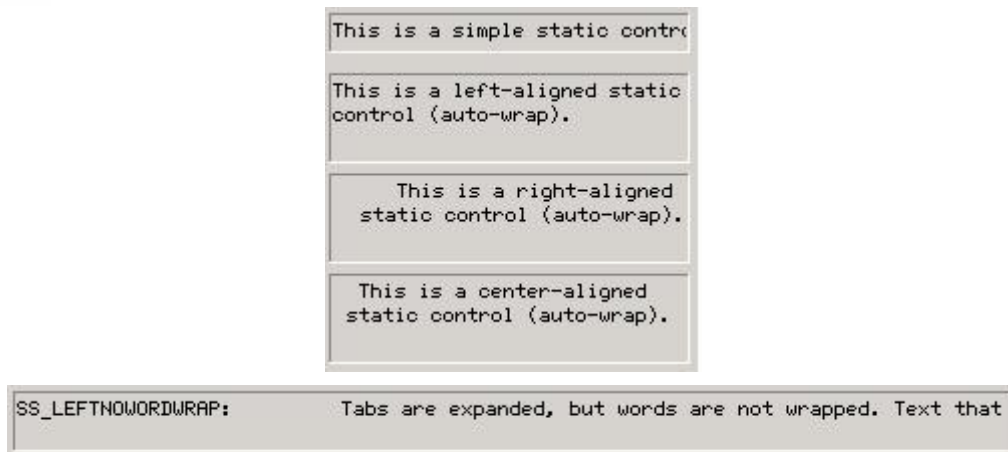


Fig. 19.2 Static controls of standard type

### 19.1.2 Bitmap Type

A static control with `SS_BITMAP` or `SS_ICON` style displays a bitmap or an icon in the control. For these two types of static control, it is needed to set the handle of an icon object or the pointer to a bitmap object to be displayed through `dwAddData` parameter when the static control is created. Styles associated with these two types of static control are `SS_CENTERIMAGE` and `SS_REALSIZEIMAGE`. These two styles are used to control the position of a bitmap or an icon in the control. By default, the bitmap or icon fills the whole static control by appropriate scale, but this scaling operation will be canceled when `SS_REALSIZEIMAGE` style is used, and the bitmap or icon aligns in the top-left of the static control. If `SS_REALSIZEIMAGE` style is used with `SS_CENTERIMAGE` style simultaneously, the bitmap or icon will be displayed in the center of the control.

The following program creates a bitmap static control and an icon static control, and creates a bitmap static control displayed in the center of the control by using `SS_REALSIZEIMAGE` and `SS_CENTERIMAGE` styles:

```
CreateWindow (CTRL_STATIC,
    "",
    WS_CHILD | SS_BITMAP | WS_VISIBLE,
    IDC_STATIC,
    280, 80, 50, 50, hWnd, (DWORD)GetSystemBitmap (SYSBMP_CHECKMARK));

CreateWindow (CTRL_STATIC,
    "",
    WS_CHILD | SS_ICON | WS_VISIBLE,
    IDC_STATIC,
    280, 20, 50, 50, hWnd,
```

```
(DWORD)GetLargeSystemIcon (IDI_INFORMATION));

CreateWindow (CTRL_STATIC,
             "",
             WS_CHILD | SS_BITMAP | SS_REALSIZEIMAGE | SS_CENTERIMAGE | WS_VISIBLE,
             IDC_STATIC,
             280, 140, 50, 50, hWnd, (DWORD)GetSystemBitmap (SYSBMP_CHECKMARK));
```

**[Note] Many predefined controls need you pass some initial parameters of them through dwAddData parameter of CreateWindowEx function. Here the first additional data of the window is used to pass these parameters during creating the control, and the application can also use the first additional data of the window to save private data after the control has been created.**

The appearance of the static controls created by the above program is shown in Fig. 19.3.



Fig. 19.3 Static control of bitmap type

### 19.1.3 Group Box

A static control with `SS_GROUPBOX` style is called a group box, which is a special type of static control. A group box is a rectangle with the caption displayed in the top, and it is usually used to include other controls. The controls that can be created in the group box include Static, Button, Edit Box, TextEdit, List Box, Track Bar, Menu Button.

The following program creates a group box, and its effect is shown in Fig. 19.4:

```
CreateWindow (CTRL_STATIC,
             "A Group Box",
```

```
WS_CHILD | SS_GROUPBOX | WS_VISIBLE,  
IDC_STATIC,  
350, 10, 200, 100, hWnd, 0);
```



Fig. 19.4 A group box

### 19.1.4 Other static control types

Besides the static controls described above, there are several other static control types not frequently used:

- **SS\_WHITERECT**: A rectangle filled with the light white color.
- **SS\_GRAYRECT**: A rectangle filled with the light gray color.
- **SS\_BLACKRECT**: A rectangle filled with the black color.
- **SS\_GRAYFRAME**: A frame drawn with the light gray color.
- **SS\_WHITEFRAME**: A frame drawn with the light white color.
- **SS\_BLACKFRAME**: A frame drawn with the black color.

Appearances of the static controls using the above styles are shown in Fig. 19.5.



Fig. 19.5 Other static control types

## 19.2 Static Control Messages

When the static control is of bitmap type, the bitmap can be gotten or changed by the following messages:

- **STM\_GETIMAGE**: This message returns the pointer to the bitmap object or the handle to the icon.



- **STM\_SETIMAGE**: To set the pointer to the bitmap or the handle to the icon through **wParam** parameter, and returns the old one.

### 19.3 Notification Codes of Static Control

When the style of a static control includes **SS\_NOTIFY**, the static control can generate the following two notification messages:

- **STN\_CLICKED**: Indicates the user clicked the left button of the mouse in a static control.
- **STN\_DBLCLK**: Indicates the user double-clicked the left button of the mouse in a static control.

### 19.4 Sample Program

The program in List 19.1 creates a bitmap static control, and changes the text when the user double-clicks the static control. Please refer to **static.c** of the demo program package **mg-samples** of this guide to get the complete source code of the program. The running effect of the program is shown in Fig. 19.6.

List 19.1 Sample program of static control

```
#include <stdio.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

static void my_notif_proc (HWND hwnd, int id, int nc, DWORD add_data)
{
    /* When the user double-clicked the static control,
     * SetWindowText is called to change the text of the control.
     */
    if (nc == STN_DBLCLK)
        SetWindowText (hwnd, "I am double-clicked. :)");
}

static int StaticDemoWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    HWND hwnd;

    switch (message) {
        case MSG_CREATE:
            /* Create a static box and set the notification callback function */
            hwnd = CreateWindow (CTRL_STATIC, "Double-click me!",
                                WS_VISIBLE | SS_CENTER | SS_NOTIFY,
                                50, 80, 100, 200, 20, hWnd, 0);
            SetNotificationCallback (hwnd, my_notif_proc);
            return 0;
    }
}
```

```

    case MSG_DESTROY:
        DestroyAllControls (hWnd);
        return 0;

    case MSG_CLOSE:
        DestroyMainWindow (hWnd);
        PostQuitMessage (hWnd);
        return 0;
}

return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

/* The Following code to create the main window are omitted */

```

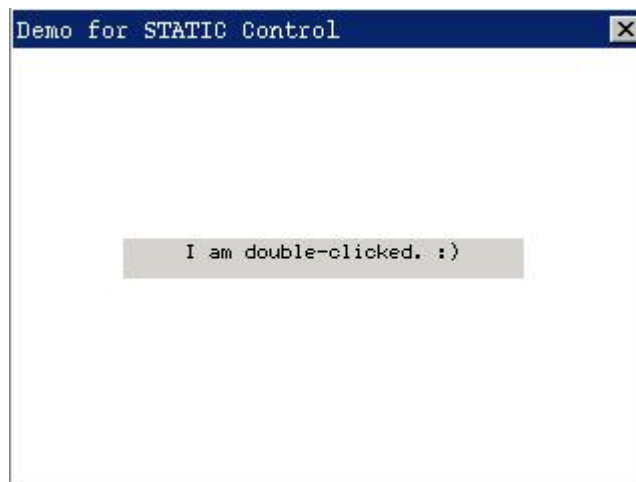


Fig. 19.6 The sample of static control



## 20 Button Control

Button control is the most frequently used control besides the static control. A button is usually used to provide switch selection for the user. The buttons of MiniGUI can be classified into push button, check box, radio button, etc. The user can select or switch the status of a button with the keyboard or the mouse. The user's input will make a button generating notification messages, and an application can also send messages to a button to change the status of it.

Calling `CreateWindow` function with `CTRL_BUTTON` as the control class name can create a button control.

### 20.1 Types and Styles of Button

#### 20.1.1 Push button

A push button is a rectangle, in which the window caption passed through `CreateWindow` is displayed. The rectangle occupies the whole height and width given when calling `CreateWindow` and the text locates in the center of the rectangle.

A push button control is mainly used to trigger an action, which responds immediately, and does not keep the switch information for long. Such a button has two window styles, namely `BS_PUSHBUTTON` and `BS_DEFPUSHBUTTON`. "DEF" in `BS_DEFPUSHBUTTON` means "default". `BS_PUSHBUTTON` and `BS_DEFPUSHBUTTON` have different effects when used in a dialog box. No matter which control the input focus locates on, the button with `BS_DEFPUSHBUTTON` style will take `ENTER` key input as the default input. However, the buttons of these two styles have the same function when used as controls in a normal main window, except that the button with `BS_DEFPUSHBUTTON` has a thicker border.

When the mouse cursor is in the button, pressing the left button of the mouse

will make the button redraw itself with three-dimensional shadow, which looks like being pushed down really. When the mouse button is released, the button recovers to its original appearance, and sends a `MSG_COMMAND` message with `BN_CLICKED` notification code to its parent window. When the button has the input focus, there are dashed lines around the text and pressing or releasing the `space` key or the left mouse button results in the same effect.

**[Prompt] The description of the control behavior and appearance in this guide takes the default classic style as the standard.**

Generally, text on the push button would display in the central of the vertical and horizontal directions with single-line form, and will not be wrapped automatically. An application also can display multiple lines text by specifying `BS_MULTILINE` style.

The following program creates two push buttons:

```

CreateWindow (CTRL_BUTTON,
    "Push Button",
    WS_CHILD | BS_PUSHBUTTON | BS_CHECKED | WS_VISIBLE,
    IDC_BUTTON,
    10, 10, 80, 30, hWnd, 0);

CreateWindow (CTRL_BUTTON,
    "Multiple Lines Push Button",
    WS_CHILD | BS_PUSHBUTTON | BS_MULTILINE | WS_VISIBLE,
    IDC_BUTTON + 1,
    100, 10, 80, 40, hWnd, 0);

```

The appearance of the two push buttons created by the program above is as shown in Fig. 20.1. Note that the text is aligning up when `BS_MULTILINE` style is used.



Fig. 20.1 Push buttons

In addition, bitmaps or icons also can also be displayed on the push button. In the case `BS_BITMAP` or `BS_ICON` style is used, the pointer to a bitmap object or the handle to an icon should be passed through `dwAddData` argument of

**CreateWindow** function. The bitmap or icon will be scaled to fill the whole button window in default; however, they are displayed in the center of the control without any scaling when **BS\_REALSIZEIMAGE** style is used. The following code fragment creates a button with a bitmap, and its effect is as shown in Fig. 20.2.

```
hWnd = CreateWindow (CTRL_BUTTON, "Close",
    WS_CHILD | BS_PUSHBUTTON | BS_BITMAP | BS_REALSIZEIMAGE | BS_NOTIFY | WS_VISIBLE,
    IDC_BUTTON + 4,
    10, 300, 60, 30, hWnd, (DWORD) GetSystemBitmap (IDI_APPLICATION));
```



Fig. 20.2 Bitmap push button

### 20.1.2 Check Box

A check box is a text block, and the text is usually on the right side of a check mark (if you specify **BS\_LEFTTEXT** style when creating the button, the text will be on the left). A check box is usually used in the application, which allows the user to make a selection among options. The commonly used function of a check box is as a switch: click once shows the checked mark, and click again clears the mark.

The two most frequently used styles for check box are **BS\_CHECKBOX** and **BS\_AUTOCHECKBOX**. When **BS\_CHECKBOX** is used, the application need send messages to the control to set the checked mark; and when **BS\_AUTOCHECKBOX** style is used, the control will switch the status between checked and unchecked.

The other two styles of check box are **BS\_3STATE** and **BS\_AUTO3STATE**. As hinted by their names, these two styles can show the third state - the color is gray within the check box, indicating the check box cannot be selected or used. The difference between **BS\_3STATE** and **BS\_AUTO3STATE** is the same as the above: the former need the application operates its state, while the latter lets the control to be in charge of the automatic state switch.

A check box is left aligned in the rectangle by default, and locates between the top and the bottom of the control window (centered vertically). Clicking the mouse button in any position in the rectangle will generate a notification message. Using `BS_LEFTTEXT` style will right-align the check box, and place the text in the left of the check mark. Styles for justifying text, such as `BS_LEFT`, `BS_CENTER`, `BS_RIGHT`, `BS_TOP`, `BS_VCENTER`, `BS_BOTTOM`, etc. all can be used for the check box.

In addition, using `BS_PUSHLIKE` style will make a check box be displayed as a push button: in pushed state when checked and normal state when unchecked.

The following program creates two check boxes, and the effect is shown in Fig. 20.3.

```

CreateWindow (CTRL_BUTTON,
              "Auto 3-state check box",
              WS_CHILD | BS_AUTO3STATE | WS_VISIBLE,
              IDC_CHECKBOX,
              10, 60, 150, 30, hWnd, 0);

CreateWindow (CTRL_BUTTON,
              "Auto check box on left",
              WS_CHILD | BS_AUTOCHECKBOX | BS_LEFTTEXT | BS_RIGHT | WS_VISIBLE,
              IDC_CHECKBOX + 1,
              170, 60, 150, 30, hWnd, 0);

```



Fig. 20.3 Check boxes

### 20.1.3 Radio Button

A radio button is just like the channel selection buttons on a radio. Each button corresponds to a channel, and each time only one button can be selected. In a dialog box, the radio button group is usually used for representing mutually exclusive options. A radio button is different from a check box, for the work manner is not as a switch; that is to say, when pushing a radio button again, its status will not change.

The shape of a radio button is a circle, not a rectangle. Except the shape difference, the behavior of a radio button is very like a check box. The

enhanced dot in the circle means the radio button is already selected. The radio button has two styles, namely `BS_RADIOBUTTON` and `BS_AUTORADIOBUTTON`. The latter will show the selection state of the user automatically, while the former will not.

By default, a radio button is left justified in the control window, and locates in the middle between the top and the bottom of the control window (centered vertically). Pressing the mouse in any position in the rectangle will generate a notification message. Using `BS_LEFTTEXT` style will right-justify the combo box, and place the text in the left of the radio button. Styles for justifying text, such as `BS_LEFT`, `BS_CENTER`, `BS_RIGHT`, `BS_TOP`, `BS_VCENTER`, `BS_BOTTOM`, etc., can be used for the radio button.

In addition, using `BS_PUSHLIKE` style will make a radio button be displayed as a push button: in pushed state when selected and normal state when unselected.

The following program creates two radio buttons, and the effect is shown in Fig. 20.4.

```
CreateWindow (CTRL_BUTTON,
    "Auto Radio Button 2",
    WS_CHILD | BS_AUTORADIOBUTTON | WS_VISIBLE,
    IDC_RADIOBUTTON + 1,
    20, 160, 130, 30, hWnd, 0);

CreateWindow (CTRL_BUTTON,
    "Auto Radio Button 2",
    WS_CHILD | BS_AUTORADIOBUTTON | BS_LEFTTEXT | BS_RIGHT | WS_VISIBLE,
    IDC_RADIOBUTTON + 4,
    180, 160, 140, 30, hWnd, 0);
```



Fig. 20.4 Radio buttons

Radio buttons are generally used in a group, and only one button can be selected among the radio buttons in the same group. When creating a group of radio buttons, we need set their status to be mutually exclusive, so `WS_GROUP` style needs to be used when creating the first radio button in order to set it as the "leader button" of the group.



## 20.2 Messages of Button

The application can do the following works by sending messages to a button:

- To get/set the check state of a radio button or check box: **BM\_GETCHECK**, **BM\_SETCHECK**.
- To get/set the pushed or released state of a push button or a check box: **BM\_GETSTATE**, **BM\_SETSTATE**.
- To get/set the bitmap or icon on the bitmap button: **BM\_GETIMAGE**, **BM\_SETIMAGE**.
- Sending **BM\_CLICK** to simulate clicking operation of the mouse button.

The application sends **BM\_SETCHECK** message with **wParam** equal to be **BST\_CHECKED** to a check box or radio box to make it to be the checked state:

```
SendMessage (hwndButton, BM_SETCHECK, BST_CHECKED, 0);
```

In fact **wParam** can be one of the three possible values shown in Table 20.1. These values are also the checked state value returned through **BM\_GETCHECK** message.

Table 20.1 States of check box or radio button

State identifier	Meaning
<b>BST_UNCHECKED</b> (0)	Indicates the button is unchecked
<b>BST_CHECKED</b> (1)	Indicates the button is checked
<b>BST_INDETERMINATE</b> (2)	Indicates the button is grayed because the state of the button is indeterminate

We can simulate the button blinking by sending **BM\_SETSTATE** message to the control. The following operation will cause the button to be pushed:

```
SendMessage (hwndButton, BM_SETSTATE, BST_PUSHED, 0);
```

The following operation will cause the button to be unpushed:

```
SendMessage (hwndButton, BM_SETSTATE, 0, 0);
```

For a bitmap button, **BM\_GETIMAGE** or **BM\_SETIMAGE** message can be used to get

or set the bitmap object or handle of the icon:

```
int image_type;
PBITMAP btn_bitmap;
HICON btn_icon;

int ret_val = SendMessage (hwndButton, BM_GETIMAGE, (WPARAM)&image_type, 0) ;

if (image_type == BM_IMAGE_BITMAP) {
    /* This button uses a bitmap object */
    btn_bitmap = (PBITMAP) ret_val;
}
else {
    /* This button uses an icon object */
    btn_icon = (HICON) ret_val;
}

/* Set the button image to be a bitmap object */
SendMessage (hwndButton, BM_SETIMAGE, BM_IMAGE_BITMAP, btn_bitmap) ;

/* Set the button image to be an icon object */
SendMessage (hwndButton, BM_SETIMAGE, BM_IMAGE_ICON, btn_icon) ;
```

In addition, in an application, we can simulate the click operation of the user by sending **BM\_CLICK** message to a button.

## 20.3 Notification Codes of Button

The notification codes generated by a button with **BS\_NOTIFY** style are as follow:

- **BN\_CLICKED**: Indicate the user clicked the button. The value of this notification code is zero, so if you want to handle **BN\_CLICKED** notification message sent by the button in the parent window, you need only determine whether **wParam** parameter of **MSG\_COMMAND** message equals to the button identifier. The generation of this notification is default and will ignore **BS\_NOTIFY** style of the button control.
- **BN\_PUSHED**: Indicate the user pushed the button.
- **BN\_UNPUSHED**: Indicate the user released the button.
- **BN\_DBLCLK**: Indicate the user double-clicked a button.
- **BN\_SETFOCUS**: Indicate the button received the keyboard focus.
- **BN\_KILLFOCUS**: Indicate a button lost the keyboard focus.

## 20.4 Sample Program

Generally, to get the click notification code of a push button, the application need only handle **BN\_CLICKED** notification code. Check boxes and radio buttons are usually set to be the automatic state, and send **BM\_GETCHECK** to get the checked state when necessary. For dialog boxes, the application can also get the state information of a button control quickly with the functions listed in Table 20.2

Table 20.2 Convenient functions handling button controls

Function	Purpose	Note
<b>CheckDlgButton</b>	Changes the check status of a button control by its identifier	
<b>CheckRadioButton</b>	Adds a check mark to (checks) a specified radio button in a group and removes a check mark from (clears) all other radio buttons in the group	Ensure to check a button mutually exclusively
<b>IsDlgButtonChecked</b>	Determines whether a button control has a check mark next to it or whether a three-state button control is grayed, checked, or neither	

The program in List 20.1 gives a comprehensive example for using button controls. This program uses a dialog box to ask the user about his taste, selects the snack type by grouped radio buttons, and then selects some special tastes for the user by check boxes. Please refer to `button.c` file in the demo program package `mg-samples` of this guide to get the full source code of this program. The running effect of this program is shown in Fig. 20.5.

List 20.1 Example for using button controls

```
#include <stdio.h>
#include <stdlib.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

#define IDC_LAMIAN          101
#define IDC_CHOUDOUFU      102
#define IDC_JIANBING        103
#define IDC_MAHUA          104
#define IDC_SHUIJIAO       105

#define IDC_XIAN            110
#define IDC_LA              111

#define IDC_PROMPT         200
```

```
static DLGTEMPLATE DlgYourTaste =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    0, 0, 370, 280,
    "What flavor snack do you like?",
    0, 0,
    12, NULL,
    0
};

static CTRLDATA CtrlYourTaste[] =
{
    {
        "static",
        WS_VISIBLE | SS_GROUPBOX,
        16, 10, 130, 160,
        IDC_STATIC,
        "optional snack",
        0
    },
    {
        "button",
        /* Using the style BS_CHECKED, make the button checked initially. */
        WS_VISIBLE | BS_AUTORADIOBUTTON | BS_CHECKED | WS_TABSTOP | WS_GROUP,
        36, 38, 88, 20,
        IDC_LAMIAN,
        "Northwest pulled noodle",
        0
    },
    {
        "button",
        WS_VISIBLE | BS_AUTORADIOBUTTON,
        36, 64, 88, 20,
        IDC_CHOUDOUFU,
        "Changsha bad-smelling bean curd",
        0
    },
    {
        "button",
        WS_VISIBLE | BS_AUTORADIOBUTTON,
        36, 90, 88, 20,
        IDC_JIANBING,
        "Shandong thin pancake",
        0
    },
    {
        "button",
        WS_VISIBLE | BS_AUTORADIOBUTTON,
        36, 116, 88, 20,
        IDC_MAHUA,
        "Tianjin fired dough twist",
        0
    },
    {
        "button",
        WS_VISIBLE | BS_AUTORADIOBUTTON,
        36, 142, 100, 20,
        IDC_SHUIJIAO,
        "Chengdu red oil boiled dumpling",
        0
    },
    {
        "static",
        WS_VISIBLE | SS_GROUPBOX | WS_GROUP,
        160, 10, 124, 160,
        IDC_STATIC,
        "Flavor",
        0
    },
    {
        "button",
        WS_VISIBLE | BS_AUTOCHECKBOX,
        170, 38, 88, 20,
    }
}
```

```

        IDC_XIAN,
        "Partial salty",
        0
    },
    {
        "button",
        /* Using the style BS_CHECKED, make the button checked initially. */
        WS_VISIBLE | BS_AUTOCHECKBOX | BS_CHECKED,
        170, 64, 88, 20,
        IDC_LA,
        "Partial spicy",
        0
    },
    {
        "static",
        WS_VISIBLE | SS_LEFT | WS_GROUP,
        16, 180, 360, 20,
        IDC_PROMPT,
        "Northwest pulled noodle is competitive product in the wheaten food",
        0
    },
    {
        "button",
        WS_VISIBLE | BS_DEFPUSHBUTTON | WS_TABSTOP | WS_GROUP,
        80, 220, 95, 28,
        IDOK,
        "OK",
        0
    },
    {
        "button",
        WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP,
        185, 220, 95, 28,
        IDCANCEL,
        "Cancel",
        0
    },
    },
};

static char* prompts [] = {
    "Northwest pulled noodle is competitive product in the wheaten food",
    "Changsha bad-smelling bean curd is very unique",
    "Shandong thin pancake is difficult to chew",
    "Tianjin fired dough twist is very fragile",
    "Chengdu red oil boiled dumpling is captivating",
};

static void my_notif_proc (HWND hwnd, int id, int nc, DWORD add_data)
{
    /* When the user selects an different snack,
     * prompt information is displayed for the snack in a static control
     */
    if (nc == BN_CLICKED) {
        SetWindowText (GetDlgItem (GetParent (hwnd), IDC_PROMPT), prompts [id - IDC_LAMIA
N]);
    }
}

static int DialogBoxProc2 (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_INITDIALOG:
        {
            int i;
            /* Set the notification callback function for the radio button of the snack */
            for (i = IDC_LAMIAN; i <= IDC_SHUIJIAO; i++)
                SetNotificationCallback (GetDlgItem (hDlg, i), my_notif_proc);
        }
        return 1;

        case MSG_COMMAND:
            switch (wParam) {
                case IDOK:
                case IDCANCEL:
                    EndDialog (hDlg, wParam);
            }
    }
}

```

```

        break;
    }
    break;

}

return DefaultDialogProc (hDlg, message, wParam, lParam);
}

int MiniGUIMain (int argc, const char* argv[])
{
#ifdef MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER, "button", 0, 0);
#endif

    DlgYourTaste.controls = CtrlYourTaste;

    DialogBoxIndirectParam (&DlgYourTaste, HWND_DESKTOP, DialogBoxProc2, 0L);

    return 0;
}

#ifndef _LITE_VERSION
#include <minigui/dti.c>
#endif

```

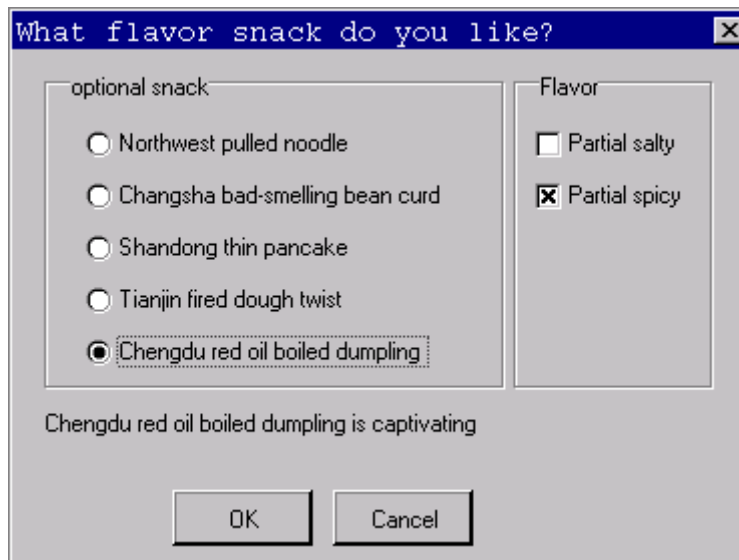


Fig. 20.5 Example for using button controls



## 21 List Box Control

A list box generally provides a series of options, which are shown in a scroll window. The user can select one or more items with the keyboard or mouse operation. The selected items are usually highlighted. The most typical use of list box is the open file dialog box, as shown in Fig. 21.1.

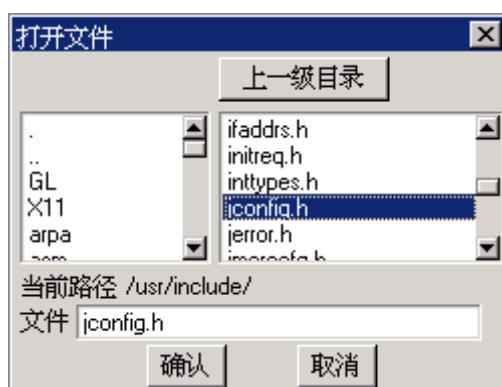


Fig. 21.1 Typical use of list box: “Open File” dialog box

We can create a list box control by calling `CreateWindow` function with `CTRL_LISTBOX` as the control class name.

### 21.1 Types and Styles of List Box

List box controls of MiniGUI can be divided into three types: single-selection list box, multiple-selection list box and bitmap list box. A list box is single-selection style by default; namely, the user can only select one item. To create a multiple-selection list box, you should use `LBS_MULTIPLESEL` style. When using this style, the user can select an item by clicking this item, and cancel the selection by clicking it again. When the list box has the input focus, you can also use the space key to select or cancel selection of an item. The effect of running a multiple-selection list box is as shown in Fig. 21.2.



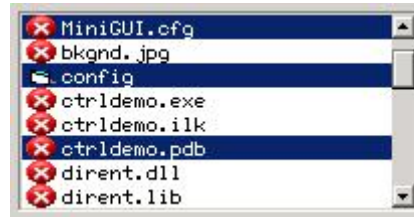


Fig. 21.2 Multiple-selection list box

Besides the two list box types described above, MiniGUI also provides an advanced list box type. In this type of list box, the list item can be not only text string, but also attached by a bitmap or an icon. With such a list box, we can also display a check box besides the list item to represent checked or uncheck status. To create such an advanced list box, `LBS_USEICON` or `LBS_CHECKBOX` style should be specified. Fig 21.3 shows the running effect of the advanced list box. If you want the selection state auto-switched when the user clicks the check box, you can use `LBS_AUTOCHECK` style. The advanced list box can also have `LBS_MULTIPLESEL` style.

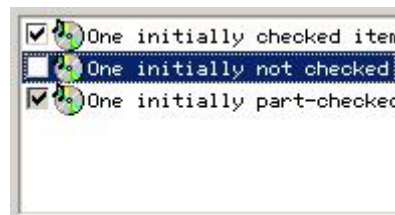


Fig. 21.3 Advanced list box control

Except the styles described above to distinguish the type of list box, you can also specify other general styles when creating a list box.

By default, the message handler of a list box only shows the list items, with no border around them. You can add border by using the window style `WS_BORDER`. In addition, you can also add a vertical scroll bar with window style `WS_VSCROLL` to scroll the list items by the mouse and add a horizontal scroll bar with window style `WS_HSCROLL`.

The default list box styles does not generate notification message when the user select a list item, so the program must send a message to the list box to get the selection state of item. Therefore, a list box control generally includes

**LBS\_NOTIFY** style, which can feed back some state information in time to the application during the user's operation.

In addition, if you want the list box control sort the list items, you can use another commonly used style **LBS\_SORT**.

Generally, the most commonly used style combination for creating list box is as follows:

```
(LBS_NOTIFY | LBS_SORT | WS_VSCROLL | WS_BORDER)
```

## 21.2 Messages of List Box

### 21.2.1 Adding Item into List Box

After a list box is created, the next step is to add text strings to it. You can realize this by sending messages to the window message handler of the list box by calling **SendMessage**. The items in a list box can be referred to by an index value; the top-most item has index value of zero. In the following example, **hwndList** is the handle of the list box control, and **index** is the index value. When **SendMessage** is used to pass the text string, **lParam** is the pointer to the NULL-terminated string.

When the stored content of the list box exceeds the available memory space, **SendMessage** returns **LB\_ERRSPACE**. If error is caused by other reasons, **SendMessage** returns **LB\_ERR**. If the operation is successful, **SendMessage** returns **LB\_OKAY**. We can determine above two errors by testing the non-zero value of **SendMessage**.

If you adopt **LBS\_SORT** style, or just want to append the new text string as the last item of a list box, the simplest approach to append string into the list box is using **LB\_ADDSTRING** message:

```
SendMessage (hwndList, LB_ADDSTRING, 0, (LPARAM)string) ;
```

We can also use **LB\_INSERTSTRING** to specify an index, and insert a text string to the specified position of the list box.

```
SendMessage (hwndList, LB_INSERTSTRING, index, (LPARAM)string) ;
```

For example, if index value is 4, string would be a text string with index 4: the fifth string counting from the beginning (due to zero-based), and all the text strings after this position will move backward. When the index is -1, the string will append to the last position. We can also use **LB\_INSERTSTRING** for a list box with **LBS\_SORT** style, but at this time the list box will ignore the index, and insert the new item according to the sorted result.

It should be noted that, after specifying **LBS\_CHECKBOX** or **LBS\_USEICON** style, when you add an item to a list box, you must use **LISTBOXITEMINFO** structure, and cannot use the string address directly, for example:

```
HICON hIcon1;          /* Declare an icon handle */
LISTBOXITEMINFO lbii;  /* Declare a structure variable of the list box item*/

hIcon1 = LoadIconFromFile (HDC_SCREEN, "res/audio.ico", 1); /* Load an icon */

/* Set the structure information, and add an item */
lbii.hIcon = hIcon1;
lbii.cmFlag = CMFLAG_CHECKED;
lbii.string = "abcdefg";
SendMessage (hChildWnd3, LB_ADDSTRING, 0, (LPARAM)&lbii);
```

Here, the value of **cmFlag** can be **CMFLAG\_CHECKED**, **CMFLAG\_BLANK**, or **CMFLAG\_PARTCHECKED**, indicating checked, unchecked, and partly checked, respectively.

We can also display bitmaps in an advanced list box, rather than icons by default. If you want a list box item display a bitmap instead of an icon, you can include **IMGFLAG\_BITMAP** in the flag, and specify the pointer to the bitmap object:

```
/* Set structure information, and add an item */
lbii.hIcon = (DWORD) GetSystemBitmap (SYSBMP_MAXIMIZE);
lbii.cmFlag = CMFLAG_CHECKED | IMGFLAG_BITMAP;
lbii.string = "abcdefg";
SendMessage (hChildWnd3, LB_ADDSTRING, 0, (LPARAM)&lbii);
```

### 21.2.2 Deleting Item from List Box

Send **LB\_DELETESTRING** message and specify the index value, and then you can delete an item with the index from the list box:

```
SendMessage (hwndList, LB_DELETESTRING, index, 0) ;
```

We can even use **LB\_RESETCONTENT** message to clear all contents in the list box:

```
SendMessage (hwndList, LB_RESETCONTENT, 0, 0) ;
```

### 21.2.3 Selecting and Getting Item

Send **LB\_GETCOUNT** to get the number of items in the list box:

```
count = SendMessage (hwndList, LB_GETCOUNT, 0, 0) ;
```

When you need to get the text string of a certain item, you can send **LB\_GETTEXTLEN** message to get the length of the string of the specified item in a list box:

```
length = SendMessage (hwndList, LB_GETTEXTLEN, index, 0) ;
```

And then, copy the item to a text buffer by sending the message **MSG\_GETTEXT**:

```
length = SendMessage (hwndList, LB_GETTEXT, index, (LPARAM)buffer) ;
```

In these two conditions, the length value returned by above messages is the length of the text. For the length of a NULL-terminate string, the buffer must be big enough. You can use the string length returned by **LB\_GETTEXTLEN** message to allocate some local memory for storing the string.

If we need to set the string of list item, you can send **LB\_SETTEXT** message:

```
SendMessage (hwndList, LB_SETTEXT, index, buffer) ;
```

For an advanced list box, we must use **LB\_GETITEMDATA** and **LB\_SETITEMDATA** to get other information of a list item, such as the bitmap object or the handle to icon, state of check box, and these messages also can be used to get or set the text string of item:

```
HICON hIcon1;          /* Declare an icon handle */
LISTBOXITEMINFO lbii;  /* Declare a struct variable of the list box item info */

hIcon1 = LoadIconFromFile (HDC_SCREEN, "res/audio.ico", 1); /* Load an icon */

/* Set the structure information, and set an item */
lbii.hIcon = hIcon1;
lbii.cmFlag = CMFLAG_CHECKED;
lbii.string = "new item";
SendMessage (hChildWnd3, LB_SETITEMDATA, index, (LPARAM)&lbii);
```

The following messages are used to retrieve the selection state of list items; these messages have different calling method for single-selection list box and multiple-selection list box. Let us to look at the single-selection list box first.

Generally, the user selects an item by mouse and keyboard. But we also can control the current selected item by program, at this time, we need send **LB\_SETCURSEL** message:

```
SendMessage (hwndList, LB_SETCURSEL, index, 0) ;
```

In contrast, we can use **LB\_GETCURSEL** to get the current selected item:

```
index = SendMessage (hwndList, LB_GETCURSEL, 0, 0) ;
```

If no item is selected, then the message returns **LB\_ERR**.

For multiple-selection list box, **LB\_SETCURSEL** and **LB\_GETCURSEL** can only be used to set or get the current highlighted item, but cannot get all selected items. But we can use **LB\_SETSEL** to set the selection state of a certain item without affecting other items:

```
SendMessage (hwndList, LB_SETSEL, wParam, (LPARAM)index) ;
```

If **wParam** is not 0, this function selects and highlights an item; if **wParam** is 0,

this function cancels the selection. In contrast, we can use `LB_GETSEL` to get the selection state of a certain item:

```
select = SendMessage (hwndList, LB_GETSEL, index, 0) ;
```

Here, if the item specified by `index` is selected, `select` is a non-zero value, else `select` is 0.

In addition, you can also use `LB_GETSELCOUNT` message to get the number of all selected items in a multiple-selection list box. Then you can send `LB_GETSELITEMS` message to get the index values of all the selected items. The following is an example:

```
int i, sel_count;
int* sel_items;

sel_count = SendMessage (hwndList, LB_GETSELCOUNT, 0, 0L) ;
if (sel_count == 0)
    return;

sel_items = alloca (sizeof(int)*sel_count);
SendMessage (hwndList, LB_GETSELITEMS, sel_count, sel_items);
for (i = 0; i < sel_count; i++) {
    /* sel_items [i] is the index of one selected item */
}
```

#### 21.2.4 Searching Item Including a Text String

```
index = SendMessage (hwndList, LB_FINDSTRING, (LPARAM)string) ;
```

Here, `string` is the pointer to a string, which should be found; the message returns the index of the fuzzy matched string, and `LB_ERR` means failure. Using `LB_FINDSTRINGEXACT` message will search the matched item exactly.

#### 21.2.5 Setting/Getting the Status of Check Mark

```
status = SendMessage (hwndList, LB_GETCHECKMARK, index, 0) ;
```

The message `LB_GETCHECKMARK` returns the check mark status of the check box specified by `index`. If corresponding item is not found, `LB_ERR` returned.

`CMFLAG_CHECKED` indicates the check box of the item is checked.

`CMFLAG_PARTCHECKED` indicates the check box of the item is partly checked.

**CMFLAG\_BLANK** indicates the check box of the item is not checked.

```
ret = SendMessage (hwndList, LB_SETCHECKMARK, index, (LPARAM)status) ;
```

The message **LB\_SETCHECKMARK** sets the check mark status of the check box specified by **index** to be the value of **status**. If the item specified by the **index** is not found, it returns **LB\_ERR** for failure, else returns **LB\_OKAY** for success.

### 21.2.6 Setting the Bold Status of Item

```
ret = SendMessage (hwndList, LB_SETITEMBOLD, index, (LPARAM)status) ;
```

The message **LB\_SETITEMBOLD** sets the bold status of the item specified by **index** to be the value of **status** (TRUE or FALSE). If the item specified by the **index** is not found, it returns **LB\_ERR** for failure.

### 21.2.7 Setting/Getting the Disable Status of Item

```
status = SendMessage (hwndList, LB_GETITEMDISABLE, index, 0) ;
```

The message **LB\_GETITEMDISABLE** returns the disable status of the item specified by **index**. If corresponding item is not found, **LB\_ERR** returned. 1 indicates the item is disabled. 0 indicates the item is not disabled.

```
ret = SendMessage (hwndList, LB_SETITEMDISABLE, index, (LPARAM)status) ;
```

The message **LB\_SETITEMDISABLE** sets the disable status of the item specified by **index** to be the value of **status**. If the item specified by the **index** is not found, it returns **LB\_ERR** for failure.

### 21.2.8 Adding Multiple Items into List Box

The message **LB\_MULTIADDITEM** is used to adding multiple items into List Box. When the stored content of the list box exceeds the available memory space, **SendMessage** returns **LB\_ERRSPACE**. If error is caused by other reasons, **SendMessage** returns **LB\_ERR**. If the operation is successful, **SendMessage** returns **LB\_OKAY**. We can determine above two errors by testing the non-zero

value of `SendMessage`.

If you adopt `LBS_SORT` style, or just want to append the new text string array as the last items of a list box, sample is as follows:

```
int num = 2;
const char text[num][ ] = {"item1", "item2"};

SendMessage (hwndList, LB_MULTIADDITEM, num, (LPARAM)text);
```

Parameter `hwndList` is the handle of the list box control; `num` is the number of item added, and `text` is the string text array address.

It should be noted that, after specifying `LBS_CHECKBOX` or `LBS_USEICON` style, when you add multiple items to a list box, you must use `LISTBOXITEMINFO` structure, and cannot use the string array address directly, for example:

```
int num = 2;
const char text[num][ ] = {"item1", "item2"};

SendMessage (hwndList, LB_MULTIADDITEM, num, (LPARAM)text);

int num = 2;
HICON hIcon1;
LISTBOXITEMINFO lbii[num];

hIcon1 = LoadIconFromFile (HDC_SCREEN, "res/audio.ico", 1);

lbii[0].hIcon = hIcon1;
lbii[0].cmFlag = CMFLAG_CHECKED;
lbii[0].string = "item1";

lbii[1].hIcon = hIcon1;
lbii[1].cmFlag = CMFLAG_CHECKED;
lbii[1].string = "item2";

SendMessage (hwndList, LB_MULTIADDITEM, num, (LPARAM)lbii);
```

### 21.2.9 Other Messages

A list box with style of `LBS_SORT` uses the standard C function `strcmp` to sort items. But we can overload the default sort method by using `LB_SETSTRCMPFUNC` in order to sort items according to the expected method. For example:

```
static int my_strcmp (const char* s1, const char* s2, size_t n)
{
    int i1 = atoi (s1);
```



```

int i2 = atoi (s2);
return (i1 - i2);
}

SendMessage (hwndList, LB_SETSTRCMPFUNC, 0, (LPARAM)my_strcmp);

```

Thus, the list box will use the user-defined function to sort items. Sort function described above can be used to sort the items in form as 1, 2, 3, 4, 10, 20 etc. according to integer values, while the default sort rule would sort the items above as 1, 10, 2, 20, 3, and 4. Generally speaking, application should use the message to set a new string comparison function before adding any item.

We also can associate an additional 32-bit data with each list item, and get the value at appropriate time. For doing this, we can use **LB\_SETITEMADDDATA** and **LB\_GETITEMADDDATA** messages. The values operated by the two messages have no meaning for the list box control. The control only takes charge to store the value and return the value when needed.

In addition, we can also use **LB\_SETITEMHEIGHT** message to set the height of items, and use **LB\_GETITEMHEIGHT** to get the height. Generally, height of items depends on the size of control font, and varies when the control font changes (call **SetWindowFont** to change). The users can also set themselves height of items. The actual height will be the maximum of the height set and control font size.

## 21.3 Notification Codes of List Box

Notification codes generated by a list box with **LBS\_NOTIFY** style and their meanings are shown in Table 21.1.

Table 21.1 Notification codes of list box

Notification code identifier	Meaning
<b>LBN_ERRSPACE</b>	Indicates that memory allocation failure.
<b>LBN_SELCHANGE</b>	Indicates the current selected item changes
<b>LBN_CLICKED</b>	Indicates click on an item
<b>LBN_DBLCLK</b>	Indicates double click on an item
<b>LBN_SELCANCEL</b>	Indicates cancel of the selection
<b>LBN_SETFOCUS</b>	Indicates gain of input focus
<b>LBN_KILLFOCUS</b>	Indicates loss of input focus

<b>LBN_CLICKCHECKMARK</b>	Indicates click on the check mark
<b>LBN_ENTER</b>	Indicates the user has pressed the ENTER key

A list box control will not send notification messages described above unless the window style of the list box have **LBS\_NOTIFY** style. Certainly, if you have called **SetNotificationCallback** function to set the notification callback function, the control will not send **MSG\_COMMAND** notification message to its parent window, but call the specified notification callback function directly.

**LBN\_ERRSPACE** indicates that the memory allocation fails. **LBN\_SELCHANGE** indicates the currently selected item has been changed, which occurs in the following cases: the user changes the highlighted item with the keyboard or mouse, or the user switches the selection status with the **space** key or mouse. **LBN\_CLICKED** indicates the mouse, which occurs when the user uses the mouse to click an item, clicks the list box. **LBN\_DBLCLK** indicates an item is double clicked by the user if **LBS\_CHECKBOX** style is set, **LBN\_CLICKCHECKMARK** indicates the user clicked the check mark, and if **LBS\_AUTOCHECK** style is set at the same time, the check box will be auto-switched between the checked or unchecked status.

## 21.4 Sample Program

The program in List 21.1 provides an example for the use of list box controls. The program imitates the "Open File" dialog box to realize the file deleting function. Initially, the program lists all the files in the current directory, and the user can also change to other directories through the directory list box. The user can select multiple files to be deleted in the file list box by select the check marks. When the user pushes the "Delete" button, the program will prompt the user. Of course, to protect the user's files, the program does not delete the files really. The effect of the dialog box created by this program is shown in Fig. 21.4. Please refer to **listbox.c** file of the demo program package of this guide for complete source code.

List 21.1 The use of list box controls

```
#include <stdio.h>
```

```

#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <pwd.h>
#include <errno.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

#define IDL_DIR      100
#define IDL_FILE     110
#define IDC_PATH     120

/* Define dialog box template */
static DLGTEMPLATE DlgDelFiles =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    100, 100, 304, 225,
    "Deleting the files",
    0, 0,
    7, NULL,
    0
};

static CTRLDATA CtrlDelFiles[] =
{
    {
        CTRL_STATIC,
        WS_VISIBLE | SS_SIMPLE,
        10, 10, 130, 15,
        IDC_STATIC,
        "Directories:",
        0
    },
    /* This list box displays directories */
    {
        CTRL_LISTBOX,
        WS_VISIBLE | WS_VSCROLL | WS_BORDER | LBS_SORT | LBS_NOTIFY,
        10, 30, 130, 100,
        IDL_DIR,
        "",
        0
    },
    {
        CTRL_STATIC,
        WS_VISIBLE | SS_SIMPLE,
        150, 10, 130, 15,
        IDC_STATIC,
        "Files:",
        0
    },
    /* This list box displays files with a check box in front for each file */
    {
        CTRL_LISTBOX,
        WS_VISIBLE | WS_VSCROLL | WS_BORDER | LBS_SORT | LBS_AUTOCHECKBOX,
        150, 30, 130, 100,
        IDL_FILE,
        "",
        0
    },
    /* This static control is used to display the current path information */
    {
        CTRL_STATIC,
        WS_VISIBLE | SS_SIMPLE,
        10, 150, 290, 15,
        IDC_PATH,
        "Path: ",
    }
}

```

```

0
},
{
    "button",
    WS_VISIBLE | BS_DEFPUSHBUTTON | WS_TABSTOP | WS_GROUP,
    10, 170, 130, 25,
    IDOK,
    "Delete",
    0
},
{
    "button",
    WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP,
    150, 170, 130, 25,
    IDCANCEL,
    "Cancel",
    0
},
},
};

/* This function gets all the directory items in the current directory,
 * and adds them to the directory list box and file list box, respectively
 */
static void fill_boxes (HWND hDlg, const char* path)
{
    struct dirent* dir_ent;
    DIR* dir;
    struct stat ftype;
    char fullpath [PATH_MAX + 1];

    SendDlgItemMessage (hDlg, IDL_DIR, LB_RESETCONTENT, 0, (LPARAM)0);
    SendDlgItemMessage (hDlg, IDL_FILE, LB_RESETCONTENT, 0, (LPARAM)0);
    SetWindowText (GetDlgItem (hDlg, IDC_PATH), path);

    if ((dir = opendir (path)) == NULL)
        return;

    while ( (dir_ent = readdir ( dir )) != NULL ) {

        /* Assemble full path name. */
        strncpy (fullpath, path, PATH_MAX);
        strcat (fullpath, "/");
        strcat (fullpath, dir_ent->d_name);

        if (stat (fullpath, &ftype) < 0 ) {
            continue;
        }

        if (S_ISDIR (ftype.st_mode))
            SendDlgItemMessage (hDlg, IDL_DIR, LB_ADDSTRING, 0, (LPARAM)dir_ent->d_name);
        else if (S_ISREG (ftype.st_mode)) {
            /* When using the list box of a checkbox,
             * the following structure need to be used */
            LISTBOXITEMINFO lbii;

            lbii.string = dir_ent->d_name;
            lbii.cmFlag = CMFLAG_BLANK;
            lbii.hIcon = 0;
            SendDlgItemMessage (hDlg, IDL_FILE, LB_ADDSTRING, 0, (LPARAM)&lbii);
        }
    }

    closedir (dir);
}

static void dir_notif_proc (HWND hwnd, int id, int nc, DWORD add_data)
{
    /* When the user double clicked the directory name or
     * pressed the ENTER key, he will enter the corresponding directory */
    if (nc == LBN_DBLCLK || nc == LBN_ENTER) {
        int cur_sel = SendMessage (hwnd, LB_GETCURSEL, 0, 0L);
        if (cur_sel >= 0) {
            char cwd [MAX_PATH + 1];
            char dir [MAX_NAME + 1];
            GetWindowText (GetDlgItem (GetParent (hwnd), IDC_PATH), cwd, MAX_PATH);

```

```

        SendMessage (hwnd, LB_GETTEXT, cur_sel, (LPARAM)dir);

        if (strcmp (dir, ".") == 0)
            return;
        strcat (cwd, "/");
        strcat (cwd, dir);
        /* Fill the two list boxes again */
        fill_boxes (GetParent (hwnd), cwd);
    }
}

static void file_notif_proc (HWND hwnd, int id, int nc, DWORD add_data)
{
    /* Do nothing */
}

static void prompt (HWND hDlg)
{
    int i;
    char files [1024] = "The files followed will be deleted\n";

    /* Get all the checked files */
    for (i = 0; i < SendDlgItemMessage (hDlg, IDL_FILE, LB_GETCOUNT, 0, 0L); i++) {
        char file [MAX_NAME + 1];
        int status = SendDlgItemMessage (hDlg, IDL_FILE, LB_GETCHECKMARK, i, 0);
        if (status == CMFLAG_CHECKED) {
            SendDlgItemMessage (hDlg, IDL_FILE, LB_GETTEXT, i, (LPARAM)file);
            strcat (files, file);
            strcat (files, "\n");
        }
    }

    /* Prompt the user */
    MessageBox (hDlg, files, "Deleting files", MB_OK | MB_ICONINFORMATION);

    /* Here these files are actually deleted */
}

static int DelFilesBoxProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_INITDIALOG:
        {
            char cwd [MAX_PATH + 1];
            SetNotificationCallback (GetDlgItem (hDlg, IDL_DIR), dir_notif_proc);
            SetNotificationCallback (GetDlgItem (hDlg, IDL_FILE), file_notif_proc);
            fill_boxes (hDlg, getcwd (cwd, MAX_PATH));
            return 1;
        }

        case MSG_COMMAND:
        {
            switch (wParam) {
                case IDOK:
                    prompt (hDlg);
                case IDCANCEL:
                    EndDialog (hDlg, wParam);
                    break;
            }
            break;
        }
    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}

int MiniGUIMain (int argc, const char* argv[])
{
#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER, "listbox", 0, 0);
#endif

    DlgDelFiles.controls = CtrlDelFiles;

    DialogBoxIndirectParam (&DlgDelFiles, HWND_DESKTOP, DelFilesBoxProc, 0L);
}

```

```
    return 0;
}

#ifdef _LITE_VERSION
#include <minigui/dti.c>
#endif
```

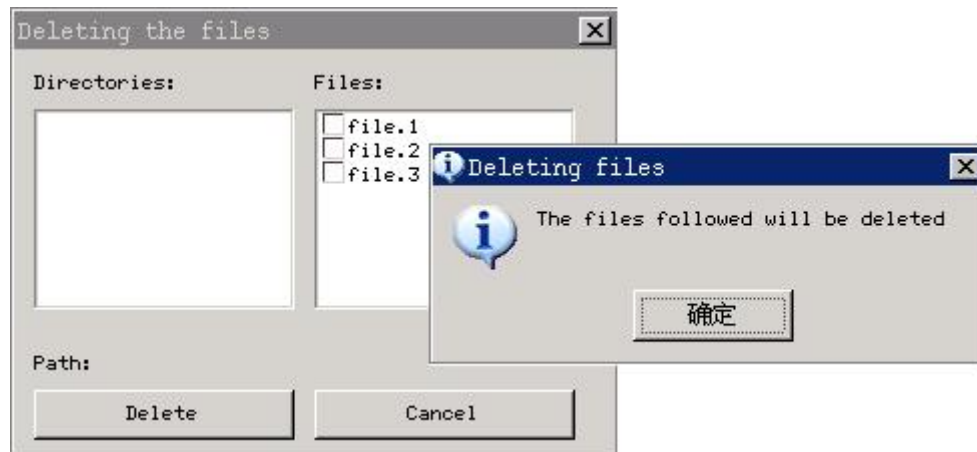


Fig. 21.4 "Delete File" dialog box

## 22 Edit Box Control

The edit box provides an important approach for the application to receive the user input. Compared with the controls mentioned above, such as static control, button, list box, etc., the behaviors of edit box is relatively simple. The edit box displays a caret, and the character input by the user would be inserted at the caret position. Besides that, the edit box also provides some simple editing functions such as selecting text, deleting text, moving the caret position, etc.

MiniGUI uses the following two control classes to implement the edit box:

- Single-line edit box: class name SLEDIT, identifier `CTRL_SLEDIT`. It can only handle single-line text, but with the help of MiniGUI logical font, it can handle arbitrary multi-byte characters, including variable-byte charset.
- Multiple-line edit box: class name TEXTEDIT, identifier `CTRL_TEXTEDIT`, `CTRL_MEDIT`, or `CTRL_MLEDIT`. It can handle arbitrary multiple-byte character input, including variable-byte charset.

Except the differences described above, the styles, messages and notification codes of the two types of edit box control classes are general similar, with only a little differences. Fig 22.1 shows the running effect of the three types of edit boxes.

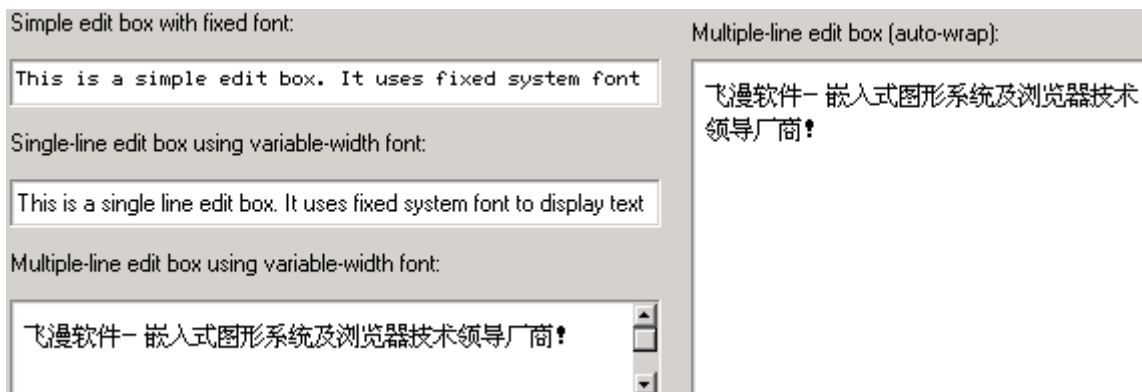


Fig. 22.1 MiniGUI edit boxes

## 22.1 Styles of Edit Box

Generally, we use the following style combination when creating an edit box:

```
WS_CHILD | WS_VISIBLE | WS_BORDER
```

Apparently, the style definition above does not use any styles specific to edit box. That is to say, we can use edit box normally without specifying any special styles of edit box. But the edit box has some special styles, mainly including:

- **ES\_UPPERCASE**: Make the edit box display all characters in uppercase.
- **ES\_LOWERCASE**: Make the edit box display all characters in lowercase.
- **ES\_PASSWORD**: Edit box is used for inputting the password, and displays an asterisk (\*) for each character typed into the edit box.
- **ES\_READONLY**: Create a read-only edit box, the user cannot change the text in the edit box, but the caret is visible.
- **ES\_BASELINE**: Draws a base line under text of the edit box.
- **ES\_AUTOWRAP**: Used for a multiple-line edit box, and automatically wraps against border when the text input exceeds the control border.
- **ES\_LEFT**: Specify the aligning style for a single line edit box, and realizes the left-align style.
- **ES\_NOHIDESEL**: Remain selected for the selected text when losing focus.
- **ES\_AUTOSELECT**: Selects all text when getting focus (only valid for single-line edit box).
- **ES\_TITLE**: Shows specified title texts in the first row, and this style is only suitable for Multiple-line edit box.
- **ES\_TIP**: When content of edit box is NULL, shows related tip texts, and this style is only suitable for SLEDIT control.
- **ES\_CENTER**: Specify the aligning style for a single line edit box, and realizes the center-align style.
- **ES\_RIGHT**: Specify the aligning style for a single line edit box, and realizes the right-align style.

For a multiple-line edit box, if you want to use the scroll bars, you can specify the styles **WS\_HSCROLL** and/or **WS\_VSCROLL**.



The following styles can be used to multiple-line edit box: `ES_UPPERCASE`, `ES_LOWERCASE`, `ES_READONLY`, `ES_BASELINE`, `ES_AUTOWRAP`, `ES_NOHIDESEL`, and `ES_TITLE`.

The following styles can be used to single-line edit box: `ES_UPPERCASE`, `ES_LOWERCASE`, `ES_READONLY`, `ES_BASELINE`, `ES_LEFT`, `ES_CENTER`, `ES_RIGHT`, `ES_PASSWORD`, `ES_NOHIDESEL`, `ES_AUTOSELECT`, and `ES_TIP`.

## 22.2 Messages of Edit Box

You can get the current text information in the edit box by using the following messages. These messages can be used for the two types of edit box classes described above:

- `MSG_GETTEXTLENGTH`: Get the length of the text with byte as unit.
- `MSG_GETTEXT`: Copy the text in the edit box.
- `MSG_SETTEXT`: Set the text in the edit box.

The application can also call the following three functions to complete corresponding work:

- `GetWindowTextLength`
- `GetWindowText`
- `SetWindowText`

In fact, the three functions above are the simple wraps of the three messages mentioned above. As we have seen in the foregoing chapters, the same messages and functions can also be used for controls such as static control and button control.

The messages to be introduced in the following sections are special for edit box.

### 22.2.1 Getting/Setting Caret Position

Sending `EM_GETCARETPOS` message to the edit box will get the current caret position:

```
int line_pos;
int char_pos;

SendMessage (hwndEdit, EM_GETCARETPOS, (WPARAM) &line_pos, (LPARAM) &char_pos);
```

After the message returns, `line_pos` and `char_pos` include the line number and the position in that line of the caret. For a single-line edit box, `line_pos` is always zero, so you can pass `NULL` value for the argument `wParam` of the message.

It should be noted that, for multi-line edit box, one lines means a character string line ended with a linefeed (carriage return), instead of a line in a paragraph when displayed with `ES_AUTOWRAP` style. Character position in edit box of MiniGUI takes multiple byte character (such as Chinese characters) as unit when displaying multiple byte text, instead of byte. This definition works for other messages of edit box.

The application can also set the caret position through `EM_SETCARETPOS` message:

```
int line_pos = 0;
int char_pos = 0;

SendMessage (hwndEdit, EM_SETCARETPOS, line_pos, char_pos);
```

The above function calling will place the caret in the beginning of the text.

### 22.2.2 Setting/Getting Selection of Text

`EM_GETSEL` message is used to get the currently selected text:

```
char buffer[buf_len];

SendMessage (hwndEdit, EM_GETSEL, buf_len, (LPARAM) buffer);
```

Here, `lParam` argument specifies the character buffer for saving the gotten text; `wParam` argument specifies the size of the buffer. If the specified buffer is

small, then excess text will be cut off.

**EM\_SETSEL** message is used to set the currently selected text:

```
int line_pos, char_pos;  
SendMessage (hwndEdit, EM_SETSEL, line_pos, char_pos);
```

Here, **lParam** argument specifies the row index of selected point; **wParam** specifies the character position in line of selected point. After this message is sent, text between the current insertion point and the selected point will be selected.

**EM\_GETSELPOS** message is used to get the position of currently selected point:

```
int line_pos;  
int char_pos;  
SendMessage (hwndEdit, EM_GETCARETPOS, (WPARAM) &line_pos, (LPARAM) &char_pos);
```

The use of **EM\_GETSELPOS** message is similar to that of **EM\_GETCARETPOS** message.

**EM\_SELECTALL** message is used to make all the text in edit box selected, as <CTRL+A> operation:

```
SendMessage (hwndEdit, EM_SELECTALL, 0, 0);
```

### 22.2.3 Copy, Cut, and Past

You can perform edit operations on an edit box control, for example, copy, cut, and paste, by key operations or sending corresponding messages.

Keyboard operations of edit box control, such as copy:

- **CTRL+C**: Copy text from edit box to clipboard
- **CTRL+V**: Copy text from clipboard to edit box
- **CTRL+X**: Cut text of edit box to clipboard

**EM\_COPYTOCB** message is used to copy the currently selected text of an edit box control to the clipboard, as <CTRL+C> operation:

```
SendMessage (hwndEdit, EM_COPYTOCB, 0, 0);
```

**EM\_CUTTOCB** message is used to cut the currently selected text of an edit box control to the clipboard, as <CTRL+X> operation:

```
SendMessage (hwndEdit, EM_CUTTOCB, 0, 0);
```

**EM\_INSERTCBTEXT** message is used to copy the text on the clipboard to an edit box, as <CTRL+V> operation:

```
SendMessage (hwndEdit, EM_INSERTCBTEXT, 0, 0);
```

#### 22.2.4 Setting/Getting Properties of Line Height and Others

Row height here represents the height of single line in wrapped display style.

**EM\_GETLINEHEIGHT** message is used to get the row height of an edit box:

```
int line_height;  
line_height = SendMessage (hwndEdit, EM_GETLINEHEIGHT, 0, 0);
```

**EM\_SETLINEHEIGHT** message is used to set the row height, If success, return old line height value, otherwise return -1:

```
int line_height;  
SendMessage (hwndEdit, EM_SETLINEHEIGHT, line_height, 0);
```

It should be noted that it is preferred to use **EM\_SETLINEHEIGHT** message before setting text of an edit box, because reset row height will redraw the whole content of the edit box.

**EM\_GETLINECOUNT** message is used to get the number of rows:

```
int line_count;
```

```
line_count = SendMessage (hwndEdit, EM_GETLINECOUNT, 0, 0);
```

Row herein represents a single line in wrapped display style.

### 22.2.5 Setting Limit of Text

Sending `EM_LIMITTEXT` to an edit box can set the text upper limit in bytes of an edit box:

```
SendMessage (hwndEdit, EM_LIMITTEXT, 10, 0L);
```

The message above will make that only 10 bytes of characters can be input to the edit box.

### 22.2.6 Setting or Canceling Read-only Status

Using `EM_SETREADONLY` message and passing `TRUE` for `wParam`, the edit box will be set in read-only status, and passing `FALSE` for `wParam` will make the edit box be set in normal editable status.

### 22.2.7 Setting/Getting Password Character

By default, MiniGUI uses asterisks (\*) to display the characters input in a password edit box. However, we can use the `EM_SETPASSWORDCHAR` message to change the password character:

```
SendMessage (hwndEdit, EM_SETPASSWORDCHAR, '%', 0L);
```

The message above will change the password character to be '%'.

Using `EM_GETPASSWORDCHAR` message would get the current password character.

**[Prompt] Password character only can be set ASCII code which can be displayed.**

Note that TEXTEDIT control does not carry out `EM_SETPASSWORDCHAR` and

**EM\_GETPASSWORDCHAR** messages.

### 22.2.8 Setting Title and Tip Text

When SLEDIT has **ES\_TIP** styles, you can set tip text of the edit box using **EM\_SETTIPTTEXT** message, and get tip text of edit box using **EM\_GETTIPTTEXT** message.

```
int len;
char *tip_text;
SendMessage (hwndEdit, EM_SETTIPTTEXT, len, (LPARAM)tip_text);
```

Here **lParam** parameter specifies tip text string, **wParam** parameter specifies the length of the string; if **tip\_text** is ended with '\0', **wParam** can be set to -1, if **wParam** is 0, **tip\_text** can be NULL. **EM\_GETTIPTTEXT** message will return the current tip text string:

```
int len;
char tip_text[len+1];
SendMessage (hwndEdit, EM_SETTIPTTEXT, len, (LPARAM)tip_text);
```

Here **lParam** specifies the buffer to store the tip text string; **wParam** specifies the length of the string that can be stored in buffer (not including '\0').

When TEXTEDIT control has **ES\_TITLE** style, you can use **EM\_SETTITLETEXT** message to set the title text of the edit box, and use **EM\_GETTITLETEXT** message to get title text of edit box:

```
int len;
char *title_text;
SendMessage (hwndEdit, EM_SETTITLETEXT, len, (LPARAM)title_text);
```

Here **lParam** parameter specifies title text string, **wParam** parameter specifies the length of the string; if **tip\_text** is ended with '\0', **wParam** can be set as -1, if **wParam** is 0, **tip\_text** can be NULL. **EM\_GETTIPTTEXT** message will return current title text string.

```
int len;
char title_text[len+1];
```

```
SendMessage (hwndEdit, EM_GETTITLETEXT, len, (LPARAM)title_text);
```

Here **lParam** parameter specifies the buffer to store the title text string;  
**wParam** parameter specifies the length of string that can be stored in the buffer (not including '\0').

### 22.2.9 Setting End of Line Symbol

In normal situation, TEXTEDIT edit box will not display linefeed symbol. If using **EM\_SETLFDISPCHAR** message sets the display symbol for linefeeds, then edit box will display linefeed as the set display symbol.

```
char disp_char;  
SendMessage (hwndEdit, EM_SETLFDISPCHAR, 0, disp_char);
```

Here **lParam** parameter is the display symbol to be set for linefeed.

For example, if you want to display linefeed with "\*", you can set as follows:

```
SendMessage (hwndEdit, EM_SETLFDISPCHAR, 0, '*');
```

### 22.2.10 Setting End of Line

In default situation, linefeed symbol of TEXTEDIT edit box is '\n'. You can use **EM\_SETLINESEP** message to change the line feed used by edit box.

```
char sep_char;  
SendMessage (hwndEdit, EM_SETLINESEP, 0, sep_char);
```

Here **lParam** parameter is the linefeed symbol to be set.

For example, if you want to indicate the line is end with **TAB** character, you can set as follows:

```
SendMessage (hwndEdit, EM_SETLINESEP, 0, '\t');
```

### 22.2.11 Getting Paragraphs Information

EM\_GETNUMOFPARAGRAPHS message is used to get the number of paragraphs.

```
int num;
num = SendMessage (hwnd, EM_GETNUMOFPARAGRAPHS, 0, 0);
```

EM\_GETPARAGRAPHLENGTH message is used to get the length of a specified paragraph. If success, return the length of a specified paragraph, otherwise return -1.

```
int len;
len = SendMessage (hwnd, EM_GETPARAGRAPHLENGTH, idx, 0);
```

Here, **wParam** argument specifies the paragraph index.

EM\_GETPARAGRAPHTEXT message is used to get the text of a specified paragraph. To show correct characters, it might adjust the number of characters can be copied appropriately.

```
TEXTPOSINFO info;
unsigned char buff [32];

info.start_pos = 5;
info.copy_len = 10;
info.buff = buff;
info.paragraph_index = -1;

SendMessage (hwnd, EM_GETPARAGRAPHTEXT, &info, 0);
```

Here, **info** is a structure of **TEXTPOSINFO** type, and specifies the related information for the copied text, as follows:

```
typedef struct _TEXTPOSINFO {
    /*The index of paragraph, if value is -1,
    *it will take effect on the whole text.*/
    int paragraph_index;
    /*The beginning byte position can be copied to the buffer.*/
    int start_pos;
    /*The maximal number of bytes can be copied to the buffer.*/
    int copy_len;
    /*The pointer to a buffer receives the text.
    *Please make sure buffer size is more than the value of copy_len.*/
    char *buff;
}TEXTPOSINFO;
```

The member **start\_pos** is the beginning byte position can be copied to the buffer. The member **copy\_len** is the maximal number of bytes can be copied to the buffer. The member **paragraph\_index** is the index of paragraph, if value is -1, it will take effect on the whole text. The member



`buff` is the character buffer for saving the gotten text.

## 22.3 Notification Codes of Edit Box

The edit box has not `ES_NOTIFY` style, so any edit box can generate notification messages, as listed below:

- **EN\_SETFOCUS**: The edit box has received the input focus.
- **EN\_KILLFOCUS**: The edit box has lost the input focus.
- **EN\_CHANGE**: The content in the edit box has been altered.
- **EN\_UPDATE**: The content in the edit box has been altered after edit box received `MSG_SETTEXT`, `EM_RESETCONTENT`, or `EM_SETLINEHEIGHT` message.
- **EN\_ENTER**: The user has pressed the **ENTER** key in the edit box.
- **EN\_MAXTEXT**: The inserting text exceeds the maximum limitation of the edit box.
- **EN\_DBLCLK**: The edit control is double clicked with the left button of the mouse.
- **EN\_CLICKED**: The edit control is clicked with the left button of the mouse.

## 22.4 Sample Program

In the foregoing chapters, we have already seen the use of edit boxes. List 22.1 gives another example of edit boxes, this program copy the input of a single-line edit box to an automatic wrapped multiple-line edit box. Please refer to `edit.c` file of the demo program package `mg-samples` of this guide for the complete source code. Effect of running the program is shown in Fig. 22.2.

List 22.1 Example for using edit boxes

```
#include <stdio.h>
#include <stdlib.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

/* Define dialog template */
```

```
static DLGTEMPLATE DlgBoxInputChar =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    120, 150, 400, 210,
    "Please input letters",
    0, 0,
    4, NULL,
    0
};

#define IDC_CHAR          100
#define IDC_CHARS         110

static CTRLDATA CtrlInputChar [] =
{
    {
        CTRL_STATIC,
        WS_VISIBLE | SS_CENTER,
        10, 10, 380, 18,
        IDC_STATIC,
        "Please input a letter:",
        0
    },
    {
        CTRL_SLEDIT,
        WS_VISIBLE | WS_TABSTOP | WS_BORDER,
        170, 40, 40, 34,
        IDC_CHAR,
        NULL,
        0
    },
    {
        CTRL_MLEDIT,
        WS_VISIBLE | WS_BORDER | WS_VSCROLL | ES_BASELINE | ES_AUTOWRAP,
        10, 80, 380, 70,
        IDC_CHARS,
        "",
        0
    },
    {
        CTRL_BUTTON,
        WS_TABSTOP | WS_VISIBLE | BS_DEFPUSHBUTTON,
        170, 160, 60, 25,
        IDOK,
        "OK",
        0
    }
};

static void my_notif_proc (HWND hwnd, int id, int nc, DWORD add_data)
{
    if (nc == EN_CHANGE) {
        char buff [5];

        /* Get the user input(the first character) of the single-line
         * edit box, and insert it to the multiple-line edit box
         */
        GetWindowText (hwnd, buff, 4);
        /* Place the caret postion of the single-line edit box in front,
         * thus to overlap the old character
         */
        SendMessage (hwnd, EM_SETCARETPOS, 0, 0);
        SendMessage (GetDlgItem (GetParent (hwnd), IDC_CHARS), MSG_CHAR, buff[0], 0L);
    }
}

static int InputCharDialogBoxProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    static PLOGFONT my_font;
    HWND hwnd;

    switch (message) {
        case MSG_INITDIALOG:
            my_font = CreateLogFont (NULL, "Arial", "ISO8859-1",

```

```

        FONT_WEIGHT_REGULAR, FONT_SLANT_ROMAN, FONT_SETWIDTH_NORMAL,
        FONT_SPACING_CHARCELL, FONT_UNDERLINE_NONE, FONT_STRUCKOUT_NONE,
        30, 0);

    hwnd = GetDlgItem (hDlg, IDC_CHAR);

    /* Set the font of the single-line edit box to be a big font */
    SetNotificationCallback (hwnd, my_notif_proc);
    SetWindowFont (hwnd, my_font);

    /* Simulate the press of INSERT key, and set the edit mode to be overlap mode */
    SendMessage (hwnd, MSG_KEYDOWN, SCANCODE_INSERT, 0L);
    return 1;

case MSG_COMMAND:
    switch (wParam) {
        case IDOK:
        case IDCANCEL:
            DestroyLogFont (my_font);
            EndDialog (hDlg, wParam);
            break;
    }
    break;
}

return DefaultDialogProc (hDlg, message, wParam, lParam);
}

int MiniGUIMain (int argc, const char* argv[])
{
#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER, "edit", 0, 0);
#endif

    DlgBoxInputChar.controls = CtrlInputChar;
    DialogBoxIndirectParam (&DlgBoxInputChar, HWND_DESKTOP, InputCharDialogBoxProc, 0L);

    return 0;
}

#ifdef _LITE_VERSION
#include <minigui/dti.c>
#endif

```

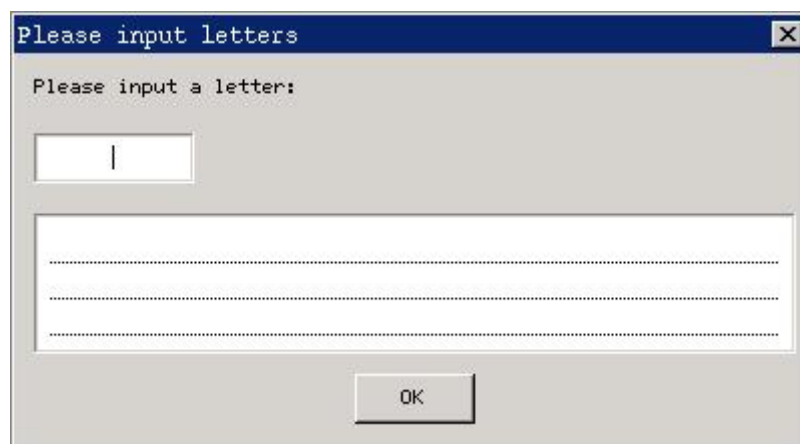


Fig. 22.2 Effect of running the edit box example program



## 23 Combo Box Control

In nature, a general combo box is the combination of an edit box and a list box. The user can type text in the edit box, and also can select one item from the options listed in the list box. The combo box in MiniGUI can be divided into three types: simple combo box, pull-down combo box, spin combo box and digital spin box. Using pull-down combo box has another advantage, i.e. it can reduce the occupied space of parent window because of using a pull-down list box.

We can create a combo box control by calling `CreateWindow` function with `CTRL_COMBOBOX` as the control class name.

### 23.1 Types and Styles of Combo Box

#### 23.1.1 Simple Combo Box, Pull-Down Combo Box, and Spin Combo Box

A simple combo box can be created with the style of `CBS_SIMPLE`. The list box of a simple combo box will always be shown below the edit box. When the user selects an item in the list box, the text of this item will fill the edit box, as shown in Fig. 23.1.



Fig. 23.1 Simple combo box control

A pull-down combo box can be created with the style of `CBS_DROPDOWNLIST`. The pull-down combo box is different from the simple combo box in that the combo box only shows an item in the rectangle region, and an icon pointing downwards presents on the right of the item content. When the user clicks the

icon, a list box will be popped up to show more items. Fig. 23.2 shows the effect of the pull-down combo boxes in normal state and pull-down state.

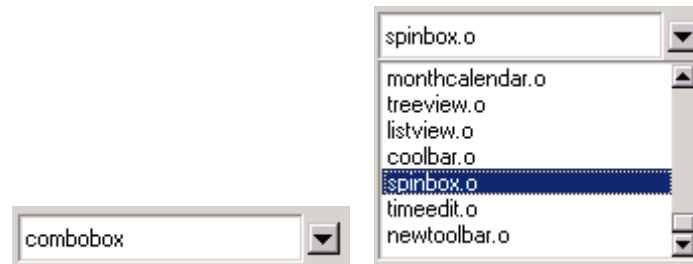


Fig. 23.2 Pull-down combo box control

When calling `CreateWindow` function to create the simple combo box and pull-down combo box, you should use `dwAddData` argument of `CreateWindow` to pass the height value of the list box, as shown in following:

```
hWnd4 = CreateWindow (CTRL_COMBOBOX,
    "0",
    WS_VISIBLE | CBS_SIMPLE | CBS_SORT | WS_TABSTOP,
    IDC_BOX4,
    10, 100, 180, 24,
    parent, 100);
/* Specify dwAddData to be 100, that is, specify the height of
 * the list box in the simple combo box to be 100
 */
```

Although behaviors of a spin combo box control differs much from that of the two types of combo boxes above, it is in nature still combination of an edit box and a list box, except that the list box in a spin combo box is always hidden, and we can select an item in the list box by the two arrow buttons beside the edit box. The spin combo box can be created with `CBS_SPINLIST` style, and its effect is shown in Fig. 23.3.

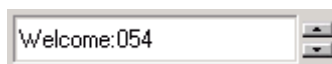


Fig. 23.3 Spin combo box

Spin combo box also has two special display styles (as shown in Fig. 13.4):



Fig. 23.4 The special styles of spin combo box

- `CBS_SPINARROW_TOPBOTTOM`: Arrow is on the top or bottom of content
- `CBS_SPINARROW_LEFTRIGHT`: Arrow is on the left or right of content

As the three types of combo boxes described above use the edit box and the list box predefined in MiniGUI, you can use similar styles of the edit box control and the list box control for the combo boxes:

- `CBS_READONLY`: The input field is read-only, corresponding to the `ES_READONLY` style of edit box.
- `CBS_UPPERCASE`: Corresponding to the `ES_UPPERCASE` style of the edit box, making the text typed in the edit box transferring into uppercase automatically.
- `CBS_LOWERCASE`: Corresponding to the `ES_LOWERCASE` style of the edit box, making the text typed in the edit box transferring into lowercase automatically.
- `CBS_EDITBASELINE`: Corresponding to the `ES_BASELINE` style of the edit box, making the edit box have a base line.
- `CBS_SORT`: Corresponding to the `LBS_SORT` style of the list box, combo box with this style will automatically sort the inserted items.

**[Note] A list box can have other advanced styles, but the list box in a combo box is only a normal single-selection list box.**

Except the styles described above, the combo box also defines the following two styles, which can be used for all types of combo box:

- `CBS_EDITNOBORDER`: Make the edit box have no `WS_BORDER` style, and make the input field have no border.
- `CBS_AUTOFOCUS`: The edit box will gain the input focus automatically when the combo box gains the input focus.

### 23.1.2 Digital Spin Box

A digital spin box is similar to a spin combo box in appearance, wherein digits are displayed instead of item of a list box. A combo box of this type can be used in many situations. However, because the operated object is digit instead of arbitrary text, it does not use a list box internally. The digital spin box can

be created with combo box style `CBS_AUTOSPIN`, which is as shown in Fig. 23.5.



Fig. 23.5 Digital spin box

The digital spin box has only a style, i.e. `CBS_AUTOLOOP`. The digit in the box will be looped automatically when this style has been used. That is to say, after the user has clicked the up arrow to the maximum value, the digit in the box will change to the minimum value if the up arrow is clicked again. The minimum and maximum values of the digital spin box are 0 and 100 by default, respectively. Each time the user clicks the button on the right side, the digit-changed step is 1 by default.

The default max and min value are 100 and 0; the default step value when click the button is 1 and the default step of click pagedown or pageup is 5.

## 23.2 Messages of Combo Box

### 23.2.1 Messages of Simple, Pull-Down, and Spin Combo Box

Because the three types of combo boxes all include list box, messages used for these three types of combo boxes basically correspond to the messages of a list box:

- `CB_ADDSTRING`: Correspond to `LB_ADDSTRING`, used for adding items to the inner list box.
- `CB_INSERTSTRING`: Correspond to `LB_INSERTSTRING`, used for inserting items to the inner list box.
- `CB_DELETETESTRING`: Correspond to `LB_DELETETESTRING`, used for deleting items from the inner list box.
- `CB_FINDSTRING`: Correspond to `LB_FINDSTRING`, used for fuzzily matching items of the list box.
- `CB_FINDSTRINGEXACT`: Correspond to `LB_FINDSTRINGEXACT`, used for



exactly matching items of the list box.

- **CB\_GETCOUNT**: Correspond to **LB\_GETCOUNT**, used for getting the number of items in the inner list box.
- **CB\_GETCURSEL**: Correspond to **LB\_GETCURSEL**, used for getting the currently selected item of the inner list box.
- **CB\_SETCURSEL**: Correspond to **LB\_SETCURSEL**, used for setting the selected item of the inner list box.
- **CB\_RESETCONTENT**: Correspond to **LB\_RESETCONTENT**, used for removing the contents of the inner list box.
- **CB\_GETITEMADDDATA**: Correspond to **LB\_GETITEMADDDATA**, used for getting the additional data associated with an item of the inner list box.
- **CB\_SETITEMADDDATA**: Correspond to **LB\_SETITEMADDDATA**, used for setting the additional data associated with an item of the inner list box.
- **CB\_GETITEMHEIGHT**: Correspond to **LB\_GETITEMHEIGHT**, used for getting the height of an item of the inner list box.
- **CB\_SETITEMHEIGHT**: Correspond to **LB\_SETITEMHEIGHT**, used for setting the height of an item of the inner list box.
- **CB\_SETSTRCMPFUNC**: Correspond to **LB\_SETSTRCMPFUNC**, used for setting the string comparing function for sorting items of the inner list box.
- **CB\_GETLBTEXT**: Correspond to **LB\_GETTEXT**, used for getting the text of an item of the inner list box.
- **CB\_GETLBTEXTLEN**: Correspond to **LB\_GETTEXTLEN**, used for getting the length of text of an item of the inner list box.
- **CB\_GETCHILDREN**: Gets the handles to the children of ComboBox control. The parameter **wParam** holds a editor pointer and **lParam** holds a list box pointer.

Combo box also provides the message for the inner edit box:

- **CB\_LIMITTEXT**: Correspond to **EM\_LIMITTEXT**, used for limiting the length of text in the inner edit box.
- **CB\_SETEDITSEL**: Correspond to **EM\_SETSEL**, set the selected text in the editor.
- **CB\_GETEDITSEL**: Correspond to **EM\_GETSEL**, get the selected text in the editor.

You can refer to the description on the list box message and the edit box message in Chapter 21 and Chapter 22 for the concrete use of the above messages. The following two messages can be used for a spin combo box:

- **CB\_SPIN**: Send the message to spin the spin box forward or backward, like the user click the up or down arrow besides the edit box (Typing the up or down arrow key in the edit box also can get the same effect). The parameter **wParam** controls the direction of the spin, 0 means up, 1 means down.
- **CB\_FASTSPIN**: Send the message to fast spin the spin box, like the user types **PageUp/PageDown** key in the edit box. The parameter **wParam** controls the direction of the spin, 0 means up, 1 means down.

Follow two message can be used for drop down combo box:

- **CB\_GETDROPPEDCONTROLRECT**: Get the rect of the drop down combo box.
- **CB\_GETDROPPEDSTATE**: Check the combo box's drop down list is displayed or not.

### 23.2.2 Messages of Digital Spin Box

Digital spin box can receive the following messages:

- **CB\_GETSPINRANGE**: Get the maximum and minimum values, which are stored in the addresses to which are pointed to by **wParam** and **lParam**, respectively.
- **CB\_SETSPINRANGE**: Set the maximum and minimum values, which are the values of **wParam** and **lParam**, respectively.
- **CB\_SETSPINVALUE**: Set the current value of the edit box, and the value to be set is passed by **wParam**.
- **CB\_GETSPINVALUE**: This message returns the current value of the edit box.
- **CB\_SPIN**: Send the message to spin the spin box forwards or backwards, the same as the user clicks the up or down arrow besides the edit box (Typing the up or down arrow key in the edit box also can get the same effect). The parameter **wParam** controls the direction of the spin: 1 means up and 0 means down.

- **CB\_FASTSPIN**: Send the message to spin the spin box quickly, like the user types **PageUp/PageDown** key in the edit box. The parameter **wParam** controls the spinning direction: 0 means up and 1 means down.
- **CB\_GETSPINPACE**: Get the pace and the fast pace of the spin box in a combo box.
- **CB\_SETSPINPACE**: Set the pace and the fast pace of the spin box in a combo box
- **CB\_SETSPINFORMAT**: Sets the format string of integer. MiniGUI uses **sprintf** and **sscanf** to transform between text string and integer, and make them have specified format. After format string is set, MiniGUI will use this format string when calling **sprintf** and **sscanf** functions to make it has a certain display format.

### 23.3 Notification Codes of Combo Box

Notifications can be generated when a combo box has the style **CBS\_NOTIFY**.

Notification codes of combo box notification messages are basically the combination of list box notification codes and edit box notification codes, which are listed as follow:

- **CBN\_ERRSPACE**: No enough memory
- **CBN\_SELCHANGE**: Current selection changes
- **CBN\_EDITCHANGE**: Text in the edit control changes
- **CBN\_DBLCLK**: User double clicked an item of the combo box
- **CBN\_CLICKED**: User clicked the combo box.
- **CBN\_SETFOCUS**: The combo box gains the input focus. If the combo box has **CBS\_AUTOFOCUS** style, the inner edit box would gain the input box simultaneously.
- **CBN\_KILLFOCUS**: The combo box loses the input focus.
- **CBN\_DROPDOWN**: The user pulled down the list box. When the user clicks the down arrow button beside the edit box or type an arrow key in the edit box, such as up or down arrow key, **PageDown** or **PageUp** key etc. the list box would be pulled down and showed.
- **CBN\_CLOSEUP**: Pull-down list box is hidden (closed up).
- **CBN\_SELENDOK**: The user selects an item from the pull-down list box.

- **CBN\_SELEND CANCEL**: The user does not select any item and close the pull-down list box.

## 23.4 Sample Program

The program in List 23.1 gives an example of use of combo boxes. The program comprises a time selection box with digital spin box, and then provides the list of heroes to be dated with a pull-down list box. When pressing "OK", the program uses **MessageBox** to display the date information. Please refer to **combobox.c** file of the demo program package of this guide for the complete source code of the program. Effect of running the program is shown in Fig. 23.6.

List 23.1 Use of combo boxes

```
#include <stdio.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

#define IDC_HOUR 100
#define IDC_MINUTE 110
#define IDC_SECOND 120
#define IDC_DAXIA 200

#define IDC_PROMPT 300

/* Define dialog box template */
static DLGTEMPLATE DlgMyDate =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    100, 100, 360, 150,
    "Meeting Plan",
    0, 0,
    9, NULL,
    0
};

static CTRLDATA CtrlMyDate[] =
{
    {
        "static",
        WS_CHILD | SS_RIGHT | WS_VISIBLE,
        10, 20, 50, 20,
        IDC_STATIC,
        "I will at ",
        0
    },
    /* the digital spin box to display hour */
    {
        CTRL_COMBOBOX,
        WS_CHILD | WS_VISIBLE |
            CBS_READONLY | CBS_AUTOSPIN | CBS_AUTOLOOP | CBS_EDITBASELINE,
        60, 18, 40, 20,
```

```

        IDC_HOUR,
        "",
        0
    },
    {
        "static",
        WS_CHILD | SS_CENTER | WS_VISIBLE,
        100, 20, 20, 20,
        IDC_STATIC,
        ":",
        0
    },
    /* The digital spin box to display minute */
    {
        CTRL_COMBOBOX,
        WS_CHILD | WS_VISIBLE |
            CBS_READONLY | CBS_AUTOSPIN | CBS_AUTOLOOP | CBS_EDITBASELINE,
        120, 18, 40, 20,
        IDC_MINUTE,
        "",
        0
    },
    {
        "static",
        WS_CHILD | SS_CENTER | WS_VISIBLE,
        160, 20, 30, 20,
        IDC_STATIC,
        "meet ",
        0
    },
    /* List the names of heroes */
    {
        CTRL_COMBOBOX,
        WS_VISIBLE | CBS_DROPDOWNLIST | CBS_NOTIFY,
        190, 20, 100, 20,
        IDL_DAXIA,
        "",
        80
    },
    /* Display characteristic of heroes */
    {
        "static",
        WS_CHILD | SS_RIGHT | WS_VISIBLE,
        10, 50, 330, 30,
        IDC_PROMPT,
        "Zeus is the leader of the gods and god of the sky and thunder in Greek mythology.",
        "",
        0
    },
    {
        CTRL_BUTTON,
        WS_VISIBLE | BS_DEFPUSHBUTTON | WS_TABSTOP | WS_GROUP,
        10, 70, 130, 25,
        IDOK,
        "OK",
        0
    },
    {
        "button",
        WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP,
        150, 70, 130, 25,
        IDCANCEL,
        "Cancel",
        0
    },
    },
};

static const char* daxia [] =
{
    "Zeus",
    "Pan",
    "Apollo",
    "Heracles",
    "Achilles",
    "Jason",

```

```

    "Theseus",
};

static const char* daxia_char [] =
{
    "Zeus is the leader of the gods and god of the sky and thunder in Greek mythology.",
    "Pan is the Greek god who watches over shepherds and their flocks.",
    "Apollo is a god in Greek and Roman mythology, the son of Zeus and Leto, and the twin
of Artemis.",
    "Heracles is the greatest of the mythical Greek heroes, best known for his superhuman
strength and many stories are told of his life.",
    "Achilles is the greatest warrior in the Trojan War.",
    "Jason is a hero of Greek mythology. His father is Aeson, the rightful king of Iolcus
.",
    "Theseus is a legendary king of Athens. Theseus is considered by Athenians as the gre
at reformer.",
};

static void daxia_notif_proc (HWND hwnd, int id, int nc, DWORD add_data)
{
    if (nc == CBN_SELCHANGE) {
        /* Display corresponding characteristic according to the hero currently selected
*/
        int cur_sel = SendMessage (hwnd, CB_GETCURSEL, 0, 0);
        if (cur_sel >= 0) {
            SetWindowText (GetDlgItem (GetParent(hwnd), IDC_PROMPT), daxia_char [cur_sel]
);
        }
    }
}

static void prompt (HWND hDlg)
{
    char date [1024];

    /* Summary the dating content */
    int hour = SendDlgItemMessage(hDlg, IDC_HOUR, CB_GETSPINVALUE, 0, 0);
    int min = SendDlgItemMessage(hDlg, IDC_MINUTE, CB_GETSPINVALUE, 0, 0);
    int sel = SendDlgItemMessage(hDlg, IDC_DAXIA, CB_GETCURSEL, 0, 0);

    sprintf (date, "You will meet %s at %02d:%02d", daxia [sel], hour, min);
    MessageBox (hDlg, date, "Meeting Plan", MB_OK | MB_ICONINFORMATION);
}

static int MyDateBoxProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    int i;
    switch (message) {
        case MSG_INITDIALOG:
            /* Set the range of hour spin box to be 0~23, digital is displayed with %02d format
*/
            SendDlgItemMessage(hDlg, IDC_HOUR, CB_SETSPINRANGE, 0, 23);
            SendDlgItemMessage(hDlg, IDC_HOUR, CB_SETSPINFORMAT, 0, (LPARAM) "%02d");
            /* Set the current value to be 20 */
            SendDlgItemMessage(hDlg, IDC_HOUR, CB_SETSPINVALUE, 20, 0);
            /* Set the pace and fast pace both to be 1 */
            SendDlgItemMessage(hDlg, IDC_HOUR, CB_SETSPINPACE, 1, 1);

            /* Set the range of hour spin box to be 0~59, digital is displayed with %02d for
mat */
            SendDlgItemMessage(hDlg, IDC_MINUTE, CB_SETSPINRANGE, 0, 59);
            SendDlgItemMessage(hDlg, IDC_MINUTE, CB_SETSPINFORMAT, 0, (LPARAM) "%02d");
            /* Set the current value to be 0 */
            SendDlgItemMessage(hDlg, IDC_MINUTE, CB_SETSPINVALUE, 0, 0);
            /* Set the pace to be 1, and set the fast pace to be 2 */
            SendDlgItemMessage(hDlg, IDC_MINUTE, CB_SETSPINPACE, 1, 2);

            /* Add the names of heroes */
            for (i = 0; i < 7; i++) {
                SendDlgItemMessage(hDlg, IDC_DAXIA, CB_ADDSTRING, 0, (LPARAM) daxia [i]);
            }

            /* Specify notification callback function */
            SetNotificationCallback (GetDlgItem (hDlg, IDC_DAXIA), daxia_notif_proc);
            /* Set initial values of names and characteristics of heroes */

```

```

    SendDlgItemMessage(hDlg, IDL_DAXIA, CB_SETCURSEL, 0, 0);
    SetWindowText (GetDlgItem (hDlg, IDC_PROMPT), daxia_char [0]);
    return 1;

case MSG_COMMAND:
    switch (wParam) {
    case IDOK:
        /* Display the current selection */
        prompt (hDlg);
    case IDCANCEL:
        EndDialog (hDlg, wParam);
        break;
    }
    break;

}

return DefaultDialogProc (hDlg, message, wParam, lParam);
}

int MiniGUIMain (int argc, const char* argv[])
{
#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER, "combobox", 0, 0);
#endif

    DlgMyDate.controls = CtrlMyDate;

    DialogBoxIndirectParam (&DlgMyDate, HWND_DESKTOP, MyDateBoxProc, 0L);

    return 0;
}

#ifdef _LITE_VERSION
#include <minigui/dti.c>
#endif

```



Fig. 23.6 Use of combo boxes





## 24 Menu Button Control

Functions of a menu button are generally the same as that of a normal pull-down combo box. Actually, in the early version of MiniGUI, the menu button acts as the alternate of the combo box. Of course, the menu button has much limitation, e.g., it cannot be edited, and does not provide scrolling of list items, etc.

In appearance, the menu button is like a normal push button. The difference is that the menu button has a small rectangle (shown as a down arrow in FLAT style) on the right side of the rectangular button region. When the user clicks the control, a menu would pop up, and when the user clicks an item of the menu with the mouse, the button content will change to the content of this item, as shown in Fig. 24.1.

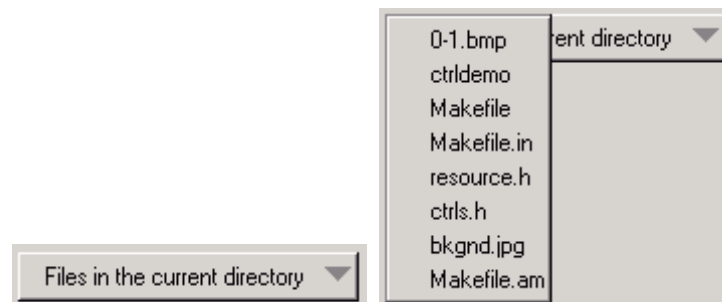


Fig. 24.1 Menu button (The left is normal status, the right is effect after menu pops up)

Calling `CreateWindow` function with `CTRL_MENUBUTTON` as the control class name can create a menu button.

### 24.1 Styles of Menu Button

The menu button has some special styles of its own. Generally, we use the following style combination when creating a menu button.

```
WS_CHILD | WS_VISIBLE | MBS_SORT
```

- **MBS\_SORT**: Sorting the menu items

- **MBS\_LEFTARROW**: The menu pull-down arrow will be display at the left of the text.
- **MBS\_NOBUTTON**: The control have not push button
- **MBS\_ALIGNLEFT**: The text on menubutton is left-align
- **MBS\_ALIGNRIGHT**: The text on menubutton is right-align
- **MBS\_ALIGNCENTER**: The text on menubutton is center-align

The position of bitmap cannot be affected by align-style when it is set

## 24.2 Messages of Menu Button

### 24.2.1 Adding Items to Menu Button Control

**MBM\_ADDITEM** message is used to add items to a menu button. You should pass an initialized **MENUBUTTONITEM** structure when calling **SendMessage**, as shown in the following:

```
MENUBUTTONITEM mbi;           // Declare a menu item structure variable
mbi.text = "item one";         // Set the item text
mbi.bmp = NULL;                // Here you can specify a bitmap object
mbi.data = 0;
pos = SendMessage (hMbtnWnd, MBM_ADDITEM, -1, (LPARAM) &mbi);
```

Here **hMbtnWnd** is the handle of a menu button control. **SendMessage** will return the index of the new menu item, and return **MB\_ERR\_SPACE** when the memory is not enough.

### 24.2.2 Deleting Items from Menu Button Control

Items can be deleted form a menu button by using **MBM\_DELITEM** message and specifying the index of the menu item to be deleted, as shown in the following:

```
SendMessage (hMbtnWnd, MBM_DELITEM, index, 0);
```

Here, **index** is the index of an item.

### 24.2.3 Deleting All Items in the Menu

As the same as a list box, the menu button also provides the message to delete all items, i.e. **MBM\_RESETCTRL** message, which is shown as follows:

```
SendMessage (hMbtnWnd, MBM_RESETCTRL, 0, 0);
```

### 24.2.4 Setting Current Selected Item

Similarly, you can use **MBM\_SETCURITEM** message to set the selected item, and the text of the selected item will be displayed on the menu button, shown as follows:

```
SendMessage (hMbtnWnd, MBM_SETCURITEM, index, 0);
```

Here, **index** is the index of the item to be set.

### 24.2.5 Getting Current Selected Item

You can get the index of the current selected item with **MBM\_GETCURITEM** message, shown as follows:

```
index = SendMessage (hMbtnWnd, MBM_GETCURITEM, 0, 0);
```

This message returns the index of the current selected item.

### 24.2.6 Getting/Setting Data of Menu Item

You can use **MBM\_GETITEMDATA** and **MBM\_SETITEMDATA** messages to get and set the data of a menu item, respectively. When using these two messages, **wParam** should be used to pass the index of the menu item to be gotten or set, **lParam** should be used to pass a pointer to **MENUBUTTONITEM** structure, which includes the text of the menu item, bitmap object, and additional data.

It should be noted that, **MENUBUTTONITEM** structure includes a member called

**which.** This member specifies which data of the menu item to be gotten or set (text, bitmap object and/or additional data), and is generally the combination of the following values:

- **MB\_WHICH\_TEXT:** Indicate the text of the menu item to be gotten or set. Here the text member of the structure must point to a valid buffer.
- **MB\_WHICH\_BMP:** Indicate the bitmap object of the menu item to be gotten or set.
- **MB\_WHICH\_ATTDATA:** Indicate the additional data of the menu item to be gotten or set.

The example is as follows:

```
MENUBUTTONITEM mbi;

mbi.which = MB_WHICH_TEXT | MB_WHICH_ATTDATA;
mbi.text = "newtext";
mbi.data = 1;
SendMessage (menubtn, MBM_SETITEMDATA, 0, (LPARAM) &mbi);
```

Here, **menubtn** is the handle of the menu button.

### 24.2.7 Other Messages

When using **MBS\_SORT** style, because item sorting is involved, MiniGUI also provides **MBM\_SETSTRCMPFUNC** message for the application to set a customized sorting function. Generally speaking, the application should use this message to set the new text string comparing function before adding items.

Usage of the message can refer to **LB\_SETSTRCMPFUNC** message of list box.

## 24.3 Notification Codes of Menu Button

The menu button has not **MBS\_NOTIFY** style, so any menu button control can generate the following notification codes:

- **MBN\_ERRSPACE:** Memory allocation error occurs, and the memory space is not enough.

- **MBN\_SELECTED**: Selection is made for the menu button control. Whether or not the selected menu item changes, this notification will be generated.
- **MBN\_CHANGED**: The selected item of the menu button control changes.
- **MBN\_STARTMENU**: The user activates the popup menu of a menu button.
- **MBN\_ENDMENU**: The popup menu closed.

## 24.4 Sample Program

List 24.1 gives an example for using menu button. This program section makes a little change on the program of List 23.1, i.e., a menu button replaces the pull-down combo box in program of List 23.1. Please refer to `menubutton.c` file of demo program package `mg-samples` of this guide for the complete source code. The running effect of the program is as shown in Fig. 24.1.

List 24.1 Example of using menu button

```

/* Define dialog box template */
static DLGTEMPLATE DlgMyDate =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    100, 100, 360, 150,
    "Meeting Plan",
    0, 0,
    9, NULL,
    0
};

static CTRLDATA CtrlMyDate[] =
{
    ...

    /* Realize the combo box for displaying the names of heroes with a menu button */
    {
        CTRL_MENUBUTTON,
        WS_CHILD | WS_VISIBLE,
        190, 20, 100, 20,
        IDL_DAXIA,
        "",
        0
    },
    ...
};

...

static void daxia_notif_proc (HWND hwnd, int id, int nc, DWORD add_data)
{
    if (nc == CBN_SELCHANGE) {
        /* Get the selected hero, and display his characteristics */
        int cur_sel = SendMessage (hwnd, MBM_GETCURITEM, 0, 0);
        if (cur_sel >= 0) {

```

```

        SetWindowText (GetDlgItem (GetParent(hwnd), IDC_PROMPT), daxia_char [cur_sel]
    );
    }
}

static void prompt (HWND hDlg)
{
    char date [1024];

    int hour = SendDlgItemMessage(hDlg, IDC_HOUR, CB_GETSPINVALUE, 0, 0);
    int min = SendDlgItemMessage(hDlg, IDC_MINUTE, CB_GETSPINVALUE, 0, 0);
    int sel = SendDlgItemMessage(hDlg, IDC_DAXIA, CB_GETCURITEM, 0, 0);

    sprintf (date, "You will meet %s at %02d:%02d", dacia [sel], hour, min);
    MessageBox (hDlg, date, "Meeting Plan", MB_OK | MB_ICONINFORMATION);
}

static int MyDateBoxProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    int i;
    switch (message) {
    case MSG_INITDIALOG:
        SendDlgItemMessage(hDlg, IDC_HOUR, CB_SETSPINFORMAT, 0, (LPARAM) "%02d");
        SendDlgItemMessage(hDlg, IDC_HOUR, CB_SETSPINRANGE, 0, 23);
        SendDlgItemMessage(hDlg, IDC_HOUR, CB_SETSPINVALUE, 20, 0);
        SendDlgItemMessage(hDlg, IDC_HOUR, CB_SETSPINPACE, 1, 1);

        SendDlgItemMessage(hDlg, IDC_MINUTE, CB_SETSPINFORMAT, 0, (LPARAM) "%02d");
        SendDlgItemMessage(hDlg, IDC_MINUTE, CB_SETSPINRANGE, 0, 59);
        SendDlgItemMessage(hDlg, IDC_MINUTE, CB_SETSPINVALUE, 0, 0);
        SendDlgItemMessage(hDlg, IDC_MINUTE, CB_SETSPINPACE, 1, 2);

        /* Add the names of the heroes to the menubutton */
        for (i = 0; i < 7; i++) {
            MENUITEM mbi;
            mbi.text = dacia[i];
            mbi.bmp = NULL;
            mbi.data = 0;
            SendDlgItemMessage(hDlg, IDC_DAXIA, CB_ADDITEM, -1, (LPARAM) &mbi);
        }

        /* Set the notification callback function of the menubutton */
        SetNotificationCallback (GetDlgItem (hDlg, IDC_DAXIA), dacia_notif_proc);
        SendDlgItemMessage(hDlg, IDC_DAXIA, CB_SETCURITEM, 0, 0);
        SetWindowText (GetDlgItem (hDlg, IDC_PROMPT), dacia_char [0]);
        return 1;

    case MSG_COMMAND:
        switch (wParam) {
        case IDOK:
            prompt (hDlg);
        case IDCANCEL:
            EndDialog (hDlg, wParam);
            break;
        }
        break;
    }

    return DefDlgProc (hDlg, message, wParam, lParam);
}

int MiniGUIMain (int argc, const char* argv[])
{
#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER, "menubutton", 0, 0);
#endif

    DlgMyDate.controls = CtrlMyDate;

    DialogBoxIndirectParam (&DlgMyDate, HWND_DESKTOP, MyDateBoxProc, 0L);

    return 0;
}

```

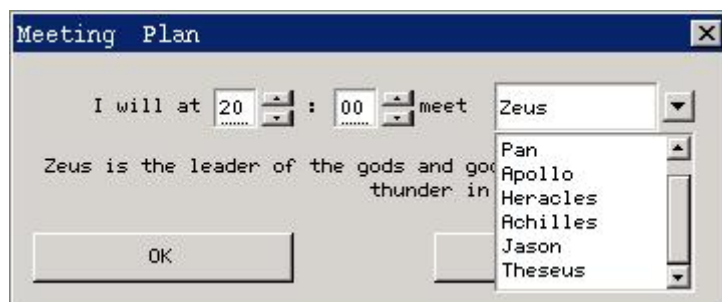


Fig. 24.2 Use of menu button





## 25 Progress Bar Control

The progress bar is generally used to prompt the progress of a task for the user, and is frequently used for tasks such as copying file, installing software. Calling `CreateWindow` function with `CTRL_PROGRESSBAR` as the control class name can create a progress bar. Fig. 25.1 is the typical running effect of a progress bar.

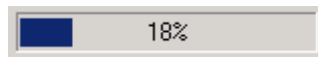


Fig. 25.1 Progress bar control

### 25.1 Styles of Progress Bar

Progress bar has only the following two styles available:

- **PBS\_NOTIFY**: The progress bar control with this style can generate notification messages.
- **PBS\_VERTICAL**: Display the progress bar vertically, as shown in Fig. 25.2.



Fig. 25.2 Vertical progress bar control

The combination of styles commonly used by a progress bar controls is:

`WS_CHILD | WS_VISIBLE | PBS_NOTIFY`

## 25.2 Messages of Progress Bar

### 25.2.1 Setting Range of Progress Bar

The range of a progress bar is 0 to 100 by default, and an application can set its own progress bar range by calling `PBM_SETRANGE` message:

```
SendMessage (hwndEdit, PBM_SETRANGE, min, max) ;
```

**[Prompt] The range of a progress bar can be set to be a negative value.**

### 25.2.2 Setting Step Value of Progress Bar

We can set the step value for a progress bar, and make the progress bar stepping forward when each stage task is complete. The step value is 10 by default, and can be changed by sending `PBM_SETSTEP` message, as shown in the following:

```
SendMessage (hwndEdit, PBM_SETSTEP, 5, 0) ;
```

The above message changes the step value of a progress bar to be 5.

**[Prompt] The step value of a progress bar can be set to be a negative value.**

You should set the position of the progress bar as the max value of its range when the step value is a negative and the progress bar will decrease from its max range to min.

### 25.2.3 Setting Position of Progress Bar

We can also set the current position of a progress bar optionally with `PBM_SETPOS` message:

```
SendMessage (hwndEdit, PBM_SETPOS, 50, 0) ;
```

The above message sets the current position of a progress bar to be 50.

#### 25.2.4 Setting Offset Based-on Current Position

We can also set the offset of the new position based on the current position to change the progress position:

```
SendMessage (hwndEdit, PBM_DELTAPOS, 10, 0) ;
```

The above message will add 10 to the new position based on the current position, i.e., new position is the current position plus 10.

**[Prompt] The offset of a progress bar can be set to be a negative value.**

#### 25.2.5 Advancing Position by One Step

`PBM_STEPIT` can be sent to advance the current position, and the new position equals the result of the current position plus the step value:

```
SendMessage (hwndEdit, PBM_STEPIT, 0, 0) ;
```

**[Note] The present progress bar control does not provide any messages for getting the current position, the current step increment, and the current position range.**

### 25.3 Notification Codes of Progress Bar

Progress bar with `PBS_NOTIFY` style may possibly generate the following notification codes:

- `PBN_REACHMAX`: Reach the maximum position.
- `PBN_REACHMIN`: Reach the minimum position.

## 25.4 Sample Program

List 25.1 gives an example of using the progress bar control. This program provides two functions. Calling `createProgressWin` function will create a main window with a progress bar and then return. We can control the progress bar of the main window in our own program, and call `destroyProgressWin` function to destroy the progress main window after completing the task. The two functions actually come from MiniGUIExt library of MiniGUI. List 25.1 gives the example of the implementation and the usage of these two functions, and the running effect is as shown in Fig. 25.3. Please refer to `progressbar.c` of the sample program package `mg-samples` of this guide for the complete source code.

List 25.1 Example of using progress bar

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

static HWND createProgressWin (HWND hParentWnd, char * title, char * label,
                               int id, int range)
{
    HWND hwnd;
    MAINWINCREATE CreateInfo;
    int ww, wh;
    HWND hStatic, hProgBar;

    /* Calculate the window width according to the width of the window client region */
    ww = ClientWidthToWindowWidth (WS_CAPTION | WS_BORDER, 400);
    /* Calculate the window height according to the height of the window client region */
    wh = ClientHeightToWindowHeight (WS_CAPTION | WS_BORDER,
                                     (range > 0) ? 70 : 35, FALSE);

    /* Create the main window */
    CreateInfo.dwStyle = WS_ABSSCRPOS | WS_CAPTION | WS_BORDER | WS_VISIBLE;
    CreateInfo.dwExStyle = WS_EX_NONE;
    CreateInfo.spCaption = title;
    CreateInfo.hMenu = 0;
    CreateInfo.hCursor = GetSystemCursor(IDC_WAIT);
    CreateInfo.hIcon = 0;
    /* The window procedure of the main window is set to
     * be the default main window procedure
     */
    CreateInfo.MainWindowProc = DefaultMainWinProc;
#ifdef _LITE_VERSION
    CreateInfo.lx = (GetGDCapability (HDC_SCREEN, GDCAP_MAXX) - ww) >> 1;
```

```

    CreateInfo.ty = (GetGDCapability (HDC_SCREEN, GDCAP_MAXY) - wh) >> 1;
#else
    CreateInfo.lx = g_rcExcluded.left + (RECTW(g_rcExcluded) - ww) >> 1;
    CreateInfo.ty = g_rcExcluded.top + (RECTH(g_rcExcluded) - wh) >> 1;
#endif
    CreateInfo.rx = CreateInfo.lx + ww;
    CreateInfo.by = CreateInfo.ty + wh;
    CreateInfo.iBkColor = COLOR_lightgray;
    CreateInfo.dwAddData = 0L;
    CreateInfo.hHosting = hParentWnd;

    hwnd = CreateMainWindow (&CreateInfo);
    if (hwnd == HWND_INVALID)
        return hwnd;

    /* Create a static control for prompting in the main window */
    hStatic = CreateWindowEx ("static",
        label,
        WS_VISIBLE | SS_SIMPLE,
        WS_EX_USEPARENTCURSOR,
        IDC_STATIC,
        10, 10, 380, 16, hwnd, 0);

    /* Create the progress bar control in the main window */
    if (range > 0) {
        hProgBar = CreateWindowEx ("progressbar",
            NULL,
            WS_VISIBLE,
            WS_EX_USEPARENTCURSOR,
            id,
            10, 30, 380, 30, hwnd, 0);
        SendDlgItemMessage (hwnd, id, PBM_SETRANGE, 0, range);
    }
    else
        hProgBar = HWND_INVALID;

    /* Update the controls */
    UpdateWindow (hwnd, TRUE);

    /* Return the handle of the main window */
    return hwnd;
}

static void destroyProgressWin (HWND hwnd)
{
    /* Destroy the controls and the main window */
    DestroyAllControls (hwnd);
    DestroyMainWindow (hwnd);
    ThrowAwayMessages (hwnd);
    MainWindowThreadCleanup (hwnd);
}

int MiniGUIMain (int argc, const char* argv[])
{
    int i, sum;
    HCURSOR hOldCursor;
    HWND hwnd;

#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER , "progressbar" , 0 , 0);
#endif

    /* Set "sandglass" mouse to indicate the system is busy */
    hOldCursor = SetDefaultCursor (GetSystemCursor (IDC_WAIT));

    /* Create the progressbar window, and specify the
     * identifier and range of the progress bar control */
    hwnd = createProgressWin (HWND_DESKTOP, "Progress Bar",
        "Calculating, please waiting...", 100, 2000);

    while (HavePendingMessage (hwnd)) {
        MSG msg;
        GetMessage (&msg, hwnd);
        DispatchMessage (&msg);
    }
}

```

```

/* Begin the long time calculating progress, and
 * refresh the position of the progressbar when completing
 * the external loop.
 */
for (i = 0; i < 2000; i++) {
    unsigned long j;

    if (i % 100 == 0) {
        SendDlgItemMessage (hwnd, 100, PBM_SETPOS, i, 0L);
        while (HavePendingMessage (hwnd)) {
            MSG msg;
            GetMessage (&msg, hwnd);
            DispatchMessage (&msg);
        }
    }

    sum = i*5000;
    for (j = 0; j < 500000; j++)
        sum *= j;
    sum += sum;
}

/* Destroy the progressbar window */
destroyProgressWin (hwnd);
/* Recover the original mouse */
SetDefaultCursor (hOldCursor);

return 0;
}

#ifdef _LITE_VERSION
#include <minigui/dti.c>
#endif

```

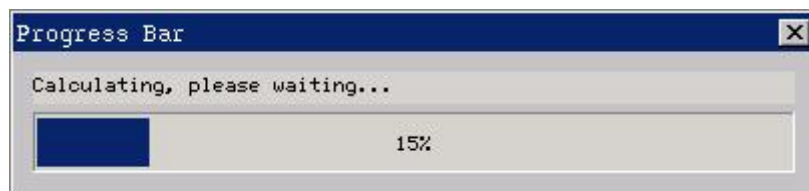


Fig. 25.3 Example of progress bar control

## 26 Track Bar Control

The track bar is generally used for adjusting brightness, volume, etc. In the situation for adjusting the value in a range, track bar can be used. Calling `CreateWindow` function with `CTRL_TRACKBAR` as the control class name can create a track bar. Fig. 26.1 shows the typical running effect of a track bar.

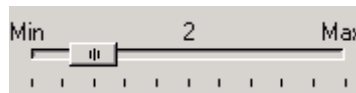


Fig. 26.1 Track bar control

### 26.1 Styles of Track Bar

The frequently used combination of styles for a track bar is:

```
WS_CHILD | WS_VISIBLE | TBS_NOTIFY
```

Specifying `TBS_NOTIFY` style can make a track bar generating notification messages.

A track bar is horizontal by default. To create a vertical track bar, you can specify `TBS_VERTICAL` style. The track bar in Fig. 26.2 is a vertical track bar.



Fig. 26.2 A vertical track bar

Other styles of track bar are illustrated as follow:

- **TBS\_TIP**: Display the tip string beside the track bar (as “Min” and “Max” in Fig. 26.1). A track bar with this style will also display the current position value in the center of the control.
- **TBS\_NOTICK**: Do not display the tick lines.
- **TBS\_BORDER**: This style makes a track bar having bordered, which is not used frequently.

## 26.2 Messages of Track Bar

Messages of a track bar are relatively simple, as summarized in the following:

- **TBM\_SETRANGE**: Set the minimum and maximum positions for the track bar through **wParam** and **lParam** parameters. The default range is 0~10.
- **TBM\_GETMIN**: Get the minimum position for the track bar.
- **TBM\_GETMAX**: Get the maximum position for the track bar.
- **TBM\_SETMIN**: Set the minimum position for the track bar.
- **TBM\_SETMAX**: Set the maximum position for the track bar.
- **TBM\_SETLINESIZE**: Sets the step value by **wParam** parameter. When the user presses up or down arrow key during the track bar has input focus, the slider will be moved up or down the step value. The default step value is 1.
- **TBM\_GETLINESIZE**: Gets the step value of the slider.
- **TBM\_SETPAGESIZE**: Set the page step value by **wParam** parameter. When the user presses **PageUp** or **PageDown** key during the track bar has input focus, the slider will be moved up or down the page step value. The default page step value is 1.
- **TBM\_GETPAGESIZE**: Get the page step value of the track bar.
- **TBM\_SETPOS**: Set the position of the slider.
- **TBM\_GETPOS**: Get the position of the slider.
- **TBM\_SETTICKFREQ**: Set the interval value of tick marks in a track bar, the default interval is 1.
- **TBM\_GETTICKFREQ**: Get the interval value of tick marks in a track bar.
- **TBM\_SETTIP**: Set the tip string at the minimum and maximum positions.
- **TBM\_GETTIP**: Set the tip strings at the minimum and maximum



positions.

## 26.3 Notification Codes of Track Bar

The track bar with `TBS_NOTIFY` style may generate the following notification codes:

- `TBN_CHANGE`: The position of the slider has changed.
- `TBN_REACHMAX`: The slider has reached the maximum position.
- `TBN_REACHMIN`: The slider has reached the minimum position.

## 26.4 Sample Program

List 26.1 gives an example program of track bar. The program draws a circle with the corresponding size according to the current slider position. When the user changes the position of the slider, the circle will also be refreshed. The running effect of the program is shown in Fig. 26.3. Please refer to `trackbar.c` file of the demo program package `mg-samples` of this guide for the complete source code.

List 26.1 Use of track bar

```
static int radius = 10;
static RECT rcCircle = {0, 60, 300, 300};

static void my_notif_proc (HWND hwnd, int id, int nc, DWORD add_data)
{
    if (nc == TBN_CHANGE) {
        /* When the position of the slider has changed,
         * save the current position, and inform the main window to redraw */
        radius = SendMessage (hwnd, TBM_GETPOS, 0, 0);
        InvalidateRect (GetParent (hwnd), &rcCircle, TRUE);
    }
}

static int TrackBarWinProc(HWND hwnd, int message, WPARAM wParam, LPARAM lParam)
{
    HWND hwnd;
    switch (message) {
        case MSG_CREATE:
            /* Create a track bar */
            hwnd = CreateWindow (CTRL_TRACKBAR,
                                "",
                                WS_VISIBLE | TBS_NOTIFY,
                                100,
                                10, 10, 280, 50, hwnd, 0);

            /* Set the range, step vaue, interval value and current position */
            SendMessage (hwnd, TBM_SETRANGE, 0, 100);
            SendMessage (hwnd, TBM_SETLINESIZE, 1, 0);
    }
}
```

```

SendMessage (hwnd, TBM_SETPAGESIZE, 10, 0);
SendMessage (hwnd, TBM_SETTICKFREQ, 10, 0);
SendMessage (hwnd, TBM_SETPOS, radius, 0);

/* Set the notification callback function of the slider */
SetNotificationCallback (hwnd, my_notif_proc);
break;

case MSG_PAINT:
{
    HDC hdc = BeginPaint (hWnd);

    /* Draw a circle with radius of the the current position of the slider */
    ClipRectIntersect (hdc, &rcCircle);
    Circle (hdc, 140, 120, radius);

    EndPaint (hWnd, hdc);
    return 0;
}

case MSG_DESTROY:
    DestroyAllControls (hWnd);
    return 0;

case MSG_CLOSE:
    DestroyMainWindow (hWnd);
    PostQuitMessage (hWnd);
    return 0;
}

return DefaultMainWinProc (hWnd, message, wParam, lParam);
}

/* Following codes to create the main window are omitted */

```

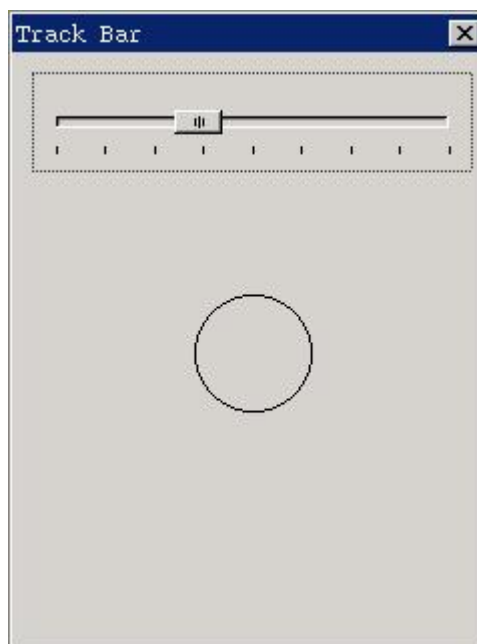


Fig. 26.3 Use of track bar

## 27 Toolbar Control

The use of toolbar is frequently seen in modern GUI applications. MiniGUI prepares the predefined toolbar control class for the application. In fact, MiniGUI provides three different predefined toolbar control classes, namely `CTRL_TOOLBAR`, `CTRL_NEWTOOLBAR`, and `CTRL_COOLBAR` (in MiniGUIExt library) control classes.

`CTRL_TOOLBAR` is the early toolbar control class, which is obsolete and replaced by `CTRL_NEWTOOLBAR` control class. Including `CTRL_TOOLBAR` class in MiniGUI is just for compatibility, and new applications should not use it. We introduce `CTRL_NEWTOOLBAR` control class in this chapter, and introduce `CTRL_COOLBAR` control class in Chapter 35.

Calling `CreateWindow` function with `CTRL_NEWTOOLBAR` as the control class name can create a toolbar control. The running effect of this control is shown in Fig. 27.1.

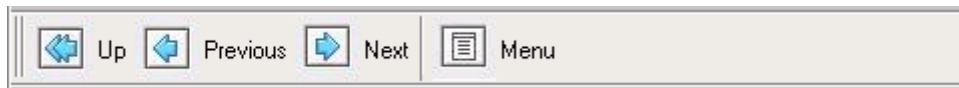


Fig. 27.1 Toolbar control

### 27.1 Creating Toolbar Control

Before creating a toolbar control, we should fill a `NTBINFO` structure first, and pass the pointer to the structure to the window procedure of the control class through `dwAddData` argument of `CreateWindow` function. `NTBINFO` structure is mainly used to define the information of the bitmap used by a toolbar, as shown in Table 27.1.

Table 27.1 NTBINFO structure

Member	Meaning	Note
image	Image for displaying the bitmaps of the buttons.	

<b>nr_cells</b>	Number of bitmap cells in the image, that is to say, number of lines in total	
<b>nr_cols</b>	Number of cell columns in the image, that is to say, number of states of each bitmap cell	1 means the cells have only normal state; 2 means the cells have normal and highlight states; 3 means the cells have not disabled state. 4 or 0 means the cells have all the four possible states.
<b>w_cell</b>	Width of bitmap cell	If w_cell is zero, it will be equal to (width_of_image / nr_cols)
<b>h_cell</b>	Height of bitmap cell	If h_cell is zero, it will be equal to (height_of_image / nr_cells)

We must organize the bitmaps on the toolbar buttons (we called bitmap cells) in a single bitmap object, and should organize them into a structure as shown in Fig. 27.2. Here, the first column illustrates the normal states of all the bitmap cells possibly used by the toolbar buttons; the second column illustrates the highlighted states of all the bitmap cells possibly used by the toolbar buttons; the third column illustrates the pushed state, and the fourth column illustrates the disabled (grayed) state. Each row of the bitmap illustrates all the states of a single button bitmap cell.



Fig. 27.2 Bitmap object used for a toolbar control

The toolbar control selects the appropriate bitmap cell in the bitmap object to display the button according to the button state in the toolbar.

## 27.2 Styles of Toolbar

The toolbar control class supports the following styles:

- **NTBS\_HORIZONTAL**: Display toolbar horizontally. This is the default style.
- **NTBS\_VERTICAL**: Display toolbar vertically, as shown in Fig. 27.3.
- **NTBS\_MULTLINE**: Toolbar can display multiple lines. When type of tool item is **NTBIF\_NEWLINE**, it will display the added tool item in another

row or column, as shown in Fig. 27.3.

- **NTBS\_WITHTEXT**: Display the text under or at the right side of the button, and display the text under the button bitmap by default. In this case, the application must specify the corresponding text when adding a button. The bitmap will be popped when the button is active with this style.
- **NTBS\_TEXTRIGHT**: When using it with **NTBS\_WITHTEXT** style, this style specifies the text to display at the right side of the button bitmap. The toolbar in Fig. 27.1 has **NTBS\_TEXTRIGHT** style. The bitmap and text will be popped when the button is active with this style.
- 
- **NTBS\_DRAWSTATES**: Draws the button states with different 3D frames, and does not use the highlight, pushed and disabled bitmap cell.
- **NTBS\_DRAWSEPARATOR**: Draws the separator bar. The separator bar used to separate buttons in the toolbar will not be drawn by default, but increase the distance between two buttons. When the toolbar has this style, it will draw a narrow separator bar.



Fig. 27.3 Toolbar control display vertically and in multiple columns

## 27.3 Messages of Toolbar

### 27.3.1 Adding an Item

An item can be added by sending **NTBM\_ADDITEM** message and passing **NTBITEMINFO** structure to the toolbar control. Table 27.2 gives the meanings of

the members of **NTBITEMINFO**.

Table 27.2 NTBITEMINFO structure

Members	Meanings	Note
which	Used for <b>NTBM_GETITEM</b> and <b>NTBM_SETITEM</b> messages.	
flags	This member is used to specify the type and state of an item. The type can be one of the following values: <ul style="list-style-type: none"> <li>■ <b>NTBIF_PUSHBUTTON</b>: A normal push button</li> <li>■ <b>NTBIF_CHECKBUTTON</b>: A check box button</li> <li>■ <b>NTBIF_HOTSPOTBUTTON</b>: A button with hotspot</li> <li>■ <b>NTBIF_NEWLINE</b>: A new line separator when the style <b>NTBS_MULTILINE</b> specified.</li> <li>■ <b>NTBIF_SEPARATOR</b>: A separator.</li> </ul> The item has only one state, namely <b>NTBIF_DISABLED</b> , indicating the item is disabled.	The value of this member should be OR'd value by one of the type identifier and the state identifier.
id	Identifier of the button. When the user clicked a button, the identifier would be sent as the notification code of the notification message of the toolbar to the parent window or the notification callback function.	
text	This member will be used to pass the text of the button when the control has <b>NTBS_WITHTEXT</b> style.	
tip	Not used and reserved.	
bmp_cell	Specify which bitmap cell in the bitmap object is used by the button. This index of first bitmap cell is zero.	This button will use the bitmap cell in the <b>bmp_cell</b> line to display the state of the button.
hotspot_proc	If the item is a button with hotspot, this member defines the callback function when the user clicked the hotspot.	
rc_hotspot	If the item is a button with hotspot, this member defines the rectangle region of the hotspot in the button, relative to the upper-left corner of the button.	When the user clicked the hotspot rectangle region, it is regarded as activating the hotspot.
add_data	Additional data of the item.	

The member will be ignored when adding an item. The following program illustrates how to add a normal push button, an initially grayed button, an separator bar, and a button with hotspot to a toolbar control:

```

HWND ntb1;
NTBINFO ntb_info;
NTBITEMINFO ntbii;
RECT hotspot = {16, 16, 32, 32};

/* Fill NTBINFO Structure */
ntb_info.nr_cells = 4;
ntb_info.w_cell = 0;
ntb_info.h_cell = 0;
ntb_info.nr_cols = 0;
ntb_info.image = &bitmap1;

```

```

/* Create the toolbar control */
ntbl = CreateWindow (CTRL_NEWTOOLBAR,
    "",
    WS_CHILD | WS_VISIBLE,
    IDC_CTRL_NEWTOOLBAR_1,
    0, 10, 1024, 0,
    hWnd,
    (DWORD) &ntb_info);

/* Add a normal button item */
ntbii.flags = NTBIF_PUSHBUTTON;
ntbii.id = IDC NTB_TWO;
ntbii.bmp_cell = 1;
SendMessage (ntbl, TBM_ADDITEM, 0, (LPARAM)&ntbii);

/* Add a grayed button item */
ntbii.flags = NTBIF_PUSHBUTTON | NTBIF_DISABLED;
ntbii.id = IDC NTB_THREE;
ntbii.bmp_cell = 2;
SendMessage (ntbl, TBM_ADDITEM, 0, (LPARAM)&ntbii);

/* Add a separator */
ntbii.flags = NTBIF_SEPARATOR;
ntbii.id = 0;
ntbii.bmp_cell = 0;
ntbii.text = NULL;
SendMessage (ntbl, TBM_ADDITEM, 0, (LPARAM)&ntbii);

/* Add a button with hotspot */
ntbii.flags = NTBIF_HOTSPOTBUTTON;
ntbii.id = IDC NTB_FOUR;
ntbii.bmp_cell = 3;
ntbii.rc_hotspot = hotspot;
ntbii.hotspot_proc = my_hotspot_proc;
SendMessage (ntbl, TBM_ADDITEM, 0, (LPARAM)&ntbii);

```

### 27.3.2 Getting/Setting Information of an Item

Using **NTBM\_GETITEM** or **NTBM\_SETITEM** message can get or set the information of an item through its identifier. Use of the two messages is similar to **NTBM\_ADDITEM**, and the difference is **wParam** as the ID of item and **lParam** as point to a **NTBITEMINFO** structure that **which** member in specifies the item information to be gotten or set. The member which can be the OR'd of the following values:

- **MTB\_WHICH\_FLAGS**: Get or set the flag of an item, which is the **flags** field in **NTBITEMINFO** structure.
- **MTB\_WHICH\_ID**: Get or set the identifier of an item
- **MTB\_WHICH\_TEXT**: Get or set the item text. Note that when getting text, you must ensure that enough buffer to be passed by **text** field. For safety consideration, it should ensure that the size of the buffer is at least "**NTB\_TEXT\_LEN+1**".
- **MTB\_WHICH\_CELL**: Get or set the bitmap cell index used by the item.

- **MTB\_WHICH\_HOTSPOT**: Get or set the hotspot rectangle of the item.
- **MTB\_WHICH\_ADDDATA**: Get or set the additional data of the item.

For convenience, MiniGUI also provides **NTBM\_ENABLEITEM** message to enable or disable an item with the specified identifier, as shown in the following:

```
SendMessage (ntb1, NTBM_ENABLEITEM, 100, FALSE);
```

The above code disables (grays) the item with identifier of 100 in **ntb1** toolbar control.

### 27.3.3 NTBM\_SETBITMAP Message

An application can send **NTBM\_SETBITMAP** message to a toolbar control, in order to change the button bitmap on the toolbar. When sending the message, an application uses **lParam** parameter to transfer a new **NTBINFO** structure, in which the new toolbar button bitmap is defined, as shown in the following code:

```
NTBINFO ntbi;
SendMessage (ntb, NTBM_SETBITMAP, 0, (LPARAM)&ntbi);
```

## 27.4 Notification Codes of Toolbar

Toolbars will not generate a notification message until the user clicks a button. The notification code of the toolbar is the identifier of the button clicked by the user. When the user clicks the button with hotspot, the toolbar control will call directly the hotspot callback function corresponding to the button, and will not generate a notification message.

## 27.5 Sample Program

List 27.1 gives a sample program of toolbar control. This program creates a toolbar with three buttons. We can control the position of the circle in the window by clicking the left item and right item on the toolbar. The other item



grayed initially is only for demonstration and has no function. The running effect of this program is shown in Fig. 27.4. Please refer to `newtoolbar.c` of the demo program package `mg-samples` of this guide for the complete source code.

List 27.1 Sample program of toolbar control

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

#define IDC NTB_LEFT 100
#define IDC NTB_RIGHT 110
#define IDC NTB_UP 120

static int offset = 0;
static RECT rcCircle = {0, 40, 300, 300};

static void my_notif_proc (HWND hwnd, int id, int nc, DWORD add_data)
{
    /* When the user clicked the left or right button, the circle in the window
     * will move left or right correspondingly
     */
    if (nc == IDC NTB_LEFT) {
        offset -= 10;
        InvalidateRect (GetParent (hwnd), &rcCircle, TRUE);
    }
    else if (nc == IDC NTB_RIGHT) {
        offset += 10;
        InvalidateRect (GetParent (hwnd), &rcCircle, TRUE);
    }
}

static BITMAP ntb_bmp;

static void create_new_toolbar (HWND hWnd)
{
    HWND ntb;
    NTBINFO ntb_info;
    NTBITEMINFO ntbii;
    gal_pixel pixel;

    ntb_info.nr_cells = 4;
    ntb_info.w_cell = 0;
    ntb_info.h_cell = 0;
    ntb_info.nr_cols = 0;
    ntb_info.image = &ntb_bmp;

    /* Create the toolbar control */
    ntb = CreateWindow (CTRL_NEWTOOLBAR,
        "",
        WS_CHILD | WS_VISIBLE,
        100,
        0, 0, 1024, 0,
        hWnd,
        (DWORD) &ntb_info);

    /* Set notification callback */
    SetNotificationCallback (ntb, my_notif_proc);

    /* Set the background color of the toolbar control,
     * and make it consistent with the background of the button bitmap
     */
}
```

```

    */
    pixel = GetPixelInBitmap (&ntb_bmp, 0, 0);
    SetWindowBkColor (ntb, pixel);
    InvalidateRect (ntb, NULL, TRUE);

    /* Add two normal button items */
    memset (&ntbii, 0, sizeof (ntbii));
    ntbii.flags = NTBIF_PUSHBUTTON;
    ntbii.id = IDC NTB_LEFT;
    ntbii.bmp_cell = 1;
    SendMessage (ntb, TBM_ADDITEM, 0, (LPARAM)&ntbii);

    ntbii.flags = NTBIF_PUSHBUTTON;
    ntbii.id = IDC NTB_RIGHT;
    ntbii.bmp_cell = 2;
    SendMessage (ntb, TBM_ADDITEM, 0, (LPARAM)&ntbii);

    /* Add a separator */
    ntbii.flags = NTBIF_SEPARATOR;
    ntbii.id = 0;
    ntbii.bmp_cell = 0;
    ntbii.text = NULL;
    SendMessage (ntb, TBM_ADDITEM, 0, (LPARAM)&ntbii);

    /* Add an initially disabled button */
    ntbii.flags = NTBIF_PUSHBUTTON | NTBIF_DISABLED;
    ntbii.id = IDC NTB_UP;
    ntbii.bmp_cell = 0;
    SendMessage (ntb, TBM_ADDITEM, 0, (LPARAM)&ntbii);
}

static int ToolBarWinProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_CREATE:
            /* Load the bitmap object used by the toolbar */
            if (LoadBitmap (HDC_SCREEN, &ntb_bmp, "new2.jpg"))
                return -1;

            create_new_toolbar (hWnd);
            break;

        case MSG_PAINT:
        {
            HDC hdc = BeginPaint (hWnd);

            ClipRectIntersect (hdc, &rcCircle);

            /* Draw a red circle */
            SetBrushColor (hdc, PIXEL_red);
            FillCircle (hdc, 140 + offset, 120, 50);

            EndPaint (hWnd, hdc);
            return 0;
        }

        case MSG_DESTROY:
            UnloadBitmap (&ntb_bmp);
            DestroyAllControls (hWnd);
            return 0;

        case MSG_CLOSE:
            DestroyMainWindow (hWnd);
            PostQuitMessage (hWnd);
            return 0;
        }

    return DefaultMainWinProc (hWnd, message, wParam, lParam);
}

/* Following codes to create the main window are omitted */

```

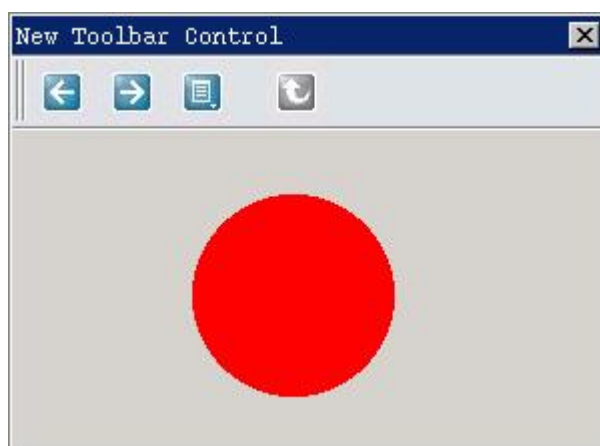


Fig. 27.4 Use of toolbar control



## 28 Property Sheet Control

The most familiar usage of property sheet is to place the interaction content belonging to different dialog boxes into one dialog box according to their catalogues. This can save space of the dialog box on the one hand, and can make the interaction interface more convenient to use on the other hand.

Figure 28.1 is a typical use of the property sheet control of MiniGUI.

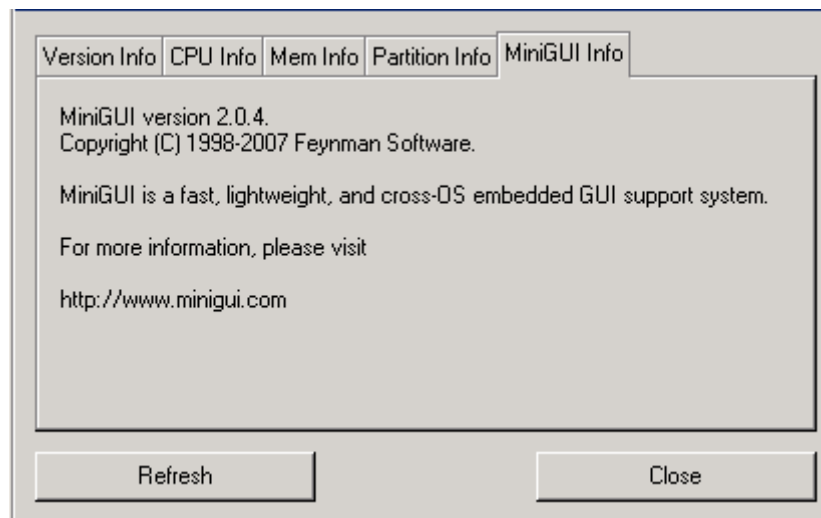


Fig. 28.1 Property sheet control

The property sheet is comprised of a group of property pages. Each property page has a tab, and we can click the tab to switch among different property pages. We can interpret the property page as a container control, in which other controls can be contained. From the point of view of an application developer, we can also interpret the property page as a dialog box in a dialog box - each property page has its own window procedure, and we can use the method similar to create a dialog box, i.e. the method to define a dialog box template, to add a property page into a property sheet control.

In an application, calling `CreateWindow` with `CTRL_PROPSHEET` as the control class name can create a property sheet control.

## 28.1 Styles of Property Sheet

At present, there are only the following two styles of property sheet, which are used to control the width of a tab of the property sheet:

- **PSS\_SIMPLE**: All tabs of the control will have the same width.
- **PSS\_COMPACTTAB**: The width of a tab depends on the length of the tab title of the property sheet.
- **PSS\_SCROLLABLE**: The width of a tab depends on the length of the tab title of the property sheet. There will be two of navigation button for scroll the tabs when the count of tab is too many.
- **PSS\_BOTTOM**: Tabs are displayed at the bottom of the property sheet. This style can be used to with other three styles.

## 28.2 Messages of Property Sheet

### 28.2.1 Adding Property Page

After the property sheet control has been created, we can send **PSM\_ADDPAGE** message to add a property page to the property sheet. **WParam** of the message is used to pass the dialog box template, and **LParam** is used to pass the window procedure function of the property page, as shown in the following code:

```
HWND pshwnd = GetDlgItem (hDlg, IDC_PROPSHEET);

/* Prepare the dialog box template */
DlgStructParams.controls = CtrlStructParams;

/* Add a property page */
SendMessage (pshwnd, PSM_ADDPAGE, (WPARAM)&DlgStructParams, (LPARAM)PageProc1);
```

The return value of this message is the index of the newly added property page, and the index is base-on zero.

### 28.2.2 Procedure Function of Property Page

Similar to the dialog box, each property page has its own procedure function to handle the related messages of the property page. The prototype of the

procedure function is the same with a normal window procedure function, but the followings are different:

- The procedure function of a property page should call `DefaultPageProc` function for the message needing default handling.
- The procedure function of a property page need handle two messages specific to the property page: `MSG_INITPAGE` and `MSG_SHOWPAGE`. The former is similar to `MSG_INITDIALOG` message of a dialog box; and the latter is sent to the procedure of the property page when the property page is hidden or showed, where `lParam` parameter is `SW_HIDE` and `SW_SHOW`, respectively. When the property page is displayed, the procedure function of the property page returns 1 to make the first control with `WS_TABSTOP` has the input focus.
- If you send `PSM_SHEETCMD` message to the property sheet control, the control will broadcast `MSG_SHEETCMD` message to all pages it contains. At this time, the page callback procedure can check the validity of the user input and save the valid input. If the input is invalid or other problems occur, the page can return -1 to break the continued broadcast of this message. After receiving a non-zero value from any property page, the property sheet control will make `PSM_SHEETCMD` message return a non-zero value, and this is equal to the page index plus one. In such a way, we can know which page includes invalid input during handling the property pages of the property sheet, and then terminate the handling and switch to this property page.

List 28.1 gives a typical procedure function of a property page, and the procedure function of the dialog box of the property sheet. When the user clicks the "OK" button of the dialog box containing the property sheet, the dialog box sends `PSM_SHEETCMD` message to the property sheet control, and according to the return value of the message, determines to close the dialog box or switch to a property page to correct the invalid input. After receiving `MSG_SHEETCMD` message, the procedure function of the property page will determine whether the user input is valid, and returns 0 or -1 correspondingly.

List 28.1 A typical procedure function of property page and a procedure function of dialog box containing property sheet

```
static int PageProc1 (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_INITPAGE:
            break;

        case MSG_SHOWPAGE:
            return 1;

        case MSG_SHEETCMD:
            if (wParam == IDOK) {
                char buffer [20];
                GetDlgItemText (hDlg, IDC_EDIT1, buffer, 18);
                buffer [18] = '\0';

                /* When the user clicked the "OK" button in
                 * the dialog box containing the property sheet,
                 * determine whether the user input is valid
                 */
                if (buffer [0] == '\0') {
                    MessageBox (hDlg,
                                "Please input something in the first edit box.",
                                "Warning!",
                                MB_OK | MB_ICONEXCLAMATION | MB_BASEDONPARENT);
                    /* The user input is invalid, return a non-zero value */
                    return -1;
                }
            }
            return 0;

        case MSG_COMMAND:
            switch (wParam) {
                case IDOK:
                case IDCANCEL:
                    break;
            }
            break;
    }

    return DefaultPageProc (hDlg, message, wParam, lParam);
}

static int PropSheetProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_INITDIALOG:
            {
                HWND pshwnd = GetDlgItem (hDlg, IDC_PROPSHEET);

                /* Add property pages to the property sheet control */

                DlgStructParams.controls = CtrlStructParams;
                SendMessage (pshwnd, PSM_ADDPAGE,
                            (WPARAM)&DlgStructParams, (LPARAM) PageProc1);

                DlgPassword.controls = CtrlPassword;
                SendMessage ( pshwnd, PSM_ADDPAGE,
                            (WPARAM)&DlgPassword, (LPARAM) PageProc2);

                DlgStartupMode.controls = CtrlStartupMode;
                SendMessage ( pshwnd, PSM_ADDPAGE,
                            (WPARAM)&DlgStartupMode, (LPARAM) PageProc3);

                DlgInitProgress.controls = CtrlInitProgress;
                SendMessage ( pshwnd, PSM_ADDPAGE,
                            (WPARAM)&DlgInitProgress, (LPARAM) PageProc4);

                break;
            }

        case MSG_COMMAND:
            switch (wParam)
            {

```



```

    case IDC_APPLY:
        break;

    case IDOK:
    {
        /* Send PSM_SHEETCMD messaeg to the property sheet control to
        * inform it that the "OK" button is clicked
        */
        int index = SendDlgItemMessage (hDlg, IDC_PROPSHEET,
                                         PSM_SHEETCMD, IDOK, 0);
        if (index) {
            /* A property page returns a non-zero value,
            * switch to this property page and prompt to input again
            */
            SendDlgItemMessage (hDlg, IDC_PROPSHEET,
                                PSM_SETACTIVEINDEX, index - 1, 0);
        }
        else
            /* Every thing is ok, close the dialog box */
            EndDialog (hDlg, wParam);

        break;
    }
    case IDCANCEL:
        EndDialog (hDlg, wParam);
        break;
    }
    break;
}

return DefaultDialogProc (hDlg, message, wParam, lParam);
}

```

### 28.2.3 Deleting Property Page

To delete a property page, you need only send **PSM\_REMOVEPAGE** message to the property sheet control, and pass the index of the property page to be deleted through **wParam** parameter:

```
SendDlgItemMessage (hDlg, IDC_PROPSHEET, PSM_REMOVEPAGE, 0, 0);
```

This message will delete the first property page in the property sheet.

**[Note] Deleting a property page may change the indices of other property pages.**

### 28.2.4 Handle and Index of Property Page

The handle of a property page is actually the handle of the parent window of the controls in the property page, i.e. the window handle passed by the procedure function of the property page, and this window is actually a child window of the property sheet control. Sending **PSM\_GETPAGE** message to the

property sheet control can get the handle of a property page with a certain index:

```
hwnd = SendDlgItemMessage (hDlg, IDC_PROPSHEET, PSM_GETPAGE, index, 0);
```

This message will return the handle of the property page with index value of "index". While the following message call returns the index of a page based on the handle of the property page:

```
index = SendDlgItemMessage (hDlg, IDC_PROPSHEET, PSM_GETPAGEINDEX, hwnd, 0);
```

After getting the handle of the property page, we can call functions such as `CreateWindow` to add a new control to the page conveniently. Of course, in the procedure function of the property page, we can complete the similar task.

### 28.2.5 Messages Relevant Property Page

MiniGUI provides the following messages to get the relevant information of property pages:

- **PSM\_GETPAGECOUNT** returns the number of pages in the property sheet.
- **PSM\_GETTITLELENGTH** gets the length of the page title according to the index value passed by **wParam** parameter, like the **MSG\_GETTEXTLENGTH** message of a window.
- **PSM\_GETTITLE** gets the page title according to the index value passed by **wParam** parameter, and save it in the buffer passed by **lParam** parameter, like the **MSG\_GETTEXT** message of a window.
- **PSM\_SETTITLE** sets the property page title according to the string passed by **lParam**, like **MSG\_SETTEXT** message of a window.

The active property page is the property page currently displayed in the property sheet, and every time only one-property page is displayed in the property sheet. MiniGUI provides the following messages to handle the active property page:

- **PSM\_GETACTIVEPAGE** returns the handle of the current active page.

- **PSM\_GETACTIVEINDEX** returns the index of the current active page
- **PSM\_SETACTIVEINDEX** sets the active property page according to the index passed by **wParam**.

## 28.3 Notification Codes of Property Sheet

At present, there is only one notification code for property sheet control:

- **PSN\_ACTIVE\_CHANGED**: When the active property page of the property sheet changed, the property sheet control will generate this notification code.

## 28.4 Sample Program

List 28.2 gives a sample program for property sheet control. This program displays some system information of the computer, such as CPU type, memory size, etc. The running effect of this program is shown in Fig. 28.2. Please refer to **propsheet.c** of the sample program package of this guide for the complete source code.

List 28.2 A sample program of property sheet control

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
#include <minigui/control.h>

#define PAGE_VERSION    1
#define PAGE_CPU        2
#define PAGE_MEMINFO    3
#define PAGE_PARTITION  4
#define PAGE_MINIGUI    5

#define IDC_PROPSHEET   100
#define IDC_SYSINFO     100

/* Define the template of the system information property page */
static DLGTEMPLATE PageSysInfo =
{
    WS_NONE,
    WS_EX_NONE,
    0, 0, 0, 0,
    "",
    0, 0,
    1, NULL,
```

```

0
};

/* There is only one static control to display the information
 * in the system information property page
 */
static CTRLDATA CtrlSysInfo [] =
{
    {
        CTRL_STATIC,
        WS_VISIBLE | SS_LEFT,
        10, 10, 370, 160,
        IDC_SYSINFO,
        "test\\ntest\\ntest\\ntest\\ntest\\n",
        0
    }
};

/* Read the system information from the specified file */
static size_t read_sysinfo (const char* file, char* buff, size_t buf_len)
{
    size_t size;
    FILE* fp = fopen (file, "r");

    if (fp == NULL) return 0;

    size = fread (buff, 1, buf_len, fp);

    fclose (fp);
    return size;
}

#define BUF_LEN 10240

/*
 * Call this function to refresh the corresponding window
 * when initializing and refreshing
 * Note, this function is called by all the property pages.
 */
static void get_systeminfo (HWND hDlg)
{
    int type;
    HWND hwnd;
    char buff [BUF_LEN + 1];
    size_t size = 0;

    /* Determine which property page according to "type" */
    type = (int)GetWindowAdditionalData (hDlg);

    /* Get the handle of the static control of the property page */
    hwnd = GetDlgItem (hDlg, IDC_SYSINFO);

    buff [BUF_LEN] = 0;
    switch (type) {
        case PAGE_VERSION:
            size = read_sysinfo ("/proc/version", buff, BUF_LEN);
            buff [size] = 0;
            break;

        case PAGE_CPU:
            size = read_sysinfo ("/proc/cpuinfo", buff, BUF_LEN);
            buff [size] = 0;
            break;

        case PAGE_MEMINFO:
            size = read_sysinfo ("/proc/meminfo", buff, BUF_LEN);
            buff [size] = 0;
            break;

        case PAGE_PARTITION:
            size = read_sysinfo ("/proc/partitions", buff, BUF_LEN);
            buff [size] = 0;
            break;

        case PAGE_MINIGUI:

```

```

    size = sprintf (buff, BUF_LEN,
        "MiniGUI version %d.%d.%d.\n"
        "Copyright (C) 1998-2003 Feynman Software and others.\n\n"
        "MiniGUI is free software, covered by the GNU General Public License, "
        "and you are welcome to change it and/or distribute copies of it "
        "under certain conditions. "
        "Please visit\n\n"
        "http://www.minigui.org\n\n"
        "to know the details.\n\n"
        "There is absolutely no warranty for MiniGUI.",
        MINIGUI_MAJOR_VERSION, MINIGUI_MINOR_VERSION, MINIGUI_MICRO_VERSION);
    break;
}

if (size) {
    SetWindowText (hwnd, buff);
}
}

/* All the property pages use the same window procedure function */
static int SysInfoPageProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_INITPAGE:
            get_systeminfo (hDlg);
            break;

        case MSG_SHOWPAGE:
            return 1;

        case MSG_SHEETCMD:
            if (wParam == IDOK)
                /* When the user clicked the "refresh" button of the
                 * dialog box, call this function to refresh.
                 */
                get_systeminfo (hDlg);
            return 0;
    }

    return DefaultPageProc (hDlg, message, wParam, lParam);
}

static int PropSheetProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_INITDIALOG:
            {
                HWND pshwnd = GetDlgItem (hDlg, IDC_PROPSHEET);

                PageSysInfo.controls = CtrlSysInfo;

                /* Add property pages. Note that each property page
                 * has different additional data
                 */

                PageSysInfo.caption = "Version Info";
                PageSysInfo.dwAddData = PAGE_VERSION;
                SendMessage (pshwnd, PSM_ADDPAGE, (WPARAM)&PageSysInfo, (LPARAM)SysInfoPagePr
oc);

                PageSysInfo.caption = "CPU Info";
                PageSysInfo.dwAddData = PAGE_CPU;
                SendMessage (pshwnd, PSM_ADDPAGE, (WPARAM)&PageSysInfo, (LPARAM)SysInfoPagePr
oc);

                PageSysInfo.caption = "Mem Info";
                PageSysInfo.dwAddData = PAGE_MEMINFO;
                SendMessage (pshwnd, PSM_ADDPAGE, (WPARAM)&PageSysInfo, (LPARAM)SysInfoPagePr
oc);

                PageSysInfo.caption = "Partition Info";
                PageSysInfo.dwAddData = PAGE_PARTITION;
                SendMessage (pshwnd, PSM_ADDPAGE, (WPARAM)&PageSysInfo, (LPARAM)SysInfoPagePr
oc);

                PageSysInfo.caption = "MiniGUI Info";
            }
    }
}

```

```

        PageSysInfo.dwAddData = PAGE_MINIGUI;
        SendMessage (pshwnd, PSM_ADDPAGE, (WPARAM)&PageSysInfo, (LPARAM)SysInfoPagePr
oc);
        break;
    }

    case MSG_COMMAND:
    switch (wParam) {
        case IDOK:
            /* When the user clicked the "refresh" button,
             * send PSM_SHEETCMD message to the property sheet control
             */
            SendDlgItemMessage (hDlg, IDC_PROPSHEET, PSM_SHEETCMD, IDOK, 0);
            break;

            case IDCANCEL:
                EndDialog (hDlg, wParam);
                break;
        }
        break;
    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}

/* Template of the main dialog box */
static DLGTEMPLATE DlgPropSheet =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    0, 0, 410, 275,
    "System Info",
    0, 0,
    3, NULL,
    0
};

/* The dialog box has only three controls:
 * the property sheet, "refresh" button and "close" button
 */
static CTRLDATA CtrlPropSheet[] =
{
    {
        CTRL_PROPSHEET,
        WS_VISIBLE | PSS_COMPACTTAB,
        10, 10, 390, 200,
        IDC_PROPSHEET,
        "",
        0
    },
    {
        CTRL_BUTTON,
        WS_VISIBLE | BS_DEFPUSHBUTTON | WS_TABSTOP | WS_GROUP,
        10, 220, 140, 25,
        IDOK,
        "Refresh",
        0
    },
    {
        CTRL_BUTTON,
        WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP,
        260, 220, 140, 25,
        IDCANCEL,
        "Close",
        0
    },
};

int MiniGUIMain (int argc, const char* argv[])
{
#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER, "propsheet", 0, 0);
#endif

    DlgPropSheet.controls = CtrlPropSheet;

```

```
DialogBoxIndirectParam (&DlgPropSheet, HWND_DESKTOP, PropSheetProc, 0L);  
  
    return 0;  
}  
  
#ifndef _LITE_VERSION  
#include <minigui/dti.c>  
#endif
```

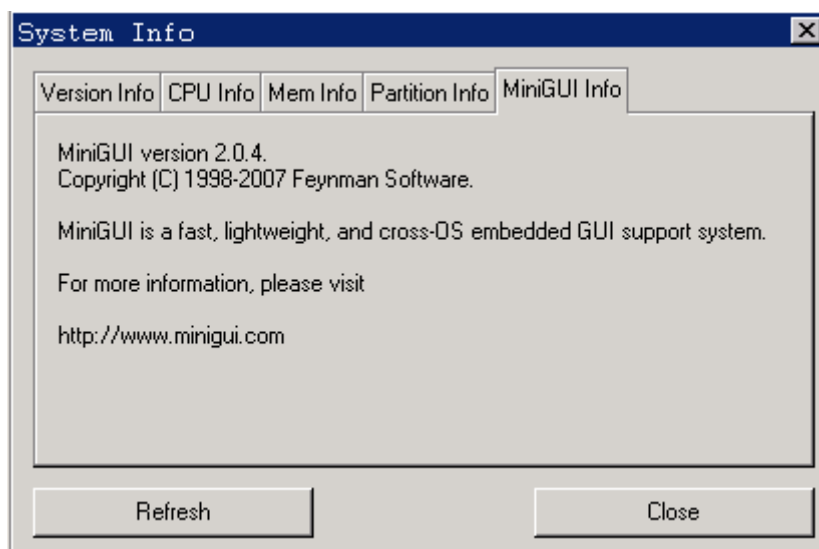


Fig. 28.2 Use of property sheet control





## 29 Scroll Window Control

Scroll window (ScrollWnd) control is a container control with scroll bars to browse children windows in it. Its basic usage is to place child controls, so make the user can check and operate many controls in a window by scroll bars. Certainly, ScrollWnd can also be used to do many other things, and it is easy to be customized. In the end of this chapter, we will see a example which uses ScrollWnd as a picture viewer.

In an application, calling `CreateWindow` function with `CTRL_SCROLLWND` as the control class name can create ScrollWnd control.

### 29.1 Scrollable Window

ScrollWnd control and ScrollView control discussed in the next chapter are both scrollable window controls, and have many similarities. A scrollable window comprises a control window (visible area) with scroll bar and content area, as shown in Fig. 29.1. When scroll bar is used to move the display content area, horizontal position value or vertical position value of content area will change. Size of content area can be controlled by application, but cannot be smaller than visible area.

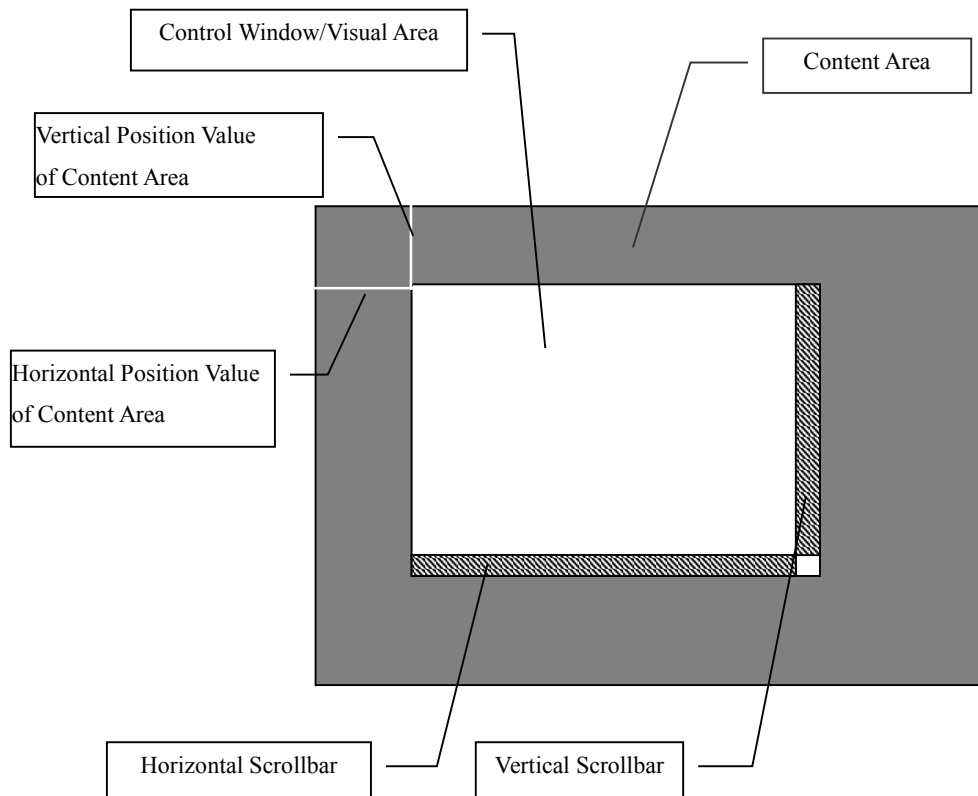


Fig. 29.1 A scrollable window

## 29.2 General Scroll Window Messages

ScrollWnd and ScrollView control all respond to some general scroll window messages, including getting and setting content range of scroll window, setting scroll value of scroll bar, getting and setting the current position of content area, getting and setting size of visible area, etc.

### 29.2.1 Get/Set the Range of Content Area and Visible Area

**SVM\_SETCONTRANGE** message is used to set the size of the content area of a scroll window:

```
int cont_w, cont_h;
SendMessage (hScrWnd, SVM_SETCONTRANGE, cont_w, cont_h);
```

Here, **cont\_w** and **cont\_h** are the width and the height of content area to be set respectively. If **cont\_w** or **cont\_h** is negative, content area will not change; if the content area width (height) to be set is less than visible area width

(height), the content area width (height) after set will equal to the visible area width (height).

**SVM\_SETCONTWIDTH** message and **SVM\_SETCONTHEIGHT** message are used to set the width and the height of a scroll window respectively:

```
int cont_w, cont_h;  
SendMessage (hScrWnd, SVM_SETCONTWIDTH, cont_w, 0);  
SendMessage (hScrWnd, SVM_SETCONTHEIGHT, cont_h, 0);
```

**SVM\_GETCONTWIDTH**, **SVM\_GETCONTHEIGHT**, **SVM\_GETVISIBLEWIDTH** and **SVM\_GETVISIBLEHEIGHT** message are used to get the content area width and height and the visible area width and height respectively.

### 29.2.2 Get Position Information and Set Current Position

**SVM\_GETCONTENTX** and **SVM\_GETCONTENTY** messages are used to get the current position value of the content area:

```
int pos_x = SendMessage (hScrWnd, SVM_GETCONTENTX, 0, 0);  
int pos_y = SendMessage (hScrWnd, SVM_GETCONTENTY, 0, 0);
```

**SVM\_SETCONTPOS** message is used to set the current position value of the content area, i.e. move the content area to a specified position in the visible area:

```
int pos_x, pos_y;  
SendMessage (hScrWnd, SVM_SETCONTPOS, pos_x, pos_y);
```

**SVM\_MAKEPOSVISIBLE** message is used to make a position point in the content area visible:

```
SendMessage (hScrWnd, SVM_MAKEPOSVISIBLE, pos_x, pos_y);
```

If this position point is invisible originally, while it becomes visible by using **SVM\_MAKEPOSVISIBLE** message, the position point will locate in the up edge of visible area (previous position point is above visible area) or down edge of

visible area (previous position point is below visible area).

### 29.2.3 Get/Set Scroll Properties

**SVM\_GETHSCROLLVAL** and **SVM\_GETVSCROLLVAL** messages are used to get the current horizontal and vertical line values of the scroll window (the scroll values when clicking the scroll arrow); **VM\_GETHSCROLLPAGEVAL** and **SVM\_GETVSCROLLPAGEVAL** messages are used to get the current horizontal and vertical page values of the scroll window (the scroll values when clicking the scroll page):

```
int val = SendMessage (hScrWnd, SVM_GETHSCROLLVAL, 0, 0);
int val = SendMessage (hScrWnd, SVM_GETVSCROLLVAL, 0, 0);
int val = SendMessage (hScrWnd, SVM_GETHSCROLLPAGEVAL, 0, 0);
int val = SendMessage (hScrWnd, SVM_GETVSCROLLPAGEVAL, 0, 0);
```

**SVM\_SETSCROLLVAL** message is used to set the horizontal and (or) vertical line value of a scroll window; **wParam** argument is the horizontal line value and **lParam** is the vertical line value; if the horizontal (vertical) line value is zero or negative value, the current horizontal (vertical) line value of the scroll window will not change:

```
int h_val, v_val;
SendMessage (hScrWnd, SVM_SETSCROLLVAL, h_val, v_val);
```

**SVM\_SETSCROLLPAGEVAL** is used to set the horizontal and (or) vertical page value of a scroll window; **wParam** argument is the horizontal page value and **lParam** is the vertical page value; if the horizontal (vertical) page value is zero or negative value, the current horizontal (vertical) page value of the scroll window will not change:

```
int h_val, v_val;
SendMessage (hScrWnd, SVM_SETSCROLLPAGEVAL, h_val, v_val);
```

## 29.3 Message of Scroll Window Control

### 29.3.1 Add Child Control

After ScrollWnd control is created, you can send **SVM\_ADDCTRLS** message to add child controls into it. The parameter **wParam** of the message is used to pass the number of controls; **lParam** is used to pass the pointer to a control array:

```
CTRLDATA controls[ctrl_nr];
SendMessage (hScrWnd, SVM_ADDCTRLS, (WPARAM)ctrl_nr, (LPARAM)controls);
```

It should be noted that adding controls to ScrollWnd control will not change the content area range of the scroll window, and if the position of a child control locates beyond the current range of the content area, you cannot see the control in the content area. Therefore, generally, you should use **SVM\_SETCONTRANGE** message to set the range of the content area before adding child controls, to make it suitable for display of controls to be added.

Except sending **SVM\_ADDCTRLS** message to add child controls after ScrollWnd control is created, you can also make ScrollWnd automatically add the child controls after passing the pointer to a **CONTAINERINFO** structure in **dwAddData** argument, when calling **CreateWindow** function to create the ScrollWnd control creates ScrollWnd:

```
typedef struct _CONTAINERINFO
{
    WNDPROC      user_proc;           /** user-defined window procedure of the container */
    int          ctrlnr;              /** number of controls */
    PCTRLDATA    controls;            /** pointer to control array */
    DWORD        dwAddData;           /** additional data */
} CONTAINERINFO;
typedef CONTAINERINFO* PCONTAINERINFO;
```

The field **ctrlnr** is the number of controls, **controls** points to a **CTRLDATA** control array; Additional data is passed through **dwAddData** field in **CONTAINERINFO** structure.

**SVM\_RESETCONTENT** message is used to reset a ScrollWnd control, including

destroying its child controls and set the range and the position values of the content area to be the default values:

```
SendMessage (hScrWnd, SVM_RESETCONTENT, 0, 0);
```

### 29.3.2 Get Handle of Child Control

**SVM\_GETCTRL** can be used to get the handle of a child control in a ScrollWnd control:

```
int id;  
HWND hCtrl;  
HCtrl = SendMessage (hScrWnd, SVM_GETCTRL, id, 0);
```

**SVM\_GETFOCUSCHILD** message can be used to get the child control gaining keyboard focus in a ScrollWnd control:

```
HWND hFocusCtrl;  
HFocusCtrl = SendMessage (hScrWnd, SVM_GETFOCUSCHILD, 0, 0);
```

### 29.3.3 Container (Content) Window Procedure

The window in which child control is contained in a scroll window is called the container window, i.e. content window (area). You can use **SVM\_SETCONTAINERPROC** message to set a new container window procedure, to achieve the goal of customizing the scroll window for an application:

```
WNDPROC myproc;  
SendMessage (hScrWnd, SVM_SETCONTAINERPROC, 0, (LPARAM)myproc);
```

The parameter **lParam** is the container window process defined by your application, and this window procedure should call the default container window procedure function **DefaultContainerProc** by default:

```
int WINAPI DefaultContainerProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam);
```

Additionally, an application can specify user-defined container window procedure by **user\_proc** field of **CONTAINERINFO** aforementioned.

## 29.4 Sample Program

We illustrate the method of using ScrollWnd control to construct a simple picture viewer using code in List 29.1. Please refer to `scrollwnd.c` program in this guide sample program packet `mg-samples` for the complete source code of this program.

List 29.1 Example program for ScrollWnd control

```
#define IDC_SCROLLWND      100
#define ID_ZOOMIN          200
#define ID_ZOOMOUT        300

static HWND hScrollWnd;
static BITMAP bmp_bkgnd;
static float current_scale = 1;

static int pic_container_proc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {

        case MSG_PAINT:
        {
            HDC hdc = BeginPaint (hWnd);
            FillBoxWithBitmap (hdc, 0, 0, current_scale * bmp_bkgnd.bmWidth,
                             current_scale * bmp_bkgnd.bmHeight, &bmp_bkgnd);
            EndPaint (hWnd, hdc);
            return 0;
        }

    }

    return DefaultContainerProc (hWnd, message, wParam, lParam);
}

static int
ImageViewerProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {

        case MSG_INITDIALOG:
        {
            hScrollWnd = GetDlgItem (hDlg, IDC_SCROLLWND);
            SendMessage (hScrollWnd, SVM_SETCONTAINERPROC, 0, (LPARAM)pic_container_proc);
            SendMessage (hScrollWnd, SVM_SETCONTRANGE, bmp_bkgnd.bmWidth, bmp_bkgnd.bmHeight
);
            break;
        }

        case MSG_COMMAND:
        {
            int id = LOWORD(wParam);

            if (id == ID_ZOOMIN || id == ID_ZOOMOUT) {
                current_scale += (id == ID_ZOOMIN) ? 0.2 : -0.2;
                if (current_scale < 0.1)
                    current_scale = 0.1;

                SendMessage (hScrollWnd, SVM_SETCONTRANGE,
                             current_scale * bmp_bkgnd.bmWidth,
                             current_scale * bmp_bkgnd.bmHeight);
                InvalidateRect (hScrollWnd, NULL, TRUE);
            }
        }
    }
}
```

```

        break;
    }

    case MSG_CLOSE:
        EndDialog (hDlg, 0);
        return 0;
    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}

static CTRLDATA CtrlViewer[] =
{
    {
        "ScrollWnd",
        WS_BORDER | WS_CHILD | WS_VISIBLE | WS_VSCROLL | WS_HSCROLL,
        10, 10, 300, 200,
        IDC_SCROLLWND,
        "image viewer",
        0
    },
    {
        CTRL_BUTTON,
        WS_TABSTOP | WS_VISIBLE | BS_DEFPUSHBUTTON,
        20, 220, 60, 25,
        ID_ZOOMIN,
        "Zoom in",
        0
    },
    {
        CTRL_BUTTON,
        WS_TABSTOP | WS_VISIBLE | BS_PUSHBUTTON,
        220, 220, 60, 25,
        ID_ZOOMOUT,
        "Zoom out",
        0
    }
};

static DLGTEMPLATE DlgViewer =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    0, 0, 350, 280,
    "Image Viewer",
    0, 0,
    TABLESIZE(CtrlViewer), CtrlViewer,
    0
};

int MiniGUIMain (int argc, const char* argv[])
{
#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER, "scrollwnd", 0, 0);
#endif

    if (LoadBitmap (HDC_SCREEN, &bmp_bkgnd, "bkgnd.jpg"))
        return 1;

    DialogBoxIndirectParam (&DlgViewer, HWND_DESKTOP, ImageViewerProc, 0L);

    UnloadBitmap (&bmp_bkgnd);
    return 0;
}

#ifdef _LITE_VERSION
#include <minigui/dti.c>
#endif

```

This simple picture viewer can be used to view a picture by using scrollbar, and



to enlarge and shrink the picture. The picture viewer sets a new container window procedure function by sending `SVM_SETCONTAINERPROC` message, and draws the picture in `MSG_PAINT` message.

Running effect of the program is shown as Fig. 29.2.

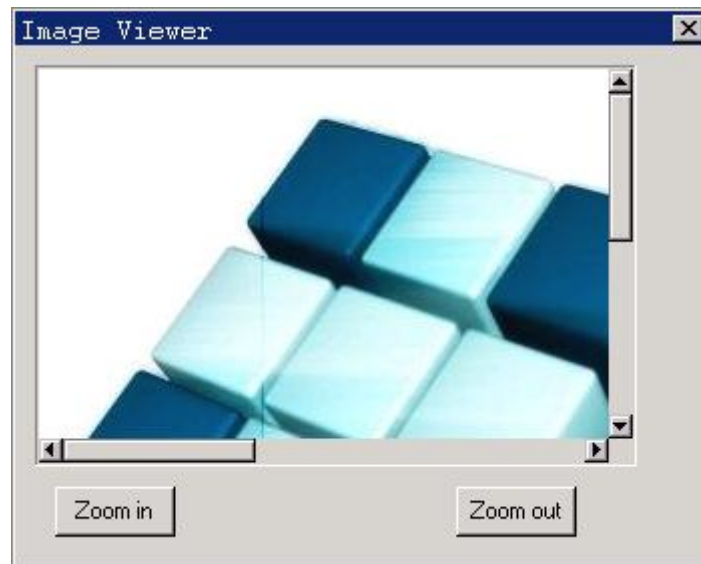


Fig. 29.2 A simple picture explorer



## 30 Scroll View Control

Scroll View (ScrollView) control is also a scroll window control, and different from ScrollWnd control in that ScrollView control displays list items instead of controls.

The major usage of ScrollView is to display and handle list items; this aspect is similar to list box and list view control. However the height of a list item in ScrollView can be specified by the user, so different list item can have different height. The most important is, draw of list items in ScrollView is completely determined by application. Totally speaking, ScrollView is a control easy to be customized, and give great free for applications, using ScrollView can perform much work that list box and list view control cannot do.

### 30.1 Styles of Scroll View Control

A ScrollView control with `SVS_AUTOSORT` style will automatically sort the list items; the precondition is that you have already set the item comparison function of the ScrollView control by using `SVM_SETITEMCMP` message:

```
SVM_SETITEMCMP myItemCmp;  
SendMessage (hScrWnd, SVM_SETITEMCMP, 0, (LPARAM)myItemCmp);
```

Here, `myItemCmp` is the item comparison function specified by an application.

List item comparison function of ScrollView control has the prototype of `VM_SETITEMCMP`, which is defined as follows:

```
typedef int (*SVITEM_CMP) (HSVITEM hsvi1, HSVITEM hsvi2);
```

Here, `hsvi1` and `hsvi2` are the handles of the two list items to be compared. If the comparison function returns a negative value, the list item `hsvi1` will be displayed before the list item `hsvi2`.

In addition, you can also use `SVM_SORTITEMS` message to sort list items in a `ScrollView` control without `SVS_AUTOSORT` style:

```
SVM_SETITEMCMP myItemCmp;
SendMessage (hScrWnd, SVM_SORTITEMS, 0, (LPARAM)myItemCmp);
```

Here, `myItemCmp` is the item comparison function used for sorting which is specified by an application.

## 30.2 Messages of Scroll View Control

Except responds to some general scroll window message as `ScrollWnd`, related messages of `ScrollView` control are mainly used to add, delete, and access list item.

### 30.2.1 Draw of List Item

Draw of list items in a `ScrollView` control is totally determined your application itself, so you must specify the content draw method of list item first before adding any list item. `SVM_SETITEMDRAW` message is used to set the draw function for list items:

```
SVITEM_DRAWFUNC myDrawItem;
SendMessage (hScrWnd, SVM_SETITEMDRAW, 0, (LPARAM)myDrawItem);
```

Content draw function of a list item has the prototype of `SVITEM_DRAWFUNC`, which is defined as follows:

```
typedef void (*SVITEM_DRAWFUNC) (HWND hWnd, HSVITEM hsvi, HDC hdc, RECT *rcDraw);
```

The arguments passed to the draw function are the handle to the `ScrollView` control (`hWnd`), the handle of list item to be drawn (`hsvi`), the graphics device context (`hdc`), and the rectangle area in which the item be drawn (`rcDraw`).

According to the actual usage of `ScrollView` control, content draw function can draw user-defined content, which can be text or picture, all decided by an

application itself, in the specified rectangle area.

### 30.2.2 Set Operation Functions of List Item

**SVM\_SETITEMOPS** message can be used to set some callback function related to list item operations, including initializing, drawing and destroying:

```
SVITEMOPS myops;
SendMessage (hScrWnd, SVM_SETITEMOPS, 0, (LPARAM)&myops);
```

Here, **myops** is a structure of **SVITEMOPS** type, and specifies the related operation functions for list items of ScrollView, as follows:

```
typedef struct _svitem_operations
{
    SVITEM_INITFUNC    initItem;    /** called when an ScrollView item is created */
    SVITEM_DESTROYFUNC destroyItem; /** called when an item is destroyed */
    SVITEM_DRAWFUNC    drawItem;    /** call this to draw an item */
} SVITEMOPS;
```

The member **initItem** is the initialization function, which is called during creating list items, and its prototype is defined as follows:

```
typedef int  (*SVITEM_INITFUNC) (HWND hWnd, HSVITEM hsvi);
```

The arguments are the handle to the control window (**hWnd**), and the handle to the created list item (**hsvi**). This function can be used to perform some operations relevant to the items during creating them.

The member **destroyItem** is the destroying function called during destroying the list items; its prototype is defined as follows:

```
typedef void (*SVITEM_DESTROYFUNC) (HWND hWnd, HSVITEM hsvi);
```

The arguments are the handle to the control window (**hWnd**), and the handle to the list item being destroyed. This function can be used to perform some cleaning up work during destroying them, such as release related resource.

The member `drawItem` specifies the draw function of the list items. Its effect is completely the same as using `SVM_SETITEMDRAW` message to set the draw function.

### 30.2.3 Operations on List Item

`SVM_ADDITEM` and `SVM_DELITEM` messages are used to add and delete a list item respectively:

```
int idx;
HSVITEM hsvi;
SVITEMINFO svii;
idx = SendMessage (hScrWnd, SVM_ADDITEM, (WPARAM)&hsvi, (LPARAM)&svii);
```

Here, `svii` is a structure of `SVITEMINFO` type, defined as follows:

```
typedef struct _SCROLLVIEWITEMINFO
{
    int        nItem;           /** index of item */
    int        nItemHeight;     /** height of an item */
    DWORD      addData;         /** item additional data */
} SVITEMINFO;
```

The field `nItem` is the desired index of the list item; if `nItem` is negative, the list item will be appended to the end. The field `nItemHeight` is the height of the list item; `addData` is the additional data value of the list item.

`Hsvi` is used to store the handle to a list item, and this handle can be used to access the list item. `SVM_ADDITEM` message returns the actual index value of the list item after inserted.

`SVM_DELITEM` message is used to delete a list item:

```
int idx;
HSVITEM hsvi;
SendMessage (hScrWnd, SVM_DELITEM, idx, hsvi);
```

Here, `hsvi` is used to specify the handle of the list item to be deleted. If `hsvi` is 0, you should specify the index value of the list item to be deleted through `idx`.

**SVM\_REFRESHITEM** message is used to refresh the area of a list item:

```
int idx;  
HSVITEM hsvi;  
SendMessage (hScrWnd, SVM_REFRESHITEM, idx, hsvi);
```

Here, **hsvi** is used to specify the handle of the list item to be refreshed. If **hsvi** is 0, you should specify the index value of the list item to be refreshed through **idx**.

**SVM\_GETITEMADDDATA** message is used to get the additional data of a list item:

```
SendMessage (hScrWnd, SVM_GETITEMADDDATA, idx, hsvi);
```

Here, **hsvi** is used to specify the handle of the list item to be accessed. If **hsvi** is 0, you should specify the index value of the list item to be accessed through **idx**.

**SVM\_SETITEMADDDATA** message is used to set the additional data of a list item:

```
int idx;  
DWORD addData;  
SendMessage (hScrWnd, SVM_SETITEMADDDATA, idx, addData);
```

Here, **idx** is used to specify the index value of the list item to be accessed, and **addData** is the additional data to be set.

**SVM\_GETITEMCOUNT** message is used to get the number of all list items:

```
int count = SendMessage (hScrWnd, SVM_GETITEMCOUNT, 0, 0);
```

**SVM\_RESETCONTENT** message is used to delete all the list items in the control:

```
SendMessage (hScrWnd, SVM_RESETCONTENT, 0, 0);
```

### 30.2.4 Get/Set Current Highlighted Item

ScrollView control has highlight property, that is to say, only one list item in list items is the current highlighted list item. An application can set and get the current highlighted list item.

It should be noted that highlight is only a property of ScrollView control. Being the current highlighted item does not mean the list item must have some specialty in appearance (such as highlighted display), and this is completely determined by the application itself.

**SVM\_SETCURSEL** message is used to set the highlighted list item of a ScrollView control:

```
SendMessage (hScrWnd, SVM_SETCURSEL, idx, bVisible);
```

Here, **idx** is used to specify the index value of the list item to be set as the highlighted item, and if **bVisible** is **TRUE**, the list item becomes a visible item.

**SVM\_GETCURSEL** message is used to get the current highlighted list item of a ScrollView control:

```
int hilighted_idx = SendMessage (hScrWnd, SVM_GETCURSEL, 0, 0);
```

The return value of **SVM\_GETCURSEL** message is the index value of current highlighted list item.

### 30.2.5 Selection and Display of List Item

Besides highlight property, ScrollView also has selection property. The highlight is unique, but selection is not unique, in other words, there may be multiple list items of ScrollView control to be selected. An application can set the selection state of list items.

Like highlight property, we should be noted that selection is only a state of a



list item. Being selected does not mean the list item must have some specialty in appearance (such as highlight display), this is also completely determined by the application itself.

**SVM\_SELECTITEM** message is used to select a list item:

```
SendMessage (hScrWnd, SVM_SELECTITEM, idx, bSel);
```

Here, **idx** is used to specify the index value of the list item to be operated, and if **bSel** is **TRUE**, the list item will be selected; otherwise will not be selected.

**SVM\_SHOWITEM** message is used to make sure a list item is visible:

```
SendMessage (hScrWnd, SVM_SHOWITEM, idx, hsvi);
```

Here, **hsvi** is used to specify the handle of list item to be displayed, while **idx** is used to specify the index value of the list item to be displayed, and **idx** only works when **hsvi** is 0.

**SVM\_CHOOSEITEM** message is combination of **SVM\_SELECTITEM** and **SVM\_SHOWITEM** messages, and is used to select a list item and make it visible:

```
SendMessage (hScrWnd, SVM_CHOOSEITEM, idx, hsvi);
```

Here, **hsvi** is the handle of list item to be selected and displayed, and **idx** is used to specify the index value of the list item to be selected and displayed, and **idx** only works when **hsvi** is 0.

### 30.2.6 Optimization of Display

When using **SVM\_ADDITEM** message or **SVM\_DELITEM** message to add or delete many list items in one time, you can use **MSG\_FREEZE** to perform certain optimization. The usage is to freeze a control before operating, and de-freeze the control after operating. When parameter **wParam** of **MSG\_FREEZE** message is

**TRUE**, the control is frozen, otherwise de-frozen.

### 30.2.7 Set Range of Visible Area

A window of a ScrollView control not only contains a visible area, but also a margin area, as shown in Fig. 30.1.

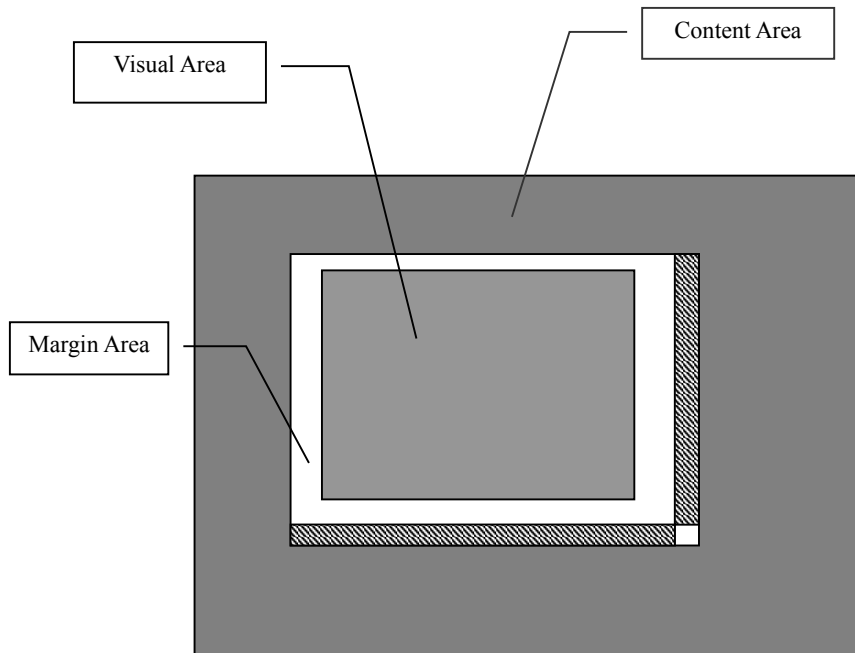


Fig. 30.1 Visible area of a ScrollView control

**SVM\_SETMARGINS** message can be used to set the margin range of a ScrollView control:

```
RECT rcMargin;
SendMessage (hScrWnd, SVM_SETMARGINS, 0, (LPARAM)&rcMargin);
```

The **left**, **top**, **right**, and **bottom** field of **rcMargin** are the size of left, top, right, and bottom margin respectively to be set, if some a margin value is negative, the corresponding set will not work.

**SVM\_GETMARGINS** message can get the margin range of a ScrollView control:

```
RECT rcMargin;
SendMessage (hScrWnd, SVM_GETMARGINS, 0, (LPARAM)&rcMargin);
```

`SVM_GETLEFTMARGIN`, `SVM_GETTOPMARGIN`, `SVM_GETRIGHTMARGIN`, and `SVM_GETBOTTOMMARGIN` messages are used to getting left, top, right, bottom margin value respectively.

### 30.3 Notification Codes of Scroll View Control

A `ScrollView` control will generate notification codes when responding to the operation such as the user clicks and some state changing happen, and the notification codes include:

- `SVN_SELCHANGE`: Currently highlighted list item changes.
- `SVN_CLICKED`: The user clicks a list item.
- `SVN_SELCHANGING`: Currently highlighted list item is changing.

An application should use `SetNotificationCallback` function to register a notification callback function, and handling on each received notification code in the function.

When you handle `SVN_CLICKED` and `SVN_SELCHANGE` in the notification callback function, the additional data passed in is the handle of the list item clicked or currently highlighted.

For `SVN_SELCHANGING`, the additional data passed in is the handle of the list item clicked or currently highlighted.

### 30.4 Sample Program

Code in List 30.1 illustrates the use of `ScrollView` control. The program constructs a simple contact person list by using `ScrollView` control. Please refer to `scrollview.c` program in the sample program package for the complete source code of the program.

List 30.1 Example program of `ScrollView` control

```
#define IDC_SCROLLVIEW    100
#define IDC_BT            200
```

```
#define IDC_BT2          300
#define IDC_BT3          400
#define IDC_BT4          500

static HWND hScrollView;

static const char *people[] =
{
    "Peter Wang",
    "Michael Li",
    "Eric Liang",
    "Hellen Zhang",
    "Tomas Zhao",
    "William Sun",
    "Alex Zhang"
};

static void myDrawItem (HWND hWnd, HSVITEM hsvi, HDC hdc, RECT *rcDraw)
{
    const char *name = (const char*)ScrollView_get_item_adddata (hsvi);

    SetBkMode (hdc, BM_TRANSPARENT);
    SetTextColor (hdc, PIXEL_black);

    if (ScrollView_is_item_hilight(hWnd, hsvi)) {
        SetBrushColor (hdc, PIXEL_blue);
        FillBox (hdc, rcDraw->left+1, rcDraw->top+1, RECTWP(rcDraw)-2, RECTHP(rcDraw)-1)
;
        SetBkColor (hdc, PIXEL_blue);
        SetTextColor (hdc, PIXEL_lightwhite);
    }

    Rectangle (hdc, rcDraw->left, rcDraw->top, rcDraw->right - 1, rcDraw->bottom);
    TextOut (hdc, rcDraw->left + 3, rcDraw->top + 2, name);
}

static int myCmpItem (HSVITEM hsvi1, HSVITEM hsvi2)
{
    const char *name1 = (const char*)ScrollView_get_item_adddata (hsvi1);
    const char *name2 = (const char*)ScrollView_get_item_adddata (hsvi2);

    return strcmp (name1, name2);
}

static int
BookProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case MSG_INITDIALOG:
        {
            SVITEMINFO svii;
            static int i = 0;

            hScrollView = GetDlgItem (hDlg, IDC_SCROLLVIEW);
            SetWindowBkColor (hScrollView, PIXEL_lightwhite);

            SendMessage (hScrollView, SVM_SETITEMCMP, 0, (LPARAM)myCmpItem);
            SendMessage (hScrollView, SVM_SETITEMDRAW, 0, (LPARAM)myDrawItem);

            for (i = 0; i < TABLESIZE(people); i++) {
                svii.nItemHeight = 32;
                svii.addData = (DWORD)people[i];
                svii.nItem = i;
                SendMessage (hScrollView, SVM_ADDITEM, 0, (LPARAM)&svii);
            }
            break;
        }

        case MSG_COMMAND:
        {
            int id = LOWORD (wParam);
            int code = HIWORD (wParam);

```

```

switch (id) {
case IDC_SCROLLVIEW:
    if (code == SVN_CLICKED) {
        int sel;
        sel = SendMessage (hScrollView, SVM_GETCURSEL, 0, 0);
        InvalidateRect (hScrollView, NULL, TRUE);
    }
    break;

}
break;
}

case MSG_CLOSE:
{
    EndDialog (hDlg, 0);
    return 0;
}

}

return DefaultDialogProc (hDlg, message, wParam, lParam);
}

static CTRLDATA CtrlBook[] =
{
    {
        "ScrollView",
        WS_BORDER | WS_CHILD | WS_VISIBLE | WS_VSCROLL | WS_HSCROLL |
        SVS_AUTOSORT,
        10, 10, 320, 150,
        IDC_SCROLLVIEW,
        "",
        0
    },
};

static DLGTEMPLATE DlgBook =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    0, 0, 350, 200,
    "My Friends",
    0, 0,
    TABLESIZE(CtrlBook), NULL,
    0
};

```

This program displays the contact person in list form, and sorts them by their names.

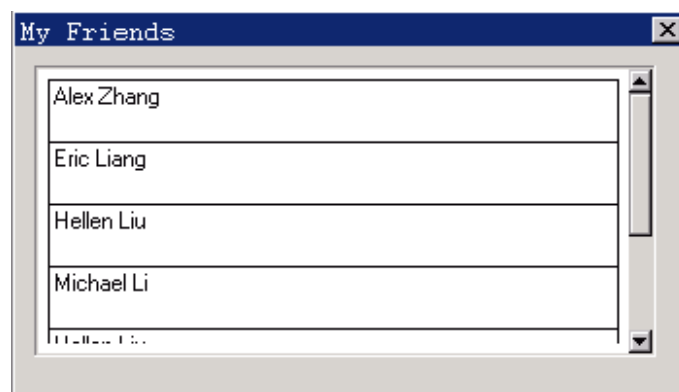


Fig. 30.2 Contact person list



## 31 Tree View Control

The tree view control displays a hierarchy items in a tree form, and each item (sub item) can include one or more child items. Each item or sub item includes the text title and an optional icon, and the user can unfold or fold the sub items of this item by clicking it. The tree view control is fit to represent objects having affiliation relationship, such as file and directory structure, or organization of an institution.

Calling `CreateWindow` function with `CTRL_TREEVIEW` as the control class name can create a tree view control. It should be noted that, the tree View control is implemented in MiniGUIExt library, so when you use this control, you must call `InitMiniGUIExt` function to initialize, and when you complete using it, call `MiniGUIExtCleanup` to do corresponding cleaning up. Controls to be illustrated in chapters below are all included in MiniGUIExt function library, so you must use the above function to initialize before using them. Of course, when compiling, you should not forget to use `-lmgext` option to link MiniGUIExt library.

After creating the tree view control, we can add, delete, set, get, or search the nodes by sending corresponding messages.

### 31.1 Styles of Tree View Control

The style of tree view control determines the appearance of the control. You can specify the initial style when creating a control, and also can then use `GetWindowStyle` to get the style and use `SetWindowStyle` to set a new style. The tree view control with `TVS_WITHICON` style uses an icon to show the folded and unfolded status of each item, and the corresponding icon can be specified when creating the node item. If you do not specify a certain icon for a node item, the tree view control will use the default icons specified in the configuration file `MiniGUI.cfg` of MiniGUI. The default icon files are `fold.ico` and `unfold.ico`. MiniGUI uses a "+" with a box around it to indicate a folded

node item and uses "-" with a box around it to indicate the unfolded node item in Tree view controls without `TVS_WITHICON` style.

Tree view control with `TVS_ICONFORSELECT` style uses an icon to indicate a selected item.

Tree view control with `TVS_SORT` style will automatically sort the items.

Tree view control with `TVS_NOTIFY` style will generate corresponding notification messages when responding to the user's operation.

## 31.2 Messages of Tree View Control

### 31.2.1 Creating and Deleting Node Item

A tree view control is comprised of a root node and a series of child nodes. When we use `CreateWindow` function to create a tree view control, we can pass a pointer to `TVITEMINFO` structure by `dwAddData` argument of this function to a tree view control to specify the properties of the root node. Please refer to the sample program in Section 31.4 for concrete example. `TVITEMINFO` structure includes the properties of the root node:

```
typedef struct _TVITEMINFO
{
    /* The text title of this item */
    char *text;

    /* The state flags of this item */
    DWORD dwFlags;

    /* The handle of the folded icon */
    HICON hIconFold;
    /* The handle of the unfolded icon */
    HICON hIconUnfold;

    /* Additional data of this item */
    DWORD dwAddData;
} TVITEMINFO;
```

Here `text` is the title of the node, and if you do not specify the properties of the root node when creating a tree view control, the title of the root node will be "root".



**dwFlags** is the state flag of a node item. **TVIF\_SELECTED** indicates that this node is a selected item, and **TVIF\_FOLD** indicates that this node item is folded initially. When adding an item, only **TVIF\_FOLD** flag is valid.

The handles **hIconFold** and **hIconUnfold** are the handles of icon used when the node is folded and unfolded, respectively. The handle **hIconFold** means selected and **hIconUnfold** means unselected and the style must be **TVS\_ICONFORSELECT**. They are meaningful only when the tree view control has **TVS\_WITHICON** style.

**TVM\_ADDITEM** (**TVM\_INSERTITEM**) message adds a node item to a tree view control:

```
TVITEMINFO tvItemInfo;  
GHANDLE item;  
item = SendMessage (hTrvWnd, TVM_ADDITEM, 0, (LPARAM) &tvItemInfo);
```

**item** is the handle of the added node returned by **SendMessage** function and we can use this handle to operate the node item.

**TVM\_DELTREE** message deletes a node and its all-descendant items (including the sub items of the item):

```
SendMessage (hTrvWnd, TVM_DELTREE, (WPARAM)item, 0);
```

Here **item** is a handle with **GHANDLE** type, which should be the handle returned by **SendMessage** function when using **TVM\_ADDITEM** message to add this node item.

### 31.2.2 Setting/Getting Properties of Node Item

**TVM\_GETITEMINFO** message is used to get the properties of a node item:

```
TVITEMINFO tvii;  
GHANDLE item;  
SendMessage (hTrvWnd, TVM_GETITEMINFO, (WPARAM)item, (LPARAM)&tvii);
```

Here, `item` is the handle of the node item the information of which we want to get, `tvii` structure is used to save the properties of the item. It should be noted that, the buffer to which `text` points in `tvii` structure should be big enough.

**TVM\_SETITEMINFO** is used to set the properties of a node item:

```
TVITEMINFO tvii;  
GHANDLE item;  
SendMessage (hTrvWnd, TVM_SETITEMINFO, (WPARAM)item, (LPARAM)&tvii);
```

Here, `item` is the handle of the node item to be set, and `tvii` structure includes the information of the node item to be set.

**TVM\_GETITEMTEXT** message gets the text title of a node item:

```
char *buffer;  
...  
SendMessage (hTrvWnd, TVM_GETITEMTEXT, (WPARAM)item, (LPARAM)buffer);
```

The buffer should be big enough to save the text title of the node item.

The length of the text title of a node item can be gotten with

**TVM\_GETITEMTEXTLEN** message:

```
int len;  
len = SendMessage (hTrvWnd, TVM_GETITEMTEXTLEN, (WPARAM)item, 0);
```

### 31.2.3 Selecting and Searching a Node Item

**TVM\_SETSELITEM** message is used to select a node item:

```
GHANDLE item;  
SendMessage (hTrvWnd, TVM_SETSELITEM, (WPARAM)item, 0);
```

Here, `item` is the handle of the node item to be selected.

**TVM\_GETSELITEM** message gets the current selected node item:

```
GHANDLE item;  
item = SendMessage (hTrvWnd, TVM_GETSELITEM, 0, 0);
```

After **SendMessage** returns, **item** is the handle of the current selected node item.

**TVM\_GETROOT** message is used to get the root node of a tree view control:

```
GHANDLE rootItem;  
rootItem = SendMessage (hTrvWnd, TVM_GETROOT, 0, 0);
```

**TVM\_GETRELATEDITEM** message is used to get the related node items of a certain node item:

```
GHANDLE item;  
int related;  
GHANDLE relItem;  
relItem = SendMessage (hTrvWnd, TVM_GETRELATEDITEM, related, (LPARAM)item);
```

Here **item** is the specified node, and **related** can be one of the following values:

- **TVIR\_PARENT**: To get the parent node of item
- **TVIR\_FIRSTCHILD**: To get the first sub item of item
- **TVIR\_NEXTSIBLING**: To get the next sibling node of item
- **TVIR\_PREVSIBLING**: To get the previous sibling node of item

**SendMessage** function returns the handle of the node item related to **item**.

**TVM\_SEARCHITEM** message is used to search a specified node item:

```
GHANDLE itemRoot;  
const char *text;  
GHANDLE found;  
found = SendMessage (hTrvWnd, TVM_SEARCHITEM, (WPARAM) itemRoot, (LPARAM) text);
```

Here, **itemRoot** specifies the searching range (a sub-tree with **itemRoot** being the root), and the string pointed to by **text** is the searching content.

**SendMessage** function will return the handle of the found node item if succeeds, and return zero if fails.

**TVM\_FINDCHILD** message is used to find a specified child node item of a node item.

```
GHANDLE itemParent;
const char *text;
GHANDLE found;
found = SendMessage (hTrvWnd, TVM_FINDCHILD, (WPARAM) itemParent, (LPARAM) text);
```

The child nodes of the node item specified by **itemParent** are the searching range, and the string pointed to by **text** is the searching content.

**SendMessage** function will return the handle of the found node item if succeeds, and return zero for if fails. The difference of **TVM\_FINDCHILD** from **TVM\_SEARCHITEM** is that **TVM\_FINDCHILD** only searches in the child nodes, while **TVM\_SEARCHITEM** searches in the whole sub-tree.

### 31.2.4 Comparing and Sorting

The tree view control with **TVS\_SORT** style automatically sorts the node items. When the application uses **TVM\_ADDITEM** message to add a node item, the item are sorted according to adding sequence if the control has not **TVS\_SORT** style; and sorted according to string comparison if the control has **TVS\_SORT** style.

The string comparison function of the tree view control determines sorting order of the strings. The initial string comparing function is **strcmp**, and the application can set a new string comparing function of the tree view control by sending **TVM\_SETSTRCMPFUNC** message:

```
SendMessage (hTrvWnd, TVM_SETSTRCMPFUNC, 0, (LPARAM) str_cmp);
```

Here, **str\_cmp** is the pointer to a function of **STRCMP** type:

```
typedef int (*STRCMP) (const char* s1, const char* s2, size_t n);
```

This string comparison function compares the first (at most) **n** characters of the two strings **s1** and **s2**, and returns an integer less than, equal to, or greater than zero according to comparison result.

### 31.3 Notification Codes of Tree View Control

The tree view control will generate notification messages when responding to the user's operation such as clicking and some state changes, including:

- **TVN\_SELCHANGE**: Current selected node item changed.
- **TVN\_DBLCLK**: The user double clicked a node item.
- **TVN\_SETFOCUS**: The tree view control gained the input focus.
- **TVN\_KILLFOCUS**: The tree view control lost the input focus.
- **TVN\_CLICKED**: The user clicked a node item.
- **TVN\_ENTER**: The user pressed the **ENTER** key.
- **TVN\_FOLDED**: A node item is folded.
- **TVN\_UNFOLDED**: The node item is unfolded.

If the application needs to know the notification from a tree view control, it needs to use **SetNotificationCallback** function to register a notification callback function for the control.

### 31.4 Sample Program

The program in List 31.1 illustrates the use of a tree view control. Please refer to **treeview.c** of the demo program package of this guide for complete source code.

List 31.1 Sample program of tree view control

```
#define IDC_TREEVIEW 100

#define CHAPTER_NUM 5

/* Define the text used by the items of the tree view control */
static const char *chapter[] =
{
    "Static Control",
    "Button Control",
    "Edit Control",
    "Listbox Control",
    "Treeview Control",
};

/* Define the text used by the items of the tree view control */
static const char *section[] =
{
    "Styles of Control",
    "Messages of Control",
    "Sample Program"
}
```

```
};

static int BookProc(HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_INITDIALOG:
        {
            TVITEMINFO tvItemInfo;
            int item;
            int i, j;

            /* Add items to the tree view control */
            for (i = 0; i < CHAPTER_NUM; i++) {
                tvItemInfo.text = (char*)chapter[i];
                item = SendMessage (GetDlgItem(hDlg, IDC_TREEVIEW), TVM_ADDITEM,
                                    0, (LPARAM)&tvItemInfo);

                /* Add subitems to each item */
                for (j = 0; j < 3; j++) {
                    tvItemInfo.text = (char*)section[j];
                    SendMessage (GetDlgItem(hDlg, IDC_TREEVIEW), TVM_ADDITEM,
                                item, (LPARAM)&tvItemInfo);
                }
            }
        }
        break;

        case MSG_CLOSE:
            EndDialog (hDlg, 0);
            return 0;
    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}

static TVITEMINFO bookInfo =
{
    "Contents"
};

/* Dialog box template */
static DLGTEMPLATE DlgBook =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    100, 100, 320, 240,
    "Book Contents",
    0, 0,
    1, NULL,
    0
};

/* The dialog box has only one control: tree view control */
static CTRLDATA CtrlBook[] =
{
    {
        CTRL_TREEVIEW,
        WS_BORDER | WS_CHILD | WS_VISIBLE |
            WS_VSCROLL | WS_HSCROLL,
        10, 10, 280, 180,
        IDC_TREEVIEW,
        "treeview control",
        (DWORD)&bookInfo
    }
};
```

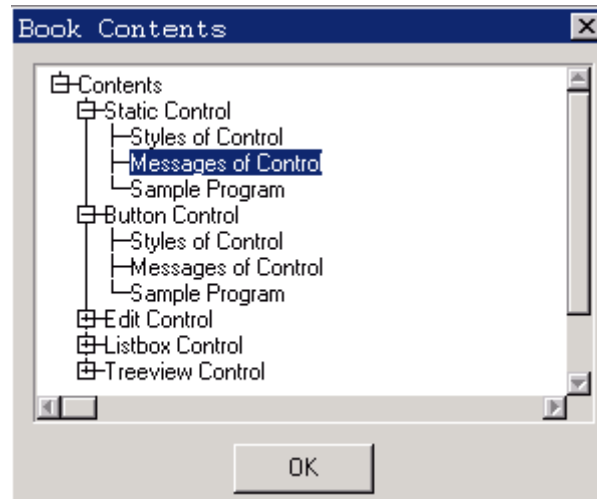


Fig. 31.1 Tree view formed by book content

The `treeview.c` program uses a tree view control to display the book contents structure. The program specifies `dwAddData` of the tree view control data structure `CTRLDATA` to be `&bookInfo` when creating the dialog box with the dialog box template. `BookInfo` is a structure of `TVITEMINFO` type, where the title of the node is "MiniGUI Programming Guide", so the title of the root node of this tree view is "MiniGUI Programming Guide".





## 32 List View Control

The list view control displays a series of data items (list item) in a table form, and the content of each list item may be comprised of one or more sub items. The sub items with the same type of different list items are organized in a column form. The header content of the list view control generally depicts the meaning of the different sub items of the list item. In appearance, the list view control is a rectangle including header and list items. The width of each sub item in the list view control can be adjusted by dragging the header, and the scrolling the control can show the content, which cannot be displayed fully in the list.

For the data including multiple properties, the list view control is a convenient and efficient tool for sorting and displaying such data. For example, the list view control is usually used as a file browser, which can display many file attributes including the file name, type, size, modified date, and so on.

The list view control is defined in mgext library, and you must first initialize mgext library to use the control. You can create a list view control by calling `CreateWindow` function with `CTRL_LISTVIEW` as the control class name. The application usually sends messages to a list view control to add, delete, sort, and operate the list items. Similar to other controls, the list view control will generate notification messages when responding to the user operations such as clicking.

### 32.1 Styles of List View Control

In the default state, the window of a list view control only displays the header and the list view items, and has no border around the displaying area. You can add border to the list view control by using `WS_BORDER` when creating the list view control with `CreateWindow` function. In addition, you can add vertical and horizontal scrollbars using `WS_VSCROLL` and `WS_HSCROLL` style in order to scroll the content of the list view control.

List view control with `LVS_TREEVIEW` style supports displaying list items in tree view mode, in other words, list view control with this style combines the function of normal list view control and tree view control.

`LVS_UPNOTIFY` style specifies the responding modes of a list view control when responding to the user mouse click operation. In the default condition, if you does not specify `LVS_UPNOTIFY` style, a list view control will send notification messages when the mouse is pressed down; if this style is specified, the control will send notification message when the mouse is released.

## 32.2 Messages of List View Control

### 32.2.1 Operations on Columns

After a list view control has been created, the next step generally is adding one or more columns successively to this control. Sending `LVM_ADDCOLUMN` to the control can complete this:

```
LVCOLUMN p;
SendMessage (hwndListView, LVM_ADDCOLUMN, 0, (LPARAM)&p) ;
```

Here, `p` is a `LVCOLUMN` structure, which includes the information related to the newly added column in the list view control. Definition and meaning of each field of `LVCOLUMN` are as follow:

```
typedef struct _LVCOLUMN
{
    /* The position of the column to be added */
    int nCols;
    /* The width of the column */
    int width;
    /* The title of the column */
    char *pszHeadText;
    /* The maximum length of the column title */
    int nTextMax;
    /* The image of the column header */
    DWORD image;
    /* The comparing function for sorting the columns */
    PFNLVCOMPARE pfnCompare;
    /* The column flag */
    DWORD colFlags;
} LVCOLUMN;
typedef LVCOLUMN *PLVCOLUMN;
```

**LVCOLUMN** structure is used to create or operate on columns of a list view control, and is used together with the messages **LVM\_ADDCOLUMN**, **LVM\_GETCOLUMN**, **LVM\_SETCOLUMN**, and **LVM\_MODIFYHEAD**.

When used with **LVM\_ADDCOLUMN** message, it is needed at least to give the value of **pszHeadText** of **LVCOLUMN** structure, i.e. the column title, and the other items can be set **NULL** or **0**. At this time, the list view control will adopt the default values for these fields.

The **nCols** item is an integer value to indicate which column the newly added column is, and the column index is from 1. If **nCols** is 0 or exceeds the range of the column index, the newly added column will be appended as the last column of the list view control. **width** is the width of the newly added column. If this value is 0 or not specified, the width of the newly added column will adopt the default value. **NTextMax** can be ignored when used for adding columns.

**Image** is the handle of a bitmap or an icon. If this value is specified, the specified image will be displayed on the header of the column. This item does not work at present.

**PfnCompare** points to a **PFNLVCOMPARE** type function, and this function is the comparing function attached to the newly added column. When the user clicks the title of the column, the list view control will determine the order of each list item according to the comparing function. If the column comparing function is not specified, the list view control will adopt the default string comparing function:

```
typedef int (*PFNLVCOMPARE) (int nItem1, int nItem2, PLVSORTDATA sortData);
```

Here **nItem1** and **nItem2** are integers, and are the indices of the two compared list items. **sortData** has no meaning at present and reserved for future. The comparing function determines the comparing result according to the indices of the two list items passed to it. The comparing basis is generally related to

the meaning of the column, and the meaning is defined by the application. Other data besides the list item index may be needed for comparison. One commonly used and feasible approach is that setting additional data useful for the comparing function when adding list items, then getting this additional data in the comparing function to handle.

**ColFlags** is the column flag, and has the following alignment flags at present: **LVCF\_LEFTALIGN**, **LVCF\_RIGHTALIGN**, and **LVCF\_CENTERALIGN**, which means left-aligned, right-aligned, and center-aligned of the column text, respectively.

After a column has been added, you can also set or change the attributes of the column by **LVM\_SETCOLUMN** message:

```
LVCOLUMN p;  
SendMessage (hwndListView, LVM_SETCOLUMN, 0, (LPARAM)&p) ;
```

Here **p** is also a **LVCOLUMN** structure, and the meanings and requirements of each item are the same as parameter **p** of **LVM\_ADDCOLUMN** message.

**LVM\_MODIFYHEAD** message is a simplification of **LVM\_SETCOLUMN**, and can be used to set the title of a list header:

```
LVCOLUMN p;  
SendMessage (hwndListView, LVM_MODIFYHEAD, 0, (LPARAM)&p) ;
```

Here **p** is also a **LVCOLUMN** structure, but only the values of **nCols** and **pszHeaderText** are needed to set.

**LVM\_GETCOLUMN** message is used to get the column attributes of a list view control:

```
LVCOLUMN p;  
int nCols;  
SendMessage (hwndListView, LVM_GETCOLUMN, nCols, (LPARAM)&p) ;
```

Here **nCols** is the integer index of the column of which the information is to be gotten, and **p** is a **LVCOLUMN** structure, used to store the gotten attributes.

**LVM\_GETCOLUMNWIDTH** message is used to get the width of a column:

```
int width;  
int nCols;  
width = SendMessage (hwndListView, LVM_GETCOLUMNWIDTH, nCols, 0) ;
```

Here **nCols** is the integer index of the column of which the information is to be gotten, and the return value of **SendMessage** function is the width of the column, and is -1 if error occurred.

**LVM\_GETCOLUMNCOUNT** is used to get the number of columns in a list view control:

```
int count;  
count = SendMessage (hwndListView, LVM_GETCOLUMNCOUNT, 0, 0) ;
```

The return value of **SendMessage** function is the number of columns.

**LVM\_DELCOLUMN** message is used to delete a column from a list view control:

```
int nCols;  
SendMessage (hwndListView, LVM_DELCOLUMN, nCols, 0) ;
```

Here **nCols** is the index of the column to be deleted.

**LVM\_SETHEADHEIGHT** is used to set the height of the column header:

```
int newHeight;  
SendMessage (hwndListView, LVM_SETHEADHEIGHT, newHeight, 0) ;
```

Here **newHeight** is the new header height.

### 32.2.2 Operations on List Item

A list view control is comprised of many list items aligned vertically. Columns to multiple sub items divide each list item, and the list item can include additional data defined by application. The application can add, change, set,

delete the list item, and get information of the list item by sending corresponding messages.

After a list view control has been created using `CreateWindow` function, the control has no items, and you need add list items to the list view control by `LVM_ADDITEM` message:

```
HLVITEM hItem;
HLVITEM hParent;
LVITEM lvItem;
hItem = SendMessage (hwndListView, LVM_ADDITEM, hParent, (LPARAM)&lvItem) ;
```

Here `hParent` specifies the parent node of newly added list item, if `hParent` is 0, this means the node is added to the root node (the topmost layer). If the control is normal list view control, 0 for `hParent` is ok. `lvItem` is a `LVITEM` structure, and contains the related information of a newly added list item in the list view control. Definition and meanings of fields of `LVITEM` are as follow:

```
typedef struct _LVITEM
{
    /**
     * The rows of the item
     */
    int nItem;
    /** Height of the item */
    int nItemHeight;
    /** Attached additional data of this item */
    DWORD itemData;

    /**
     * State flags of the item, can be OR'ed by the following values:
     *
     * - LVIF_FOLD\n
     *   The item is folded.
     *
     * When adding an item to the listview control, only LVIF_FOLD
     * flag is valid.
     */
    DWORD dwFlags;
} LVITEM;
```

Here `nItem` is the position value of the newly added list item. If this value is 0 or exceeds the index range, the newly added item will be added in the last of the list view. If `wParam` argument of `LVM_ADDITEM` message specifies the parent node of the newly added node, `nItem` specifies the position of newly added node in the parent node.

The return value of `LVM_ADDITEM` message is the handle of the newly added list

item, and the handle can be used in other messages to access the item.

The newly added item by **LVM\_ADDITEM** has no content yet, and you need use **LVM\_FILLSUBITEM** or **LVM\_SETSUBITEM** message to set the content of each sub item of the list item.

**LVM\_GETITEM** message is used to get the information of a list item:

```
LVITEM lvItem;  
HLVITEM hItem;  
SendMessage (hwndListView, LVM_GETITEM, hItem, (LPARAM)&lvItem) ;
```

Here **hItem** is the handle of target list type; **lvItem** is a structure of **LVITEM** type, and this structure is used to save the gotten list item information. If **hItem** is 0, **nItem** of **lvItem** structure should be set as the index value of list item to be gotten.

**LVM\_GETITEMCOUNT** message is used to get the number of all the items in the list view control:

```
int count;  
count = SendMessage (hwndListView, LVM_GETITEMCOUNT, 0, 0) ;
```

The return value of **SendMessage** function is the number of list items in the list view control.

**LVM\_GETITEMADDDATA** message is used to get the additional data of a list item:

```
DWORD addData;  
int nItem;  
HLVITEM hItem;  
addData = SendMessage (hwndListView, LVM_GETITEMADDDATA, nItem, hItem) ;
```

Here **hItem** is the handle of list item to be gotten, if **hItem** is zero, **nItem** should be used to specify the index value of list item to be gotten. **SendMessage** function returns the additional data of list item.

**LVM\_SETITEMADDDATA** message sets the additional data of a list item:

```
HLVITEM hItem;  
DWORD addData;  
SendMessage (hwndListView, LVM_SETITEMADDDATA, hItem, (LPARAM)addData) ;
```

Here **hItem** is the handle of list item to be set; **addData** is the additional data, if set successfully, **SendMessage** returns **LV\_OKAY**, else **LV\_ERR**.

**LVM\_SETITEMHEIGHT** message can be used to set the height of list items in a list view control. If the height is not set, the height of list items in the list view control will adopt the default value:

```
HLVITEM hItem;  
int newHeight;  
SendMessage (hwndListView, LVM_SETITEMHEIGHT, hItem, newHeight) ;
```

Here **hItem** is the handle of the list item to be set; **newHeight** is the new height value of the list item. Setting successfully, **SendMessage** function will return **TRUE**; otherwise return **FALSE**.

**LVM\_DELITEM** message is used to delete an item from the list view control, and **LVM\_DELALLITEM** message is used to delete all the list items:

```
HLVITEM hItem;  
int nItem;  
SendMessage (hwndListView, LVM_DELITEM, nItem, hItem) ;  
SendMessage (hwndListView, LVM_DELALLITEM, 0, 0) ;
```

Here **hItem** is the handle of list item to be deleted. It **hItem** is 0; **nItem** should be used to specify the index value of the list item to be deleted.

Each list item contains one or multiple sub items, and the number of sub items is the same as the number of columns in the list view control. A sub item contains string and bitmap, and can use the messages **LVM\_SETSUBITEM**, **LVM\_SETSUBITEMTEXT**, **LVM\_SETSUBITEMCOLOR**, and **LVM\_GETSUBITEMTEXT** to get and set the sub item attributes.



**LVM\_SETSUBITEM** (**LVM\_FILLSUBITEM**) message is used to set the attributes of a sub item:

```
LVSUBITEM subItem;
HLVITEM hItem;
SendMessage (hwndListView, LVM_SETSUBITEM, hItem, (LPARAM)&subItem) ;
```

Here **hItem** is the handle of the list item to be set; **sub item** is a structure of **LVSUBITEM** type, in which relevant information required for creating a sub item is included:

```
typedef struct _LVSUBITEM
{
    /* The flags of a subitem */
    DWORD flags;
    /* The vertical index of a subitem */
    int nItem;
    /* The horizontal index of a subitem */
    int subItem;
    /* The text content of a subitem */
    char *pszText;
    /* The text length of a subitem */
    int nTextMax;
    /* The text color of a subitem */
    int nTextColor;
    /* The image of a subitem */
    DWORD image;
} LVSUBITEM;
typedef LVSUBITEM *PLVSUBITEM;
```

Here, **flags** is the flag value of a sub item, can be **LVFLAG\_BITMAP** or **LVFLAG\_ICON** at present. If a bitmap or an icon is to be displayed in the sub item, the corresponding flag should be set, for example, **flags |= LVFLAG\_BITMAP**.

**NItem** and **sub item** are the vertical index and horizontal index of an item, i.e. the positions of the row and the column, respectively. **PszText** is the text content to be displayed in the sub item. **NTextMax** is the maximum length of the text of the sub item, and can be ignored when used for **LVM\_SETSUBITEM** message. When **LVSUBITEM** structure is used to get information of a sub item, **pszText** points to a buffer storing the text content, and **nTextMax** indicates the size of the buffer.

**NTextColor** specifies the color of the text in a sub item, and we can also use **LVM\_SETSUBITEMCOLOR** to set the color of text in a sub item. **Image** specifies

the bitmap or icon to be displayed in a sub item, which works only when flags item is set `LVFLAG_BITMAP` or `LVFLAG_ICON`.

`LVM_GETSUBITEMTEXT` and `LVM_SETSUBITEMTEXT` messages are used to get and set the text content of a sub item, respectively; `LVM_GETSUBITEMLEN` message is used to get the length of the string in a sub item:

```
LVSUBITEM subItem;  
HLVITEM hItem;  
int len;  
  
SendMessage (hwndListView, LVM_GETSUBITEMTEXT, hItem, (LPARAM)&subItem) ;  
SendMessage (hwndListView, LVM_SETSUBITEMTEXT, hItem, (LPARAM)&subItem) ;  
len = SendMessage (hwndListView, LVM_GETSUBITEMLEN, hItem, (LPARAM)&subItem) ;
```

### 32.2.3 Selecting, Displaying, and Searching List Item

`LVM_SELECTITEM` message is used to select a list item, and the selected item will be highlighted. It should be noted that, the selected item might not be visible.

```
int nItem;  
HLVITEM hItem;  
  
SendMessage (hwndListView, LVM_SELECTITEM, nItem, hItem) ;
```

Here `hItem` is the handle of list item to be selected; if `hItem` is 0, `nItem` specifies the index value of list item to be selected.

`LVM_GETSELECTEDITEM` message is used to get the current selected list item:

```
HLVITEM hItemSelected;  
hItemSelected = SendMessage (hwndListView, LVM_GETSELECTEDITEM, 0, 0) ;
```

`SendMessage` function returns the handle of a list item in a list view control currently selected. If no item is selected, 0 is returned.

`LVM_SHOWITEM` message makes a list item visible in the list view control. Making a list item visible will not make it selected.

```
HLVITEM hItem;
int nItem;
SendMessage (hwndListView, LVM_SHOWITEM, nItem, hItem) ;
```

Here **hItem** is the handle of the list item to be displayed; if **hItem** is 0, **nItem** should be used to specify the index value of list item to be displayed. If the item to be displayed is not visible or not totally visible, the item will become the first or last visible item of visible area and is totally visible after sending **LVM\_SHOWITEM** message.

**LVM\_CHOOSEITEM** is the combination of **LVM\_SELECTITEM** and **LVM\_SHOWITEM**. It makes a list item be selected and visible:

```
int nItem;
HHLVITEM hItem;
SendMessage (hwndListView, LVM_CHOOSEITEM, nItem, hItem) ;
```

Here **hItem** is the handle of the list item to be selected and displayed; if **hItem** is 0, **nItem** should be used to specify the index value of the list item to be selected and displayed.

**LVM\_FINDITEM** message is used to search a certain list item in a list view control. If successful for searching, **SendMessage** returns the handle of the found list item.

```
HLVITEM hFound;
HLVITEM hParent;
LVFINDINFO findInfo;
hFound = SendMessage (hwndListView, LVM_FINDITEM, hParent, (LPARAM)&findInfo) ;
```

Here **hParent** specifies the root node of target node tree to be searched for. **FindInfo** is a **LVFINDINFO** structure, and contains the information needed for searching:

```
typedef struct _LVFINDINFO
{
    /* The searching flags */
    DWORD flags;
    /* The starting index for searching */
    int iStart;
    /* pszInfo field includes how many columns */
    int nCols;
    /* The text contents of several subitems to be found */
    char **pszInfo;
```

```

/* The additional data of a list item */
DWORD addData;

/** The found item's row, reserved */
int nItem;
/** The found subitem's column, reserved */
int nSubitem;

} LVFINDINFO;
typedef LVFINDINFO *PLVFINDINFO;

```

The field **flags** is the search flag, and can be **LVFF\_TEXT** and/or **LVFF\_ADDDATA**, which means to search according to sub item text and (or) additional data of list item. If the root node **hParent** specified by **wParam** argument of **LVM\_FINDITEM** message is zero, **iStart** is the start index value for searching, and if it is zero then search from the start.

The pointer **pszInfo** points to a multiple character strings to be searched for, value of **nCols** means text content of the first **nCols** column sub item in matching list item should be consistent to the character string in **pszInfo**. If searching according to the additional data, **addData** field should include the additional data to be searched for.

### 32.2.4 Comparing and Sorting

After a list view control has added items with **LVM\_ADDITEM** message, the items are sorted according to their sequence and specified index during being added. When the user clicks the header of a list view control, i.e. the column title, the control will determine the orders of the list items according to the comparing function associated with this column, and sort them. As mentioned earlier, when you use **LVM\_ADDCOLUMN** message to add a column, you can specify the comparing function of the new column; after this, you can also set a new comparing function by **LVM\_SETCOLUMN** function.

We can also sort the list items by sending **LVM\_SORTITEMS** message to the list view control:

```
SendMessage (hwndListView, LVM_SORTITEMS, 0, (LPARAM)pfnCompare) ;
```

Here `pfnCompare` points to a function with `PFNLVCOMPARE` type, and this function is the comparing function to sort list items and should be defined by application.

In addition, we can also make the list view control sort all the sub items according to a certain column by sending `LVM_COLUMNSORT` message:

```
int nCol;  
SendMessage (hwndListView, LVM_COLUMNSORT, nCol, 0) ;
```

Here `nCol` is the index of the specified column, and the list view control will compare and sort the sub items according to the comparing function associated with the column.

When the comparing function is not specified, the list view control uses the default string comparing function to sort. The initial string comparing function is `strcasecmp`. We can set a user-defined string comparing function by `LVM_SETSTRCMPFUNC` message:

```
SendMessage (hwndListView, LVM_SETSTRCMPFUNC, 0, (LPARAM)pfnStrCmp) ;
```

Here `pfnStrCmp` is a pointer to the function with `STRCMP` type:

```
typedef int (*STRCMP) (const char* s1, const char* s2, size_t n);
```

This string comparing function compares the first (at most) `n` characters of the two compared strings `s1` and `s2`, and returns an integer less than, equal to, or greater than 0 according to the comparing result.

### 32.2.5 Operation of Tree View Node

We can perform some operations on the tree view node in a list view control with `LVS_TREEVIEW` style, including getting the related node and/or fold/unfold a node.

**LVM\_GETRELATEDITEM** message is used to get the related tree view nodes of a node, such as the parent node, the sibling nodes and its first child node etc:

```
int related;  
HLVITEM hItem;  
HLVITEM hRelatedItem;  
hRelatedItem = SendMessage (hwndListView, LVM_GETRELATEDITEM, related, hItem) ;
```

Here **related** specifies the relationship between the node and the target node, including:

- **LVIR\_PARENT**: Get the parent node.
- **LVIR\_FIRSTCHILD**: Get the first child node.
- **LVIR\_NEXTSIBLING**: Get the next sibling node.
- **LVIR\_PREVSIBLING**: Get the previous sibling node.

**hItem** is the handle of target node. **LVM\_GETRELATEDITEM** message returns the handle to the gotten related node.

**LVM\_FOLDITEM** message is used to fold or unfold a node item including child nodes:

```
HLVITEM hItem;  
BOOL bFold;  
SendMessage (hwndListView, LVM_FOLDITEM, bFold, hItem) ;
```

If **bFold** is **TRUE**, the node item is folded, else unfolded it. **hItem** is the handle of the node.

### 32.3 Handling of Key Messages

When the user clicks the up or down arrow key, the current selected list item will change, moving forward or backward by one item, and the newly selected item will change to be visible (if it is invisible originally). When the user presses the **PageUp** or **PageDown** key, the list item will go to the next page, and the range is the same as clicking the scrollbar to the next page, i.e. the last item of the former page becomes the first item of the latter page. If the **HOME** key is pressed down, the first list item will be selected and visible; and if the

**END** key is pressed down, the last item will be selected and visible.

## 32.4 Notification Codes of List View Control

The list view control will generate notification messages when responding to the user operation such as clicking and some state changes, including:

- **LVN\_ITEMRDOWN**: The right mouse button is pressed down on a list view item.
- **LVN\_ITEMRUP**: The right mouse button is released on a list view item.
- **LVN\_HEADRDOWN**: The right mouse button is pressed down on the list view header.
- **LVN\_HEADRUP**: The right mouse button is released up on the list view header.
- **LVN\_KEYDOWN**: A key is pressed down.
- **LVN\_ITEMDBCLK**: The user double clicked a list item.
- **LVN\_ITEMCLK**: The user clicked a list item (reserved).
- **LVN\_SELCHANGE**: The current selected item changed.
- **LVN\_FOLDED**: The user clicks some list item by mouse to fold it.
- **LVN\_UNFOLDED**: The user clicks some list item by mouse to unfold it.

When the right mouse button is pressed down on a list item, the item is selected, and two notification codes **LVN\_SELCHANGE** and **LVN\_ITEMRDOWN** are generated.

If the application need to know the notification codes generated by a list view control, it is better to use **SetNotificationCallback** function to register a notification callback function.

## 32.5 Sample Program

The program in List 32.1 illustrates the use of a list view control. Please refer to **listview.c** file of the demo program package of this guide for the complete source code.

List 32.1 Sample program of list view control

```
#define IDC_LISTVIEW    10
#define IDC_CTRL1      20
#define IDC_CTRL2      30

#define SUB_NUM        3

static char * caption [] =
{
    "Name", "Chinese", "Math", "English"
};

#define COL_NR          TABLESIZE(caption)

static char *classes [] =
{
    "Grade 1", "Grade 2", "Grade 3"
};

typedef struct _SCORE
{
    char *name;
    int scr[SUB_NUM];
} SCORE;

static SCORE scores[] =
{
    {"Tom", {81, 96, 75}},
    {"Jack", {98, 62, 84}},
    {"Merry", {79, 88, 89}},
    {"Bob", {79, 88, 89}},
};

#define SCORE_NUM       TABLESIZE(scores)

static GHANDLE add_class_item (HWND hlist, PLVITEM lvItem, GHANDLE classent)
{
    LVSUBITEM subdata;
    GHANDLE item = SendMessage (hlist, LVM_ADDITEM, classent, (LPARAM)lvItem);

    subdata.nItem = lvItem->nItem;
    subdata.subItem = 0;
    subdata.pszText = classes[lvItem->nItem];
    subdata.nTextColor = 0;
    subdata.flags = 0;
    subdata.image = 0;
    SendMessage (hlist, LVM_SETSUBITEM, item, (LPARAM) & subdata);

    return item;
}

static GHANDLE add_score_item (HWND hlist, PLVITEM lvItem, GHANDLE classent)
{
    char buff[20];
    LVSUBITEM subdata;
    GHANDLE item = SendMessage (hlist, LVM_ADDITEM, classent, (LPARAM)lvItem);
    int i = lvItem->nItem;
    int j;

    subdata.flags = 0;
    subdata.image = 0;
    subdata.nItem = lvItem->nItem;

    for (j = 0; j < 4; j++) {

        subdata.subItem = j;
        if (j == 0) {
            subdata.pszText = scores[i].name;
            subdata.nTextColor = 0;
        }
        else {
            sprintf (buff, "%d", scores[i].scr[j-1]);

```



```

        subdata.pszText = buff;
        if (scores[i].scr[j-1] > 90)
            subdata.nTextColor = PIXEL_red;
        else
            subdata.nTextColor = 0;
    }
    SendMessage (hlist, LVM_SETSUBITEM, item, (LPARAM) & subdata);

}

return item;
}

static int
ScoreProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    HWND hListView;
    hListView = GetDlgItem (hDlg, IDC_LISTVIEW);

    switch (message)
    {
    case MSG_INITDIALOG:
    {
        int i, j;
        LVITEM item;
        LVCOLUMN lvcol;
        GHANDLE hitem;

        for (i = 0; i < COL_NR; i++) {
            lvcol.nCols = i;
            lvcol.pszHeadText = caption[i];
            lvcol.width = 120;
            lvcol.pfnCompare = NULL;
            lvcol.colFlags = 0;
            SendMessage (hListView, LVM_ADDCOLUMN, 0, (LPARAM) &lvcol);
        }

        item.nItemHeight = 25;

        SendMessage (hListView, MSG_FREEZECTRL, TRUE, 0);
        hitem = 0;
        for (i = 0; i < 3; i++) {
            item.nItem = i;
            hitem = add_class_item (hListView, &item, 0);

            for (j = 0; j < SCORE_NUM; j++) {
                item.nItem = j;
                add_score_item (hListView, &item, hitem);
            }
        }

        SendMessage (hListView, MSG_FREEZECTRL, FALSE, 0);
        break;
    }

    case MSG_COMMAND:
    {
        int id = LOWORD (wParam);
        int i, j;

        if (id == IDC_CTRL2) {
            float average = 0;
            char buff[20];
            for (i = 0; i < SCORE_NUM; i++) {
                for (j = 0; j < SUB_NUM; j++) {
                    average += scores[i].scr[j];
                }
            }
            average = average / (SCORE_NUM * SUB_NUM);

            sprintf (buff, "%4.1f", average);
            SendDlgItemMessage (hDlg, IDC_CTRL1, MSG_SETTEXT, 0, (LPARAM)buff);
        }
    }
}

```

```

        break;
    }

    case MSG_CLOSE:
    {
        EndDialog (hDlg, 0);
        break;
    }

    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}

static CTRLDATA CtrlScore[] =
{
    {
        "button",
        WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,
        80, 260, 80, 20,
        IDC_CTRL2,
        "Everage score",
        0
    },
    {
        "edit",
        WS_CHILD | WS_VISIBLE | WS_BORDER,
        10, 260, 50, 20,
        IDC_CTRL1,
        "",
        0
    },
    {
        "listview",
        WS_BORDER | WS_CHILD | WS_VISIBLE | WS_VSCROLL | WS_HSCROLL | LVS_TREEVIEW,
        10, 10, 320, 220,
        IDC_LISTVIEW,
        "score table",
        0
    },
};

static DLGTEMPLATE DlgScore =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    0, 0, 480, 340,
    "Getting the average score",
    0, 0,
    0, NULL,
    0
};

```

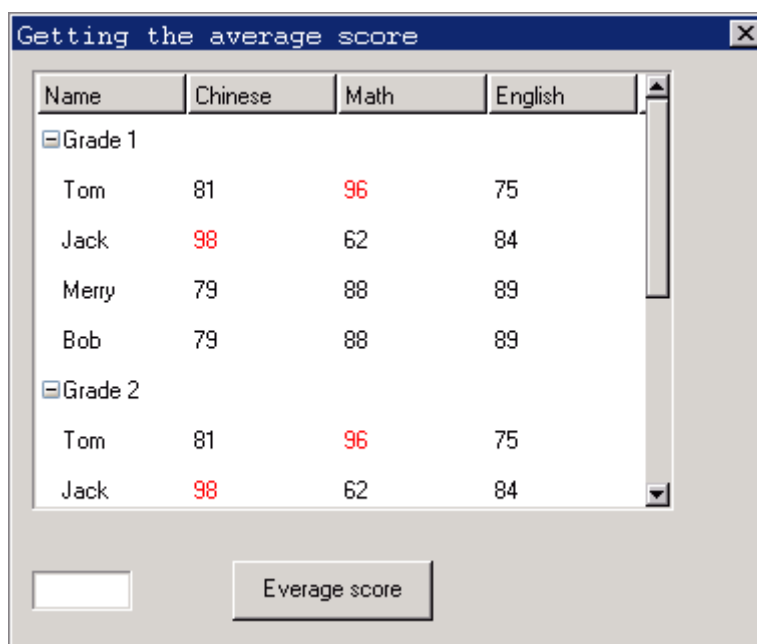


Fig. 32.1 Use of list view control

The `listview.c` program creates a list view control in dialog box for displaying each class scores of students, and you can get the average of total classes of students by clicking the button.



## 33 Month Calendar Control

The month calendar control provides a user interface similar to a calendar, and makes the user be able to select and set the date conveniently. The application can get or set the date by sending message to a month calendar control.

Month calendar control is defined in `mgext` library, so you must initialize `mgext` library to use this control. You can create a month calendar control by calling `CreateWindow` function with `CTRL_MONTHCALENDAR` as the control class name.

### 33.1 Styles of Month Calendar

A month calendar control can use several forms of Chinese or English and so on to display the date information such as week, month, etc. Specifying the control style to be `MCS_CHN`, `MCS_ENG_L`, or `MCS_ENG_S` can complete this. If a month calendar control has the style `MCS_CHN`, the control will display the date information in Chinese; if having the style `MCS_ENG_L`, the control will display the date information in English; and if having the style `MCS_ENG_S`, the control displays the date information in abbreviate English.

If having the style `MCS_NOTIFY`, the month calendar control will generate corresponding notification messages in cases such as responding to the user's operation.

### 33.2 Messages of Month Calendar

#### 33.2.1 Getting Date

`MCM_GETCURDAY` message is used to get the day of the month of the current selected date:

```
int day;  
day = SendMessage (hwndMonthcal, MCM_GETCURDAY, 0, 0) ;
```

The return value of **SendMessage** is the current day of the month, in the range 1 to 31.

**MCM\_GETCURMONTH** message is used to get the number of months of the current selected date:

```
int month;  
month = SendMessage (hwndMonthcal, MCM_GETCURMONTH, 0, 0) ;
```

The return value of **SendMessage** is the number of months since January, in the range 0 to 11.

**MCM\_GETCURYEAR** message is used to get the number of years of the current selected date:

```
int year;  
year = SendMessage (hwndMonthcal, MCM_GETCURYEAR, 0, 0) ;
```

The return value of **SendMessage** is the current number of years.

**MCM\_GETCURMONLEN** message is used to get the length (how many days in a month) of the current month:

```
int monthlen;  
monthlen = SendMessage (hwndMonthcal, MCM_GETCURMONLEN, 0, 0) ;
```

The return value of **SendMessage** is the length of the current month.

**MCM\_GETFIRSTWEEKDAY** message is used to determine which weekday is the first day in the current month:

```
int weekday;  
weekday = SendMessage (hwndMonthcal, MCM_GETFIRSTWEEKDAY, 0, 0) ;
```

The return value of **SendMessage** is the weekday number of the first day in the current month. The weekday number is the number of days since Sunday, in the range 0 to 6.

**MCM\_GETCURDATE** message gets the current selected date in a month calendar control:

```
SYSTEMTIME systime;  
SendMessage (hwndMonthcal, MCM_GETCURDATE, 0, (LPARAM)&systime) ;
```

Here **systime** is a structure of **SYSTEMTIME** type, which stores the gotten date information such as year, month, day, and week, etc. This structure is also used for messages such as **MCM\_GETTODAY** and so on. The definition of **SYSTEMTIME** structure is as follows:

```
typedef struct _SYSTEMTIME  
{  
    int year;  
    int month;  
    int day;  
    int weekday;  
} SYSTEMTIME;  
typedef SYSTEMTIME *PSYSTEMTIME;
```

**MCM\_GETTODAY** message gets the date of "today".

```
SYSTEMTIME systime;  
SendMessage (hwndMonthcal, MCM_GETTODAY, 0, (LPARAM)&systime) ;
```

Here **systime** is also a structure of **SYSTEMTIME** type.

### 33.2.2 Setting Date

It should be noted that special user right (such as root) may needed to set the date in Linux/UNIX system.

**MCM\_SETCURDAY** message sets the current day, **MCM\_SETCURMONTH** message sets the current month, and **MCM\_SETCURYEAR** sets the current year:

```
int day;  
int month;  
int year;  
SendMessage (hwndMonthcal, MCM_SETCURDAY, day, 0) ;  
SendMessage (hwndMonthcal, MCM_SETCURMONTH, month, 0) ;  
SendMessage (hwndMonthcal, MCM_SETCURYEAR, year, 0) ;
```

Here, day, month, and year specify the new day, month and year respectively,

and if these values exceed the rational values, the control will adopt the most approached day, month, or year.

**MCM\_SETCURDATE** message sets a specified date as the current selected data:

```
SYSTEMTIME systime;
SendMessage (hwndMonthcal, MCM_SETCURDATE, 0, (LPARAM)&systime) ;
```

**MCM\_SETTODAY** sets today as the current selected date:

```
SendMessage (hwndMonthcal, MCM_SETTODAY, 0, 0) ;
```

### 33.2.3 Adjusting Colors

An application can set or get the color of each element in a month calendar control by **MCM\_GETCOLOR** and **MCM\_SETCOLOR** messages:

```
MCCOLORINFO color;

SendMessage (hwndMonthcal, MCM_GETCOLOR, 0, (LPARAM)&color) ;
SendMessage (hwndMonthcal, MCM_SETCOLOR, 0, (LPARAM)&color) ;
```

Here **color** is a structure of **MCCOLORINFO**, and is used to store color information:

```
typedef struct _MCCOLORINFO
{
    /* background color of the title */
    int clr_titlebk;
    /* text color of the title */
    int clr_titletext;
    /* color of the arrows */
    int clr_arrow;
    /* background color of the highlighted arrows */
    int clr_arrowHibk;

    /* background color of the weekday */
    int clr_weekcaptbk;
    /* text color of the weekday */
    int clr_weekcapttext;

    /* backgroud color of the day */
    int clr_daybk;
    /* background color of the highlighted day */
    int clr_dayHibk;
    /* text color of the current day */
    int clr_daytext;
    /* text color of the other day */
    int clr_trailingtext;
    /* text color of the hilighted day */
    int clr_dayHitext;
} MCCOLORINFO;
```



### 33.2.4 Size of Control

A month calendar has a minimum limit for the window to display content in it normally. `MCM_GETMINREQRECTW` and `MCM_GETMINREQRECTH` messages are used to get the minimum width and the minimum height respectively:

```
int minw, minh;
minw = SendMessage (hwndMonthcal, MCM_GETMINREQRECTW, 0, 0) ;
minh = SendMessage (hwndMonthcal, MCM_GETMINREQRECTH, 0, 0) ;
```

The return values of `SendMessage` functions are the minimum width and height.

### 33.3 Notification Codes of Month Calendar

When user clicked month calendar and the currently selected date is changed, the control will generate `MCN_DATECHANGE` notification code.

### 33.4 Sample Program

Program in List 33.1 illustrates the use of month calendar control. Please refer to `monthcal.c` file of the demo program package of this guide for the complete source code.

List 33.1 Sample Program of month calendar control

```
#define IDC_MC          100
#define IDC_OK          200

/* Dialog box template: only two controls: month calendar control and "OK" button */
static CTRLDATA CtrlTime[] =
{
    {
        "monthcalendar",
        WS_CHILD | WS_VISIBLE | MCS_NOTIFY | MCS_ENG_L,
        10, 10, 240, 180,
        IDC_MC,
        "",
        0
    },
    {
        "button",
        WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,
```

```

        260, 180, 50, 22,
        IDC_OK,
        "OK",
        0
    }
};

static DLGTEMPLATE DlgTime =
{
    WS_VISIBLE | WS_CAPTION | WS_BORDER,
    WS_EX_NONE,
    0, 0, 320, 240,
    "Date time",
    0, 0,
    2, CtrlTime,
    0
};

static int TimeWinProc(HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_INITDIALOG:
            break;

        case MSG_COMMAND:
        {
            int id = LOWORD(wParam);
            if (id == IDC_OK) {
                char info[100];
                SYSTEMTIME date;
                /* Get the current date of month calendar */
                SendMessage (GetDlgItem(hDlg, IDC_MC), MCM_GETCURDATE, 0, (LPARAM)&date);
                sprintf (info, "You will meet Bush president on %d.%d.%d",
                        date.year, date.month, date.day);
                MessageBox (hDlg, info, "Date", MB_OK);
                EndDialog (hDlg, 0);
            }
        }
        break;

        case MSG_CLOSE:
        {
            EndDialog (hDlg, 0);
        }
        return 0;
    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}

```

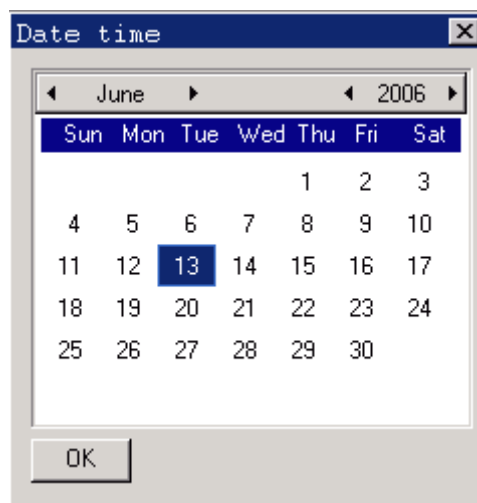


Fig. 33.1 Use of month calendar

## 34 Spin Box Control

The spin box control described in this chapter makes the user be able to select from a group of predefined values. The interface of a spin box control contains up and down arrows, and the user can select a value by clicking the arrows.

The behavior of a spin box is similar with a digital spin box described in Chapter 23. However, a spin box does not display the digit and commonly is used to control other controls, like `timeeditor` described in Section 6.3 “Combined Use of Controls”.

The spin box control is in `mgext` library, so you must initialize `mgext` library to use this control. You can create a spin box control by calling `CreateWindow` function with `CTRL_SPINBOX` as the control class name.

It should be noted that the size of a spin box control is fixed, that is to say, the width and height of the window passed through `CreateWindow` function will be ignored.

### 34.1 Styles of Spin Box

Generally, a spin box control has one style `SPS_AUTOSCROLL`. Spin boxes with this style can automatically determine the present scrolling state of the spin box, and disable (gray) the up and down arrows when it reaches the maximum or the minimum value. The scrolling states of the spin box without this style should be mastered by the application.

If you specify `SPS_TYPE_UPARROW` or `SPS_TYPE_DOWNARROW` style when creating a spin box control, the control will only display one arrow for increasing or decreasing the value, respectively.

If you specify `SPS_HORIZONTAL` style when creating a spin box control, the

control will display the arrows horizontally (vertically by default).

## 34.2 Messages of Spin Box

### 34.2.1 Setting/Getting Position

Generally, after a spin box has been created, we can set the attributes and states of the control by sending `SPM_SETINFO` message to it. Certainly, we can also use this message to reset the attributes of the control:

```
SPININFO spinfo;
SendMessage (hwndSpinBox, SPM_SETINFO, 0, (LPARAM)&spinfo) ;
Here spininfo is a SPININFO structure:
typedef struct _SPININFO
{
    /* The maximum position value */
    int max;
    /* The minimum position value */
    int min;
    /* The current position value */
    int cur;
} SPININFO;
typedef SPININFO *PSPININFO;
```

The fields of `SPININFO` structure specify the maximum position value, the minimum position value, and the current position value of a spin box control. For the spin box control with `SPS_AUTOSCROLL`, the following condition must be satisfied: **maximum position value >= current position value >= minimum position value.**

`SPM_GETINFO` message is used to get the attributes of a spin box control:

```
SPININFO spinfo;
SendMessage (hwndSpinBox, SPM_GETINFO, 0, (LPARAM)&spinfo) ;
```

Here `spinfo` is used to store the gotten attribute values.

`SPM_SETCUR` message is used to set the current position of a spin box control:

```
int cur;
SendMessage (hwndSpinBox, SPM_SETCUR, cur, 0) ;
```

Here `cur` is the desired current position of a spin box to be set, and `cur` should be in the range of the minimum and the maximum when its style is `SPS_AUTOSCROLL`, otherwise the set will fail, and `SendMessage` returns -1.

`SPM_GETCUR` message is used to get the current position:

```
int cur;  
cur = SendMessage (hwndSpinBox, SPM_GETCUR, 0, 0) ;
```

### 34.2.2 Disabling and Enabling

`SPM_DISABLEDOWN`, `SPM_ENABLEDOWN`, `SPM_DISABLEUP` and `SPM_ENABLEUP` are used to disable and enable the scrolling ability of the up/down arrow respectively but never mind the current position reached the maximum or minimum. These messages only work for spin boxes without `SPS_AUTOSCROLL` style. For spin boxes with `SPS_AUTOSCROLL` style, the scrolling ability and states of the arrows are controlled by the control itself:

```
SendMessage (hwndSpinBox, SPM_DISABLEDOWN, 0, 0) ;  
SendMessage (hwndSpinBox, SPM_ENABLEDOWN, 0, 0) ;  
SendMessage (hwndSpinBox, SPM_DISABLEUP, 0, 0) ;  
SendMessage (hwndSpinBox, SPM_ENABLEUP, 0, 0) ;
```

### 34.2.3 Target Window

`SPM_SETTARGET` message sets the target window of a spin box:

```
HWND hTarget;  
SendMessage (hwndSpinBox, SPM_SETTARGET, 0, (LPARAM)hTarget) ;
```

When the user clicks the up/down arrow of the spin box, the spin box will send `MSG_KEYDOWN` and `MSG_KEYUP` message to the target window. `wParam` parameter is `SCANCODE_CURSORBLOCKUP` (when clicking the up arrow) or `SCANCODE_CURSORBLOCKDOWN` (when clicking the down arrow), and `lParam` parameter will have the flag of `KS_SPINPOST` to indicate that the message comes from a spin box.

`SPM_GETTARGET` message gets the target window of a spin box:

```
HWND hTarget;
hTarget = SendMessage (hwndSpinBox, SPM_SETTARGET, 0, 0) ;
```

### 34.3 Notification Codes of Spin Box

A spin box will generate a `SPN_REACHMAX` notification code when it greater than or equal to the maximum. It will generate a `SPN_REACHMIN` notification code when it less than or equal to the minimum.

### 34.4 Sample Program

The program in List 34.1 illustrates the use of a spin box control. Please refer to `spinbox.c` file of the demo program package of this guide for the complete source code.

List 34.1 Example of spin box

```
#define IDC_SPIN10
#define IDC_CTRL1      20
#define IDC_CTRL2      30
#define IDC_CTRL3      40
#define IDC_CTRL4      50

static int
SpinProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    SPININFO spinfo;
    HWND hSpin = GetDlgItem (hDlg, IDC_SPIN);

    switch (message) {
    case MSG_INITDIALOG:
    {
        /* Set the range and current position of a spin box */
        spinfo.min = 1;
        spinfo.max = 10;
        spinfo.cur = 1;
        SendMessage (hSpin, SPM_SETTARGET, 0, (LPARAM)hDlg);
        SendMessage (hSpin, SPM_SETINFO, 0, (LPARAM)&spinfo);
    }
    break;

    case MSG_KEYDOWN:
    {
        /* Handle the key-pressed messagees,
         * including the emulate key stroke messages from the spin box
         */
        if (wParam == SCANCODE_CURSORBLOCKUP ||
            wParam == SCANCODE_CURSORBLOCKDOWN) {
            if (!(lParam & KS_SPINPOST)) {
                int cur;
                cur = SendMessage (hSpin, SPM_GETCUR, 0, 0);
                if (wParam == SCANCODE_CURSORBLOCKUP)
                    cur --;
                else
                    cur ++;
            }
        }
    }
    }
}
```

```

        SendMessage (hSpin, SPM_SETCUR, cur, 0);
    }
    /* Invalidate the window */
    InvalidateRect (hDlg, NULL, TRUE);
}
break;

case MSG_PAINT:
{
    HDC hdc;
    int x, y, w, h;
    int cur;

    cur = SendMessage (hSpin, SPM_GETCUR, 0, (LPARAM)&spinfo);
    x = 10;
    y = cur*10;
    w = 60;
    h = 10;
    if (y < 10)
        y = 10;
    else if (y > 100)
        y = 100;

    /* Draw the window to reflect the current position of the spin box */
    hdc = BeginPaint (hDlg);
    MoveTo (hdc, 2, 10);
    LineTo (hdc, 100, 10);
    Rectangle (hdc, x, y, x+w, y+h);
    SetBrushColor (hdc, PIXEL_black);
    FillBox (hdc, x, y, w, h);
    MoveTo (hdc, 2, 110);
    LineTo (hdc, 100, 110);
    EndPaint (hDlg, hdc);
}
break;

case MSG_CLOSE:
{
    EndDialog (hDlg, 0);
}
break;

}

return DefaultDialogProc (hDlg, message, wParam, lParam);
}

/* The dialog box template */
static DLGTEMPLATE DlgSpin =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    100, 100, 320, 240,
    "Spinbox and black block",
    0, 0,
    1, NULL,
    0
};

/* The dialog box has only one control: the spin box control */
static CTRLDATA CtrlSpin[] =
{
    {
        CTRL_SPINBOX,
        SPS_AUTOSCROLL | WS_BORDER | WS_CHILD | WS_VISIBLE,
        200, 120, 0, 0,
        IDC_SPIN,
        "",
        0
    }
};

```

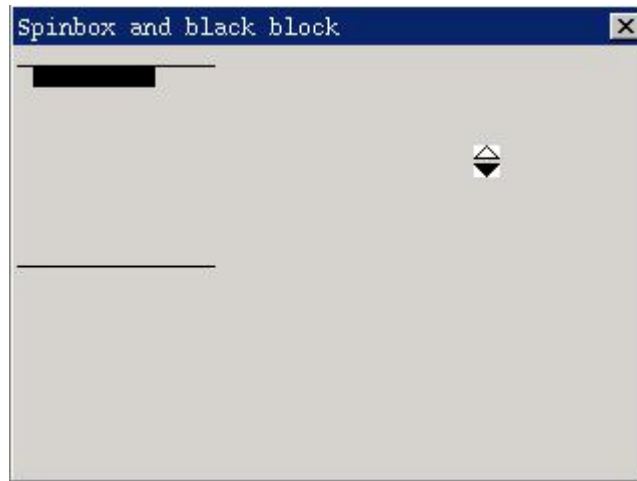


Fig. 34.1 The use of spin box control

The `spinbox.c` program creates a spin box control with `SPS_AUTOSCROLL` in a dialog box, and the user can operate the black block to move between the top and bottom lines by clicking the up and down arrows of the spin box.



## 35 Cool Bar Control

The cool bar control is a toolbar which can display a line of text or icon buttons. The cool bar is very simple and is easy to use.

The cool bar control is defined in `mgext` library, so you must initialize MiniGUIExt library to use this control. You can create a cool bar control by calling `CreateWindow` function with `CTRL_COOLBAR` as the control class name.

### 35.1 Styles of Cool Bar

The button items of cool bar with the styles `CBS_BMP_16X16` and `CBS_BMP_32X32` will display bitmaps with size of 16x16 and 32x32 respectively. The button items of a cool bar with `CBS_BMP_CUSTOM` style will use bitmaps with customized size. For cool bar controls with this style, you should pass the height and width of the bitmap to the control through `dwAddData` argument when calling `CreateWindow` to create the control:

```
CreateWindowEx (CTRL_COOLBAR, ..., MAKELONG (item_width, item_height));
```

A cool bar with `CBS_USEBKBMF` style has a background bitmap, and you should pass the path of the bitmap file to the control by `spCaption` argument of `CreateWindow` function when creating the control.

```
CreateWindowEx (CTRL_COOLBAR, "res/bk.bmp", ...);
```

Cool bar cannot accept height when create it.

### 35.2 Messages of Cool Bar

After a cool bar has been created, we can use `CBM_ADDITEM` message to add items to the toolbar:

```
COOLBARITEMINFO itemInfo;
SendMessage (hwndCoolBar, CBM_ADDITEM, 0, (LPARAM)&itemInfo) ;
```

Here `itemInfo` is a structure of `COOLBARITEMINFO` type:

```
typedef struct _COOLBARITEMINFO
{
    /* Reserved */
    int insPos;
    /* The identifier of the button item */
    int id;
    /* The type of the button item */
    int ItemType;
    /* The bitmap object used by the button */
    PBITMAP Bmp;
    /* The hint text of the button */
    const char *ItemHint;
    /* The caption of the */
    const char *Caption;
    /* The additional data of the button item */
    DWORD dwAddData;
} COOLBARITEMINFO;
```

Here `id` is the identifier of the item in a toolbar. When the user clicks the item, the cool bar will generate notification messages, and the high word of `wParam` is the identifier value of the corresponding item, the low word of `wParam` is the identifier value of the tool item.

`ItemType` specifies the type of the item, value of which can be one of `TYPE_BARITEM`, `TYPE_BMPITEM`, and `TYPE_TEXTITEM`. Item of `TYPE_BARITEM` is a vertical separator; item of `TYPE_BMPITEM` is a bitmap button; and item of `TYPE_TEXTITEM` is a text button.

If the type of an item is `TYPE_BMPITEM`, `bmp` specifies the bitmap object used by the item. `ItemHint` is the prompt text to be displayed when the mouse move onto the item. If the style of an item is `TYPE_TEXTITEM`, `Caption` should point to the text string displayed on the item. `dwAddData` is the additional data of an item.

`CBM_ENABLE` message disables or enables an item:

```
int id;
BOOL beEnabled;
SendMessage (hwndCoolBar, CBM_ENABLE, id, beEnabled) ;
```

Here `id` is the identifier value of the item to be set. If `beEnabled` is `TRUE`, enable the item and `FALSE` disable the item.

### 35.3 Sample Program

The program in List 35.1 illustrates the use of a cool bar control. Please refer to `coolbar.c` file of the demo program package of this guide for the complete source code.

List 35.1 Use of cool bar control

```
#define ITEM_NUM 10

/* The text to be displayed on the coolbar */
static const char* caption[] =
{
    "0", "1", "2", "3", "4", "5", "6", "7", "8", "9"
};

/* Specify the hint text */
static const char* hint[] =
{
    "Number 0", "Number 1", "Number 2", "Number 3", "Number 4",
    "Number 5", "Number 6", "Number 7", "Number 8", "Number 9"
};

/* Create the coolbar, and add the tool items */
static void create_coolbar (HWND hWnd)
{
    HWND cb;
    COOLBARITEMINFO item;
    int i;

    cb = CreateWindow (CTRL_COOLBAR,
        "",
        WS_CHILD | WS_VISIBLE | WS_BORDER,
        100,
        10, 100, 100, 20,
        hWnd,
        0);

    item.ItemType = TYPE_TEXTITEM;
    item.Bmp = NULL;
    item.dwAddData = 0;
    for (i = 0; i < ITEM_NUM; i++) {
        item.insPos = i;
        item.id = i;
        item.Caption = caption[i];
        item.ItemHint = hint[i];
        SendMessage (cb, CBM_ADDITEM, 0, (LPARAM)&item);
    }
}

static int CoolbarWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    static HWND ed;

    switch (message) {
    case MSG_CREATE:
        /* Create an edit box to feedback the user's operation on the toolbar */
        ed = CreateWindow (CTRL_EDIT,
            "",
```

```

        WS_CHILD | WS_VISIBLE | WS_BORDER,
        200,
        10, 10, 100, 20,
        hWnd,
        0);

    create_coolbar (hWnd);
    break;

case MSG_COMMAND:
{
    int id = LOWORD (wParam);
    int code = HIWORD (wParam);

    if (id == 100) {
        static char buffer[100];
        char buf[2];

        /* Type appropriate character into the edit box according to
         * the tool item clicked by the user
         */
        sprintf (buf, "%d", code);
        SendMessage (ed, MSG_GETTEXT, 90, (LPARAM)buffer);
        strcat (buffer, buf);
        SendMessage (ed, MSG_SETTEXT, 0, (LPARAM)buffer);
    }
}
break;

case MSG_DESTROY:
    DestroyAllControls (hWnd);
    return 0;

case MSG_CLOSE:
    DestroyMainWindow (hWnd);
    PostQuitMessage (hWnd);
    return 0;
}

return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

/* Following codes to create the main window are omitted */

```



Fig. 35.1 Use of cool bar control

This program creates a tool bar comprised of "0-9" figures in a dialog box. When the user clicks the item in the toolbar, the corresponding figure is typed into the above edit box.

## 36 Animation Control

Animation control is one control, which can be used to display animations; it is very simple and easy to use.

Animation control is included in `mgext` library, so you must initialize MiniGUIExt library for using the control. You can create an animation control by calling `CreateWindow` function with `CTRL_ANIMATION` as the control class name.

### 36.1 ANIMATION Object

Before creating an animation control, you must create an object of `ANIMATION`. This object is actually a linked list structure, and each node of the linked list represents a frame of the animation object. An object of `ANIMATION` is represented by the following two structures:

```

/** Animation frame structure. */
typedef struct _ANIMATIONFRAME
{
    /** The disposal method (from GIF89a specification):
     *  Indicates the way in which the graphic is to be treated after being displayed.
     *  - 0\n No disposal specified. The decoder is not required to take any action.
     *  - 1\n Do not dispose. The graphic is to be left in place.
     *  - 2\n Restore to background color. The area used by the frame must be restored to
     *      the background color.
     *  - 3\n Restore to previous. The decoder is required to restore the area overwritten
    by
     *      the frame with what was there prior to rendering the frame.
     */
    int disposal;
    /** The x-coordinate of top-left corner of the frame in whole animation screen. */
    int off_x;
    /** The y-coordinate of top-left corner of the frame in whole animation screen. */
    int off_y;
    /** The width of the frame. */
    unsigned int width;
    /** The height of the frame. */
    unsigned int height;

    /** The time of the frame will be display, in the unit of animation time_unit. */
    unsigned int delay_time;
#ifdef _USE_NEWGAL
    /** The memdc compatible with the gif image. */
    HDC mem_dc;
    /** The bits of the mem_dc, should be freed after deleting the mem_dc. */
    UInt8* bits;
#else
    /** The bitmap of the frame. */
    BITMAP bmp;
#endif

    /** The next frame */
    struct _ANIMATIONFRAME* next;
}
  
```

```

/** The previous frame */
struct _ANIMATIONFRAME* prev;
} ANIMATIONFRAME;

/** Animation structure */
typedef struct _ANIMATION
{
    /** The width of the animation. */
    unsigned int width;
    /** The height of the animation. */
    unsigned int height;

    /** The background color */
    RGB bk;

    /** The number of all frames. */
    int nr_frames;
    /**
     * The unit of the time will be used count the delay time of every frame.
     * The default is 1, equal to 10ms.
     */
    int time_unit;
    /** Pointer to the animation frame.*/
    ANIMATIONFRAME* frames;
} ANIMATION;

```

What the **ANIMATION** structure describes are the global properties of objects of animation, and includes the width and height of an animation, number of frames, time scale used for indicating delaying (1 means 10ms), and the header pointer to the linked list of animation frames.

**ANIMATIONFRAME** structure presents a single animation frame, and includes the following information:

- Offset of the current animation frame in the global animation, and the width and height of the frame. Since an animation frame may only change partial image information, so including only the changed part in the frame structure will significantly reduce the amount of data for frames.
- Delay time of the current frame. Calculate the display time of the current frame with `time_unit` in the **ANIMATION** object as the scale.
- Image information of the current frame. When you use NEWGAL interface, the image is represented by a memory DC; otherwise a BITMAP object represents the image.

An application can construct an **ANIMATION** object itself, and can also call the following function to directly create an **ANIMATION** object from an image file in GIF98a format.

```
ANIMATION* CreateAnimationFromGIF89a (HDC hdc, MG_RWops* area);  
ANIMATION* CreateAnimationFromGIF89aFile (HDC hdc, const char* file);  
ANIMATION* CreateAnimationFromGIF89aMem (HDC hdc, const void* mem, int size);
```

The above-mentioned function will read the image information of the animation GIF from the data area in GIF89a format, and then create an **ANIMATION** object.

After an application creates an **ANIMATION** object, and can both display it, also can create an animation control to display the animation. When you call **CreateWindow** function to create an animation control, you can pass the created **ANIMATION** object to the animation control and the animation control will use the **ANIMATION** object to display the animation automatically. For example, an **ANIMATION** object is created from a GIF file in the following code fragment, and then using the object creates an animation control:

```
ANIMATION* anim = CreateAnimationFromGIF89aFile (HDC_SCREEN, "banner.gif");  
if (anim == NULL)  
    return 1;  
  
CreateWindow (CTRL_ANIMATION,  
             "",  
             WS_VISIBLE | ANS_AUTOLOOP,  
             100,  
             10, 10, 300, 200, hWnd, (DWORD)anim);
```

It should be noted that you could pass the pointer to the **ANIMATION** object into an animation control through **dwAddData** argument when calling **CreateWindow** function.

## 36.2 Styles of Animation Control

At present, there are three styles for an animation control:

- **ANS\_AUTOLOOP**: An animation control will display the animation circularly after the style is used.
- **ANS\_SCALED**: The display image can be scaled.
- **ANS\_FITTOANI**: Fit the control size follows the animation.

### 36.3 Messages of Animation Control

Messages of animation controls are very simple. There are the following several message currently, which can be used to control the display action of an animation control.

- **ANM\_SETANIMATION**: Set an **ANIMATION** object.
- **ANM\_GETANIMATION**: Get the current **ANIMATION** object.
- **ANM\_STARTPLAY**: Start playing the animation. Before sending **ANM\_STARTPLAY** to an animation control, the animation control will only display the first frame image of the **ANIMATION** object; and only after **ANM\_STARTPLAY** message is sent, the animation control can display an animation according to the information in **ANIMATION** object.
- **ANM\_PAUSE\_RESUME**: Pause/Resume playing. This message is used to pause the play of an animation (during playing), or used to resume the play of an animation (when the animation has been paused).
- **ANM\_STOPPLAY**: Stop the play of an animation. The animation control displays the first frame of the **ANIMATION**.

### 36.4 Sample Program

Code in List 36.1 illustrates the use of an animation control. Please refer to **animation.c** file of the demo program package **mg-samples** of this guide for complete source code.

List 36.1 Use of animation control

```
static int AnimationWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_CREATE:
        {
            ANIMATION* anim = CreateAnimationFromGIF89aFile (HDC_SCREEN, "banner.gif");
            if (anim == NULL)
                return 1;

            SetWindowAdditionalData (hWnd, (DWORD) anim);
            CreateWindow (CTRL_ANIMATION,
                        "",
                        WS_VISIBLE | ANS_AUTOLOOP,
                        100,
                        10, 10, 300, 200, hWnd, (DWORD) anim);
            SendMessage (GetDlgItem (hWnd, 100), ANM_STARTPLAY, 0, 0);

            CreateWindow (CTRL_ANIMATION,
```



```

        "",
        WS_VISIBLE | ANS_AUTOLOOP,
        200,
        10, 210, 300, 200, hWnd, (DWORD)anim);

    break;
}

case MSG_LBUTTONDOWN:
    SendMessage (GetDlgItem (hWnd, 200), ANM_STARTPLAY, 0, 0);
    break;

case MSG_DESTROY:
    DestroyAnimation ((ANIMATION*)GetWindowAdditionalData (hWnd), TRUE);
    DestroyAllControls (hWnd);
    return 0;

case MSG_CLOSE:
    DestroyMainWindow (hWnd);
    PostQuitMessage (hWnd);
    return 0;
}

return DefaultMainWinProc(hWnd, message, wParam, lParam);
}

/* Following codes to create the main window are omitted */

```

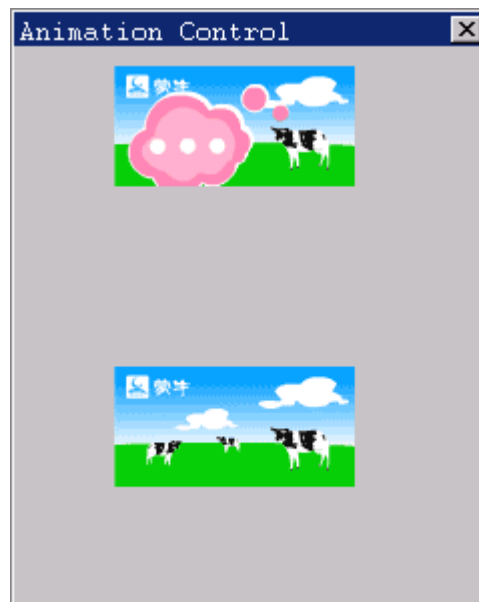


Fig. 36.1 Use of an animation control

In the program within List 36.1, **banner.gif** file in the current directory is loaded when the main window is created, and the corresponding **ANIMATION** object is created. Then two animation controls are created. The first animation control start playing the animation immediately after created, and the second animation control won't play until the user clicks the window. Fig. 36.1 shows the running effect of the sample program, in which what the first animation control displays is the second frame of **banner.gif** file, and the second animation control displays the first frame.



## 37 GridView Control

GridView displays a series of data items (cells) in table form, and the contents of every cell are independent each other. The contents of gridview header, including a column of header and a row of header, usually express the meanings of the column and the row. From the appearance, gridview is a rectangle box including header cells. You can adjust the height of the row and the width of the column in the gridview by dragging the header, and use the scrollbar to display the content, which was out of the display window.

GridView is a convenient and effective tool, which can arrange and display the data item. It suits to deal with a great deal of datum with different attribute, such as experiment data or account table.

GridView is contained in `mgext` library; you must initialize the `mgext` library at first. You can call `CreateWindow` function and use the control name `CTRL_GRID` to create a GridView. Applications usually add, delete or operate the table items by sending messages to the GridView. Like other controls, GridView will not generate messages until it responds user clicking or other operations.

### 37.1 Styles of Gridview

By default, GridView window only displays header and cell, and there is no border in the display region. When creating GridView with `CreateWindow` function, you can use window style `WS_BORDER` to add the border in it. Otherwise, use window style `WS_VSCROLL` and `WS_HSCROLL` to add upright and horizontal scroll bars. It is convenient for using mouse to display all contents in the GridView by scrolling the bar.

### 37.2 Messages of GridView

When creating GridView, you can set a structure `GRIDVIEWDATA` and transfer

this structure as a parameter. The definition of this structure and the meanings of every member are:

```
typedef struct _GRIDVIEWDATA
{
    /** The number of the rows */
    int nr_rows;
    /** The number of the columns */
    int nr_cols;
    /** The default height of a row */
    int row_height;
    /** The default width of a column */
    int col_width;
} GRIDVIEWDATA;
```

### 37.2.1 Column Operations

After creating the GridView, user demands to add a column to the control, which can be finished by application sending **GRIDM\_ADDCOLUMN** message to the control.

```
int index;
GRIDCELLEDATA celldata;
GRIDCELLEDATAHEADER cellheader;
SendMessage(hWndGrid, GRIDM_ADDCOLUMN, index, &celldata);
```

Here **celldata** is a **GRIDCELLEDATA** structure, including the information about the new added column. The **GRIDCELLEDATA** structure definition and the meanings of every member are:

```
typedef struct _GRIDCELLEDATA
{
    /** mask of properties, can be OR'ed with following values:
     * set or get a cell style
     * - GVITEM_STYLE\n
     * set or get a cell text color
     * - GVITEM_FGCOLOR\n
     * set or get a cell background color
     * - GVITEM_BGCOLOR\n
     * set or get a cell text font
     * - GVITEM_FONT\n
     * set or get a cell's image
     * - GVITEM_IMAGE\n
     * set or get all of the content of a cell
     * - GVITEM_ALLCONTENT\n
     * set or get the main content of a cell
     * - GVITEM_MAINCONTENT\n
     */
    DWORD mask;

    /** the style of the cell */
    DWORD style;
    /** text color */
    gal_pixel color_fg;
    /** the background color */
    gal_pixel color_bg;
```

```

/** text font */
PLOGFONT font;
/** Pointer to the bitmap of one cell */
PBITMAP image;
/** the concrete data of one cell */
void* content;
}GRIDCELLDATA;

```

**content** field in the above structure points to the address of another structure **GRIDCELLDATAHEADER**. The definition of this structure and the meanings of every member are:

```

typedef struct _GRIDCELLDATAHEADER
{
    /** the height of a row or the width of a column */
    int size;
    /** the caption of the row or column */
    char* buff;
    /** the length of the buff string */
    int len_buff;
}GRIDCELLDATAHEADER;

```

Before adding a new column, you should set the number **size** of the structure **GRIDCELLDATAHEADE** and the number **buff** point to the caption of the column. The number **len\_buff** is the length of the caption. Adding a row is same as adding a column. But the number **size** is the height of the new row.

Before adding a new column, you should set the **size** in this structure, which expresses the width of this new column. Then, you should set **buff**, which expresses the caption of this column header. The operation to add a new row is same as adding column, and the difference is to set **height** in **GRIDCELLDATAHEADER** structure, which is the height of the new row.

**GRIDCELLDATA** structure is to set the attributes of the row, column and cell in GridView, which is used by many messages, such as **GRIDM\_SETCELLPROPERTY**, **GRIDM\_GETCELLPROPERTY**, **GRIDM\_ADDROW** and **GRIDM\_ADDCOLUMN** etc.

**style** field in the **GRIDCELLDATA** structure is the style of cell. Every times, when you set, you should point out which type is among below options: **GV\_TYPE\_HEADER**, **GV\_TYPE\_TEXT**, **GV\_TYPE\_NUMBER**, **GV\_TYPE\_SELECTION** and **GV\_TYPE\_CHECKBOX**. It can be used with cell style, such as **GVS\_READONLY** etc.

**content** field also can point to other structures, which are **GRIDCELLDATATEXT** (text cell), **GRIDCELLDATANUMBER** (data cell), **GRIDCELLDATASELECTION** (combo boxes cell), **GRIDCELLDATAcheckbox**(selection cell). The definition and the meaning of every member are:

```
typedef struct _GRIDCELLDATATEXT
{
    /** the caption of the row or column */
    char* buff;
    /** the length of the buff string */
    int len_buff;
}GRIDCELLDATATEXT;

typedef struct _GRIDCELLDATANUMBER
{
    /** the value of the number type cell*/
    double number;
    /** the format of the number to display */
    char* format;
    /** the length of the format string */
    int len_format;
}GRIDCELLDATANUMBER;

typedef struct _GRIDCELLDATAcheckbox
{
    /** whether this checkbox is checked */
    BOOL checked;
    /** the text after checkbox */
    char* text;
    /** when used for set, -1 means null-terminated */
    int len_text;
}GRIDCELLDATAcheckbox;

typedef struct _GRIDCELLDATASELECTION
{
    /** the index of the combobox that current selected */
    int cur_index;
    /** "Yes\nNo\n"*/
    char* selections;
    /** when used for set, -1 means null-terminated */
    int len_sel;
}GRIDCELLDATASELECTION;
```

**GRIDM\_SETCOLWIDTH** can be used to set the width of the control column:

```
int index;
int width;
SendMessage (hwndGrid, GRIDM_SETCOLWIDTH, index, width) ;
```

Here **index** is the integral index value of the column needed to set, and **width** is the width of the column.

**GRIDM\_GETCOLWIDTH** can get the width of the control column:

```
int width;
int index;
width = SendMessage (hwndGrid, GRIDM_GETCOLWIDTH, 0, index);
```

Here **index** is the integral index value of the column needed to get, and the return value of the **SendMessage** function is the width of column. If error occurs, returns -1.

**GRIDM\_ADDCOLUMN** message is used to add a column.

```
int index;  
GRIDCELldata* celldata;  
SendMessage (hwndGrid, GRIDM_ADDCOLUMN, index, celldata) ;
```

Here **index** is the integral index value of the column upon the added column, and **celldata** is a pointer of **GRIDCELldata** structure, which is used to set initial value for new column. **GRIDM\_DELCOLUMN** message is used to delete a column in GridView.

```
int index;  
SendMessage (hwndGrid, GRIDM_DELCOLUMN, 0, index) ;
```

Here **index** is the index value of the deleted column.

**GRIDM\_GETCOLCOUNT** message is used to get the number of columns in GridView.

```
int count;  
count = SendMessage (hwndGrid, GRIDM_GETCOLCOUNT, 0, 0) ;
```

The return value of **SendMessage** function is the number of columns. This message will return -1 on error.

### 37.2.2 Row Operations

Row operations are same as column operations.

**GRIDM\_SETROWHEIGHT** can be used to set the height of the row.

```
int index;
```

```
int height;  
SendMessage (hwndGrid, GRIDM_SETROWHEIGHT, index, height) ;
```

Here, **index** is the integral index value of the row needed to set, and **height** is the set height.

**GRIDM\_GETROWHEIGHT** can get the width of the row.

```
int height;  
int index;  
height = SendMessage (hwndGrid, GRIDM_GETROWHEIGHT, 0, index);
```

Here, **index** is the integral index value of the row needed to get, and the return value of the **SendMessage** function is the height of the row. The message will return -1 on error.

**GRIDM\_ADDROW** message is used to add a new row in GridView.

```
int index;  
GRIDCELLEDATA* celldata;  
SendMessage (hwndGrid, GRIDM_ADDROW, index, celldata) ;
```

Here, **index** is the integral index value of row upon the added row; **celldata** is a pointer of **GRIDCELLEDATA** structure, which is used to set the initial value of the new row.

**GRIDM\_DELROW** message is used to delete a row in GridView.

```
int index;  
SendMessage (hwndGrid, GRIDM_DELROW, 0, index) ;
```

Here **index** is the index value of the deleted row.

**GRIDM\_GETROWCOUNT** is used to get the number of the rows in GridView.

```
int count;  
count = SendMessage (hwndGrid, GRIDM_GETROWCOUNT, 0, 0) ;
```

The return value of **SendMessage** function is the number of the rows.



### 37.2.3 Cell Operations

**GRIDM\_SETCELLPROPERTY** message is used to set one or many cell.

```
GRIDCELLS* cells;
GRIDCELldata* celldata;
SendMessage (hwndGrid, GRIDM_SETCELLPROPERTY, cells, celldata) ;
```

Here, **cells** is a pointer of **GRIDCELLS** structure, which expresses the range of the cell needed to set. The definition of the **GRIDCELLS** structure and the meanings of every member are:

```
typedef struct _GRIDCELLS
{
    /** the start row of the selected cell(s) */
    int row;
    /** the start column of the selected cell(s) */
    int column;
    /** the number of the column(s) which contain(s) selected cell(s) */
    int width;
    /** the number of the row(s) which contain(s) selected cell(s) */
    int height;
}GRIDCELLS;
```

If **SendMessage** function is ok, return **GRID\_OKAY**; otherwise, return **GRID\_ERR**.

**GRIDM\_GETCELLPROPERTY** message is used to obtain the attribute of cell.

```
GRIDCELLS* cells;
GRIDCELldata* celldata;
SendMessage (hwndGrid, GRIDM_GETCELLPROPERTY, cells, celldata) ;
```

Here, **cells** is an idiographic cell, which is not multi-cell. After the content of certain cell is set successfully, the function **SendMessage** will return **GRID\_OKAY**. The structure **celldata** contains the information of the certain cell. If error occurred, the message will return **GRID\_ERR**.

Otherwise, there are some other messages for cells with different format, such as **GRIDM\_SETNUMFORMAT** message, which is used to set the data format of data cell (**GRIDCELldataNUMBER**).

```
GRIDCELLS* cells;
char* format = "%3.2f";
```

```
SendMessage (hwndGrid, GRIDM_SETNUMFORMAT, cells, format);
```

Here, `cells` is the cell to be set, and `format` is the data format to be set.

For all kinds of cells, `GRIDM_SETSELECTED` is used to set highlighted cell.

```
GRIDCELLS* cells;
SendMessage (hwndGrid, GRIDM_SETSELECTED, 0, cells);
```

Here, `cells` is the cell to be set highlighted. If the cell is set highlighted successfully, the function `SendMessage` will return `GRID_OKAY`, otherwise it will return `GRID_ERROR`.

`GRIDM_GETSELECTED` is used to get all highlighted cells.

```
GRIDCELLS* cells;
SendMessage (hwndGrid, GRIDM_GETSELECTED, 0, cells);
```

With this option, the function `SendMessage` will return all the highlighted cells.

### 37.3 Other Messages

When user presses the `up/down` or `left/right` arrow key, the selected cell will change, and the new selected option will turn visible (if it has been invisible). When user presses the `PAGEUP/PAGEDOWN` key, the column cell will turn to another page. The page change range is the same as the scrollbar; the last item on first page will turn to second one's first item. If `HOME` key is pressed, the first cell in the column will be selected and become visible. If `END` key is pressed, the last cell will be selected and become visible. When all the above keys are pressed with `SHIFT` on the same time, the operation on highlighted area will be carried out. When cell is double clicked or selected, it will edit the content of the cell to input character.

The grid control is also able to associate some cells (source cells) with other cells (target cells). Then the target cells will refresh themselves according to

the given operation function when the source cells' data is changed. The structure to carry out this operation is listed below:

```
typedef struct _GRIDCELLDEPENDENCE
{
    /* source cells */
    GRIDCELLS source;
    /* target cells */
    GRIDCELLS target;
    /* data operation function */
    GRIDCELLEVALCALLBACK callback;
    /* additional information */
    DWORD dwAddData;
}GRIDCELLDEPENDENCE;

/* prototype of the data operation function */
typedef int (*GRIDCELLEVALCALLBACK)(GRIDCELLS* target,
    GRIDCELLS* source, DWORD dwAddData);
```

The message, **GRIDM\_ADDDEPENDENCE**, is used to add an association of cells.(It should be noted that source cell and target cell can not intersect and target cell also can not intersect with other target cells of the control).

```
GRIDCELLDEPENDENCE* dependence;
SendMessage (hwndGrid, GRIDM_ADDDEPENDENCE, 0, dependence);
```

If success, the message will return the index of the association; otherwise it will return **GRID\_ERROR**.

The message **GRIDM\_DELDEPENDENCE** is used to delete a cell association in the GridView.

```
int dependence_id;
SendMessage (hwndGrid, GRIDM_DELDEPENDENCE, 0, dependence_id);
```

Here the **dependence\_id** is the index of the associated cell to be deleted. The message function will return **GRID\_OKAY** when deletion is successful, otherwise it will return **GRID\_ERR**.

## 37.4 Notification Codes of GridView

The GridView will generate notification code when it responds to user's operation such as clicking or some status is changed. The notification codes

include:

- **GRIDN\_HEADLDOWN**: the left button of the user's mouse is pressed on the table head
- **GRIDN\_HEADLUP**: the left button of the user's mouse is released on the table head
- **GRIDN\_KEYDOWN**: key is pressed down
- **GRIDN\_CELLDBCLK**: user double click one cell
- **GRIDN\_CELLCLK**: user single click one cell
- **GRIDN\_FOCUSCHANGED**: the focus cell is changed
- **GRIDN\_CELLTEXTCHANGED**: the content of the cell is changed

When the left button of the user's mouse is pressed on some cell, the cell will be selected, and two notification codes **GRIDN\_FOCUSCHANGED** and **GRIDN\_CELLCLK** will be generated.

If an application needs to know the notification code of the grid control, a notification handle function should be registered using **SetNotificationCallback** to handle all the received notification code.

## 37.5 Sample Program

Code in List 37.1 illustrates the use of a GridView control. Please refer to **grid.c** file of the demo program package **mg-samples** of this guide for complete source code.

List 37.1 Use of GridView Control

```
int ww = 800;
int wh = 600;

enum {
    IDC_GRIDVIEW,
};

static HWND hGVWnd;

static char* colnames[] = {"Chinese", "Math", "English", "Total"};
static char* scores[] = {"Rose", "Joan", "Rob", "John", "Average"};

int total(GRIDCELLS* target, GRIDCELLS* source, DWORD dwAddData)
{
    int i, j;
```

```

double value = 0;
GRIDCELLEDATA data;
GRIDCELLS cells;
GRIDCELLDATANUMBER num;
memset(&data, 0, sizeof(data));
memset(&num, 0, sizeof(num));
data.mask = GVITEM_MAINCONTENT|GVITEM_STYLE;
data.content = &num;
data.style = GV_TYPE_NUMBER;
cells.width = 1;
cells.height = 1;
for(i = 0; i<source->width; i++)
{
    cells.column = source->column + i;
    for (j = 0; j<source->height; j++)
    {
        cells.row = source->row + j;
        SendMessage(hGVWnd, GRIDM_GETCELLPROPERTY, (WPARAM)&cells, (LPARAM)&data);
        value += num.number;
    }
}
num.number = value;
num.len_format = -1;
cells.row = target->row;
cells.column = target->column;
SendMessage(hGVWnd, GRIDM_SETCELLPROPERTY, (WPARAM)&cells, (LPARAM)&data);

return 0;
}

int averge(GRIDCELLS* target, GRIDCELLS* source, DWORD dwAddData)
{
    int i, j;
    int count = 0;
    double value = 0;
    GRIDCELLEDATA data;
    GRIDCELLS cells;
    GRIDCELLDATANUMBER num;
    memset(&data, 0, sizeof(data));
    memset(&num, 0, sizeof(num));
    data.content = &num;
    data.style = GV_TYPE_NUMBER;
    cells.width = 1;
    cells.height = 1;
    for(i = 0; i<source->width; i++)
    {
        cells.column = source->column + i;
        for (j = 0; j<source->height; j++)
        {
            data.content = &num;
            data.style = GV_TYPE_NUMBER;
            cells.row = source->row + j;
            SendMessage(hGVWnd, GRIDM_GETCELLPROPERTY, (WPARAM)&cells, (LPARAM)&data);
            value += num.number;
            count++;
        }
    }
    data.mask = GVITEM_MAINCONTENT;
    num.number = value/count;
    cells.row = target->row;
    cells.column = target->column;
    SendMessage(hGVWnd, GRIDM_SETCELLPROPERTY, (WPARAM)&cells, (LPARAM)&data);

    return 0;
    return 0;
}

static int
ControlTestWinProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case MSG_CREATE:
        {
            GRIDVIEWDATA gvdata;

```

```

gvdata.nr_rows = 10;
gvdata.nr_cols = 10;
gvdata.row_height = 30;
gvdata.col_width = 60;
hGVWnd = CreateWindowEx (CTRL_GRIDVIEW, "Grid View",
                        WS_CHILD | WS_VISIBLE | WS_VSCROLL |
                        WS_HSCROLL | WS_BORDER, WS_EX_NONE, IDC_GRIDVIEW, 2
0, 20, 600,
                        300, hWnd, (DWORD)&gvdata);

int i;
GRIDCELLS cellsel;
GRIDCELLDEPENDENCE dep;
GRIDCELLDATA celldata;
GRIDCELLDATAHEADER header;
GRIDCELLDATANUMBER cellnum;
memset(&header, 0, sizeof(header));
memset(&celldata, 0, sizeof(celldata));
/* set the attribute of the column header */
for (i = 1; i<= 3; i++)
{
    header.buff = colnames[i-1];
    header.len_buff = -1;
    celldata.content = &header;
    celldata.mask = GVITEM_MAINCONTENT;
    celldata.style = GV_TYPE_HEADER;
    cellsel.row = 0;
    cellsel.column = i;
    cellsel.width = 1;
    cellsel.height = 1;
    SendMessage(hGVWnd, GRIDM_SETCELLPROPERTY, (WPARAM)&cellsel, (LPARAM)&
celldata);

}

/* set the attribute of the column header */
memset(&header, 0, sizeof(header));
memset(&celldata, 0, sizeof(celldata));
for (i = 1; i<= 4; i++)
{
    header.buff = scores[i-1];
    celldata.content = &header;
    celldata.mask = GVITEM_MAINCONTENT;
    celldata.style = GV_TYPE_HEADER;
    cellsel.row = i;
    cellsel.column = 0;
    cellsel.width = 1;
    cellsel.height = 1;
    SendMessage(hGVWnd, GRIDM_SETCELLPROPERTY, (WPARAM)&cellsel, (LPARAM)&
celldata);

}

/* set the attribute of the cell */
memset(&celldata, 0, sizeof(celldata));
memset(&cellnum, 0, sizeof(cellnum));
cellnum.number = 50;
cellnum.format = NULL;
celldata.content = &cellnum;
celldata.mask = GVITEM_MAINCONTENT;
celldata.style = GV_TYPE_NUMBER;
cellsel.row = 1;
cellsel.column = 1;
cellsel.width = 3;
cellsel.height = 4;
SendMessage(hGVWnd, GRIDM_SETCELLPROPERTY, (WPARAM)&cellsel, (LPARAM)&cel
ldata);

/* operation to add one column */
memset(&header, 0, sizeof(header));
memset(&celldata, 0, sizeof(celldata));
header.buff = "Total";
header.size = -1;
celldata.mask = GVITEM_MAINCONTENT;
celldata.content = &header;
celldata.style = GV_TYPE_HEADER;

```

```

        SendMessage(hGVWnd, GRIDM_ADDCOLUMN, 3, (LPARAM)&celldata);

        file://operation to add one row
        memset(&header, 0, sizeof(header));
        memset(&celldata, 0, sizeof(celldata));
        header.buff = "Average";
        header.size = -1;
        celldata.mask = GVITEM_MAINCONTENT;
        celldata.content = &header;
        celldata.style = GV_TYPE_HEADER;
        SendMessage(hGVWnd, GRIDM_ADDROW, 4, (LPARAM)&celldata);

        memset(&celldata, 0, sizeof(celldata));
        memset(&cellnum, 0, sizeof(cellnum));
        cellnum.number = 0;
        cellnum.format = NULL;
        celldata.content = &cellnum;
        celldata.mask = GVITEM_MAINCONTENT;
        celldata.style = GV_TYPE_NUMBER;
        cellsel.row = 1;
        cellsel.column = 4;
        cellsel.width = 1;
        cellsel.height = 4;
        SendMessage(hGVWnd, GRIDM_SETCELLPROPERTY, (WPARAM)&cellsel, (LPARAM)&celldata);

        cellsel.row = 5;
        cellsel.column = 1;
        cellsel.width = 4;
        cellsel.height = 1;
        SendMessage(hGVWnd, GRIDM_SETCELLPROPERTY, (WPARAM)&cellsel, (LPARAM)&celldata);

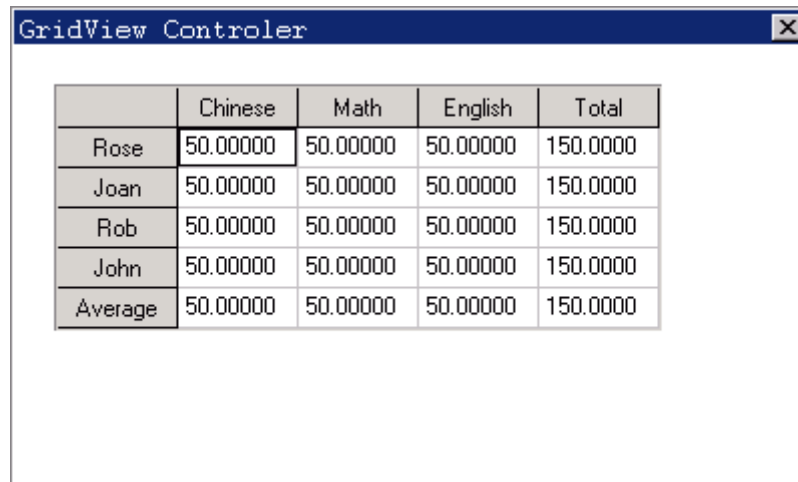
        // set the sum function for this cell.
        memset(&dep, 0, sizeof(dep));
        dep.callback = total;
        for (i = 1; i<= 4; i++)
        {
            dep.source.row = i;
            dep.source.column = 1;
            dep.source.width = 3;
            dep.source.height = 1;
            dep.target.row = i;
            dep.target.column = 4;
            dep.target.width = 1;
            dep.target.height = 1;
            SendMessage(hGVWnd, GRIDM_ADDDEPENDENCE, 0, (LPARAM)&dep);
        }

        dep.callback = averge;
        // set average function for this cell.
        for (i = 1; i<= 4; i++)
        {
            dep.source.row = 1;
            dep.source.column = i;
            dep.source.width = 1;
            dep.source.height = 4;
            dep.target.row = 5;
            dep.target.column = i;
            dep.target.width = 1;
            dep.target.height = 1;
            SendMessage(hGVWnd, GRIDM_ADDDEPENDENCE, 0, (LPARAM)&dep);
        }

        return 0;
    }
    case MSG_COMMAND:
        break;
    case MSG_CLOSE:
        DestroyMainWindow (hWnd);
        MainWindowCleanup (hWnd);
        return 0;
}

```

```
return DefaultMainWinProc (hWnd, message, wParam, lParam);  
}
```



The screenshot shows a window titled "GridView Controller" with a close button in the top right corner. Inside the window is a table with 5 columns and 6 rows. The columns are labeled "Chinese", "Math", "English", and "Total". The rows represent individual students (Rose, Joan, Rob, John) and an "Average" row. Each cell in the table contains a numerical value, specifically 50.00000 for the individual subjects and 150.0000 for the total and average.

	Chinese	Math	English	Total
Rose	50.00000	50.00000	50.00000	150.0000
Joan	50.00000	50.00000	50.00000	150.0000
Rob	50.00000	50.00000	50.00000	150.0000
John	50.00000	50.00000	50.00000	150.0000
Average	50.00000	50.00000	50.00000	150.0000

Fig. 37.1 Use of a GridView control



## 38 IconView Control

**IconView** control offers an interface for user to surf entry in icon and label mode. These entire icon items can be shown in a scroll child window. User can select one or some items using keyboard and mouse operation and the selected icon usually are shown high lightly. The representative usage of **IconView** control is to be the container of the desktop icons and display of files under some directory.

**CreateWindow** using **CTRL\_ICONVIEW** as control class name can create **IconView** control.

We can add, delete, set the size of icon (should be done before adding this icon) and get label text of the icon, etc. by sending corresponding message after creating **IconView** control.

### 38.1 Styles of IconView

By default, the **IconView** control window will only display the icon and its label and there will be no border in display region. You can add border for the **IconView** control by using window style **WS\_BORDER** when create control using **CreatWindow**. On the other hand, you can also use window style **WS\_VSCROLL** and **WS\_HSCROLL** to add vertical and horizontal scroll bar so that you can display all the content in the list control.

**IconView** control is based on the **scrollView** Control and it remains the style of the **scrollView** Control.

## 38.2 Messages of IconView

### 38.2.1 Icon operation

After an `IconView` control is created, the next step is to add icon to the control. Sending an `IVM_ADDITEM` message to the control does this work.

```
IVITEMINFO ivii;
SendMessage (hIconView, IVM_ADDITEM, 0, (LPARAM)&ivii) ;
```

In above program, `ivii` is an `IVITEMINFO` structure and it is used to display the icon information needed to be set. The definition and meaning of the `IVITEMINFO` structure are as below:

```
typedef struct _IVITEMINFO
{
    /**
     * the Index of the item
     */
    int nItem;

    /**
     * the bitmap icon of the item
     */
    PBITMAP bmp;

    /**
     * the text label of the item
     */
    const char *label;

    /** attached additional data of this item */
    DWORD addData;

    /**
     * reserved
     */
    DWORD dwFlags;
} IVITEMINFO;
```

The index value of the icon shows this icon control's position in the parent window. It will return the handle of the icon control when success, otherwise return 0.

The height and width of the icon can be specified by `IVM_SETITEMSIZE` before adding an icon, and all the icons will be displayed in this height and width.

```
int width;
int height;
```

```
SendMessage (hIconView, IVM_SETITEMSIZE, width, height) ;
```

In above code, `width` is the width to be set and `height` is the height to be set.

Since the `IconView` control is based on the `ScrollView`, the rest messages of the `IconView` control are almost the same as the `ScrollView`.

- `IVM_RESETCONTENT` corresponding to `SVM_RESETCONTENT`: used to clear the icon item in `IconView` control.
- `IVM_DELITEM` corresponding to `SVM_DELITEM`: used to delete the icon item in `IconView` control.
- `IVM_SETITEMDRAW` corresponding to `SVM_SETITEMDRAW`: used to set drawing function of the icon item.
- `IVM_SETCONTWIDTH` corresponding to `SVM_SETCONTWIDTH`: used to set the width of the scrollable window.
- `IVM_SETCONTHEIGHT` corresponding to `SVM_SETCONTHEIGHT`: used to set the height of the scrollable window.
- `IVM_SETITEMOPS` corresponding to `SVM_SETITEMOPS`: used to set the callback function of correlation operation of the icon item.
- `IVM_GETMARGINS` corresponding to `SVM_GETMARGINS`: used to get the margin range of the `IconView`.
- `IVM_SETMARGINS` corresponding to `SVM_SETMARGINS`: used to set the margin range of the `IconView`.
- `IVM_GETLEFTMARGIN`, `IVM_GETTOPMARGIN`, `IVM_GETRIGHTMARGIN` and `IVM_GETBOTTOMMARGIN` corresponding to `SVM_GETLEFTMARGIN`, `SVM_GETTOPMARGIN`, `SVM_GETRIGHTMARGIN`, and `SVM_GETBOTTOMMARGIN`: used to get the left, up, right, down margin range of the `IconView` control. `IVM_GETCONTWIDTH`, `IVM_GETCONTHEIGHT`, `IVM_GETVISIBLEWIDTH` and `IVM_GETVISIBLEHEIGHT` corresponding to `SVM_GETCONTWIDTH`, `SVM_GETCONTHEIGHT`, `SVM_GETVISIBLEWIDTH` and `SVM_GETVISIBLEHEIGHT`: Used to get the width and height of the content area and visual area.
- `IVM_SETCONTRANGE` corresponding to `SVM_SETCONTRANGE`: used to set the range of the content area in the scrollable window.
- `IVM_GETCONTENTX` and `IVM_GETCONTENTY` corresponding to

**SVM\_GETCONTENTX** and **SVM\_GETCONTENTY**: used to get the current position of the content area.

- **IVM\_SETCONTPOS** corresponding to **SVM\_SETCONTPOS**: used to set the current position of the content area, in other word, to move content area to a specific position in the visual area.
- **IVM\_GETCURSEL** and **IVM\_SETCURSEL** corresponding to **SVM\_GETCURSEL** and **SVM\_SETCURSEL**: used to get and set the current highlighted icon of the icon control.
- **IVM\_SELECTITEM** corresponding to **SVM\_SELECTITEM**: used to select a column item, and the selected item will be highlighted displayed.
- **IVM\_SHOWITEM** corresponding to **SVM\_SHOWITEM**: used to show an icon item.
- **IVM\_CHOOSEITEM** corresponding to **SVM\_CHOOSEITEM** is the combination of **IVM\_SELECTITEM** and **IVM\_SHOWITEM** message: used to select an icon and visualize it.
- **IVM\_SETITEMINIT** corresponding to **SVM\_SETITEMINIT**: used to set the initial operation of the icon item.
- **IVM\_SETITEMDESTROY** corresponding to **SVM\_SETITEMDESTROY**: used to set the destroy operation of the icon item.
- **IVM\_SETITEMCMP** corresponding to **SVM\_SETITEMCMP**: used to set the comparison function of the iconview control item.
- **IVM\_MAKEPOSVISIBLE** corresponding to **SVM\_MAKEPOSVISIBLE**: used to visualize one position in the content area.
- **IVM\_GETHSCROLLVAL** and **IVM\_GETVSCROLLVAL** corresponding to **SVM\_GETHSCROLLVAL** and **SVM\_GETVSCROLLVAL**: used to get the current horizontal and vertical scroll data (the scroll range by clicking the scroll bar arrow).
- **IVM\_GETHSCROLLPAGEVAL** and **IVM\_GETVSCROLLPAGEVAL** corresponding to **SVM\_GETHSCROLLPAGEVAL** and **SVM\_GETVSCROLLPAGEVAL**: used to get the current horizontal and vertical scroll data (the scroll range of changing page).
- **IVM\_SETSCROLLVAL** corresponding to **SVM\_SETSCROLLVAL**: used to set the horizontal or vertical scroll data of the scroll window.
- **IVM\_SETSCROLLPAGEVAL** corresponding to **SVM\_SETSCROLLPAGEVAL**: used to set the horizontal or vertical scroll data of the scroll window.

- `IVM_SORTITEMS` corresponding to `SVM_SORTITEMS`: used to sort the items of the icon.
- `IVM_GETITEMCOUNT` corresponding to `SVM_GETITEMCOUNT`: used to get the item count of current icon control.
- `IVM_GETITEMADDDATA` corresponding to `SVM_GETITEMADDDATA`: used to get the additional data of current icon item.
- `IVM_SETITEMADDDATA` corresponding to `SVM_SETITEMADDDATA`: used to set the additional data of current icon item.
- `IVM_REFRESHITEM` corresponding to `SVM_REFRESHITEM`: used to refresh a icon item area.
- `IVM_GETFIRSTVISIBLEITEM` corresponding to `SVM_GETFIRSTVISIBLEITEM`: used to get the first visible icon item.

### 38.3 Notification Codes of IconView

The IconView control will generate notification code when it responds to user's operations such as clicking or some status changed. The notification codes include:

- `LVN_SELCHANGE` corresponding to `SVN_SELCHANGE`: current highlighted icon item is changed.
- `LVN_CLICKED` corresponding to `SVN_CLICKED`: user clicks the icon item.

A notification handle function should be registered by application using `SetNotificationCallback` to handle all the received notification code. `LVN_CLICKED` and `LVN_SELCHANGE` are used to inform the message handle function that the additional data is the clicked or highlighted icon handle.

### 38.4 Sample Program

Code in List 38.1 illustrates the use of an `IconView` control to construct a simple icon explore window. Please refer to `iconview.c` file of the demo program package `mg-samples` of this guide for complete source code.

List 38.1 Use of IconView control

```

#define IDC_ICONVIEW      100
#define IDC_BT            200
#define IDC_BT2           300
#define IDC_BT3           400
#define IDC_BT4           500

#define IDC_ADD           600
#define IDC_DELETE        601

static HWND hIconView;

static BITMAP myicons [12];

static const char* iconfiles[12] =
{
    "./res/acroread.png",
    "./res/icons.png",
    "./res/looknfeel.png",
    "./res/package_games.png",
    "./res/tux.png",
    "./res/xemacs.png",
    "./res/gimp.png",
    "./res/kpilot.png",
    "./res/multimedia.png",
    "./res/realplayer.png",
    "./res/usb.png",
    "./res/xmms.png"
};

static const char *iconlabels[12] =
{
    "acroread",
    "icons",
    "looknfeel",
    "games",
    "tux",
    "xemacs",
    "gimp",
    "kpilot",
    "multimedia",
    "realplayer",
    "usb",
    "xmms"
};

static void myDrawItem (HWND hWnd, GHANDLE hsvi, HDC hdc, RECT *rcDraw)
{
    const PBITMAP pbmp = (PBITMAP)iconview_get_item_bitmap (hsvi);
    const char *label = (const char*)iconview_get_item_label (hsvi);

    SetBkMode (hdc, BM_TRANSPARENT);
    SetTextColor (hdc, PIXEL_black);

    if (iconview_is_item_highlight(hWnd, hsvi)) {
        SetBrushColor (hdc, PIXEL_blue);
    }
    else {
        SetBrushColor (hdc, PIXEL_lightwhite);
    }
    FillBox (hdc, rcDraw->left, rcDraw->top, RECTWP(rcDraw), RECTHP(rcDraw));
    SetBkColor (hdc, PIXEL_blue);

    if (label) {
        RECT rcTxt = *rcDraw;
        rcTxt.top = rcTxt.bottom - GetWindowFont (hWnd)->size * 2;
        rcTxt.left = rcTxt.left - (GetWindowFont (hWnd)->size) + 2;

        DrawText (hdc, label, -1, &rcTxt, DT_SINGLELINE | DT_CENTER | DT_VCENTER);
    }
    FillBoxWithBitmap (hdc, rcDraw->left, rcDraw->top, 0, 0, pbmp);
}

static int
BookProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)

```

```

{
    switch (message)
    {

    case MSG_INITDIALOG:
    {
        IVITEMINFO ivii;
        static int i = 0, j = 0;

        hIconView = GetDlgItem (hDlg, IDC_ICONVIEW);
        SetWindowBkColor (hIconView, PIXEL_lightwhite);
        //SendMessage (hIconView, IVM_SETITEMDRAW, 0, (LPARAM)myDrawItem);
        SendMessage (hIconView, IVM_SETITEMSIZE, 55, 65);
        //SendMessage (hIconView, IVM_SETITEMSIZE, 35, 35);
        for (j = 0; j < 3; j++) {
            for (i = 0; i < TABLESIZE(myicons); i++) {
                memset (&ivii, 0, sizeof(IVITEMINFO));
                ivii.bmp = &myicons[i];
                ivii.nItem = 12 * j + i;
                ivii.label = iconlabels[i];
                ivii.addData = (DWORD)iconlabels[i];
                SendMessage (hIconView, IVM_ADDITEM, 0, (LPARAM)&ivii);
            }
        }
        break;
    }

    case MSG_COMMAND:
    {
        int id = LOWORD (wParam);
        int code = HIWORD (wParam);

        switch (id) {
        case IDC_ICONVIEW:
            if (code == IVN_CLICKED) {
                int sel;
                sel = SendMessage (hIconView, IVM_GETCURSEL, 0, 0);
                printf ("clicking %d\n", sel);
            }
            break;
        case IDC_ADD:
        {
            IVITEMINFO ivii;
            char buff [10];
            int idx;
            int count = SendMessage (hIconView, IVM_GETITEMCOUNT, 0, 0);

            sprintf (buff, "NewIcon%i", count);
            memset (&ivii, 0, sizeof (IVITEMINFO));
            ivii.bmp = &myicons [0];
            ivii.nItem = count;
            ivii.label = buff;
            ivii.addData = (DWORD)"NewIcon";

            idx = SendMessage (hIconView, IVM_ADDITEM, 0, (LPARAM)&ivii);
            SendMessage (hIconView, IVM_SETCURSEL, idx, 1);
            break;
        }

        case IDC_DELETE:
        {
            int sel = SendMessage (hIconView, IVM_GETCURSEL, 0, 0);
            int count = SendMessage (hIconView, IVM_GETITEMCOUNT, 0, 0);
            char *label = NULL;

            if (sel >= 0){
                label = (char *) SendMessage (hIconView, IVM_GETITEMADDDATA, sel, 0);
                if (label && strlen (label))
                    printf ("delete item:%s\n", label);
                SendMessage (hIconView, IVM_DELITEM, sel, 0);
                if (sel == count - 1)
                    sel--;
                SendMessage (hIconView, IVM_SETCURSEL, sel, 1);
            }
            break;
        }
    }
}

```

```

    }

    } /* end command switch */
    break;
}

case MSG_KEYDOWN:
    if (wParam == SCANCODE_REMOVE) {
        int cursel = SendMessage (hIconView, IVM_GETCURSEL, 0, 0);

        if (cursel >= 0){
            SendMessage (hIconView, IVM_DELITEM, cursel, 0);
            SendMessage (hIconView, IVM_SETCURSEL, cursel, 0);
        }
    }
    break;

case MSG_CLOSE:
{
    EndDialog (hDlg, 0);
    return 0;
}

} /* end switch */

return DefaultDialogProc (hDlg, message, wParam, lParam);
}

static CTRLDATA CtrlBook[] =
{
    {
        CTRL_ICONVIEW,
        WS_BORDER | WS_CHILD | WS_VISIBLE | WS_VSCROLL | WS_HSCROLL,
        10, 10, 290, 300,
        IDC_ICONVIEW,
        "",
        0
    },
    {
        CTRL_BUTTON,
        WS_CHILD | WS_VISIBLE | BS_DEFPUSHBUTTON | WS_TABSTOP,
        90, 330, 50, 30,
        IDC_ADD,
        "Add",
        0
    },
    {
        CTRL_BUTTON,
        WS_CHILD | WS_VISIBLE | WS_TABSTOP | BS_PUSHBUTTON,
        170, 330, 50, 30,
        IDC_DELETE,
        "Delete",
        0
    }
};

static DLGTEMPLATE DlgIcon =
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    0, 0, 310, 400,
    "My Friends",
    0, 0,
    TABLESIZE(CtrlBook), CtrlBook,
    0
};

```



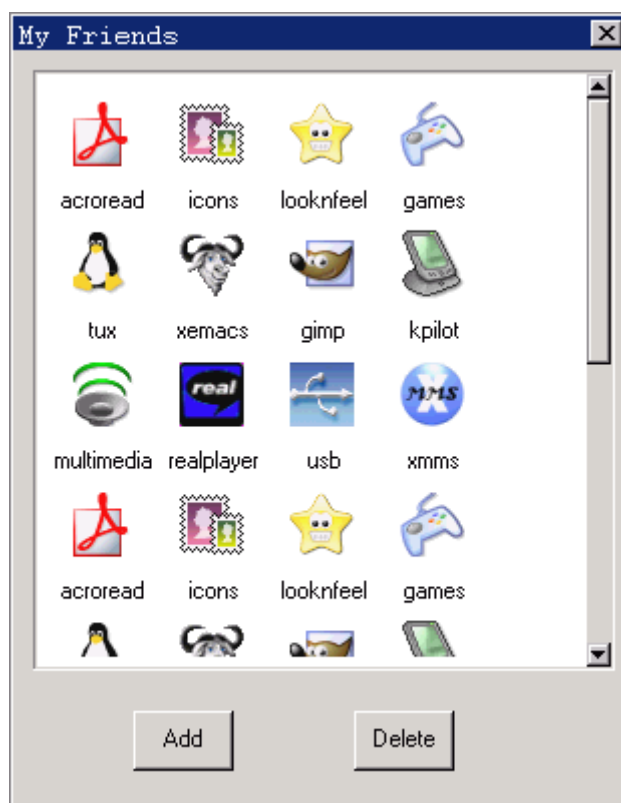


Fig. 38.1 Use of an Icon View control

## Appendix A Universal Startup API for RTOSes

MiniGUI provides universal startup API for RTOSes and spare startup reference files in V2.0.4/1.6.9. These files are saved in the MiniGUI source code `rtos/` directory. The rules of nomenclature in these files are like `<os>_startup.c`.

In startup file, there are an introduction to malloc, printf and POSIX pthread initialization interface. The following is detailed description.

### A.1 Malloc Initialization Interface

MiniGUI provides an own-implemented malloc set of functions. We can use macro `_USE_OWN_MALLOC` to open it. MiniGUI provides the following interface to initialize MiniGUI malloc set of functions.

```
int init_minigui_malloc (unsigned char* heap, unsigned int heap_size,
int (*lock_heap) (void), int (*unlock_heap) (void));
```

To use MiniGUI malloc set of functions, you should pass the information of heap address, lock and unlock heap function pointer, and stack size.

The following is the reference implementation on ThreadX OS.

```
#include "tx_api.h"

static TX_MUTEX __threadx_malloc_mutex;

#define NAME_MUTEX_MALLOC    "Mutex4Malloc"

static int __threadx_heap_lock (void)
{
    UINT status;

    status = tx_mutex_get (&__threadx_malloc_mutex, TX_WAIT_FOREVER);
    if (status == TX_SUCCESS)
        return 0;

    return -1;
}

static int __threadx_heap_unlock (void)
{
    UINT status;

    status = tx_mutex_put (&__threadx_malloc_mutex);
    if (status == TX_SUCCESS)
        return 0;

    return -1;
}

...

/* create the mutex for heap */
```

```
tx_mutex_create (&__threadx_malloc_mutex, NAME_MUTEX_MALLOC, TX_NO_INHERIT);

/* init MiniGUI malloc system */
init_minigui_malloc (heap, heap_size, __threadx_heap_lock, __threadx_heap_unlock);
```

## A.2 Standard Output Initialization Interface

MiniGUI provides an own-implemented printf set of functions. We can use macro `_USE_OWN_STDIO` to open it. MiniGUI provides the following interface to initialize MiniGUI printf set of functions.

```
int init_minigui_printf (int (*output_char) (int ch), int (*input_char) (void));
```

The following is a reference implementation.

```
extern int serial_write (int ch);
extern int serial_read (void);

/* init MiniGUI's own printf system */
init_minigui_printf (serial_write, serial_read);
```

## A.3 POSIX Threads Initialization Interface

MiniGUI provides an own-implemented POSIX threads system. We can use macro `_USE_OWN_PTHREAD` to open it. MiniGUI provides the following interface to initialize RTOS POSIX threads system.

```
#define MAIN_PTH_MIN_STACK_SIZE (1024)
#define MAIN_PTH_DEF_STACK_SIZE (1024*4)

int start_minigui_pthread (int (* pth_entry) (int argc, const char* argv []), int argc,
const char* argv[],
char* stack_base, unsigned int stack_size);
```

The first argument `pth_entry` will run as main thread. In reality, we can pass `minigui_entry` for it. For example:

```
...
static char main_stack [MAIN_PTH_DEF_STACK_SIZE];

char* argv[] = {"pth_entry", NULL};

start_minigui_pthread (minigui_entry, 1, argv, main_stack, MAIN_PTH_DEF_STACK_SIZE);
```

The following list shows the relationship between RTOS and macros above.

Non-Linux RTOS	Macro
VxWorks	<code>_USE_OWN_STDIO</code> <code>_USE_OWN_MALLOC</code> <code>USE_OWN_PTHREAD</code> (only for vxWorks5.5 below, not included)

	vxWorks5.5)
uC/OS-II	_USE_OWN_STDIO _USE_OWN_MALLOC _USE_OWN_PTHREAD
eCos	none
pSOS	_USE_OWN_PTHREAD
Win32	_USE_OWN_STDIO
OSE	_USE_OWN_PTHREAD
ThreadX	_USE_OWN_STDIO _USE_OWN_MALLOC _USE_OWN_PTHREAD
Nucleus	_USE_OWN_STDIO _USE_OWN_MALLOC _USE_OWN_PTHREAD