## HiGV

# Development Guide

**Issue** 00B02

**Date** 2019-07-30

## HiSilicon (Shanghai) Technologies Co., Ltd.

Address: New R&D Center, 49 Wuhe Road, Bantian,
Longgang District,
Shenzhen 518129 P. R. China

Website: http://www.hisilicon.com/en/

Email: support@hisilicon.com

# About This Document

## Purpose

This document describes the basic architecture of the HiGV graphics component as well as the features, processes, scenario design, and usage of each functional module.

## Related Versions

The following table lists the product versions related to this document.

| Product Name | Version |
|---|---|
| Hi3559A | V100 |
| Hi3559C | V100 |
| Hi3556A | V100 |
| Hi3559 | V200 |
| Hi3556 | V200 |
| Hi3519A | V100 |
| Hi3518E | V300 |
| Hi3516D | V300 |
| Hi3516C | V500 |

## Intended Audience

This document is intended for:

- Technical support engineers
- Software development engineers

# Change History

Changes between document issues are cumulative. The latest document issue contains all changes made in previous issues.

## Issue 00B02 (2019-07-30)

This issue is the second draft release, which incorporates the following changes:

Section 3.7.4 is added.

## Issue 00B01 (2018-07-20)

This issue is the first draft release.

# Contents



# Figures



# Tables



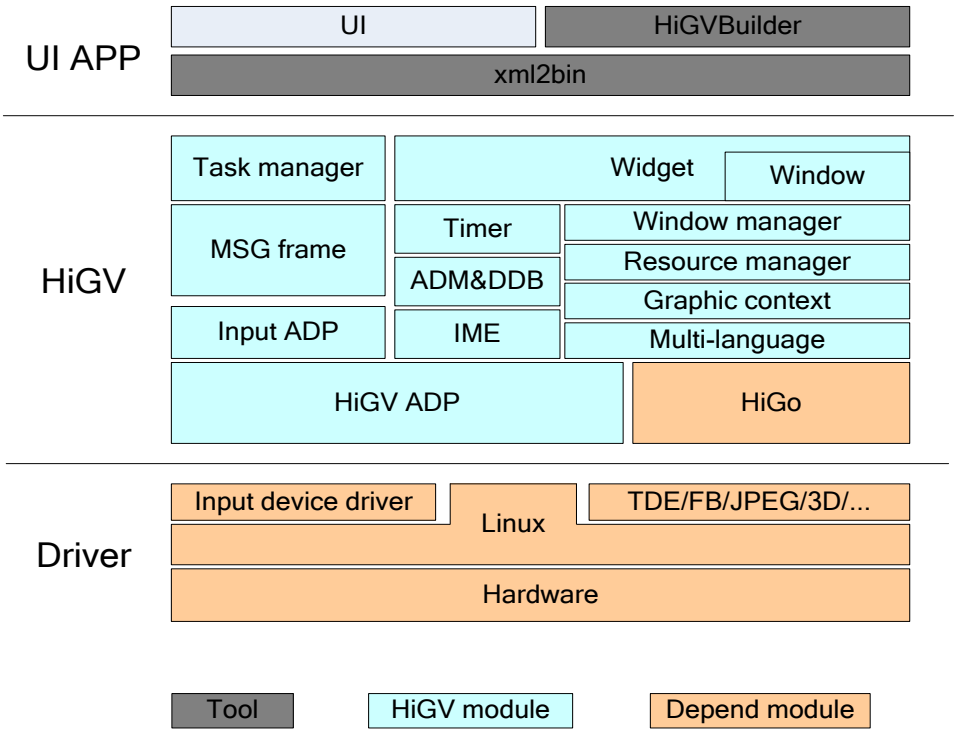
# 1 Introduction

## 1.1 HiGV Architecture

The HiGV is a unified lightweight, highly efficient, and easy-to-use graphical user interface (GUI) system for HiSilicon chip platforms. It uses the layer-based mechanism. The underlying graphics libraries depend on the HiGo library in the software development kit (SDK), whereas the HiGo lays its foundation on basic graphics drivers such as the FrameBuffer (FB), two-dimensional engine (TDE), and graphics CODEC, as shown in Figure 1-1.

Based on the excellent design ideas of modern graphics systems, the HiGV uses various mature design patterns and extensible markup language (XML) GUI description, supports highly efficient and unified system resource management, and provides a rich set of controls. It also provides a quick and visual GUI development tool HiGVBuilder to facilitate easy GUI development.

**Figure 1-1** HiGV architecture



The major modules in Figure 1-1 are described as follows:

- HiGo: basic graphics component, which provides basic APIs for drawing lines, filling, and transferring images
- TDE: 2D acceleration driver interface for the chip
- FB: graphics layer driver interface
- HiGV modules: see chapter 3 "Modules."
- xml2bin: XML file parsing tool
- HiGVBuilder: auxiliary tool for visual GUI development

# 1.2 Important Concepts

You need to know about the following basic concepts about the graphics system before HiGV system development:

- Layer: independent graphics buffer area. Multiple layers can be combined for output by using the alpha value. The graphics layer corresponds to the hardware layer.
- Surface: memory area for storing pixel data, which can be considered as canvas. Typically graphics are drawn on the surface and then transferred to the graphics layer for presentation.
- View: virtual ID of the GUI description XML file, which has no actual meaning for GUI data
- Control: basic element of a GUI. A GUI consists of several controls, such as buttons, text boxes, and image boxes. Each control has its own rectangular region, which determines the display position on the GUI and its size. Controls are classified into container controls and non-container controls. Non-container controls must be placed in container controls.
- Window: basic element of a GUI, a container control. Any controls except windows can be placed in a window, and windows cannot be placed in any other container controls. Windows of the same level can overlap each other. A window cannot contain other windows, and it must be bound to a graphics layer.
- Parent and children: When control A (not a window) is placed in container control B, B is called A's parent, and A is called B's child. If two controls in the same parent container overlap, it is recommended that the controls do not overlap with each other and do not be displayed at the same time.
- Z order: window level, which determines the sequence when windows overlap with each other
- Skin: control appearance. Each state of the control corresponds to a skin. Typically there are four states:
  - Common: basic state
  - Activated: The control is focused.
  - Highlighted: The control is not focused but is also different from the common state.
  - Disabled: The control is forbidden from being activated.
- Abstract data model (ADM): It is used to abstract data operations shown by controls and provide unified data operation APIs for users.
- Default data base (DDB): It provides simple data buffers and management for data models.
- Resource: font libraries, images, and multi-language character strings

- Handle: identifier for instances such as controls, ADMs, multi-language character strings, and fonts
- Message (MSG): an event triggered by the GUI system or user that leads to system behavior changes. When a message is generated, the GUI system or application responds accordingly based on the message type and calls back the message event registered by the user.
- Message callback event: Users can bind functions with controls by registering message callback events. When a message is transmitted to a control, the message callback function bound to the control is called back. Typically the specific service events are used as the message callback events. For details, see section 3.2.4 "Message Callback Event."

# 1.3 Application Development

## 1.3.1 Components

Typically a HiGV project has the following important components apart from the main function:

- XML GUI description files
- Service processing files
- Resource files used by the GUI

A GUI consists of several HiGV controls. These controls can be described by using XML files or created by calling the HiGV APIs.

Typically XML files are used to describe the GUI and resources used by the GUI because the XML description files are convenient and easy-to-use. In addition, a large amount of control creation code is not required, and the HiGV can automatically generate control handles. For details, see section 3.8 "XML File."

You can also create controls by calling the HiGV API. If controls are created by calling this API, a large number of variables are required for storing the control handles to facilitate control operations. The API is typically called to dynamically create or destroy controls.

## 1.3.2 Application Instance

This section describes the association among the XML files, GUI event description files, GUI element attributes, and events by using the Hello World program as an example.

The program requirements are as follows:

- There is an **OK** button and a text box in a window. The text box is empty by default.
- When you click the **OK** button, the text box content is set to **Hello World!** and displayed.

The XML description file is as follows:

```
<view
id = "hello_sample"            <!—view name-->
onload = ""
unload = "">
   <window
      id = "hello_sample"          <!—window name-->
```

```
        top = “0”              <!—widget pos-->
        left = “0”
        width = “720”
        height = “576”
        normalskin = “commonpic_skin_colorkey” <!—normal skin-->
        transparent = “no”
        colorkey = “0xff0000ff”
        isrelease = “yes”
        opacity = “255”
        winlevel = “0”
        onshow = “hello_window_onshow”>  <!—onshow call back-->
        <button
            id = “hello_button_ok”         <!—ok button ID-->
            top = “285”
            left = “235”
            width = “246”
            height = “40”
            normalskin = “commonpic_normalskin_button”
            disableskin = “”
            highlightskin = “”
            activeskin = “commonpic_activeskin_button”    <!—active state
skin-->
            transparent = “no”
            isrelease = “no”
            text = “ID_STR_OK”     <!—multi language ID-->
            alignment = “hcenter|vcenter”
            onclick = “hello_button_onclick”/>  <!—onclick call back-->

        <label
            id = “hello_label_helpinfo”   <!—label ID-->
            top = “200”
            left = “235”
            width = “246”
            height = “50”
            normalskin = “common_skin_black”
            disableskin = “”
            highlightskin = “”
            activeskin = “”
            transparent = “yes”
            isrelease = “no”
            text = ""
            alignment = "hcenter|vcenter"/>
    </window>
</view>
```

```
UI_XXX.c :
/**Window onshow call back*/
HI_S32 hello_window_onshow (HI_HANDLE hWidget,HI_U32 wParam, HI_U32lParam)
{
    return HIGV_PROC_GOON;
}
/**OK button onclick call back*/
HI_S32 hello_button_onclick (HI_HANDLE hWidget,HI_U32 wParam,
HI_U32lParam)
{
    HI_S32 s32Ret;

    /**Change the label content*/
    s32Ret = HI_GV_Widget_SetText(hello_label_helpinfo, "Hello World!");
    return HIGV_PROC_GOON;
}

Main.c:
#define SCREEN_WIDTH  1280
#define SCREEN_HEIGHT  720
#define PROJECT_ID    hello_sample //from xml

HI_S32 main(HI_S32 argc, HI_CHAR *argv[])
{
    HI_S32 Ret;
    HI_S32 g_hApp ;

    /** The layer info, layer width is 1280 and height is 720. The pixel
format is HIGO_PF_8888, Each pixel occupies 32 bits, and the A/R/G/B
components each occupies 8 bits. Use Dual buffers  supported .*/
    HIGO_LAYER_INFO_S LayerInfo = {SCREEN_WIDTH, SCREEN_HEIGHT,
SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_WIDTH, SCREEN_HEIGHT,

(HIGO_LAYER_FLUSHTYPE_E)(HIGO_LAYER_BUFFER_DOUBLE),
                                HIGO_LAYER_DEFLICKER_AUTO,
                                HIGO_PF_8888, HIGO_LAYER_HD_0};


    /**higv init*/
    Ret = HI_GV_Init();
    if (HI_SUCCESS != Ret)
    {
        return Ret;
```

```
    }

    /**Init parser module to parser binary file from xml.*/
    Ret = HI_GV_PARSER_Init();
    if (HI_SUCCESS != Ret)
    {
       printf("HI_GV_PARSER_Init failed! Return: %d\n",Ret);
       return Ret;
    }

    /**Load higv.bin*/
    Ret = HI_GV_PARSER_LoadFile("./higv.bin");
    if (HI_SUCCESS != Ret)
    {
        HI_GV_Deinit();
        HI_GV_PARSER_Deinit();
        return Ret;
    }
    /**Create layer for draw graphics*/
    HI_HANDLE LAYER_0 = INVALID_HANDLE;
    Ret = HI_GV_Layer_Create(&LayerInfo, &LAYER_0);
    if (HI_SUCCESS != Ret)
    {
        printf("HI_GV_Layer_CreateEx failed! Return: %x  \n",Ret);
        HI_GV_Deinit();
        HI_GV_PARSER_Deinit();
        return Ret;
    }
    else
        printf("LAYER_0 create ok ...\n");

    /**Load view,PROJECT_ID is view ID from xml file. Create all child
widget of PROJECT_ID.*/
    Ret = HI_GV_PARSER_LoadViewById(PROJECT_ID);
    if (HI_SUCCESS != Ret)
    {
        printf("HI_GV_PARSER_LoadViewById failed! Return: %x\n",Ret);
        HI_GV_Deinit();
        HI_GV_PARSER_Deinit();
        return Ret;
    }

    /**Create HiGV app.*/
    Ret = HI_GV_App_Create("MainApp", (HI_HANDLE*)&g_hApp);
```

```
    if (HI_SUCCESS != Ret)
    {
        printf("HI_GV_App_Create failed! Return: %d\n",Ret);
        HI_GV_Deinit();
        HI_GV_PARSER_Deinit();
        return Ret;
    }

    /**Show window.*/
    Ret = HI_GV_Widget_Show(PROJECT_ID);
    if(Ret !=0)
        printf("HI_GV_Widget_Show() failed ....: %d \n",Ret);

    Ret = HI_GV_Widget_Active(PROJECT_ID);
    if(Ret !=0)
        printf("HI_GV_Widget_Active() failed ....: %d \n",Ret);

    /*Start HiGV app*/
    HI_GV_App_Start(g_hApp);
    /** If the HiGV app over, the HI_GV_App_Start will be return.*/
    HI_GV_App_Destroy(g_hApp);
    HI_GV_PARSER_Deinit();
    HI_GV_App_Stop(g_hApp);

    return 0;
}
```

After the program is started, click the **OK** button. The onclick event is generated, and the hello_button_onclick function is called. The text box content is changed to "Hello World!" in the function. See Figure 1-2 and Figure 1-3.

**Figure 1-2** Before the OK button is clicked


**Figure 1-3** After the OK button is clicked

# 2 Environment Configurations

## 2.1 Development Environment

The HiSilicon digital media SDK is recommended for development based on the HiSilicon chip platforms.

- All the header files required for development are stored in the **pub/include** directory of the SDK.
- All the libraries required for development are stored in the **pub/lib** directory of the SDK.
- All the drivers (.ko files) are stored in the **rootbox/kmod** directory of the SDK.

The HiSilicon digital media SDK provides only the basic development environment, which does not contain the HiGV. The HiGV development libraries and tools (xml2bin and HiGVBuilder) are required to develop GUI-related applications on the HiGV platform.

- The HiGVBuilder is a quick and visual GUI development tool based on the Eclipse platform. You can generate a GUI by simple mouse operations (drag, move, click, and release) on the PC. After attributes and resource paths are configured, the GUI layout and resource XML description files are automatically generated.
- The xml2bin is a tool that converts the GUI and resource XML description files into the binary file (**higv.bin**) and C file (**higv_cestfile.c**) that can be identified by the HiGV.

The HiGV library **libhigv** provides the following functions:

- Provides core functions and basic graphic framework of the HiGV. It is the core library of the HiGV.
- Provides basic HiGV controls.
- Parses the binary files converted by using the xml2bin tool.
- Provides the adaptation function.

## 2.2 Running Environment

Set the running environment as follows before running the program:

- Burn the fastboot, hi_kernel (Linux kernel), and rootbox images before the system starts. For details, see the *Hi35xx Vx00 Development Environment User Guide*, and *HiTool Quick Start Video*.

- Connect the development board to the liquid-crystal display (LCD), or connect the development board to the TV through a video signal cable.
- Turn on the LCD or TV and select the correct input source.

# 2.3 Running and Debugging

- Load all kernel drivers required by the HiGV.

  Load the kernel drivers required by the graphics system (such as the TDE and FB) before running the program.
- Run the program.

  Run the executable program **/xxx_sample** under the **xxx_sample** directory. You can view the output controls on the display terminal.
- Debug and verify the program.

  You can monitor the running status over the debugging serial port. If an exception occurs, analyze the causes based on the information output by the debugging serial port and the error codes returned by the system.

**NOTICE**

Run the HiGo sample in the SDK before debugging the HiGV to ensure that the HiGo runs properly as the drawing foundation of the HiGV.

# 3 Modules

## 3.1 Control

Controls are the most important GUI elements due to the following reasons:

- The display of graphics and texts of the HiGV is dependent on controls.
- The generation and execution of message events are based on controls.
- The change of the GUI behavior is the change of control data.

### 3.1.1 Basic Attributes

The basic attributes of a control determine the functions, appearance, and position of the control. The following lists the basic common attributes that must be configured:

- **Type**: control type, which determines the basic functions of a control and is represented by the control label in the XML file
- **Rect(rectangle)**: rectangular area of a control, which determines the width and height of a control and the coordinate offset of the control relative to the upper left corner of the parent container
- **Parent**: parent container. A window does not have any parent container. The coordinate offset of a window is that relative to the graphics layer.
- **Handle**: identifier of the control, skin, font, multi-language character string, and data model. The HiGV finds the corresponding instance by identifying the handle.
- **Skin**: basic appearance of a control. The skin (at least common skin) must be configured; otherwise, the control may fail to be displayed.

Besides the mandatory basic attributes listed above, there are also attributes such as the font, text, alignment mode, message callback event, and special style for brother controls. In addition, different control types have their own private attributes. **hi_gv_widget.h** contains the public APIs, message definitions, and other public attributes of controls.

### 3.1.2 Usage

You need to create a control before using it. The method of creating a control by using the XML description file is described in section 1.3.2 "Application Instance." You can also generate a control by calling the HiGV creation API. The following is a simple example of the window creation code:

```
HIGV_WCREATE_S infoWindow;
```

```
HIGV_WINCREATE_S WinCreate;
HI_HANDLE hWindow;
memset(&infoWindow, 0x00, sizeof(infoWindow));
infoWindow.rect.x = 20;//X coordinate of the window relative to the
graphics layer
infoWindow.rect.y = 20;//Y coordinate of the window relative to the
graphics layer
infoWindow.rect.w = 400;//Window width
infoWindow.rect.h = 400;//Window height
WinCreate.hLayer = HIGO_LAYER_HD_0;//Graphics layer to which the window
belongs
WinCreate.PixelFormat = HIGO_PF_BUTT;//The pixel format is not configured.
The pixel format of the graphics layer is used by default.
infoWindow.pPrivate = &WinCreate;
infoWindow.hParent = INVALID_HANDLE;//The window does not have a parent
container.
infoWindow.style = 0;//No special style is configured.
infoWindow.type = HIGV_WIDGET_WINDOW;//The control is a window.
/**Create the control by calling the interface and obtain the dinwo
handle hWindow.*/
if(HI_SUCCESS == HI_GV_Widget_Create(&infoWindow,  &hWindow))
{
    /**Set a common skin window_skin for hWindow. The skin creation
process is not described here.*/
    HI_GV_Widget_SetSkin(hWindow, HIGV_SKIN_NORMAL, window_skin);
}
```

You can use a control after it is created successfully. Take the instance in section 1.3.2 "Application Instance" as an example:

- To display a control, call HI_GV_Widget_Show.
- To hide a control, call HI_GV_Widget_Hide.
- To set the text to be displayed on a control, press a button to trigger HI_GV_Widget_SetText; to display the configured text on a control, call HI_GV_Widget_Paint.
- To change the GUI graphics, call various HiGV APIs to change the control behavior and appearance.
- To destroy a control, call HI_GV_Widget_Destroy.

## 3.1.3 Inheritance Relationship

Controls are classified into the following categories:

- Common control

  Common controls are the leaf nodes in the control inheritance tree.
- Container control

  A container control can contain multiple common controls.

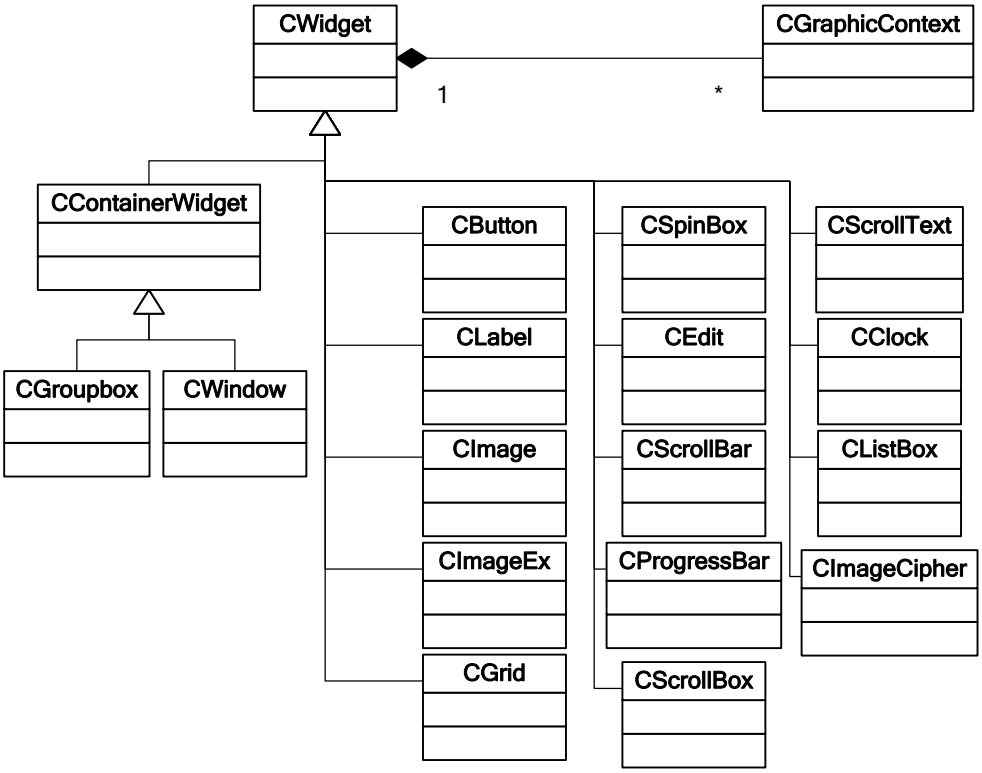
  

- Combo control

  A combo control consists of multiple common controls or container controls. For example, the combo box control consists of controls such as the window, spin, and list box.

The HiGV controls are designed by using the C++ object-oriented design method. Figure 3-1 shows the inheritance relationship among controls.

**Figure 3-1** Inheritance relationship




# 3.2 Message

A message is an event triggered by the GUI system or user that leads to system behavior changes. When a message is generated, the GUI system or application responds accordingly based on the message type and calls back the message event registered by the user.

The main thread of the HiGV starts to capture and process messages after HI_GV_App_Start is called and stops these operations after HI_GV_App_Stop is called. After obtaining the message, the HiGV transfers the message to the focus control for processing and continues to capture the next message after the processing is complete.

The HiGV messages are defined in **hi_gv_widget.h** with a unified prefix "HIGV_MSG_".

## 3.2.1 Message Generation

A message is generated when the HiGV internal data changes or the user triggers it. The following are the main sources of messages:

- Message events generated by input devices such as the keyboard, mouse, and touchscreen. The input device events are received through HiGV adaptation or user adaptation and then converted into messages and transferred to the message queue. The common input device events include the following:

1. Remote Control and Panel Message Type
   - HIGV_MSG_KEYDOWN: generated when a key/button on the remote control or front panel is pressed
   - HIGV_MSG_KEYUP: generated when a key/button on the remote control or front panel is released
   - HIGV_MSG_MOUSEDOWN: generated when the left or right mouse button is pressed
   - HIGV_MSG_MOUSEUP: generated when the left or right mouse button is released
2. Mouse Message Type
   - HIGV_MSG_MOUSEWHEEL: generated when the mouse wheel is rolled
   - HIGV_MSG_MOUSELONGDOWN: generated when the left mouse button is held down
   - HIGV_MSG_MOUSEDBCLICK: generated when the mouse button is double-clicked
   - HIGV_MSG_MOUSEIN: generated when the pointer moves to the region of a specific control
   - HIGV_MSG_MOUSEOUT: generated when the pointer moves out of the region of a specific control
3. Touchscreen Message Type
   - HIGV_MSG_TOUCH: generated when a specific control region of the touchscreen is touched.
   - HIGV_MSG_GESTURE_TAP: generated when a specific control region of the touchscreen is tapped.
   - HIGV_MSG_GESTURE_LONGTAP: generated when a specific control region of the touchscreen is tapped for a while.
   - HIGV_MSG_GESTURE_FLING: generated when a specific control region of the touchscreen is flicked.
   - HIGV_MSG_GESTURE_SCROLL: generated when a specific control region of the touchscreen is scrolled.
   - HIGV_MSG_GESTURE_PINCH: generated when a specific control region of the touchscreen is pinched with two fingers.

Touch event processing means that the target control judges a touch event and then processes this event. The basic processing is similar to the Hittest of a mouse, but the difference is that touch event processing needs to capture the touch event in the control tree so that the parent control can process the touch event first. The gesture event is the advanced package of the touch event, and touch event flows is the basis for judging the gesture events. The recognized gesture events are as follows:

HIGV_GESTURE_TAP: Tap gesture on the touchscreen

HIGV_GESTURE_LONGTAP: Long tap gesture (An event is triggered when the control region is tapped for more than 2 seconds.)

HIGV_GESTURE_FLING: Flick gesture (You can perform this gesture toward any direction.)

HIGV_GESTURE_SCROLL: Scroll gesture (The finger scrolls on the touchscreen.)

HIGV_GESTURE_PINCH: Pinch gesture (Two fingers are pinched on the touchscreen.)

- Messages directly transmitted to controls by calling message transfer APIs such as HI_GV_Msg_SendAsync

**NOTE**

For details about all message transfer APIs, see the header file **hi_gv_msg.h**.

- HiGV events triggered by calling control APIs. For example, calling HI_GV_Widget_Show generates the HIGV_MSG_SHOW and HIGV_MSG_PAINT events, and calling HI_GV_List_SetSelItem generates the HIGV_MSG_ITEM_SELECT event.
- HIGV_MSG_TIMER generated by the timer
- Events generated due to internal data changes of controls. For example, when a control is focused, the HIGV_MSG_GET_FOCUS event is triggered; when it is unfocused, the HIGV_MSG_LOST_FOCUS event is triggered.

## 3.2.2 Message Features

The message has the following features:

- Messages that are triggered first are executed first. The sequence is stable.
- Repeated messages transmitted to the same target in a short time are combined to avoid unnecessary operations.

  Messages that can be combined include the following:

  - HIGV_MSG_PAINT: drawing message
  - HIGV_MSG_REFRESH_WINDOW: window refreshing message
  - HIGV_MSG_FORCE_REFRESH_WINDOW: window forcible refreshing message
  - HIGV_MSG_TIMER: timer message
  - HIGV_MSG_DATA_CHANGE: data change message
  - HIGV_MSG_ST_UPDATE: scrolling teletext update message
  - HIGV_MSG_MOUSEMOVE: mouse moving message
  - HIGV_MSG_MOUSEWHEEL: mouse wheel message
- The asynchronously transmitted messages trigger the GUI service but do not block the main thread.
- Associated parameters are transmitted with messages so that the HiGV and callback functions can obtain accurate message event data.

**NOTE**

The parameters vary according to messages. For details, see the header file **hi_gv_widget.h**.

## 3.2.3 Message Transfer Process

A message is directly distributed to the target control for processing after it is generated.

However, the target control for input events triggered by input devices (such as the mouse and key) is not specified when the message is generated. The transfer process of these messages is as follows:

- Key message event transfer process

**Step 1** The key message is distributed to the current focus window.

**Step 2** If the focus window has child controls, the window transfers the key message to the child control. If the child control is a container control, it continues to transfer the key message level by level until the message is transferred to the terminal focus child control.

**Step 3** The terminal control processes the key message.

**Step 4** If the return value of the key message callback function registered by the user is HIGV_PROC_GOON, the message is transferred level by level to the parent container of the control that processes the message until the message reaches the window or the return value of the message callback function is HIGV_PROC_STOP.

**----End**

- Mouse message event transfer process

**Step 1** The current coordinates of the pointer are calculated and the message is transferred to the top window to which the coordinates belong.

**Step 2** The window transfers the message to the terminal control where the pointer is located.

**Step 3** The terminal control processes the mouse message.

**Step 4** If the return value of the mouse message callback function registered by the user is HIGV_PROC_GOON, the message is transferred level by level to the parent container of the control that processes the message until the message reaches the window or the return value of the message callback function is HIGV_PROC_STOP.

**----End**

## 3.2.4 Message Callback Event

**NOTICE**

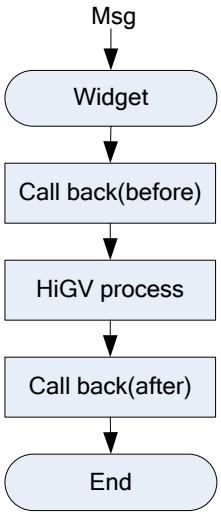
As the message callback event is executed in the HiGV thread, the HiGV thread is blocked if the message callback event block occurs.

If you want the HiGV control to process some service events when it receives messages, you can use the message callback function. Figure 3-2 shows the message callback process.

The callback event is the main approach for interaction between controls and services. Service functions are put into callback functions to implement functions such as man-machine interaction and user operations.

**Figure 3-2** Message callback process




HI_GV_Widget_SetMsgProc registers the message callback event with controls. This API has four parameters:

- **hWidget**: registered control handle
- **Msg**: ID of the registered message that triggers the callback event
- **CustomProc**: callback event of receiving **Msg**
- **ProcOrder**: event processing priority. If the HiGV processes a message internally:
  - **HIGV_PROCORDER_BEFORE** indicates that **CustomProc** is called back before the internal processing of the control.
  - **HIGV_PROCORDER_AFTER** indicates that **CustomProc** is called back after the internal processing of the control.

The general format of a callback function is as follows:

```
typedef HI_S32 (*HIGV_MSG_PROC)(HI_HANDLE hWidget,
HI_U32 wParam, HI_U32 lParam)
```

where

- **hWidget**: control handle that is bound to the callback function
- **wParam**: first event parameter. The HiGV processes some events internally, and this parameter may be used. For details, see **hi_gv_widget.h**.
- **lParam**: second event parameter, typically as the private data parameter
- Return value: Because the callback function is interleaved in the control data change process, its return value affects the event processing of the control. The return value **HIGV_PROC_GOON** indicates that the message continues to be transferred, which specifies whether the key or mouse event is transferred to the parent container. The return value **HIGV_PROC_STOP** indicates that event processing will be stopped.

Calling the API for transferring messages triggers the message event. You can specify the parameters when transferring message events.

- HI_GV_Msg_SendAsync(HI_HANDLE hWidget, HI_U32 MsgId, HI_U32 Param1, HI_U32 Param2)

  - **Param1** corresponds to the callback event **wParam**, and it affects the internal processing of controls.
  - **Param2** corresponds to the callback event **lParam**, and it may be overwritten as 0 if the control processes the event internally.
- HI_GV_Msg_SendAsyncWithData(HI_HANDLE hWidget, HI_U32 MsgId, HI_VOID *pBuf, HI_U32 BufLen)

  This API can carry private data. **pBuf** corresponds to **lParam** of the callback function.

## 3.2.5 Typical Messages

### HIGV_MSG_SHOW

HIGV_MSG_SHOW is a control display message. This message event can be registered before the control is displayed or for other services.

- Event triggered by the control API: HI_GV_Widget_Show can display hidden controls. It calls back the HIGV_MSG_SHOW event synchronously and then draws controls asynchronously. The parameters **wParam** and **lParam** are **0**.
- Event transferred by the MSG interface: The event is called back but the control is not displayed. The parameter **wParam** is **Param1** and **lParam** is **Param2** or **pBuf**.
- Impact of the return value: none

### HIGV_MSG_PAINT

HIGV_MSG_PAINT is a control drawing message. This message event can be registered for service processing before and after control drawing.

- Trigger condition: all operations and events that lead to control drawing
- Parameter definition: region for drawing the control. The parameter can be converted into **HI_RECT** by using HIGV_U32PARAM_TORECT.
- Event transferred by the MSG interface: The event is called back but the control is not drawn. The parameter **wParam** is **Param1** and **lParam** is **Param2** or **pBuf**.
- Impact of the return value: The control drawing is stopped if the return value is not **HIGV_PROC_GOON** for the callback event with the priority HIGV_PROCORDER_BEFORE.

### HIGV_MSG_KEYDOWN/HIGV_MSG_MOUSEDOWN

HIGV_MSG_KEYDOWN/HIGV_MSG_MOUSEDOWN is a control press service event.

- Event triggered by the input device: **wParam** is the key value or mouse button value, and **lParam** is 0. The control is also triggered to respond to the input event.
- Event transferred by the MSG interface: same as the input device trigger behavior
- Impact of the return value: For the callback event with the priority HIGV_PROCORDER_BEFORE:
  - If the return value is not HIGV_PROC_GOON, the control stops responding to the message.
  - If the return value is HIGV_PROC_GOON, the message continues to be transferred to the parent container of the control.

### HIGV_MSG_TOUCH

HIGV_MSG_TOUCH is a control press service event.

- Event triggered by the input device: **wParam** is the length of the touch event structure, and **lParam** is the structure pointer of the touch event. The control is also triggered to respond to the input event.
- Event transferred by the MSG interface: same as the input device trigger behavior
- Impact of the return value: For the callback event with the priority HIGV_PROCORDER_BEFORE:
  - If the return value is not HIGV_PROC_GOON, the control stops responding to the message.
  - If the return value is HIGV_PROC_GOON, the message continues to be transferred to the parent container of the control.

### HIGV_MSG_GESTURE_TAP/HIGV_MSG_GESTURE_LONGTAP/HIGV_MSG_GESTURE_FLING/HIGV_MSG_GESTURE_SCROLL/HIGV_MSG_GESTURE_PINCH

HIGV_MSG_GESTURE_TAP/HIGV_MSG_GESTURE_LONGTAP/HIGV_MSG_GESTURE_FLING/HIGV_MSG_GESTURE_SCROLL/HIGV_MSG_GESTURE_PINCH is a control press service event.

- Event triggered by the input device: **wParam** is the length of the gesture event structure, and **lParam** is the structure pointer of the touch event. The control is also triggered to respond to the input event.
- Event transferred by the MSG interface: same as the input device trigger behavior
- Impact of the return value: For the callback event with the priority HIGV_PROCORDER_BEFORE:
  - If the return value is not HIGV_PROC_GOON, the control stops responding to the message.
  - If the return value is HIGV_PROC_GOON, the message continues to be transferred to the parent container of the control.

## 3.2.6 Internal Message Processing

Not all messages of all controls are processed internally. Table 3-1 lists the messages that are internally processed by the HiGV.

**Table 3-1** List of internally processed messages

| Message ID | Class | HiGV Process |
|---|---|---|
| HIGV_MSG_PAINT | All | Paint |
| HIGV_MSG_SHOW | All | Show widget |
| HIGV_MSG_HIDE | All | Hide widget |
| HIGV_MSG_KEYDOWN | All | Switch HIGV_FOCUS_STATE_E;<br>Enter;<br>Switch focus |

| Message ID | Class | HiGV Process |
|---|---|---|
| HIGV_MSG_LAN_CHANGE | All | Renewedly paint |
| HIGV_MSG_DATA_CHANGE | ADM<br>Listbox<br>Spin<br>Combobox<br>Scrollgrid | Data update |
| HIGV_MSG_STATE_CHANGE | Containerwidget | Change HIGV_STATENAME_E |
| HIGV_MSG_ST_UPDATE | Listbox<br>Multiedit<br>Scrolltext | Listbox: update scroll text<br>Scrolltext: paint<br>Multiedit: update cursor |
| HIGV_MSG_SCROLLBAR_CHANGE | Listbox<br>Multiedit<br>Scrollview | Update scrollbar |
| HIGV_MSG_REFRESH_WINDOW | Window | Refresh window |
| HIGV_MSG_FORCE_REFRESH_WINDOW | Window | Force update the window to the screen |
| HIGV_MSG_MOUSEIN | All | Switch HIGV_SKIN_HIGHLIGHT |
| HIGV_MSG_MOUSEDOWN | All | Switch HIGV_SKIN_MOUSEDOWN |
| HIGV_MSG_MOUSEOUT | All | Lost HIGV_MSG_MOUSEIN |
| HIGV_MSG_MOUSEUP | All | Lost HIGV_MSG_MOUSEDOWN |
| HIGV_MSG_MOUSEMOVE | All | Move event |
| HIGV_MSG_MOUSEDBCLICK | All | Mouse event |
| HIGV_MSG_MOUSEWHEEL | All | Mouse event |
| HIGV_MSG_MOUSELONGDOWN | All | Mouse event |
| HIGV_MSG_TOUCH | All | Touch event |
| HIGV_MSG_GESTURE_TAP | All | Gesture event |
| HIGV_MSG_GESTURE_LONGTAP | All | Gesture event |
| HIGV_MSG_GESTURE_FLING | All | Gesture event |
| HIGV_MSG_GESTURE_SCROLL | All | Gesture event |
| HIGV_MSG_VALUEONCHANGE | Trackbar | Value change event |
| HIGV_MSG_FINISHSEEK | Trackbar | Finish seek event |
| HIGV_MSG_UNLOCK | Slideunlock | Unlock event |
| HIGV_MSG_MOVE | Slideunlock | Move event |
| HIGV_MSG_KICKBACK | Slideunlock | Kick back to origin event |

## 3.2.7 Message Callback Events Defined in the XML File

Frequently used message callback events can be registered in the XML files. If the callback event is not registered in the XML files, you can register it by calling HI_GV_Widget_SetMsgProc. Table 3-2 lists the callback events that apply to all controls.

**Table 3-2** List of callback events in the XML files

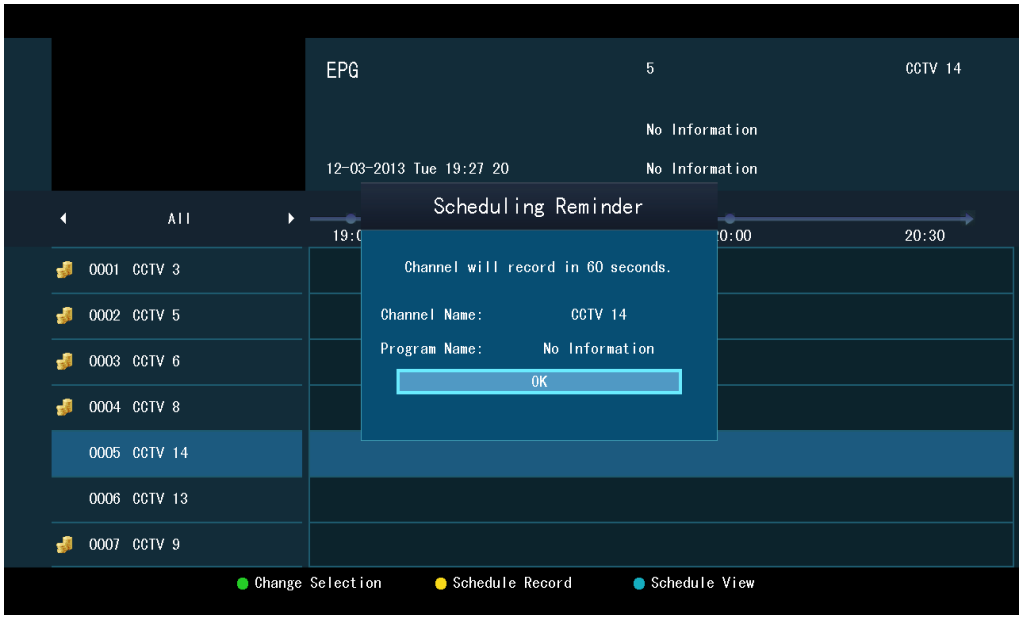
| XML Attribute | Message ID | Message Order |
|---|---|---|
| onkeydown | HIGV_MSG_KEYDOWN | HIGV_PROCORDER_AFTER |
| onkeyup | HIGV_MSG_KEYUP | HIGV_PROCORDER_AFTER |
| ontimer | HIGV_MSG_TIMER | HIGV_PROCORDER_AFTER |
| ongetfocus | HIGV_MSG_GET_FOCUS | HIGV_PROCORDER_AFTER |
| onlostfocus | HIGV_MSG_LOST_FOCUS | HIGV_PROCORDER_AFTER |
| onmousein | HIGV_MSG_MOUSEIN | HIGV_PROCORDER_AFTER |
| onmousedown | HIGV_MSG_MOUSEDOWN | HIGV_PROCORDER_AFTER |
| onmouseout | HIGV_MSG_MOUSEOUT | HIGV_PROCORDER_AFTER |
| onmouseup | HIGV_MSG_MOUSEUP | HIGV_PROCORDER_AFTER |
| onmousemove | HIGV_MSG_MOUSEMOVE | HIGV_PROCORDER_AFTER |
| onmousewheel | HIGV_MSG_MOUSEWHEEL | HIGV_PROCORDER_AFTER |
| onmousedbclick | HIGV_MSG_MOUSEDBCLICK | HIGV_PROCORDER_AFTER |
| onfocuseditexit | HIGV_MSG_FOCUS_EDIT_EXIT | HIGV_PROCORDER_BEFORE |
| onlanchange | HIGV_MSG_LAN_CHANGE | HIGV_PROCORDER_AFTER |
| ontouchaction | HIGV_MSG_TOUCH | HIGV_PROCORDER_AFTER |
| ongesturetap | HIGV_MSG_GESTURE_TAP | HIGV_PROCORDER_AFTER |
| ongesturelongtap | HIGV_MSG_GESTURE_LONGTAP | HIGV_PROCORDER_AFTER |
| ongesturefling | HIGV_MSG_GESTURE_FLING | HIGV_PROCORDER_AFTER |
| ongesturescroll | HIGV_MSG_GESTURE_SCROLL | HIGV_PROCORDER_AFTER |
| onvaluechange | HIGV_MSG_VALUEONCHANGE | HIGV_PROCORDER_BEFORE |
| onfinishseek | HIGV_MSG_FINISHSEEK | HIGV_PROCORDER_BEFORE |
| onunlock | HIGV_MSG_UNLOCK | HIGV_PROCORDER_BEFORE |
| onmove | HIGV_MSG_MOVE | HIGV_PROCORDER_BEFORE |
| onkickback | HIGV_MSG_KICKBACK | HIGV_PROCORDER_BEFORE |

# 3.3 Drawing

Figure 3-3 shows the GUI of a HiGV. This section provides guidance for creating a complete and complex GUI by describing how to draw a control and what is the relationship between a window and its child controls.

**Figure 3-3** HiGV GUI instance




## 3.3.1 Drawing a Control

The control appearance consists of the skin and text, and the drawing can be synchronous or asynchronous. Control drawing is triggered when the HiGV internal data changes, the user calls the drawing API, or the user sends the drawing message to the control.

For some operations, a displayed control needs to be redrawn. For example, after you configure the text dynamically for a label control, the newly configured text is displayed only after the control is redrawn.

The drawing operation triggered by HI_GV_Widget_Paint is asynchronous. The basic process is as follows:

**Step 1** Calculate the drawing region.

**Step 2** Send the drawing message. The drawing region is the one obtained in Step 1.

**Step 3** Draw the background of the parent container after the drawing message is received if the control is a non-window control with the transparent style or forcible parent style.

**Step 4** Call back the message callback event with the priority HIGV_PROCORDER_BEFORE.

**Step 5** Draw the skin of the current state.

**Step 6** Draw other contents of the control (including the image and text).

**Step 7** Call back the message callback event with the priority HIGV_PROCORDER_AFTER.

**----End**

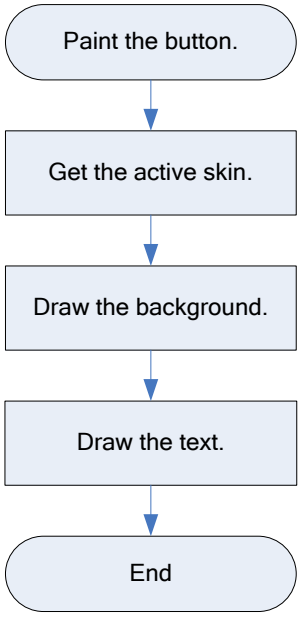
For the drawing operation triggered by other modes:

- The process starts directly from Step 4 after the drawing message is transferred by calling the message transfer API.
- Calling HI_GV_Widget_Update skips the message transfer process and implements drawing synchronously.

In normal scenarios, HI_GV_Widget_Paint is used because calling this API draws a control asynchronously, which does not block the main thread for other processing. However, some special scenarios may require that the control be drawn synchronously. In this case, you can call HI_GV_Widget_Update.

Take the **OK** button in the displayed dialog box in Figure 3-3 as an example. Its parent container is the displayed dialog box **Scheduling Reminder**, and its position in the parent container is the rectangle to be drawn.

**Figure 3-4** Process for drawing the OK button




## 3.3.2 Drawing a Window

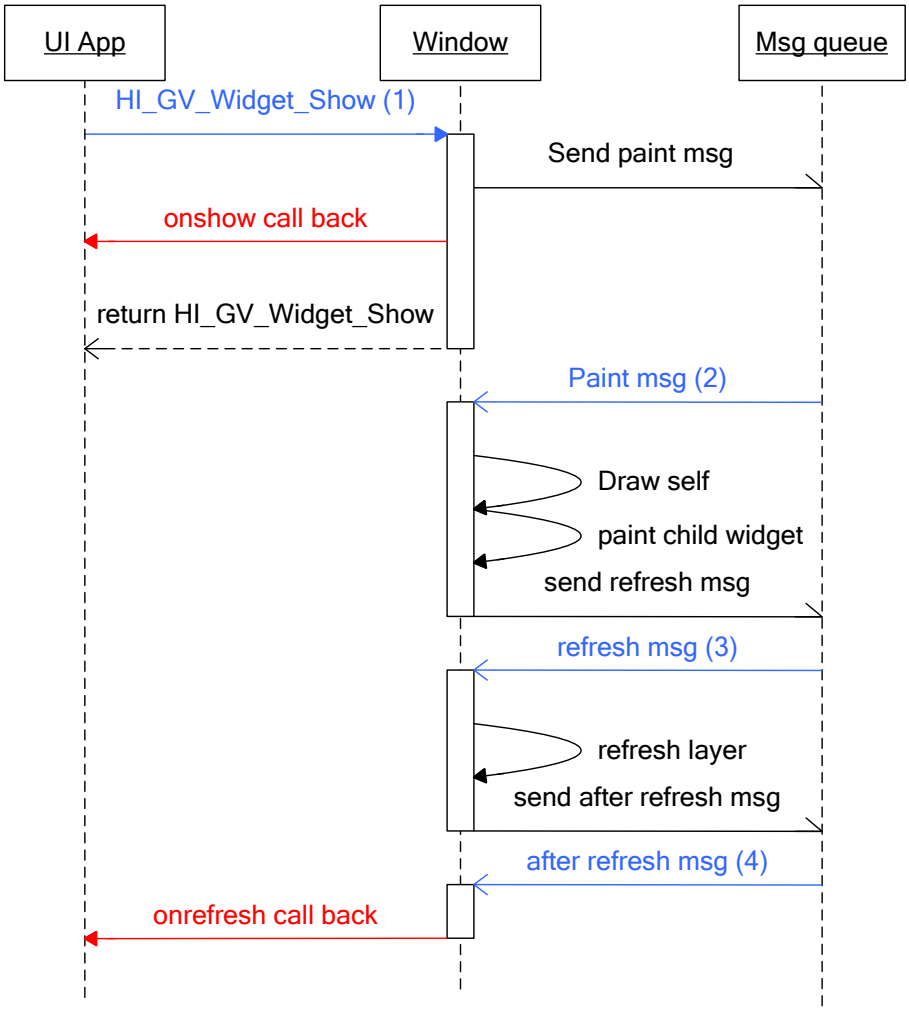
**NOTICE**

Do not obtain the window display status by calling HI_GV_Widget_IsShow in the onshow event because the draw operation is not successful yet and the display status is incorrect. Do not hide the window in the onshow event because exceptions may occur during drawing and display.

Figure 3-5 shows the window display process.

**Figure 3-5** Window display timing




The window display process is as follows:

**Step 1** HI_GV_Widget_Show is called. If the window is hidden and can be displayed, it sends an async drawing message to itself. Async drawing can avoid short delay due to long drawing duration. Then the window calls back the onshow event registered by the user and returns the HI_GV_Widget_Show interface. The user can make preparations for drawing the window in the onshow event.

**Step 2** After receiving the drawing message, the window draws itself, draws child controls (displayed), and then sends a graphics layer refresh message to itself.

**Step 3** The window checks the current drawing status after receiving the graphics layer refresh message in Step 2 because the drawing and refresh messages are async messages.

- If the drawing process is not complete or the window is hidden, the message processing is terminated.

- If the window status meets the refresh conditions, the graphics layer is refreshed. Then the window sends a refresh completion message to itself. The window is displayed on the screen after the graphics layer is refreshed.

**Step 4** After receiving the refresh completion message in Step 3, the window calls back the onrefresh event registered by the user. The user then can implement other processing after the window is drawn in the onrefresh event.

**----End**

### 3.3.3 Hiding a Control

HI_GV_Widget_Hide is used to hide a control.

A control can be drawn only when it is displayed, and the display status of a control depends on that of its parent container.

A non-window control can be displayed or hidden only when its parent container is displayed. For example, if a window contains a group box, and the group box has a button:

- When the window is hidden, the group box and button are also hidden and cannot be displayed.
- When the window is displayed, the group box can be displayed or hidden; when the window and group box are both displayed, the button can be displayed or hidden.

Displaying or hiding a control involves the process of loading or releasing resources. For details, see section 3.5 "Resources."

## 3.4 Timer

The timer is used to trigger events periodically or as scheduled. HiGV timers are bound to controls. One timer can be bound to only one control, whereas a control can have multiple timers at the same time. For details about the timer, see the header file **hi_gv_timer.h**.

### 3.4.1 Usage

To use a timer, perform the following steps:

**Step 1** Create a timer by calling HI_GV_Timer_Create(HI_HANDLE hWidget, HI_U32 TimerID, HI_U32 Speed).

- **hWidget**: control to which the timer is bound
- **TimerID**: timer ID. A control can be bound to multiple timers. The IDs of timers that are bound to the same control must be different, but the IDs of timers that are bound to different controls can be the same.
- **Speed**: timer trigger time, in ms

**Step 2** Register the timer callback event.

Register a callback function OnTimer(HI_HANDLE hWidget, HI_U32 wParam, HI_U32 lParam) for the timer event in the **ontimer** attribute of the XML file or HI_GV_Widget_SetMsgProc.

- **hWidget**: control to which the timer is bound

- **wParam**: timer ID, which corresponds to **TimerID** in the timer creation API. It is used to identify the timer that triggers this callback when a control is bound to multiple timers.
- **lParam**: **0**

**Step 3** Use the timer.

A HiGV timer is a one-time timer. That is, the timer executes only one callback event after being started. It must be restarted if it needs to be activated again.

1. Start the timer by calling HI_GV_Timer_Start(HI_HANDLE hWidget, HI_U32 TimerID). If the timer has been started, calling this API cannot enable the timer to recount.
2. Stop the timer by calling HI_GV_Timer_Stop(HI_HANDLE hWidget, HI_U32 TimerID) if the timer is not started for the first time.

   **NOTE**

   Stop the timer before starting it in case that the timer fails to start properly due to incorrect timer status.
3. Restart the timer by calling HI_GV_Timer_Reset(HI_HANDLE hWidget, HI_U32 TimerID). The timer recounts when it is started. This API is typically called in OnTimer to implement the cyclic timer.

**Step 4** Destroy the timer by calling HI_GV_Timer_Destroy(HI_HANDLE hWidget, HI_U32 TimerID).

You need to destroy the timer in time to avoid waste of resources.

**----End**

## 3.4.2 Relationship Between the Timer and the HiGV Main Thread

The timer starts counting after being started (by calling HI_GV_Timer_Start), and it sends the HIGV_MSG_TIMER event to the bound control after the configured time is reached. The HiGV main thread distributes the IGV_MSG_TIMER event to the bound control upon reception and calls back the registered OnTimer function. The callback function is executed in the HiGV main thread. Therefore, you need to ensure the smoothness of the HiGV main thread so that the timer callback function can be executed on time. Some time-consuming services may delay the response to the timer event.

# 3.5 Resources

The HiGV provides a unified resource management mechanism. The major resources include fonts, images, skins, and animation information. The resource management module counts the use of resources so that the same resource needs to be loaded only once when it is used in multiple places. This avoids the waste of system memory and performance. For details about the APIs of HiGV resource management, see **hi_gv_resm.h**.

## 3.5.1 Creating Resources

To use the font libraries or images, you need to convert them into resources that can be identified by the HiGV first.

Create a resource based on the resource path and resource type and assign a resource ID by calling HI_GV_Res_CreateID(const HI_CHAR* pFileName, HIGV_RESTYPE_E ResType, HI_RESID* pResID).

- The resource path is relative to the final executable program.
- **ResID** is used to identify a resource. Its functions are similar to those of the control handle.

To facilitate the configuration of resource paths, the HiGV provides the resource environment variable. You can enter the following commands in the console:

- **export HIGV_RES_IMAGE_PATH=***xxx*: Configures the prefix of an image resource path.
- **export HIGV_RES_FONT_PATH=***xxx*: Configures the prefix of a font resource path.

For example, if the command is **export HIGV_RES_IMAGE_PATH=./res/image/** and **pFileName** is **button.png**, the HiGV uses **./res/image/button.png** as the resource path. Note that "/" is not duplicated. After the resource environment variable is configured, HI_GV_Res_CreateID_NoPrefixPath can be called to create a **ResID** without reading the environment variable.

## 3.5.2 Using Resources

### Font

```
typedef struct hiHIGV_FONT_S
{
    HI_RESID SbFontID;
    HI_RESID MbFontID;
    HI_U32   Size;
    HI_BOOL  bBold;
    HI_BOOL  bItalic;
}HIGV_FONT_S;
```

- **HIGV_FONT_S**: data structure for creating a font
- **SbFontID** and **MbFontID**: resource IDs for two physical device font libraries, obtained from HI_GV_Res_CreateID
- **Size**: font size
- **bBold**: bold
- **bItalic**: italic

You need at least one font resource to create a font. After the font is created successfully, set the font handle to the control instance. The reference code is as follows:

```
#define SBFONT_FILE "./res/sbfont.ttf"
#define MBFONT_FILE "./res/mbfont.ttf"

HI_S32 AppCreateSysFont(HI_HANDLE *pFont)
{
    HI_S32 Ret;
    HI_RESID SbFont, MbFont;
    HI_HANDLE hFont;
    HIGV_FONT_S FontInfo;

    Ret = HI_GV_Res_CreateID(SBFONT_FILE, HIGV_RESTYPE_FONT, &SbFont);
```

```
    if (HI_SUCCESS != Ret)
    {
        return Ret;
    }

    Ret = HI_GV_Res_CreateID(MBFONT_FILE, HIGV_RESTYPE_FONT, &MbFont);
    if (HI_SUCCESS != Ret)
    {
        HI_GV_Res_DestroyID(SbFont);
        return Ret;
    }

    FontInfo.MbFontID = MbFont;
    FontInfo.SbFontID = SbFont;
    FontInfo.Size = 22;
    FontInfo.bBold = HI_FALSE;
    FontInfo.bItalic = HI_FALSE;

    Ret = HI_GV_Font_Create((const HIGV_FONT_S *)&FontInfo, &hFont);
    if (HI_SUCCESS != Ret)
    {
        HI_GV_Res_DestroyID(SbFont);
        HI_GV_Res_DestroyID(MbFont);
        return Ret;
    }

    *pFont = hFont;
    return HI_SUCCESS;
}
```

## Image and Skin

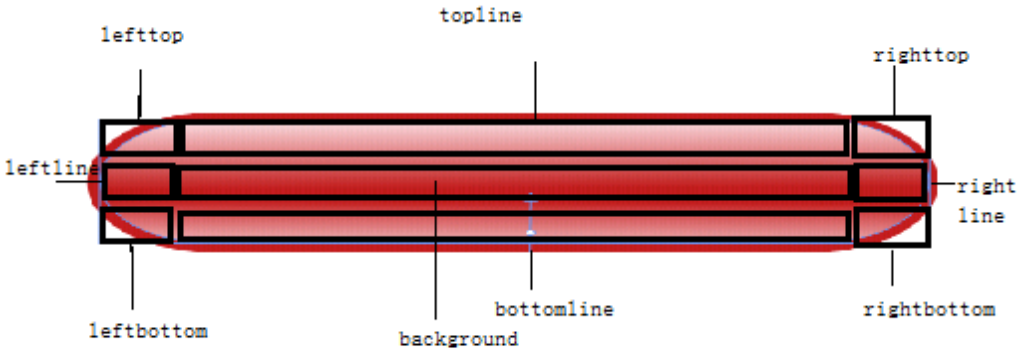
```
typedef struct hiHIGV_STYLE_S
{
    HIGV_STYLE_TYPE_E      StyleType;
    HIGV_STYLE_MEMBER_U   Top;
    HIGV_STYLE_MEMBER_U   Bottom;
    HIGV_STYLE_MEMBER_U   Left;
    HIGV_STYLE_MEMBER_U   Right;
    HIGV_STYLE_MEMBER_U   LeftTop;
    HIGV_STYLE_MEMBER_U   LeftBottom;
    HIGV_STYLE_MEMBER_U   RightTop;
    HIGV_STYLE_MEMBER_U   RightBottom;
    HIGV_STYLE_MEMBER_U   BackGround;
    HI_COLOR               FontColor;
```

```
    HI_U32                  bNoDrawBg;
    HI_U32                  LineWidth;
} HIGV_STYLE_S;
```

- **HIGV_STYLE_S**: data structure for creating the skin. The HiGV control skin consists of nine squares and is classified into the color block skin and image skin.
- **HIGV_STYLE_MEMBER_U**: hexadecimal ARGB color value for the color block skin or image resource ID for the image skin
- **FontColor**: control font color, also called foreground color
- **bNoDrawBg**: whether to draw the skin background. If it is set to true, the background color is not drawn.
- **LineWidth**: line width of the color block skin

**Figure 3-6** Nine-square skin




The image resources can also be used directly for controls such as the image, scroll bar, and spin controls.

## Animation Information

```
typedef struct hiHIGV_ANIM_INFO_S
{
    HI_U32          AnimHandle;
    HI_U32          DurationMs;
    HI_U32          RepeatCount;
    HI_U32          DelayStart;
    HIGV_ANIM_TYPE_E AnimType;
    union
    {
        HIGV_ANIM_TRANSLATE_INFO_S Translate;
        HIGV_ANIM_ALPHA_INFO_S Alpha;
        HIGV_ANIM_ROLL_INFO_S Roll;
        HIGV_ANIM_ANY_INFO_S Any;
    }AnimParam;
```

```
} HIGV_ANIM_INFO_S;
```

- HIGV_ANIM_INFO_S is a data structure for creating an animation.
- **AnimHandle** is the handle to animation information.
- **DurationMs** indicates the duration of the animation.
- **RepeatCount** indicates the loop count, and the value **–1** indicates an infinite loop.
- **RepeatMode** indicates the loop mode and is valid only when **RepeatCount** is greater than **0**. **0** indicates that the loop starts anew, and **1** indicates the loop starts reversely from the end position.

It is recommended that the animation information be defined in the XML file. (The animation information can be created by code, which is not recommended.) After the handle to animation information is available, you can create an animation and start it. The code example is as follows:

```
HI_S32 AppCreateStartAnimation(HI_HANDLE *phAnim)
{
HI_S32 s32Ret;

//Create an animation.
//ANIM_BTN_MOVE: handle to animation information; ANIM_BUTTON: handle to
the control; phAnim: handle to the animation
   s32Ret = HI_GV_Anim_CreateInstance(ANIM_BTN_MOVE, ANIM_BUTTON, phAnim);
if (HI_SUCCESS != s32Ret)
{
       printf("create animation error, s32Ret=%d\n", s32Ret);
    return s32Ret;
}

//Starts the animation.
//WND_HANDLE: handle to the current window; BUTTON1: handle to the
control; phAnim: handle to the animation
   s32Ret = HI_GV_Anim_Start(WND_HANDLE, *phAnim);
if (HI_SUCCESS != s32Ret)
{
       printf("start animation error, s32Ret=%d\n", s32Ret);
}
return s32Ret;
}
```

**NOTICE**

Destroy the resource by calling HI_GV_Res_Destroy when it is not used.

## 3.5.3 Configuring Resources in the XML File

To create fonts, animations, and skins using an XML description file, you need only to set the resource path in the XML file. The creation process is implemented by the HiGV, which is very convenient.

```
<font
    id="common_font_text_18"
    sbname="../resource/font/ttf/sbfont.ttf"
    mbname="../resource/font/ttf/mbfont.ttf"
    size="18"
    isbold=""
    isitalic=""/>
    <translate
    id="ANIM_BTN_MOVE"
    duration_ms=”500”
    repeatcount=”3”
    repeatmode=”1”
    delaystart=”20”
    animtype=”0”
    fromx=”20”
    fromy=”6”
    tox=”252”
    toy=”260”
    />

skin
    id="button_default"
    type="pic"
    btmlineidx="./res/default/button_bottom.png"
    toplineidx="./res/default/button_top.png"
    llineidx="./res/default/button_midleft.png"
    rlineidx="./res/default/button_midright.png"
    ltopidx="./res/default/button_topleft.png"
    rbtmidx="./res/default/button_bottomright.png"
    rtopidx="./res/default/button_topright.png"
    lbtmidx="./res/default/button_bottomleft.png"
    bgidx="./res/default/button_mid.png"
    isnodrawbg="no"
    linewidth="2"
    fgidx=""/>

<skin
    id="gray_skin"
    type="color"
```

```
    btmlineidx="0xFFC0C0C0"
    toplineidx="0xFFC0C0C0"
    llineidx="0xFFC0C0C0"
    rlineidx="0xFFC0C0C0"
    ltopidx="0xFFC0C0C0"
    rbtmidx="0xFFC0C0C0"
    rtopidx="0xFFC0C0C0"
    lbtmidx="0xFFC0C0C0"
    bgidx="0xFFC0C0C0"
    isnodrawbg="no"
    linewidth="2"
    fgidx=""/>
```

The structure attributes in the XML attribute configuration APIs can be directly used. For example, **common_font_text_18**, **button_default**, and **gray_skin** in the following instance are the handles of the created font and skin resources.

```
HI_GV_Widget_SetFont(hButton0, common_fonttext_18);//Set a font for
hButton0.
HI_GV_Widget_SetSkin(hButton0, HIGV_SKIN_NORMAL, button_default);//Set
the normal skin for hButton0.
```

## 3.5.4 Loading and Releasing Resources

The resources that can be loaded and released mainly include skins and images. The HiGV controls load resources in either of the following two modes:

- The resources are not released after being loaded. The resources are loaded when the control is displayed for the first time and released when the control is destroyed (HI_GV_Widget_Destroy).

  This mode is recommended when the memory is sufficient or the control is used frequently. This mode requires more memory, but the resources do not need to be loaded again when the control is displayed again.
- The resources are loaded when the control is displayed and released when the control is hidden.

  This mode requires less memory. However, the resources are loaded every time the control is displayed, and therefore the control display speed is low.

The first mode is used by default. If you want to use the second mode, set the common attribute **isrelease** in the XML file. **isrelease** is a bool type attribute and the value **yes** indicates the second mode. If the control is created by calling an API, set the **style** member in the HIGV_WCREATE_S structure to **HIGV_STYLE_HIDE_UNLODRES** when the control is created.

### Loading Resources

When a control is loaded, if the skin is the image skin, all image members of the skin are decoded before other private images of the control are loaded. The HiGV counts the use of each resource. Each time a resource is used, the count value increases by one.

To display a control, HI_GV_Widget_Show() must be called. If the displayed control is a container control, all displayed child controls in the container are loaded. The HIGV_MSG_SHOW event is triggered when the control is displayed, and it calls back the register event function (onshow event). The sequence is as follows:

1. The resources are loaded first.
2. The onshow event function is called back after the resources are loaded successfully.
3. The HIGV_MSG_PAINT message is sent.

**NOTE**

If the resources are not released when the control is hidden, the resources are loaded after the control receives the HIGV_MSG_PAINT event for the first time, and then the control is drawn.

### Releasing Resources

A resource may be used by other controls when it is released. Therefore, the HiGV only counts down the use of the resource when it is released. The resource is truly released when the value is counted down to 0.

### Setting Resources to Resident

A resource may be referenced by other controls when it is released. Therefore, the value of the resource reference times is counted down when a resource is released. A resource is truly released only when the value of reference times is counted down to 0.

**NOTICE**

If resources are released when the control is hidden, when the UI is switched, display the new UI by calling HI_GV_Widget_Show before hiding the previous UI by calling HI_GV_Widget_Hide; otherwise, the resources shared by the two UIs are released and then loaded again.

## 3.6 Data Model

The ADM is used to display batch data. The controls that use the ADM include the list box, spin, and scroll grid. The UI display and data management of the HiGV are separate. The control instance displays data, and the ADM controls data.

The system provides a DDB for basic data management. You can also specify external DBs, such as the Berkeley DB frequently used in the embedded system or the SQLite DB.

ADMs can be bound to or unbound from controls dynamically. When the data of an ADM changes, the system instructs the controls to which the ADM is bound to refresh automatically. For details about the ADM, see the header file **hi_gv_adm.h**.

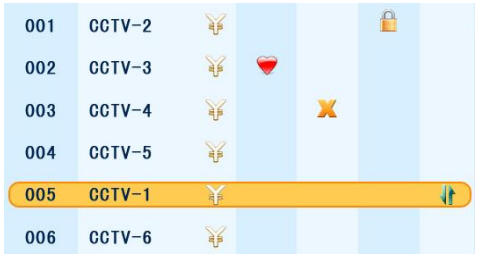
### 3.6.1 Data Form

Figure 3-7 shows a list box that is bound to an ADM. All character strings and images in the list box come from the ADM. As shown in Figure 3-7, the type of data in each column is the same, which corresponds to a field in the ADM. Each row contains all types of fields and is

called a piece of data. A field involves two elements: data type and maximum length. A data model consists of several fields.

**Figure 3-7** List box instance




The following describes the structure of fields in the code:

```
typedef struct hiHIGV_CELLATTR_S
{
    HIGV_DT_E eDataType;
    HI_U32 MaxSize;
}HIGV_FIELDATTR_S;

typedef enum
{
    HIGV_DT_S8 = 0, /**< char */
    HIGV_DT_U8,     /**< unsigned char */
    HIGV_DT_S16,    /**< short */
    HIGV_DT_U16,    /**< unsigned short */
    HIGV_DT_S32,    /**< int */
    HIGV_DT_U32,    /**< unsigned int */
    HIGV_DT_S64,    /**< long long */
    HIGV_DT_U64,    /**< unsigned long long */
    HIGV_DT_F32,    /**< float */
    HIGV_DT_D64,    /**< double */
    HIGV_DT_STRING, /**< char * */
    HIGV_DT_HIMAGE, /**< image handle */
    HIGV_DT_STRID,  /**< multi-language  string ID*/
    HIGV_DT_BUTT
} HIGV_DT_E;
```

- For **HIGV_DT_S8** to **HIGV_DT_D64**, the length is the number divided by 8. For example, the length of **HIGV_DT_S8** is 1, and that of **HIGV_DT_D64** is 8.
- **HIGV_DT_STRING** is the character string type. The length is the maximum length of the character string, which must be an integral multiple of 4.
- **HIGV_DT_HIMAGE** and **HIGV_DT_STRID** are the image resource ID and multi-language character string ID respectively, and the length is 4.

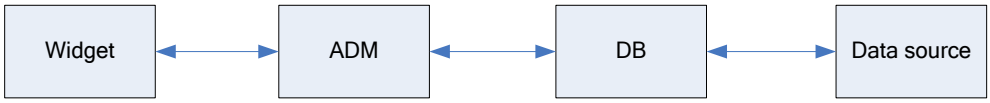
# 3.6.2 ADM Usage

The ADM data comes from two sources:

- OwnDB: DDB provided by the HiGV. Data is stored in the DB in advance. The OwnDB is used for static data or a small amount of data.
- UserDB: DB specified by the user. The data model obtains data by using the callback function registered by the user.

No matter which DB is used, the control is bound to the ADM, the ADM is bound to the DB, and the DB manages data.

**Figure 3-8** Binding relationship




## Creating an ADM

The following are the parameters for creating an ADM:

```
typedef struct hiADM_OPT_S
{
    HI_U32 DBSource;
    HI_U32 FieldCount;
    HIGV_FIELDATTR_S *pFieldAttr;
    HI_U32 BufferRows;
    GetCountFunc GetCount;
    GetRowValueFunc GetRowValue;
    RegisterDataChangeFunc RegisterDataChange;
    UnregisterDataChangeFunc UnregisterDataChange;
}HIGV_ADMOPT_S;
```

HIGV_ADMOPT_S is the parameter structure for creating the data model. Its parameters are described as follows:

- **DBSource**: handle of the HiGV DB to which the ADM is bound when the OwnDB is used. This parameter is not used for the UserDB.
- **FieldCount**: number of fields of the ADM
- **pFieldAttr**: attribute of each field. The number of nodes of this pointer equals the value of **FieldCount**.
- **BufferRows**: number of data entries obtained from the DB each time when the UserDB is used. This parameter is not used for the OwnDB.
- **GetCount**: function for obtaining the number of data entries when the UserDB is used. This parameter is not used for the OwnDB.
- **GetRowValue**: function for obtaining data when the UserDB is used. This parameter is not used for the OwnDB.
- **RegisterDataChange**: registered event callback function when the ADM is created

- **UnregisterDataChange**: unregistered event callback function when the ADM is destroyed

To be specific, the parameters **DBSource**, **FieldCount**, and **pFieldAttr** must be configured for the OwnDB, and the parameters **FieldCount**, **pFieldAttr**, **BufferRows**, **GetCount**, and **GetRowValue** must be configured for the UserDB.

## OwnDB

The HiGV data buffer header file is **hi_gv_ddb.h**, which contains all data buffer APIs.

To use the OwnDB, you need to create a DB first by calling the following API:

```
HI_GV_DDB_Create(HI_U32 FieldCount, const HIGV_FIELDATTR_S* pFieldAttr,
HI_HANDLE* phDDB)
```

- **FieldCount**: number of DB fields, same as that of the bound ADM
- **pFieldAttr**: attribute of each field. The number of nodes of this pointer equals the value of **FieldCount**.
- **phDDB**: DB handle returned to the user by the HiGV after the DB is successfully created. The handle is the ID of the DB.

Use the DB handle as **DBSource** (data source of the ADM) of HIGV_ADMOPT_S. Then you can operate the DB data by calling APIs such as HI_GV_DDB_Append, HI_GV_DDB_Modify, and HI_GV_DDB_Insert.

## UserDB

The HiGV does not store data that is managed by the external DB specified by the user. When a control needs to use the data, the bound ADM is instructed to directly obtain the data by using the callback event registered by the user.

The callback function for obtaining the total number of data entries is typedef HI_S32 (*GetCountFunc)(HI_U32 DBSource, HI_U32 *RowCnt).

- **DBSource**: handle of the DB that is bound to the ADM. This parameter is not required because the UserDB is used and the callback function has determined the bound DB.
- **RowCnt**: number of external data entries. The user needs to implement this function and transfer the number of data entries as ***RowCnt** in the function.

  The callback function for obtaining the data is typedef HI_S32 (*GetRowValueFunc)(HI_U32 DBSource, HI_U32 Row, HI_U32 Num, HI_VOID *pData, HI_U32 *pRowNum). The user needs to implement this function to transfer data in the external DB to the HiGV controls in the function.
- **DBSource**: handle of the DB that is bound to the ADM. This parameter is not required because the UserDB is used and the callback function has determined the bound DB.
- **Row**: start row for obtaining data. The region for drawing the control may not start from row 0 in some scenarios, for example, after the list box is scrolled.
- **Num**: number of rows that the HiGV wants to obtain
- **pData**: data organized by the user
- **pRowNum**: number of rows that are transferred, which may be less than **Num**. The HiGV also needs to be notified of the actual number of data entries.

### Binding an ADM to Controls

Currently the ADM can be bound to the following controls:

- List box
- Scroll grid
- Spin
- Combo box

An ADM can be bound to multiple controls at the same time, but a control can be bound to only one ADM. Controls that are bound to the same ADM have the same data source.

- To bind a control to an ADM, call HI_GV_Widget_BindDataSource.
- To unbind a control from an ADM, call HI_GV_Widget_UnbindDataSource.

### Synchronizing Data

To display the DB contents on controls, you need to instruct the control to obtain data from the DB. The method of obtaining data varies according to the DB (OwnDB or UserDB), but that of notifying the control is the same.

Any of the following four methods can be used to display new data on a control. You need only to use one of them; otherwise, unnecessary time is consumed.

- Call HI_GV_Widget_SyncDB to instruct the control to synchronize and display new data. This API takes effect on only a single control.
- Call HI_GV_ADM_Sync to synchronize all controls that are bound to the ADM. This API takes effect on all controls that are bound to the ADM.
- Send the HIGV_MSG_DATA_CHANGE message to the control. The effect is the same as that of calling HI_GV_Widget_SyncDB.
- Send the HIGV_MSG_ADM_DATACHANGE message to the ADM. The effect is the same as that of calling HI_GV_ADM_Sync.

When data is synchronized, the ADM is instructed to obtain data from the DB and store the data in the buffer. The buffer is requested when the ADM is created. When the related control needs to be drawn, the data is obtained from the buffer.

## 3.6.3 Data Model XML

If you create a GUI by using the XML files, the ADMs, DBs, and empty callback functions are created at a time, and controls are bound to the related ADM when they are created. This method is easy and convenient.

```
/** Data model description*/
<datamodel
id = "datamodel_record"
field="
ID:u32:4;Name:string:20;HasExperience:u32:4;Mobile:string:12;hImg:resid:4
"
datasource = "owndb"
getrowcount = ""
getrowvalue = ""
registerdatachange = ""
```

```
unregisterdatachange = ""/>
```

The data model is similar to HIGV_ADMOPT_S. Note the following:

- The ID of the data model is the ADM handle, and **DBsource** is the DDB handle.
- **field** is the data field stored in the data model.

  Two adjacent fields are separated by using a semicolon (;). A field consists of three members, which are separated by using colons (:).

  - The first member is the field name, which does not affect the actual function of the field.
  - The second and third members are similar to HIGV_FIELDATTR_S. For example, **u32** corresponds to HIGV_DT_U32. For details, see the *HiGV Label User Guide*.

  You can consider the fields of the data model as a structure:

```
struct
{
    HI_U32 ID;
    HI_CHAR Name[20];
    HI_U32 HasExperience;
    HI_CHAR Mobile[12];
    HI_RESID hImg;
}
```

This is a list box that is bound to the data model datamodel_record. It has five columns of data, and columns 0 to 4 correspond to fields 0 to 4 in the data model.

```
/**GUI control description*/
<view
    id = "allwidget"
    onload = "onload"
    unload = "unload">
    <window
           id = "allwidget"
           top = "0"
           left = "0"
           width = "720"
           height = "576"
           normalskin = "common_skin_previewbg"
           isrelease = "yes"
           isnofocus = "no"
           islanchange = "yes"
           iswinmodel = "no"
           isskinforcedraw = "no"
           opacity = "255"
           onshow = "allwidget_window_onshow"
           onhide = "allwidget_window_onhide"
           onrefresh = "allwidget_window_onrefresh"
           onevent = "allwidget_window_onevent"
```

```
onkeydown = "allwidget _window_onkeydown"
ontimer ="allwidget_window_ontime"
onlanchange="allwidget_window_lanchange"
layer = "LAYER_0" >
 <listbox
       id = "listbox_record"
       top = "150"
       left = "70"
       width = "480"
       height = "192"
       normalskin = "common_skin_buttonbg"
       transparent = "no"
       font = ""
       rownum = "6"
       colnum = "5"
       leftorderobj = ""
       rightorderobj = ""
       uporderobj = "allwidget_button1"
       downorderobj = "spin_array"
       datamodel = "datamodel_record"
       hlineheight = "2"
       hlinecolor = "0xFF0000FF"
       vlinewidth = "2"
       vlinecolor = "0xFF0000FF"
       rowselectskin = "commonpic_skin_rowsel1"
       rownormalskin = "commonpic_skin_rowsel2"
       oncellselect = "listbox_record_oncellsel">
       <listcol
          id = "listbox_record_ID"
          colindex = "0"
          coltype = "text"
          colwidth = "30"
          colalignment = "center"
          colbinddb = "yes"
          coldbindex = "0"
          colimage = ""
          coldataconv = ""/>
       <listcol
          id = "listbox_record_name"
          colindex = "1"
          coltype = "text"
          colwidth = "120"
          colalignment = "center"
          colbinddb = "yes"
```

```
                    coldbindex = "1"
                    coldataconv = ""/>
                <listcol
                    id = "listbox_record_experience"
                    colindex = "2"
                    coltype = "image"
                    colwidth = "40"
                    colalignment = "center"
                    colbinddb = "yes"
                    coldbindex = "2"
                    colimage = "./res/Fav.gif"/>
                <listcol
                    id = "listbox_record_mobile"
                    colindex = "3"
                    coltype = "text"
                    colwidth = "250"
                    colalignment = "center"
                    colbinddb = "yes"
                    coldbindex = "3"
                    coldataconv = ""/>
                <listcol
                    id = "listbox_record_himg"
                    colindex = "4"
                    coltype = "image"
                    colwidth = "40"
                    colalignment = "center"
                    colbinddb = "yes"
                    coldbindex = "4"/>
            </listbox>
    </window>
</view>
```

```
/**Main program app.c*/
typedef struct {
    HI_U32 ID;
    HI_CHAR Name[20];
    HI_BOOL HasExperience;
    HI_CHAR Mobile[12];
    HI_HANDLE hImg;
}TestRecord;
/**Adding data*/
static HI_S32 ListBox_GenerateRecord(HI_U32 Num)
{
```

```
    HI_S32 Ret;
    TestRecord Record;
    HI_U32 i;
    HIGV_DBROW_S DBRow;
    HI_HANDLE hDDB = INVALID_HANDLE;

    ListBox_CreateImgRes();

    DBRow.Size = sizeof(Record);
    DBRow.pData = &Record;

    HI_GV_ADM_GetDDBHandle(datamodel_record, &hDDB);
    printf("hDDB is %d\n", hDDB);

    HI_GV_DDB_EnableDataChange(hDDB, HI_FALSE);//Disable data notification

    for(i = 0; i < Num; i++)
    {
        Record.ID = i;
        snprintf(Record.Name, 20, "Name%d", i);
        snprintf(Record.Mobile, 20, "Tel%d", i);
        Record.HasExperience    = (HI_BOOL)(i%2);
        Record.hImg = s_hRecord2Img[i%4];
        Ret = HI_GV_DDB_Append(hDDB, &DBRow);//Add data
        assert(Ret == HI_SUCCESS);
    }

    HI_GV_DDB_EnableDataChange(hDDB, HI_TRUE); //Enable data notification
    return Ret;
}

HI_S32 main(HI_S32 argc, HI_CHAR *argv[])
{
…

    HI_GV_Widget_Show(allwidget);
    HI_GV_Widget_Active(allwidget);

    /**Adding data*/
    Ret = ListBox_GenerateRecord(20);
    assert(Ret == HI_SUCCESS);
    Ret = HI_GV_App_Start(hApp);
    if (HI_SUCCESS != Ret)
    {
```

```
        HI_GV_PARSER_Deinit();
        (HI_VOID)HI_GV_App_Destroy(hApp);
        HI_GV_Deinit();
        return NULL;
    }

    HI_GV_PARSER_Deinit();
    (HI_VOID)HI_GV_App_Destroy(hApp);
    HI_GV_Deinit();

    return 0;
}
```

**NOTICE**

- The length and type of fields in the data model must correspond to those written to the data model. For example:
  - The lengths of u32 and resid must be 4.
  - The length of the character string written to the "Text:string:12" code must be 12.
- When adding data in batches, you need to disable the data notification message function, and enable it after data is added.
- If the control needs to be refreshed immediately after data is updated, you need to call the API to synchronize data to the control.
- HI_GV_Widget_SyncDB is used to synchronize the DB of a specific control, and HI_GV_ADM_Sync is used to synchronize all controls that are bound to the data model. If data is synchronized by using a non-HiGV thread, a segment error may occur when the preceding APIs are called directly. The right approach is to send the HIGV_MSG_ADM_DATACHANGE message to the control, instructing the control to synchronize the DB.

# 3.7 Multi-Language

Character strings can be configured and displayed on some HiGV controls, such as the label, button, and multiedit. The HiGV provides the function of switching the language for the entire GUI, that is, the multi-language function. For details about the multi-language function, see the header file **hi_gv_lan.h.**

## 3.7.1 Multi-Language XML Description

The XML file is used to describe the language types and multi-language character strings, and the xml2bin tool is used to automatically generate the character string resource file. A multi-language character string ID corresponds to character strings in all language environments. A control can display the corresponding character string in different language environments if you configure the multi-language character string ID for the control. Currently the HiGV supports the generation of multi-language character strings by using only the XML file.

The following is an example of the multi-language XML file:

```
<language
    id = "language_sample"
    languageinfo = "ru:russian;en:english;ar:arabic"
    locale = "en">



    <lanstr
    id = "STRID_AMPM_AM"
    en = "AM."
    ar = "ص"
    />

    <lanstr
    id = "STRID_AMPM_PM"
    en = "PM."
    ar = "م"
    />

    <strset
    id = "STRSET_AMPM"
    tt = "STRID_AMPM_AM;STRID_AMPM_PM;"
    />
    <timefmt
    id = "TIMEFMT_SHORT_TIME"
    en = "<tt> [h]:[m]:[s]"
    ar = "<tt> [h]:[m]:[s]"/>
</language>
```

The preceding instance is described as follows:

- <language

  The tag <language indicates the language environment, which must be one and one only.

  - The **languageinfo** attribute describes the multi-language types and corresponding abbreviations in the format of "xx:xx;xx:xx;...". Languages are separated by using semicolons (;).

    The language abbreviation is on the left of the colon (:) and is the actually used language environment tag; the language full name is on the right of the colon (:) and has no actual usage.

  - **locale** indicates the local language, which is the most important and frequently used language. It is a language environment in **languageinfo** and must be configured.

- <lanstr

  The <lanstr part describes the multi-language character strings in detail. IDs such as STRID_AMPM_AM and STRID_AMPM_PM in the instance are the multi-language character string IDs that can be used. You can set the multi-language IDs for controls by

calling HI_GV_Widget_SetTextByID. **en**, **zh**, and **ar** correspond to the language environments described in **languageinfo**.

**locale** has no default value and must be configured. Other language environments can be empty. If another language environment is empty, it is replaced with the value of **locale**.

- <strset and <timefmt

  <strset and <timefmt are dedicated to control clocks. The time format and character string vary according to the country or region. **tt** is the combination of multi-language character string IDs, and **timefmt** is the time display sequences of different language environments.

## 3.7.2 Registering and Switching the Language Environment

If you use the XML files to describe multiple languages, you need to register and use them by calling APIs in the C code. XML files can be converted by using the xml2bin tool to obtain the .lang binary files, such as **en.lang** and **ar.lang**. You can perform the following operations before the HiGV thread starts:

- Register languages with the HiGV system by calling HI_GV_Lan_Register.
- Switch the language by calling HI_GV_Lan_Change.

## 3.7.3 Changing the Language Writing Direction

Some languages such as the Arabic and Hebrew are written from right to left. When the entire language direction is changed due to the switchover of the language environment, the entire control layout must change accordingly. The HiGV provides the automatic control mirroring function.

You can set the automatic mirroring attribute for controls by calling HI_GV_Widget_EnableMirror(HI_HANDLE hWidget, HI_BOOL bPosMirror, HI_BOOL bInteriorMirror).

- **bPosMirror** specifies whether the coordinate position of the control relative to the parent container (window relative to the graphics layer) is mirrored.
- **bInteriorMirror** specifies whether the control content (such as text alignment mode and table left/right direction) is mirrored.

When the language environment is changed, the HiGV determines whether to mirror controls automatically based on the language environments before and after the change.

## 3.7.4 Supporting Multiple Languages

Table 3-3 lists the languages supported by HiGV.

**Table 3-3** Languages supported by HiGV

| Language Type | Macro Definition |
|---|---|
| Simplified Chinese | LAN_ZH |
| Traditional Chinese | LAN_ZH_TW |
| Arabic | LAN_AR |
| Czech | LAN_CS |

| Language Type | Macro Definition |
|---|---|
| Danish | LAN_DA |
| German | LAN_DE |
| Greek | LAN_EL |
| English | LAN_EN |
| Spanish | LAN_ES |
| Persian | LAN_FA |
| Finnish | LAN_FI |
| French | LAN_FR |
| Italian | LAN_IT |
| Hebrew | LAN_HE |
| Japanese | LAN_JA |
| Korean (South Korea) | LAN_KO |
| Dutch | LAN_NL |
| Portuguese | LAN_PT |
| Russian | LAN_RU |
| Swedish | LAN_SV |
| Thai | LAN_TH |
| Turkish | LAN_TR |
| Polish | LAN_PO |
| Vietnamese | LAN_VN |
| Hungarian | LAN_HU |

# 3.8 XML File

This section introduces the XML files that have been mentioned for multiple times in the preceding sections.

The XML files can be used to describe GUI controls, skins, multi-language character strings, fonts, data models, and frequently used callback events. These files can be converted into binary (.bin and .lang) files by using the xml2bin tool and generate the handle header files, callback event header files, and empty function c files for callback events.

By using the XML files, the code for creating and configuring various controls and resources can be greatly reduced, and manager handles and callback functions can be set in a unified manner.

The XML files are classified into the following five types:

- Skin XML: skin description file for generating control skins. For details, see section 3.5.3 "Configuring Resources in the XML File."
- Language XML: multi-language description file for generating multi-language character strings. For details, see section 3.7 "Multi-Language."
- Font XML: font description file. The font determines the size and style of UI texts. For details, see section 3.5.3 "Configuring Resources in the XML File."
- Data model XML: ADM description file. For details, see section 3.6 "Data Model."
- View XML: GUI control description file for generating HiGV controls and registering frequently used message callback events. Typically multiple view XML files are used. Controls that are closely related are described in the same view XML file, and the file is named based on the control function.

**NOTE**

For details about the XML tags, see the *HiGV Label User Guide*.

## 3.8.1 XML GUI Description

```
<view

onload = ""
unload = "">
   <window
      id = "hello_sample"          <!—Window name-->
      top = "0"               <!—Control coordinate-->
      left = "0"
      width = "720"
      height = "576"
      normalskin = "commonpic_skin_colorkey" <!—Common skin-->
      transparent = "no"
      colorkey = "0xff0000ff"
      isrelease = "yes"
      opacity = "255"
      winlevel = "0"
      onshow = "hello_window_onshow">  <!—Registered onshow event
callback-->
      <button
         id = "hello_button_ok"         <!—OK button name-->
         top = "285"
         left = "235"
         width = "246"
         height = "40"
         normalskin = "commonpic_normalskin_button"
         disableskin = ""
         highlightskin = ""
         activeskin = "commonpic_activeskin_button"    <!—Focused skin-->
```

```
            transparent = "no"
            isrelease = "no"
            text = "ID_STR_OK"     <!—Multi-language character string ID-->
            alignment = "hcenter|vcenter"
            onclick = "hello_button_onclick"/>  <!— Button onclick event
callback-->

        <label
            id = "hello_label_helpinfo"   <!—Textbox ID-->
            top = "200"
            left = "235"
            width = "246"
            height = "50"
            normalskin = "common_skin_black"
            disableskin = ""
            highlightskin = ""
            activeskin = ""
            transparent = "yes"
            isrelease = "no"
            text = ""
            alignment = "hcenter|vcenter"/>
    </window>
</view>
```

- Tag: The description starts with a tag, for example <window>, ">" indicates the end of the description of the control itself, which may be followed by the description of child elements. "</Tag name>" or "/>" indicates the ending of the entire control. Typically the ending of a container control contains the tag name. For example:

  ```
  <window
  Window description>
  <label>
  Label description
  />
  Description of other child controls in the Window
  </window>
  ```

  The tag name determines the description type. All GUI descriptions can be divided into multiple XML files by module or function to facilitate modification and management.

- View: Actually a view is not a control instance. The view provides an ID for the GUI XML file. The HiGV provides the HI_GV_PARSER_LoadViewById API, which allows you to create all control instances contained in a view by loading the view ID.

**NOTICE**

Windows must be described in a view. Only one window has the same ID as the view ID.

## 3.8.2 Parsing of XML Files

By using the xml2bin tool, the XML description files can generate the binary files **higv.bin** and *xx***.lang**, header files **higv_cextfile.h** and **higv_language.h**, and **higv_cextfile.c** (corresponding to **higv_cextfile.h**). If callback events are registered in the view XML file, a C file with the same name as the XML file is generated (for example, **view.c** for **view.xml**). If the files also contain empty functions for callback events, a C file for data models is also generated.

- The handles of multi-language character strings are defined in **higv_language.h**.
- The control and skin handles are declared in **higv_cextfile.h** by using the event callback function directly registered in the XML file.
- The multi-language information is stored in the .lang binary file.
- All information about the XML GUI, skins and data models is stored in the binary file **higv.bin**. You can load **higv.bin** by calling HI_GV_PARSER_LoadFile to obtain all XML description information except the multi-language character strings.

# 4 Controls

## 4.1 Window

A window is a drawn rectangular region on the graphics layer, and its coordinate rectangle is the region that is mapped to the graphics layer. A graphics layer may contain multiple windows at the same time, and the windows can overlap with each other. The HiGo manages window overlapping in a unified manner, creates and uses functional controls in windows, and creates rich UI graphics to associate services with related controls. The window header file is **hi_gv_win.h**.

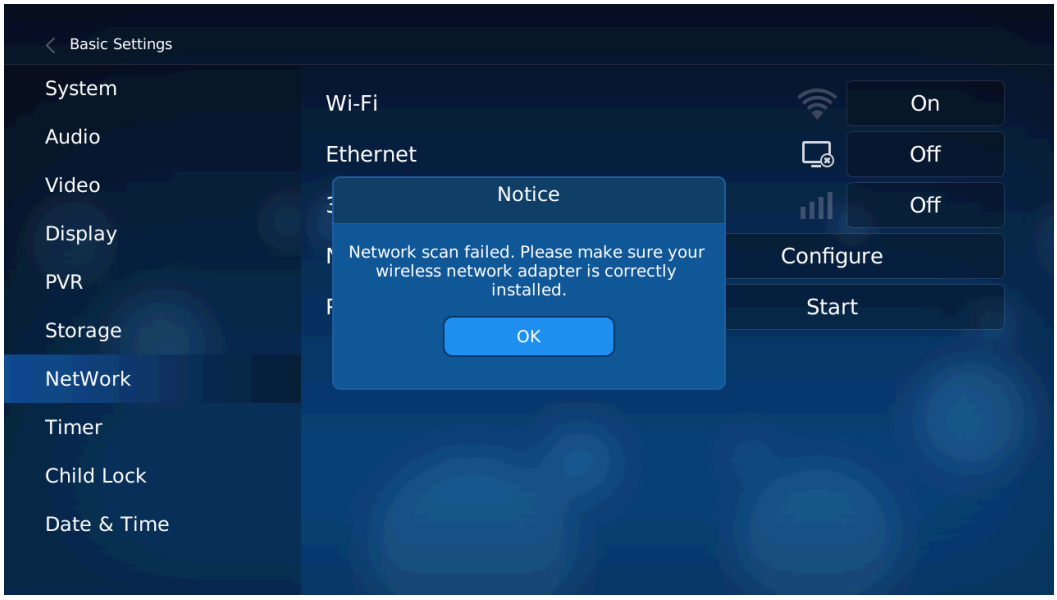
### 4.1.1 Window Overlapping

Windows have different coordinates and sizes. The overlapping is irregular and involves the overlapping sequence. If window A and window B overlap with each other, the Z orders (window level) of the two windows determine their display sequence. The Z order ranges from 0 to 15. A larger value indicates higher order. That is, the overlapping part of the window with the largest Z order value is displayed. If the Z orders of two windows are the same, the focused window has higher display priority.

In Figure 4-1, multiple windows overlap with each other. The Z order value of the **Notice** dialog box is the largest, and therefore the entire dialog box is displayed, covering part of the main UI. The drawing orders of child controls in the window are determined by the window.

**Figure 4-1** GUI instance



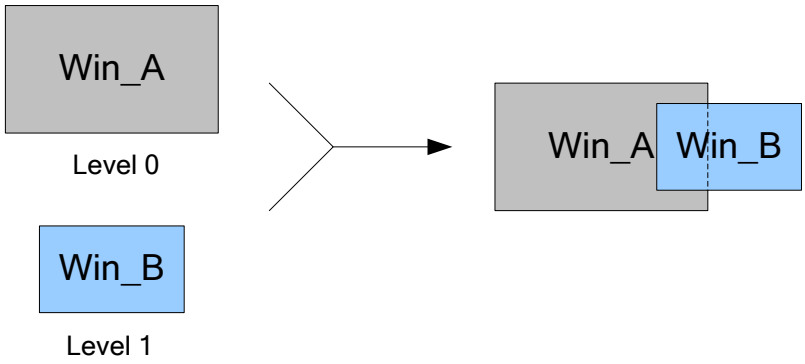

## 4.1.2 Window and Surface

The HiGV/HiGo window surface allocation modes (drawing modes) include the following:

- Non-sharing mode

  In non-sharing mode, each window has its own independent surface, and each surface occupies certain memory according to its size. The drawing of windows in separate surfaces does not interfere with each other. After drawing is complete, windows are transferred to the graphics layer in a unified manner for display. The drawing speed is higher and the overlapping effect is better in non-sharing mode. However, more memory is required. See Figure 4-2.

**Figure 4-2** Window overlapping in non-sharing mode



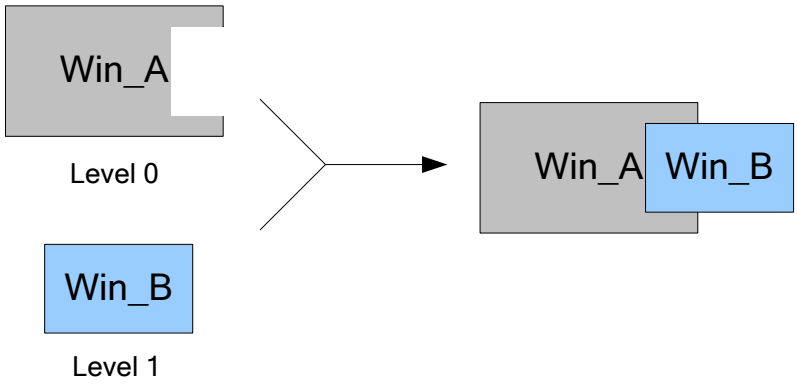

- Sharing mode

  All windows on the UI share the same surface in sharing mode. When windows overlap with each other, the overlapping region of only the window with the highest order is drawn. The sharing mode requires less memory. However, the drawing speed is lower

and the drawing of the overlapping region is more complex than that in non-sharing mode.

**Figure 4-3** Window overlapping in sharing mode




As shown in Figure 4-3, the region of window A that is covered by window B cannot be drawn in sharing mode. When window B is transparent or translucent, the update of the covered region of window A cannot be seen. In sharing mode, the original data on the surface is not erased during drawing, and the overlapping of transparent windows needs to be implemented by using transparent image resources. Window A is displayed when window B is hidden, and window B is displayed after window A is updated to the graphics layer. In this way, the transparency effect is presented. In non-sharing mode, the drawing sequence of windows is not considered when the transparency effect is implemented.

Color blocks and images are drawn on a surface by calling the underlying drawing interface, and then the surface is transferred to the graphics layer for display. A surface is a display buffer. Creating a surface requires certain MMZ memory with the size of Width x Height x Number of bytes of the pixel format (for example, 4 bytes for ARGB8888).

- Each window has an independent surface with the same size as that of the window in non-sharing mode. The window is drawn on its own surface. When windows overlap with each other, the region covered by upper-level windows is drawn but cannot be seen.
- All windows are drawn on the same surface in sharing mode. When multiple windows are displayed at the same time and overlap with each other, only the visible part but not the covered part of the overlapping region is drawn.

Therefore, the window overlapping effect is better in non-sharing mode, especially when the windows are transparent or translucent. However, more memory is required.

When the window XML attribute **isrelease** is **yes** (HIGV_STYLE_HIDE_UNLODRES style), the surface is created when the window is displayed and is destroyed when the window is hidden; otherwise, the surface is created when the window is created and is released when the controls are destroyed (HI_GV_Widget_Destroy).

## 4.1.3 Private Window Attributes

- opacity

  Window opacity can be configured in non-sharing mode. The configured opacity affects all controls in the window. The opacity ranges from 0 to 255. The value 0 indicates transparent and the value 255 indicates opaque. You can set the opacity by using the XML attribute **opacity** or calling HI_GV_Win_SetOpacity.

- colorkey

  A 32-bit color value can be configured as the colorkey of a window. All colorkey colors of the window are transparent on the graphics layer. You can set the colorkey by configuring the XML attribute **colorkey**.

- pixelformat

  The pixel format of each window can be separately configured. The default value is the pixel format for the graphics layer to which the window belongs. You can set the pixel format by configuring the XML attribute **pixelformat**.

- onrefresh

  Window drawing triggers the graphics layer refresh event. onrefresh is the callback function used after the graphics layer is refreshed.

- onshow

  onshow is the display triggered event callback function before the control is drawn. The onshow event can be directly configured in the XML file only for windows. For other controls, you need to register the HIGV_MSG_SHOW event by calling HI_GV_Widget_SetMsgPorc.

### 4.1.4 Transparency in Sharing Mode

In sharing mode, new graphics are drawn directly on the surface without erasing. Therefore, the transparency and translucency effects can also be implemented in sharing mode. Typically images with transparent colors are required to implement the overlapping transparency effect. The background window is drawn first. After the background window is refreshed, the window with the transparency effect is drawn to cover the background.

For the round corners of the dialog box in Figure 4-1, ensure that the dialog box is drawn after the background window is refreshed to the graphics layer. If the background window is redrawn when the dialog box is displayed, the HiGV considers that the dialog box but not the background window should be drawn at the round corners, and therefore the background window cannot refresh the regions covered by the round corners.

If a child control is transparent but the window is not, you need to set the HIGV_STYLE_FORCE_PAINT_PARENT style for the child control by configuring the **isskinforcedraw** attribute in the XML file. If this attribute is configured, the parent background of the child control region is drawn before the child control is drawn to erase the previous graphics of this region.

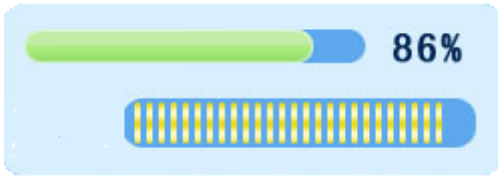
## 4.2 Group Box

The group box is a control container that houses any other controls (except the window). The group box is used to separately manage a group of child controls in a window, and it has no special attributes and events.

Figure 4-4 shows a group box that contains two progress bars and a label.

**Figure 4-4** Group box instance




# 4.3 Button

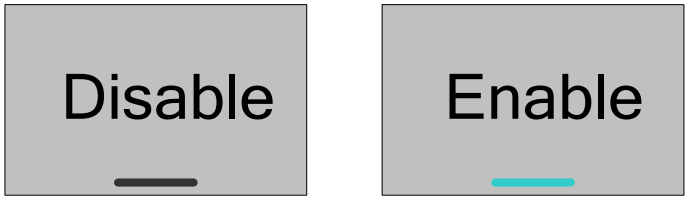
The button is a frequently used control for interaction between the terminal and service processing. It responds to user click operations. The button header file is **hi_gv_button.h**.

The background skins of the button in different status can be configured. The button also supports the display of foreground text, and the alignment mode of the foreground text can be configured. Different buttons have different functions. Figure 4-5 to Figure 4-7 show some frequently used buttons.

**Figure 4-5** Normal button


**Figure 4-6** Switch button


**Figure 4-7** Toggle button



## 4.3.1 Button Types

There are several types of buttons. The button functions vary according to the type. You need to choose the button type as required.

- Common button

  The common button is the most frequently used button for basic input device interaction.

- Radio button

  Radio buttons are used in batches. When a radio button is clicked, its check state is configured or canceled. The button skin varies according to the state. Only one radio button in a parent container is in the check state at a time.

- Check button

  Check buttons are also used in batches. Multiple brother check buttons in the same parent container can be in the check state at the same time.

- Switch button

  The switch button has two character strings which indicate enabled and disabled respectively. The skin and the display position of the character string vary according to the button state. The display of two character strings is mutually exclusive, that is, only one character string is displayed at a time.

- Toggle button

  The toggle button is similar to the switch button, except that the character string position and skin remain unchanged when the button state changes. The toggle button has a rodlike button color bar for distinguishing the button state.

- Soft keyboard button

  Common buttons are automatically converted into soft keyboard buttons when they are used for the soft keyboard. A character is saved as the button value. For details, see section 5.7 "Input Method."

## 4.3.2 Private Attributes

- Private skins

  There are five button skins for matching the check states: checkednormal, checkeddisable, checkedhighlight, checkedactive, and checkedmousedown.

- Text margin

  Texts can be drawn on the entire button region by default. You can specify the virtual region for drawing texts on the button by setting the top, bottom, left, and right margins. The texts are aligned to the drawing region.

- Configuration of the switch button and toggle button

  The character strings and layout when the switch button or toggle button is enabled or disabled, and the toggle color can be configured.

## 4.3.3 Special Events

The callback function onstatuschange can be configured to respond to the change of the check states. This event corresponds to HIGV_MSG_BUTTON_STATUSCHANGE.

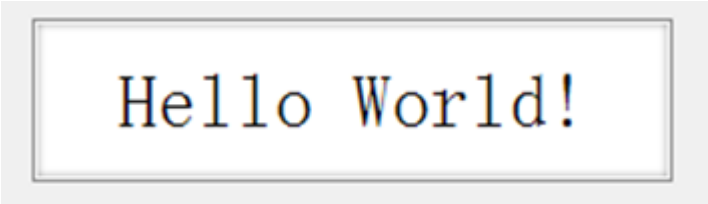
  

# 4.4 Label

The label is a basic text control for displaying static text. It has the following features:

- Supports the display of single-row or multi-row text.
- Supports various alignment modes.
- Supports page turning for multi-row text.
- Supports the scroll bar.
- Cannot be focused.

**Figure 4-8** Label instance




The region for drawing text on a label can also be restricted by setting the margin. Besides that, the label has no other special attribute or event. The label header file is **hi_gv_label.h**.

# 4.5 Image

The image is used for displaying static images, and it cannot be focused. The image resources described in section 3.5 "Resources" can be used as the display content of the image. The image header file is **hi_gv_image.h**.

**Figure 4-9** Image instance

**NOTICE**

If the image resolution is too high, the memory may fail to be applied for. Therefore, when the image control is used to display static images, the capability status of the system resources needs to be considered.

## 4.5.1 Images Stored in the Memory

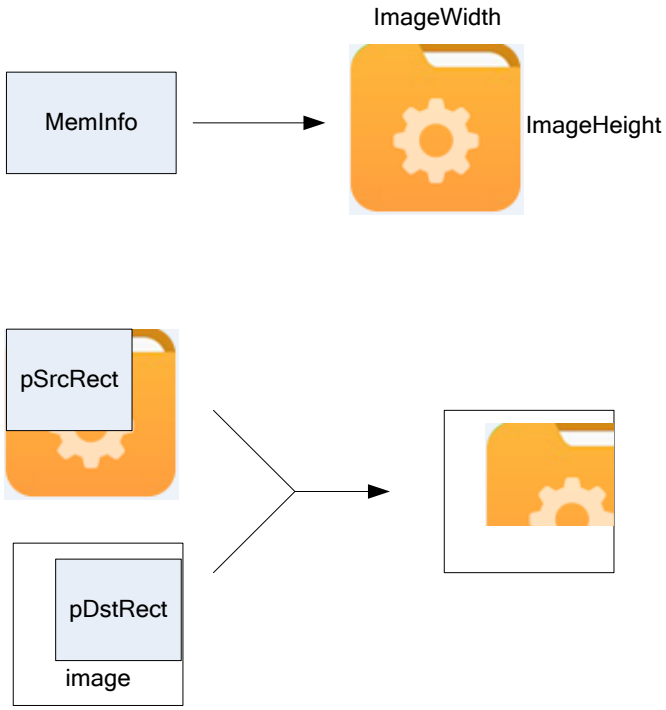
Images may not be stored as files. The image control can also display image data in the MMZ memory. The image can be displayed only if the start address and data length of the memory image are specified.

```
HI_GV_Image_DrawMemImage(HI_HANDLE hImage, HI_GV_MemInfo*  MemInfo,
HI_U32 ImageHeight, HI_U32 ImageWidth, HI_RECT* pSrcRect, HI_RECT*
pDstRect, HIGO_BLTOPT_S* pBlitOpt, HI_BOOL Transparent)
```

where

- **hImage**: handle of the image control
- **MemInfo**: start address and length of the image
- **ImageHeight**: image height
- **ImageWidth**: image width
- **pSrcRect**: source rectangle of the image
- **pDstREct**: destination rectangle to be displayed on the image control
- **pBlitOpt**: blit operation attribute of the image, such as mirroring, scaling, and rotation
- **Transparent**: whether the background skin of the image control is transparent

**Figure 4-10** Memory image instance




MemInfo is decoded as a surface when the image is transferred. Therefore, you need to release the memory image by calling HI_GV_Image_FreeMemSurface after drawing is complete. The memory needs to be released by the object that applies for it. Therefore, HI_GV_Image_FreeMemSurface releases only the generated image surface but not MemInfo.

# 4.6 ImageEx

The ImageEx control is used for displaying dynamic GIF images. The displayed image can be centered or tiled, and is clipped if it is larger than the display area. The ImageEx control cannot be focused. Its header file is **hi_gv_imageex.h**.

The ImageEx has the following functions:

- Sets the display type of an image.
- Sets the frame display interval of a dynamic image.
- Sets the number of repeat times of a dynamic image.

## 4.6.1 Special Event of ImageEx

The onendofrepeat event is reported after a dynamic image is repeatedly played. This corresponds to HIGV_MSG_IMAGEEX_ENDOF_REPEAT.

# 4.7 List Box

The list box is used to display batch data (which can be stored in the database or memory). The list box header file is **hi_gv_listbox.h**. Figure 4-11 shows a list box. It has the following features:

- It can be bound to the data model.
- The source of data in each column can be specified as a field in the data model or customized by the user.
- Each cell displays images or texts, and displays nothing if the cell width is 0.
- Multiple rows and columns are supported.
- The column width can be configured.
- Scroll bars can be bound to the list box.
- The focus can be switched by using the **Up** and **Down** keys, and the page can be turned by using **Page Up** and **Page Down** keys.
- The background of the item in highlighted state or focus state can be configured.
- The cell focus mode is supported.
- The cell width can be customized in cell focus mode.
- The text can be scrolled in cell focus mode (the text contents must exceed the cell width).
- The scrolling modes from left to right and from right to left are supported.
- The scrolling step can be configured.
- The scrolling time interval can be configured.
- Whether to draw grid frames can be configured.
- The column text color can be configured.
- The non-cyclic focus mode is supported when the top or bottom item is focused in item focus mode.
- Icons can be added to the text.
- The scrolling column can be configured in item focus mode.

**Figure 4-11** List box instance


## 4.7.1 Adding Data

Data of the list box comes from the ADM. For details about how to add data, see section 3.6 "Data Model."

## 4.7.2 Private Attributes

```
typedef struct
{
    HI_U32 RowNum;
    HI_U32 ColNum;
    HI_BOOL NoFrame;
    HI_BOOL Cyc;
    HI_BOOL IsCellActive;
    HI_BOOL AutoSwitchCell;
    HI_BOOL Scroll;
    HI_BOOL Fromleft;
    HI_U32 Timeinterval;
    HI_U32 Step;
    HI_U32 ScrollCol;
    HIGV_GET_WIDTH_CB GetWidthCb;
    HIGV_LIST_COLATTR_S *pColAttr;
} HIGV_LIST_ATTRIBUTE_S;
```

- Number of displayed rows

  **RowNum** indicates the number of displayed rows in the list box on the GUI.
- Number of columns

  **ColNum** is the number of columns of the list box, which corresponds to the number of fields in the database.
- Line frame

  The width and color of the horizontal and vertical lines of the list box can be configured. **NoFrame** indicates whether to draw the outer frame of the list box.
- Focus switch

When the list box is focused, the internal focus of the list box can be switched. The internal focus modes of the list box include the row focus mode and cell focus mode, and the row focus mode is used by default. When the focus is switched by using keys, you can specify whether to use cyclic focus. If cyclic focus is configured, the focus jumps to the other end of the list box when you continue to press the key after the focus reaches one end of the list box.

- Scrolling text

  If the cell text length exceeds the cell length, you can set the display mode to scrolling mode. The scrolling direction, interval, and step can be configured.

  In item focus mode, you need to specify the index of the column to be scrolled (**ScrollCol**). Only this specified column can be scrolled in item focus mode.

  In cell focus mode, the content in the focused cell can be scrolled.

**NOTICE**

Do not use a translucent skin as the skin of the highlighted scrolling bar. Otherwise, the highlighted skin in scrolling mode is inconsistent with that in non-scrolling mode.

- Column attributes

  Each column of the list box corresponds to a field in the database, and the contents of each column are different. You need to specify the type of displayed content of each column (text, image, or text+image), column width, text alignment mode, text color, and field index of the list box.

- Internal focus skin

  To distinguish the focused region from the non-focused region, you need to specify the focus skin, including **rowselectskin** when the list box is focused and **rownormalskin** when the list box is not focused.

## 4.7.3 Special Events

- Internal focus change event onselect, which corresponds to HIGV_MSG_ITEM_SELECT and is triggered when the internal focus changes
- Focus select event oncellselect, which corresponds to HIGV_MSG_LISTBOX_CELLSEL and is triggered when a focus is selected
- Data change event ondatachange, which corresponds to HIGV_MSG_DATA_CHANGE and is triggered when data is synchronized

## 4.7.4 Cell Focus

You can set the cell focus mode by setting the XML attribute **iscellactive** to **yes** or by setting **IsCellActive** in the API. In cell focus mode, the width of each cell can be customized. You can register a cell width obtaining function GetWidthCb in cell focus mode as follows:

```
typedef HI_S32 (* HIGV_GET_WIDTH_CB)(HI_HANDLE hList,HI_U32 Item,HI_U32
Col)
```

This callback function is called when the list box data is synchronized. The number of invocation times equals the number of list box columns multiplied by that of the list box rows. Each time GetWidthCb is called, the user returns the expected cell width. The return value ranges from 0 to 100, indicating the percentage of the cell width relative to the row width. The

value 0 indicates that the cell is hidden, and the value 100 indicates that the cell occupies the entire row.

If GetWidthCb is not registered, the cell width of each column is the same, which is the one configured in the column attribute. If the return value does not fall within 0−100, the previous cell width is used.

When switching cell focus, the user uses the API for receiving key calling to set focus switching function in onkeydown event by default. Cells with the width of 0 cannot be displayed but actually exist. The focus should not be switched to a cell that cannot be displayed. You can also configure AutoSwitchCell to enable the HiGV to implement focus switching internally.

## 4.7.5 Small Icon in the Cell

The list box allows you to set a small icon as the flag in the cell. In many scenarios, a flag indicating that a cell in the list box has been selected is required, which can be implemented by using the small icon of the cell.

The small icon is a tiny image in the left or right part of the cell which does not affect the original data and line frame of the cell. The resource handles of these images are obtained from the ADM. You need to set the column type to **LIST_COLTYPE_TXTICONLEFT** or **LIST_COLTYPE_TXTICONRIGHT** and set **Fgidx** to *a*|*b* to indicate the ADM field index corresponding to the text and icon.

The following is a list box XML description instance:

```
<listbox
isrelease="yes"
id="test_listbox01"
top="20"
left="20"
width="200"
height="150"
normalskin="black_skin"
disableskin = ""
activeskin="button_default"
transparent="no"
iscellactive="no"
widgetposmirror="yes"
widgetinteriormirror="yes"
oncellselect="listbox_cellselect"
rownum="5"
colnum="2"
font="common_font_text_22"
datamodel="test_datamodel1"
rowselectskin="common_skin_combobox_row_select"
rownormalskin="common_skin_combobox_row_select"
leftorderobj=""
rightorderobj=""
onkeydown=""
```

```
hlineheight="1"
vlinewidth="1"
hlinecolor="0xFF000000"
vlinecolor="0xFF000000"
scrollbar=""
ontimer=""
noframe=""
ongetfocus=""
onlostfocus=""
onselect="">
<!—coldbindex field described by using text|icon -->
<listcol
id="test_listcol01"
coltype="texticonleft"
colwidth="100"
colalignment="hcenter|vcenter"
colbinddb="yes"
fgidx=""
coldbindex="0|1"
coldataconv=""/>
<!--coldbindex field described by using text|icon -->
<listcol
id="test_listcol02"
coltype="texticonright"
colwidth="100"
colalignment="hcenter|vcenter"
colbinddb="yes"
fgidx=""
coldbindex="2|3"
coldataconv=""/>
</listbox>

<!—Store the list box icon in the DB. The field for storing the image
resource IDs needs to be preconfigured in the DB.-->
<datamodel
   id="test_datamodel1"
   field="name:string:20;icon0:resid:4;num:u32:4;icon1:resid:4"
   datasource="owndb"
   getrowcount=""
   getrowvalue=""
   cacherows=""
   registerdatachange=""
   unregisterdatachange="" />
```

## 4.7.6 Column Callback Function

In the column attributes, you can register the callback function HIGV_LIST_CONV_CB ConvString, which corresponds to the **coldataconv** attribute in the XML file. This callback function is used to convert character strings. When the content of a cell in the list box is drawn, data stored in the DB is put into this callback function, converted, and then transferred as parameter values for display.

This callback function is called when each cell is drawn, and you can use this wisely.

```
typedef HI_S32 (*HIGV_LIST_CONV_CB)(HI_HANDLE hList, HI_U32 Col, HI_U32
Item, const HI_CHAR *pSrcStr, HI_CHAR *pDstStr, HI_U32 Length)
```

where

- **Col** and **Item** specify the position of the cell that calls the function.
- **pSrcStr** indicates the data corresponding to the current cell.
- **pDstStr** indicates the final displayed data of the list box.
- **Length** indicates the length of **pSrcStr**.

When all data in a column is the same image resource, you do not need to bind the column to an ADM field. Instead, you can directly set the attribute **hImage** of the structure HIGV_LIST_COLATTR_S because the column does not rely on the DB in this situation. The list box provides the following special processing:

- pDstStr[0]=1: Display the image.
- pDstStr[0]=0: Hide the image.

# 4.8 Scroll Bar

The scroll bar is bound to controls with the content that may exceed the display region. It is used to assist the control display and operation and show the position of the current displayed region in the entire region that needs to be displayed. The scroll bar cannot be focused. The scroll bar header file is **hi_gv_scrollbar.h**.

**Figure 4-12** Scroll bar instance


You need to set the up and down arrows, slider, and background skin for the scroll bar. The scroll bar has no special event. Note that the start offset position of the slider depends on the image on the upper left corner of the background skin. Therefore, ensure that the upper left corner of the background skin is consistent with the up arrow.

# 4.9 Spin

The spin is used to switch between a group of entries. The entries come from DB tables or views, or are added by users. When the spin is focused, you can switch the entry by pressing the left or right arrow key. The spin header file is **hi_gv_spin.h**.

**Figure 4-13** Spin instance


## 4.9.1 Data Source Types

There are three types of spins:

- Digital type

  The spin content is determined by configuring the start digit, step, and number of entries. For example, if the start digit is 3, the step is 2, and the number of entries is 4, the spin content is 3, 5, 7, and 9.

- Character string type

  You can add character strings for a spin by calling HI_GV_Spin_AddItem or add multi-languange character string IDs for a spin by calling HI_GV_Spin_AddItemByID. Note that the character strings and multi-languange character string IDs cannot be added at the same time. You can directly configure the character string IDs in the XML file.

- DB data type

  DB data can be obtained from the bound ADM.

## 4.9.2 Spin Arrows

The up and down arrows of the spin can be configured in the XML file or by calling HI_GV_Spin_SetButtonImg. You can switch the spin entry by clicking the arrows or scrolling the mouse wheel.

## 4.9.3 Special Event

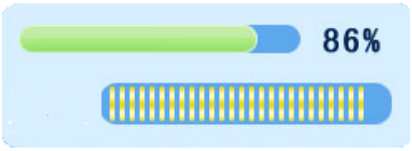
When the spin entry is switched, the onselect event is generated, corresponding to HIGV_MSG_ITEM_SELECT.

# 4.10 Progress Bar

The progress bar is used to show the progress, for example, the progress of file compression or program playback. The progress bar is filled with images. The progress bar header file is **hi_gv_progressbar.h**.

**Figure 4-14** Progress bar instance



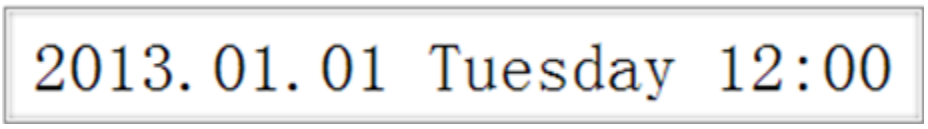

The progress bar can be horizontal or vertical (the configurations are similar). The background color of the progress bar is its own skin. Besides that, you need to configure the progress skin, step, and maximum and minimum values.

When the progress bar is focused, you can change the progress by pressing the left or right key. An event is generated when the progress changes, corresponding to HIGV_MSG_ITEM_SELECT.

# 4.11 Clock

The clock control is used to display the time and date. It supports the configuration of texts or data in a fixed format. The clock header file is **hi_gv_clock.h**.

**Figure 4-15** Clock instance




## 4.11.1 Internal Timer

Before the clock control is displayed, you need to specify the minimum unit for the clock by setting the attribute **minunit** in the XML file (if the clock is created by using the XML file) or calling HI_GV_Clock_SetTimeUnit after the clock is initialized by calling HI_GV_Clock_Init (if the clock is created by calling an API). If the minimum unit is second, the clock automatically creates a timer that is triggered every second; otherwise, it creates a timer that is triggered every minute.

The clock creates an internal timer when loading resources. That is:

- When the clock uses the HIGV_STYLE_HIDE_UNLODRES style, the timer is created when the clock is displayed and destroyed when the clock is hidden.
- When the clock does not use the HIGV_STYLE_HIDE_UNLODRES style, the internal timer is created when the clock is created and has the same lifecycle as the clock.

After the timer is successfully created, it is started cyclically and the HIGV_MSG_TIMER (ontimer) event is called back. The timer ID is 0x1001. The clock timer is started and stopped by calling HI_GV_Clock_Run. If the HIGV_STYLE_HIDE_UNLODRES style is not used, the clock timer is stopped first, and started when the clock control is displayed.

NOTICE

**minunit** must be configured. If it is not configured, the control does not automatically specify the timer update interval.

## 4.11.2 Clock Display

The display format must be specified when the clock is created. You can set the attribute **dispmode** in the XML file or **HIGV_CLOCK_MODE_E DispMode** in the API.

### Text Mode

If **DispMode** is **HIGV_CLOCK_MODE_TEXT**, you can set the clock timer by calling HI_GV_Widget_SetText or HI_GV_Widget_SetTextByID.

### Fixed Format Mode

If **DispMode** is **HIGV_CLOCK_MODE_FORMAT**, you need to set the fixed display format for the clock in the XML file. The display format is typically defined in the multi-language XML file. **timefmt** is the tag and the ID of **timefmt** is used as the text attribute of the control.

Do not set the text for the clock in fixed format by calling an API. Table 4-1 lists the clock display formats (that will be replaced) in **timefmt**.

**Table 4-1** Clock display formats

| Complementary Format | Default Format | Definition | Editable or Not | Separator |
|---|---|---|---|---|
| yyyy | yy | year | Yes | [ ] |
| MM | M | month | Yes | [ ] ,< > |
| dd | d | day | Yes | [ ] |
| dddd | ddd | week | No | < > |
| HH | H | hour/24-hour format | Yes | [ ] |
| hh | h | hour/12-hour format | Yes | [ ] |
| mm | m | minute | Yes | [ ] |
| ss | s | second | Yes | [ ] |
| tt | - | a.m./p.m. | No | < > |

The characters in the table will be replaced by other characters, which are distinguished by using square brackets ([]) and angle brackets (<>).

- The characters in square brackets ([]) are digits.
- The characters in angle brackets (<>) indicate an ID character set.

**[<strset>fmt]** indicates that the digits 0 to 9 in **fmt** will be replaced by the ID character set **strset** (which must be 10, corresponding to digits 0 to 9).

In the complementary format, complete character strings are displayed. Some unnecessary characters are not displayed in default mode. For example, year 2015 is displayed as **2015** in yyyy format but **15** in yy format, and day 9 is displayed as **09** in dd format but **9** in d format.

The following is an XML instance:

```
<language
    id = "language"
    languageinfo = "en:english;ar:arabic"
    locale = "en">
    <strset
        id = "STRSET_MONTH"
        MM =
"STRID_MM_JAN;STRID_MM_FEB;STRID_MM_MAR;STRID_MM_APR;STRID_MM_MAY;STRID_M
M_JUN;STRID_MM_JUL;STRID_MM_AUG;STRID_MM_SEP;STRID_MM_OCT;STRID_MM_NOV;ST
RID_MM_DEC;"
    />
    <strset
        id = "STRSET_SHORT_MONTH"
        M =
"STRID_MM_JAN;STRID_MM_FEB;STRID_MM_MAR;STRID_MM_APR;STRID_MM_MAY;STRID_M
M_JUN;STRID_MM_JUL;STRID_MM_AUG;STRID_MM_SEP;STRID_MM_OCT;STRID_MM_NOV;ST
RID_MM_DEC;"
    />

    <strset
        id = "STRSET_AMPM"
        tt = "STRID_AMPM_AM;STRID_AMPM_PM;"
    />
    <strset
        id = "STRSET_DIGIT"
        digit =
"STRSET_DIGIT_0 ;STRSET_DIGIT_1;STRSET_DIGIT_2;STRSET_DIGIT_3;STRSET_DIGI
T_4;STRSET_DIGIT_5;STRSET_DIGIT_6;STRSET_DIGIT_7;STRSET_DIGIT_8;STRSET_DI
GIT_9;"
    />

    <lanstr
        id = "STRSET_DIGIT_0"
        en = "0"
        ar = "٠"
    />
    <lanstr
        id = "STRSET_DIGIT_1"
        en = "1"
        ar = "١"
```

```
    />
    <lanstr
        id = "STRSET_DIGIT_2"
        en = "2"
        ar = "٢"
    />
    <lanstr
        id = "STRSET_DIGIT_3"
        en = "3"
        ar = "٣"
    />
    <lanstr
        id = "STRSET_DIGIT_4"
        en = "4"
        ar = "٤"
    />
    <lanstr
        id = "STRSET_DIGIT_5"
        en = "5"
        ar = "٥"
    />
    <lanstr
        id = "STRSET_DIGIT_6"
        en = "6"
        ar = "٦"
    />
    <lanstr
        id = "STRSET_DIGIT_7"
        en = "7"
        ar = "٧"
    />
    <lanstr
        id = "STRSET_DIGIT_8"
        en = "8"
        ar = "٨"
    />
    <lanstr
        id = "STRSET_DIGIT_9"
        en = "9"
        ar = "٩"
    />

    <lanstr
        id = "STRID_AMPM_AM"
```

```
    en = "AM."
    ar = "ص"
/>

<lanstr
    id = "STRID_AMPM_PM"
    en = "PM."
    ar = "م"
/>

<lanstr
    id = "STRID_MM_JAN"
    en = "January"
    />

<lanstr
    id = "STRID_MM_FEB"
    en = "February"
    />

<lanstr
    id = "STRID_MM_MAR"
    en = "March"
    />

<lanstr
    id = "STRID_MM_APR"
    en = "April"
    />
<lanstr
    id = "STRID_MM_MAY"
    en = "May"
    />

<lanstr
    id = "STRID_MM_JUN"
    en = "June"
    />

<lanstr
    id = "STRID_MM_JUL"
    en = "July"
    />
```

```
<lanstr
    id = "STRID_MM_AUG"
    en = "August"
    />

<lanstr
    id = "STRID_MM_SEP"
    en = "September"
    />
<lanstr
    id = "STRID_MM_OCT"
    en = "October"
    />
<lanstr
    id = "STRID_MM_NOV"
    en = "November"
    />

<lanstr
    id = "STRID_MM_DEC"
    en = "sunday"
    />

<lanstr
    id = "STRID_WEEK_SUN"
    en = "sunday"
    ar = "الأحـد يـوم"
    />

<lanstr
    id = "STRID_WEEK_MON"
    en = "monday"
    ar = "الاثـنين يـوم"
    />

<lanstr
    id = "STRID_WEEK_TUE"
    en = "Tuesday"
    ar = "الـثـلاثـاء يـوم"
    />

<lanstr
    id = "STRID_WEEK_WED"
    en = "wednesday"
```

```
        ar = "الأربعاء يوم"
        />

    <lanstr
        id = "STRID_WEEK_THUR"
        en = "thursday"
        ar = "الخميس يوم"
        />

    <lanstr
        id = "STRID_WEEK_FRI"
        en = "friday"
        ar = "الجمعة يوم"
        />

    <lanstr
        id = "STRID_WEEK_SAT"
        en = "saturday"
        ar = "السبت يوم"
        />

    <lanstr
        id = "STRID_WEEK_SHORT_SUN"
        en = "sun."
        ar = "الأحد يوم"
        />

    <lanstr
        id = "STRID_WEEK_SHORT_MON"
        en = "mon."
        ar = "الاثنين يوم"
        />

    <lanstr
        id = "STRID_WEEK_SHORT_TUE"
        en = "Tues."
        ar = "الثلاثاء يوم"
        />

    <lanstr
        id = "STRID_WEEK_SHORT_WED"
        en = "wed."
        ar = "الأربعاء يوم"
        />
```

```
    <lanstr
        id = "STRID_WEEK_SHORT_THUR"
        en = "thu."
        ar = "الخميس يوم"
        />

    <lanstr
        id = "STRID_WEEK_SHORT_FRI"
        en = "fri."
        ar = "الجمعة يوم"
        />

    <lanstr
        id = "STRID_WEEK_SHORT_SAT"
        en = "sat."
        ar = "السبت يوم"
        />
    <strset
        id = "STRSET_SHORT_WEEK"
        ddd =
"STRID_WEEK_SHORT_SUN;STRID_WEEK_SHORT_MON;STRID_WEEK_SHORT_TUE;STRID_WEEK_SHORT_WED;STRID_WEEK_SHORT_THUR;STRID_WEEK_SHORT_FRI;STRID_WEEK_SHORT_SAT;"/>

    <strset
        id = "STRSET_WEEK"
        dddd =
"STRID_WEEK_SUN;STRID_WEEK_MON;STRID_WEEK_TUE;STRID_WEEK_WED;STRID_WEEK_THUR;STRID_WEEK_FRI;STRID_WEEK_SAT;"/>

    <timefmt
        id = "TIMEFMT_DATE"
        en = "[yyyy]-<MM>-[dd] week: <dddd>"
        ar = "<dddd>,[dd] <MM>,[yyyy]"
        />
    <timefmt
        id = "TIMEFMT_TIME"
        en = "[HH]:[mm]:[s]"
        ar = "<tt> [hh]:[mm]:[s]"
        />


    <timefmt
```

```
        id = "TIMEFMT_DATETIME"
        en = "[yyyy].<MM>.[dd] week: <dddd>  [HH]:[mm]:[s]"
        ar = "<dddd>,[dd] <MM>,[yyyy] <tt> [h]:[mm]:[s]"
        />

    <timefmt
        id = "TIMEFMT_SHORT_TIME"
        en = "<tt> [h]:[m]:[s]"
        ar = "<tt> [h]:[m]:[s]"
        />

    <timefmt
        id = "TIMEFMT_SHORT_DATE"
        en = "[yy]-<M>-[d] week: <ddd>"
        ar = "<ddd>,[d] <M>,[yy]"
        />
    <timefmt
        id = "TIMEFMT_SHORT_DATETIME"
        en = "[yy].<M>.[d] [H]:[m]:[s]"
        ar = "<ddd>,[d] <M>,[yy] <tt> [h]:[m]:[s]"
        />
</language>
```

## Time Format TIMEFMT_DATETIME

```
    <timefmt
        id = "TIMEFMT_DATETIME"
        en = "[yyyy].<MM>.[dd] week: <dddd>  [HH]:[mm]:[s]"
        ar = "<dddd>,[dd] <MM>,[yyyy] <tt> [h]:[mm]:[s]"
        />
```

where

- **[yyyy]** indicates the year in complementary format, such as 2015.
- **<MM>** indicates a month from January to December. Because the angle brackets (<>) are used, the MM text in **strset** is displayed.

  ```
      <strset
          id = "STRSET_MONTH"
          MM =
  "STRID_MM_JAN;STRID_MM_FEB;STRID_MM_MAR;STRID_MM_APR;STRID_MM_MAY;STRID_MM_JUN;STRID_MM_JUL;STRID_MM_AUG;STRID_MM_SEP;STRID_MM_OCT;STRID_MM_NOV;STRID_MM_DEC;"
      />
  ```

  The strset ID is STRSET_MONTH. This ID is not used. The xml2bin tool automatically searches for the MM data to replace the content in the angle brackets (<>). In different language environments, <MM> is translated into multi-language character strings for the

12 months. For example, for **STRID_MM_JAN**, **January** is found in the en environment, and **January** is also found as the default value for the ar environment.

- **[dd]** indicates the date in a month.
- **<dddd>** indicates a day in a week. The character strings under the language environment corresponding to the multi-language character string ID are automatically searched. Multiple multi-language character string IDs are separated by using semicolons (;) in **strset**.

```
    <strset
        id = "STRSET_WEEK"
        dddd =
"STRID_WEEK_SUN;STRID_WEEK_MON;STRID_WEEK_TUE;STRID_WEEK_WED;STRID_WE
EK_THUR;STRID_WEEK_FRI;STRID_WEEK_SAT;"/>
```

- **[HH]** indicates the hour in the 24-hour format.
- **[hh]** indicates the hour in the 12-hour format.
- **[mm]** indicates the minute ranging from 0 to 59 (00 to 59 in the complementary format).
- **[s]** indicates the second ranging from 0 to 59 (0 to 59 by default).
- **<tt>** indicates AM/PM. The multi-language character set tt is automatically searched.
- **[<digit>yyyy]** indicates the year in complementary format. The character string for the digits in the corresponding language environment is automatically searched.

Assume that the current time is 15:40:05 on April 7th, 2015, Tuesday:

- The display format in the en environment is "[yyyy].<MM>.[dd] week: <dddd> [HH]:[mm]:[s]", which is displayed as "2015. April.07 week: Tuesday 15:40: 5".
- The display format in the ar environment is "<dddd>,[dd] <MM>,[yyyy] <tt> [h]:[mm]:[s]", which is displayed as "الـ ثلاثـ اء يـ وم,07 April,2015 م 3:40:5".

## 4.11.3 Time Configuration and Modification

The event function registered by the user is called back when the clock internal timer is started cyclically. Typically the current time is configured in the following callback events:

- ontimer (corresponding to HIGV_MSG_TIMER)
- ontimeupdate (corresponding to HIGV_MSG_CLOCK_UPDATE)

Both callback events can be used as update events. Their differences are as follows:

- The parameters for the ontimer event are the timer ID (0x1001) and 0.
- The parameters for the ontimeupdate event are the current system time and 0.

The system time is the HIGV_TIME_S pointer that is forcibly converted into the HI_U32 type.

In text mode, you need only to set the text for the clock; in fixed format mode, you need to convert the preconfigured time and date into time_t in the callback function, and update the clock by calling HI_GV_Clock_SetUTC.

In addition to updating the clock in the callback event, you can also adjust the time by using keys. When the clock is focused, pressing the **OK** key enters the editing mode. In editing mode, you can modify the digit where the cursor locates by pressing the up and down arrow keys and change the cursor position by pressing the left and right arrow keys. The week and

AM/PM are automatically calculated by the system and cannot be edited. After editing, pressing the **OK** key again exits the editing mode. You can then switch the focus.

### 4.11.4 Private Attributes

- Display configuration

  The minimum time unit and display mode need to be configured. For the fixed format mode, you also need to set the time format in the XML file.

- Internal focus configuration

  The internal focus is used to indicate the current editing position of the cursor. You can specify a color block or an image resource. The color value or image resource path also needs to be specified based on the internal focus type.

### 4.11.5 Special Events

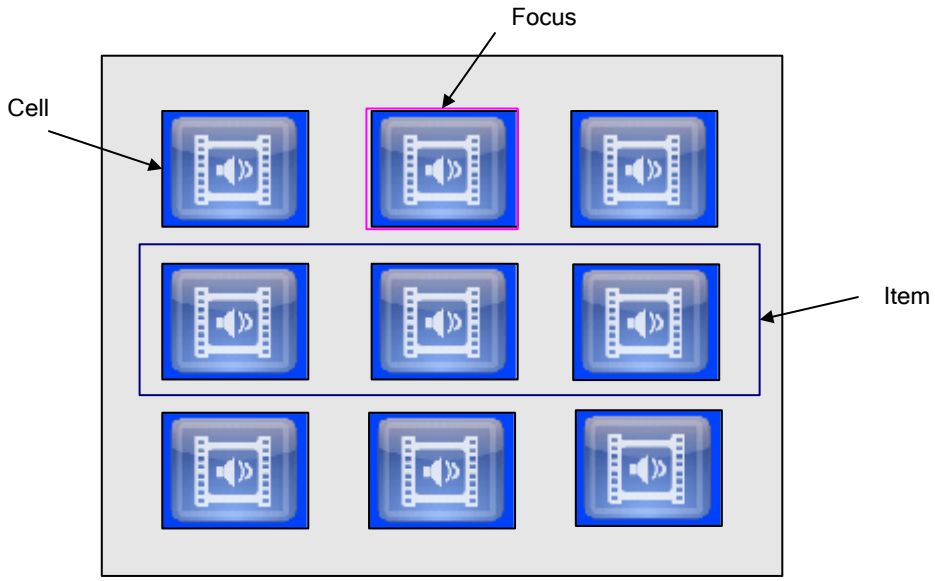
- onfocusmove event, which corresponds to HIGV_MSG_CLOCK_FOCUS_MOVE and is triggered when the internal focus changes
- ontimeadjust event, which corresponds to HIGV_MSG_CLOCK_TIME_ADJUST and is triggered when the data of the internal focus is modified
- ontimeupdate event, which corresponds to HIGV_MSG_CLOCK_UPDATE and is triggered cyclically
- onposition event, which corresponds to HIGV_MSG_POSITION and reports the position information after the focus is switched

## 4.12 Scroll Grid

The scroll grid is a grid control with the 2D scrolling effect. Data of this control comes from the data model. The scroll grid displays data more vividly, but also consumes a lot of resources. The scroll grid header file is **hi_gv_scrollgrid.h**. This control is complex and you need to register it by calling HI_GV_ScrollGrid_RegisterWidget() before using it.

**Figure 4-16** Scroll grid instance




The scroll grid provides functions similar to those of the list box, except that the scroll grid shows the move of the internal focus with the 2D animation effect. Therefore, more attributes need to be configured for the scroll grid to describe the control appearance.

## 4.12.1 Private Attributes

- Grid page
  - **RowNum** and **ColNum**: row quantity and column quantity on a page. Both of them are 3 in Figure 4-16.
  - **CellColNum**: number of columns
  - **LeftMargin**, **RightMargin**, **TopMargin**, and **BtmMargin**: left, right, top, and bottom margins for restricting the drawing region
  - **RowSpace** and **ColSpace**: row spacing and column spacing, which determine the spacing between cells
- Internal focus
  - **FocusRectAnchor**: position of the focus anchor when the focus move event is triggered
  - **FocusRectSkin**: skin of the focus when the control is focused
  - **FocusRectNormSkin**: skin of the focus when the control is not focused
- Scrolling
  - **ItemStep**: item scrolling step
  - **ItemInterval**: item scrolling interval
  - **FocusRectStep**: focus scrolling step
  - **FocusRectInterval**: focus scrolling interval
- Grid column
  - **Coltype**: column type, image or text

- **Colindex**: column index
- **ColTop**: vertical offset of the content relative to the cell
- **ColLeft**: horizontal offset of the content relative to the cell
- **ColWidth**: column width
- **ColHeight**: column height
- **Colalignment**: column alignment mode
- **Colbinddb**: whether the content comes from the ADM
- **Coldbindex**: ADM field index
- **Colimage**: image when the data does not come from the index and the image type is used

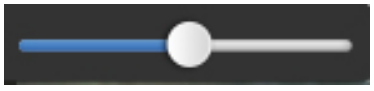
## 4.12.2 Special Events

- Focus switch event onfocusmove, which corresponds to HIGV_MSG_ITEM_SELECT
- Cell attribute column select event oncellcolselect, which corresponds to HIGV_MSG_SCROLLGRID_CELL_COLSEL
- Focus reaching top edge event onfocusreachtop, which corresponds to HIGV_MSG_SCROLLGRID_REACH_TOP
- Focus reaching bottom edge event onfocusreachbtm, which corresponds to HIGV_MSG_SCROLLGRID_REACH_BTM
- Focus reaching left edge event onfocusreachleft, which corresponds to HIGV_MSG_SCROLLGRID_REACH_LEFT
- Focus reaching right edge event onfocusreachright, which corresponds to HIGV_MSG_SCROLLGRID_REACH_RIGHT
- Data status change event ondatachange, which corresponds to HIGV_MSG_DATA_CHANGE
- Dynamic image decoding start event ondecimgstart, which corresponds to HIGV_MSG_SCROLLGRID_DECIMG_BEGIN
- Dynamic image decoding end event ondecimgfinish, which corresponds to HIGV_MSG_SCROLLGRID_DECIMG_FINISH
- Fling gesture end event onfinishfling, which corresponds to HIGV_MSG_SCROLLGRID_FLING_FINISH

# 4.13 Track Bar

The track bar allows you to select from a range of values by moving a slider. The track bar header file is **hi_gv_trackbar.h**. The track bar is not used frequently. You need to register the track bar by calling HI_GV_Track_RegisterWidget before using it.

**Figure 4-17** Track bar instance



The track bar is typically operated by using the mouse. You can click and drag the slider by using the mouse or pressing the left and right keys to control the slider.

### 4.13.1 Private Attributes

- Style: sliding style, vertical or horizontal
- Slide block
  - **Normaltrack**: image resource for the slide block in normal state
  - **Activetrack**: image resource for the slide block in activated state
  - **Mousedowntrack**: image resource for the slide block when it is being clicked
- Slider
  - **Slider**: background image resource for the slider
  - **Margin**: slider margins for specifying the drawing region of the slider
- Sliding
  - **CurValue**: current scale
  - **MaxValue**: maximum scale
  - **MinValue**: minimum scale
  - **Step**: sliding step

### 4.13.2 Special Event

The special event for the track bar is the scale change event onvaluechange, which corresponds to HIGV_MSG_VALUEONCHANGE.

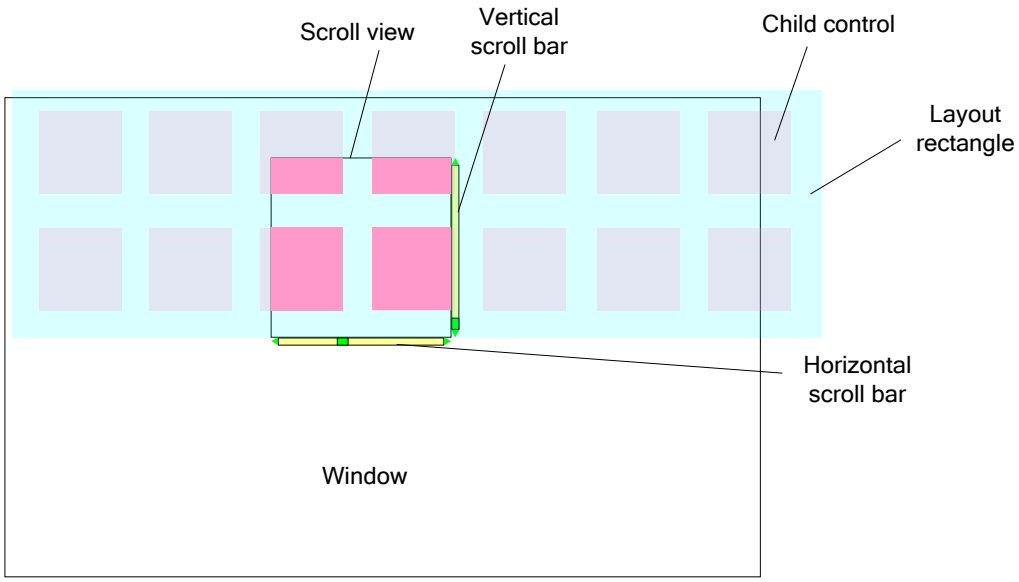
## 4.14 Scroll View

The scroll view is a container control that can house other controls except the window. The scroll view header file is **hi_gv_scrollview.h**. It has the following features:

- Allows the child control layout to exceed the view.
- Supports binding to the vertical scroll bar.
- Supports binding to the horizontal scroll bar.
- Controls the vertical scroll bar by using the mouse wheel.
- Automatically moves when the focus is switched to a child control that is not displayed completely.
- Notifies the view move event.

**Figure 4-18** Scroll view instance




## 4.14.1 Private Attributes

- View (visible region)
  - **Margin**: top, bottom, left, and right margins for specifying the region of the view for displaying the child controls in the scroll view background
  - **ScrollViewStyle**: view style, including:
    - **SCROLLVIEW_STYLE_DEFAULTE**: default style, ensuring that the child control is completely displayed in the view when the focus is switched
    - **SCROLLVIEW_STYLE_ALWAY_CENTER**: center style, ensuring that the child control is always in the center of the view when the focus is switched
- Total range of child controls
  - **ScrollContentWidth**: maximum width of the total range of child controls. If it is 0, the width is not restricted.
  - **ScrollContentHeight**: maximum height of the total range of child controls. If it is 0, the height is not restricted.
- Scrolling
  - **Step**: View scrolling step. Currently smooth scrolling is not supported. Refresh is performed only once.
  - **Interval**: View scrolling interval. Currently smooth scrolling is not supported. Refresh is performed only once.
  - **hVerticalScrollBar**: bound vertical scroll bar
  - **hHorizontalScrollBar**: bound horizontal scroll bar
  - **direction**: scroll direction of the control

## 4.14.2 Special Event

- The special event for the scroll view is the view move event onviewmove, which corresponds to HIGV_MSG_SCROLLVIEW_SCROLL.

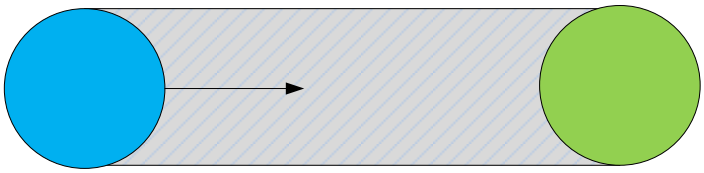
  

- The fling gesture end event onfinishfling for the ScrollView controll corresponds to HIGV_MSG_SCROLLVIEW_FINISHFLING.

# 4.15 Slide Unlock Control

SlideUnlock is a control that unlocks the screen by moving the slide block. The screen can be unlocked by dragging the slide block from the start point to the end point. If the slide block is released during the dragging process, the slide block returns to the original position with animation effect.

**Figure 4-19** SlideUnlock instance



## 4.15.2 Private Attributes

- **Style**: sliding style, vertical or horizontal
- Slide block
  - **Normalthumb**: image resource for the slide block in initial state
  - **Touchthumb**: image resource for the slide block in activated state
  - **Donethumb**: image resource for the slide block in completion state
- Slide bar
  - **Slider**: background image resource for the slide bar
  - **Margin**: margins of the slide bar for specifying the drawing region of the slide bar

## 4.15.3 Special Events

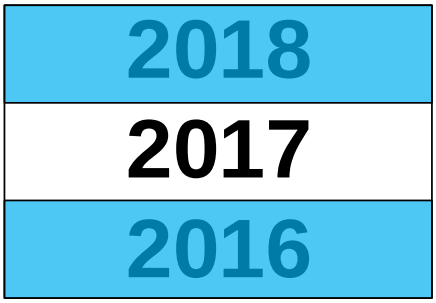
- The special event for the SlideUnlock control is the slide unlock completion event onunlock, which corresponds to HIGV_MSG_UNLOCK.
- The special event for the SlideUnlock control is the slide movement event onmove, which corresponds to HIGV_MSG_MOVE.
- The special event for the SlideUnlock control is the slide back-to-origin event onkickback, which corresponds to HIGV_MSG_KICKBACK.

# 4.16 Wheel View Control

The WheelView control is a scrolling selector, which is used to switch focus by sliding up or down. The middle row is the focus row. The upper and lower texts are covered with semitransparent skins for you to slide and choose.

**Figure 4-20** WheelView diagram




## 4.16.2 Private Attributes

- **Rownum**: number of displayed rows
- **Datamodel**: handle to data model
- **Upcoverskin**: handle to the upper cover text skin
- **Downcoverskin**: handle to the lower cover text skin
- **Iscirclescroll**: whether to scroll the text cyclically

## 4.16.3 Special Events

The special event for the WheelView control is a reported event of the focus row onfocusSelect after sliding, which corresponds to HIGV_MSG_WHEELVIEW_ITEM.

# 4.17 Image View Control

The ImageView control is used to play the big image data of Multi-Picture Format (MPF) images or play 4K and 8K images.

## 4.17.1 Private Attributes

- **Imagefile**: directory of JPEG images
- **Imageindex**: subimage index of an image file
- **Vouttype**: image output mode
- **Disphandle**: handle to the display
- **Wndhandle**: handle to the window

## 4.17.2 Special Events

If an internal error occurs when an image is completely displayed or being displayed, the HI_GV_IMAGEVIEW_ON_EVENT_FN function is called back.

# 5 HiGV Programming

## 5.1 Customized Drawing

The HiGV provides a set of customized APIs for drawing graphics, which facilitates special graphics customization based on existing controls. These APIs provide functions such as drawing lines, filling the background, drawing texts, and decoding images.

### 5.1.1 Setup of the Drawing Environment

**NOTE**

The controls required for building a temporary drawing environment are referred to as dependent controls in the following sections.

Customized drawing is implemented as follows: create a temporary drawing environment based on the control, and then draw on the surface of the control by calling the HiGV-encapsulated drawing API.

Before customized drawing, create a temporary drawing environment by calling HI_GV_GraphicContext_Create(HI_HANDLE hWidget, HI_HANDLE *phGC). A handle for the temporary drawing environment (phGC) is obtained based on the control handle.

If the temporary drawing environment is not required any more, destroy it by calling HI_GV_GraphicContext_Destroy(HI_HANDLE hGC) to prevent any memory leak.

### 5.1.2 Customized Drawing APIs

- HI_GV_GraphicContext_DecodeImg or HI_GV_GraphicContext_DecodeMemImg: Decodes the specified memory image or file into the graphical surface that can be identified by the HiGV and HiGo. This API does not rely on the temporary drawing environment and can be widely used in graphics services.
- HI_GV_GraphicContext_FreeImageSurface: Releases the decoded image surface. The decoded image must be released in time if it is not used to prevent any memory leak.
- HI_GV_GraphicContext_SetFgColor: Configures the foreground color for the temporary drawing environment. The foreground color takes effect on lines and texts. The default value is the foreground color of the dependent control.
- HI_GV_GraphicContext_SetBgColor: Configures the background color for the temporary drawing environment. The background color takes effect on the background. The default value is the background color of the dependent control.

- HI_GV_GraphicContext_DrawLine: Draws lines. It can be used to draw foreground color lines.
- HI_GV_GraphicContext_DrawImage: Draws images. It can be used to transfer the image resource ID or decoded image surface to the surface of the temporary drawing environment.
- HI_GV_GraphicContext_DrawText or HI_GV_GraphicContext_DrawTextByID: Draws texts. It can be used to draw character strings, and the color of the character strings is the foreground color.
- HI_GV_GraphicContext_SetFont: Configures the text font.
- HI_GV_GraphicContext_FillRect: Fills the background for a region.

## 5.1.3 Customized Drawing Process

### Process

The customized drawing process is as follows:

**Step 1** Start drawing by calling HI_GV_GraphicContext_Begin.

**Step 2** Add a clip rectangle for the drawing environment. The rectangle determines the drawing region.

**Step 3** Draw graphics by calling the drawing APIs.

**Step 4** Stop drawing by calling HI_GV_GraphicContext_End.

**----End**

### Drawing Timing

For customized drawing based on controls, the display/hide state of the controls and the overlapping relationship between controls must be considered. If the related operations and management are all implemented by the application layer, customized drawing will be complicated.

The best timing for customized drawing is when the control drawing state and internal data have been configured internally and before the control itself is drawn or right after the control is drawn but the drawing state data is not cleared.

HIGV_MSG_PAINT is a control drawing event, which has been introduced in the preceding sections. You can register an event callback function by calling HI_GV_Widget_SetMsgProc and implement customized drawing in the callback function.

- If the priority of the callback function is HIGV_PROCORDER_BEFORE, the callback function can affect the drawing of the control itself before the control drawing event.
  - If the return value is HIGV_PROC_GOON, the control drawing continues.
  - If the return value is HIGV_PROC_STOP, the control drawing is interrupted.

  At this time, the clip rectangle is the drawing region of the control by default, and configuring the clip rectangle changes the drawing region of the control.
- If the priority of the callback function is HIGV_PROCORDER_AFTER, the callback function does not affect the drawing of the control itself after the control drawing event. At this time, the clip rectangle is the drawing region of the control by default.

Customized drawing is more secure and accurate if it is performed after the control is drawn. The clip rectangle does not need to be configured because its drawing region is the same as that of the control.

## 5.1.4 Reference Code

```
static HI_HANDLE s_ImgHandle = 0;
static HI_HANDLE s_WinGC = 0;

staitc HI_S32 TestPaintProc(HI_HANDLE hWidget, HI_U32 wParam, HI_U32
lParam)
{
    HI_S32 Ret;
    HI_RECT Rect = {50,150,207,100};

    HI_GV_GraphicContext_Begin(s_WinGC);
    HI_GV_GraphicContext_DrawImage(s_WinGC, &Rect, s_ImgHandle, 0, 0);
    HI_GV_GraphicContext_SetFgColor(s_WinGC, 0xFFFFF000);
    Rect.x += 120;
    HI_GV_GraphicContext_DrawText(s_WinGC, "Test graphiccontext.", &Rect,
0);

    HI_GV_GraphicContext_DrawLine(s_WinGC, 5, 5, 150, 5);
    HI_GV_GraphicContext_DrawLine(s_WinGC, 5, 6, 150, 6);
    HI_GV_GraphicContext_DrawLine(s_WinGC, 5, 7, 150, 7);
    HI_GV_GraphicContext_End(s_WinGC);

    return HI_SUCCESS;
}

HI_S32 main(HI_S32 argc, HI_CHAR *argv[])
{
    HI_GV_Init();
    ...
    ...

    Ret = HI_GV_GraphicContext_Create(TestWin, &s_WinGC);
    if(HI_SUCCESS == Ret)
    {
        Ret = HI_GV_GraphicContext_DecodeImg("./res/test.png", 0,0,
&s_ImgHandle);
        if(HI_SUCCESS != Ret)
        {
            HI_GV_GraphicContext_Destroy(s_WinGC);
        }
```

```
    }

    HI_GV_Widget_SetMsgProc(TestWin, HIGV_MSG_PAINT, TestPaintProc,
HIGV_PROCORDER_AFTER);
    ...
    HI_GV_App_Start(Gui_App);
    ...
    if(s_WinGC)
    {
        HI_GV_GraphicContext_Destroy(s_WinGC);
    }

    if(s_ImgHandle)
    {
        HI_GV_GraphicContext_FreeImageSurface(s_ImgHandle);
    }
    ...
    return HI_SUCCESS;
}
```

# 5.2 Input Device Adaptation

There are many input devices for man-machine interaction, including the remote controls, panel buttons, and mouse devices. The HiGV provides an input device management framework, which easily integrates various input devices.

## 5.2.1 Input Adaptation Event

The HiGV implements the development framework of input devices. If you want to customize an input device, you must implement the following interfaces and register the customized input device with the system. When the system starts, information about the customized device is sent to the GUI system.

```
/** Input devece structure*//** CNcomment: Input device structure*/
typedef struct hiHIGV_INPUT_S
{
    HI_S32 (*InitInputDevice)(HI_VOID);
    HI_S32 (*DeinitInputDevice)(HI_VOID);
    /** timeout is us */
    HI_S32 (*GetKeyEvent)(HIGV_INPUTEVENT_S *pInputEvent, HI_U32 TimeOut);
    /** timeout is us */
HI_S32 (*GetMouseEvent)(HIGV_INPUTEVENT_S *pInputEvent, HI_U32 TimeOut);
    /** timeout is us */
    HI_S32 (*GetTouchEvent)(HIGV_TOUCH_GESTURE_EVENT_S* pInputEvent,
HI_U32 TimeOut);
```

```
}HIGV_INPUT_S;
```

- InitInputDevice: Points to the input device initialization interface implemented by the user.
- DeinitInputDevice: Points to the input device deinitialization interface implemented by the user.
- GetKeyEvent: Points to the key value reception event interface implemented by the user.
- GetMouseEvent: Points to the mouse reception event interface implemented by the user.
- GetTouchEvent: Points to the touch reception event interface implemented by the user.

The HiGV initializes device management by calling HI_GV_InitInputSuite(HIGV_INPUT_S* pInputDevice). The user implements this API at the application layer and points the members of pInputDevice to the corresponding functions in the API.

## 5.2.2 Reference Code for Key Adaptation

```
HI_S32 SX_InitInputDevice(HI_VOID)
{
    HI_S32 Ret = HI_SUCCESS;
    Ret = HI_UNF_IR_Open();
    if (HI_SUCCESS != Ret)
    {
        IM_ERROR(HI_ERR_IM_OPENIRDEVICE, "Open Ir Device Failed.\n");
        return HI_ERR_IM_OPENIRDEVICE;
    }

    Ret = HI_UNF_IR_SetCodeType(HI_UNF_IR_CODE_NEC_SIMPLE);
    if (HI_SUCCESS != Ret)
    {
        HI_UNF_IR_Close();
        IM_ERROR(HI_ERR_IM_OPENIRDEVICE, "HI_UNF_IR_SetCodeType
Failed.\n");
        return HI_ERR_IM_OPENIRDEVICE;
    }

    Ret = HI_UNF_IR_EnableKeyUp(HI_TRUE);
    if (HI_SUCCESS != Ret)
    {
        HI_UNF_IR_Close();
        IM_ERROR(HI_ERR_IM_OPENIRDEVICE, "HI_UNF_IR_EnableKeyUp
Failed.\n");
        return HI_ERR_IM_OPENIRDEVICE;
    }

    Ret = HI_UNF_IR_SetRepKeyTimeoutAttr(108);
    if (HI_SUCCESS != Ret)
```

```
    {
        HI_UNF_IR_Close();
        IM_ERROR(HI_ERR_IM_OPENIRDEVICE, "HI_UNF_IR_SetRepKeyTimeoutAttr
Failed.\n");
        return HI_ERR_IM_OPENIRDEVICE;
    }

    Ret = HI_UNF_IR_EnableRepKey(HI_TRUE);
    if (HI_SUCCESS != Ret)
    {
        HI_UNF_IR_Close();
        IM_ERROR(HI_ERR_IM_OPENIRDEVICE, "HI_UNF_IR_EnableRepKey
Failed.\n");
        return HI_ERR_IM_OPENIRDEVICE;
    }

    Ret = HI_UNF_IR_Enable(HI_TRUE);
    if (HI_SUCCESS != Ret)
    {
        HI_UNF_IR_Close();
        IM_ERROR(HI_ERR_IM_OPENIRDEVICE, "HI_UNF_IR_Enable Failed.\n");
        return HI_ERR_IM_OPENIRDEVICE;
    }
    printf("SX_InitInputDevice success.\n");
    return Ret;
}

HI_S32 SX_DeinitInputDevice(HI_VOID)
{
    HI_UNF_IR_Close();
    return HI_SUCCESS;
}
HI_S32 SX_GetKeyEvent(HIGV_INPUTEVENT_S *pInputEvent, HI_U32 TimeOut)
{
    HI_S32 Ret = HI_FAILURE;
    HI_U32 KeyId[2] = {0};
    HI_U32 PressStatus = 0;
    HI_S32 Index;

    Ret = HI_UNF_IR_GetValue(&PressStatus, KeyId);
    if (HI_SUCCESS == Ret)
    {
        Ret = IM_GetIrdaMapItem ((HI_S32)KeyId[0], &Index);
        if (HI_SUCCESS == Ret)
```

```
        {
            pInputEvent->value  = s_PanIRMap_Key[Index].CommonValue;
            pInputEvent->msg = (PressStatus == 0) ? HIGV_MSG_KEYUP :
HIGV_MSG_KEYDOWN;
            return HI_SUCCESS;
        }
        else
        {
            IM_DEBUG("Don't match vkey.\n");
            return HI_ERR_IM_NOVIRKEY;
        }
    }
    else
    {
        IM_DEBUG("HI_UNF_IR_GetValue Ret:%x\n", Ret);
        return HI_ERR_EMPTY;
    }
}

HI_S32 HI_GV_InitInputSuite(HIGV_INPUT_S *pInputDevice)
{
    pInputDevice->InitInputDevice = SX_InitInputDevice;
    pInputDevice->DeinitInputDevice = SX_DeinitInputDevice;
    pInputDevice->GetKeyEvent = SX_GetKeyEvent;

    return HI_SUCCESS;
}
```

## 5.2.3 Reference Code for Mouse Adaptation

The mouse is not a HiGV control, and it is drawn by the HiGo directly. Besides input adaptation, you also need to configure the sources of the mouse images and the graphics layer to which the mouse belongs.

```
HI_S32 main(HI_S32 argc, HI_CHAR *argv[])
{
    …

    Ret = HI_GV_PARSER_LoadViewById(allwidget);
    if (HI_SUCCESS != Ret)
    {
        printf("HI_GV_PARSER_LoadViewById fail Ret:%x\n", Ret);
        HI_GV_Deinit();
        HI_GV_PARSER_Deinit();
        return -1;
```

```
    }

#ifdef MOUSE_SUPPORT

    HIGV_CURSORINFO_S CursorInfo;
    Ret = HI_GV_GraphicContext_DecodeImg ("./res/image/cursor.png",  0, 0,
&CursorInfo.hImage);
    assert(Ret == HI_SUCCESS);

    CursorInfo.HotspotX = 0;
    CursorInfo.HotspotY = 0;

    HI_GO_EnableSurfaceColorKey(CursorInfo.hImage, HI_TRUE);
    Ret = HI_GO_SetSurfaceColorKey(CursorInfo.hImage, 0x00);
    assert(Ret == HI_SUCCESS);

    Ret = HI_GV_Cursor_SetImage(&CursorInfo);
    assert(Ret == HI_SUCCESS);

    Ret = HI_GV_Cursor_AttchLayer(HIGO_LAYER_SD_0);
#endif

…
}

#define MOUSE_READ_BUFLEN 256
#define MOUSE_PACKDET_LEN 4

#define PACKET_IS_NOT_SYNC(data)       ((data) & 0x08)
#define PACKET_GET_BUTTONS(data)       ((data) & 0x07)
#define PACKET_GET_DELTAX(data0, data1) (((data0) & 0x10) ? (data1) -
256  :  (data1))
#define PACKET_GET_DELTAY(data0, data1) (((data0) & 0x20) ? -((data1) -
256) :  -(data1))
#define PACKET_GET_DELTAZ(data)        ((HI_S8)(((data) & 0x80) ? (data)
| 0xf0 :  (data) & 0x0F))


static HI_VOID UpdateMouseEvent(HIGV_MOUSE_EVENT_S* pEvent,
                                HI_U8 ChangedButtons, HI_S32 Buttons)
{
    if ( ChangedButtons & 0x01 )
    {
        pEvent->type = (Buttons & 0x01) ?
```

```
            HIGV_MOUSE_DOWN: HIGV_MOUSE_UP;
        pEvent->Buttons |= HIGV_MOUSE_B_LEFT;
    }

    if ( ChangedButtons & 0x02 )
    {
        pEvent->type = (Buttons & 0x02) ?
            HIGV_MOUSE_DOWN : HIGV_MOUSE_UP;
        pEvent->Buttons |= HIGV_MOUSE_B_RIGHT;
    }

    if ( ChangedButtons & 0x04 )
    {
        pEvent->type = (Buttons & 0x04) ?
            HIGV_MOUSE_DOWN : HIGV_MOUSE_UP;
        pEvent->Buttons |= HIGV_MOUSE_B_MIDDLE;
    }
}

static HI_S32 GetEvent(HIGV_MOUSE_EVENT_S* pEvent)
{
    HIGV_MOUSE_DATA_S *pMouseData  = &s_MouseData;
    struct timeval CurButtonDownTime = {0};
    static struct timeval s_LastButtonDownTime = {0};
    static HI_U8 LastButtons = 0;
    static HI_U32 s_LastButton = 0;
    HI_U8 PacketBuf[MOUSE_PACKDET_LEN] = {0}, ReadBuffer[MOUSE_READ_BUFLEN]
= {0};
    HI_U8 index = 0;
    HI_S32 ReadLength = 0;
    HI_S32 interval = 0;
    HI_S32 Deltax, Deltay, Deltaz;
    HI_S32 Buttons = 0;
    HI_U8 ChangedButtons = 0;
    HI_S32 i;

    memset(pEvent, 0x00, sizeof(*pEvent));

    ReadLength = read(pMouseData->fd, ReadBuffer, MOUSE_READ_BUFLEN);
    if (ReadLength > 0)
    {
        for ( i = 0; i < ReadLength; i++ )
        {
            if ( index == 0  &&  (ReadBuffer[i] & 0xc0) )
```

```
        {
            return HI_FAILURE;
        }

        PacketBuf[index++] = ReadBuffer[i];
        if ( index == pMouseData->packetLength )
        {
            index = 0;

            if (!PACKET_IS_NOT_SYNC(PacketBuf[0]))
            {

                i--;

                return HI_FAILURE;
            }

            Buttons = PACKET_GET_BUTTONS(PacketBuf[0]);

            Deltax = PACKET_GET_DELTAX(PacketBuf[0], PacketBuf[1]);
            Deltay = PACKET_GET_DELTAY(PacketBuf[0], PacketBuf[2]);
            Deltaz = 0;

            if (PS2_PROTOCOL_IMPS2 == pMouseData->mouseId)
            {
                Deltaz = PACKET_GET_DELTAZ(PacketBuf[3]);
                if (Deltaz)
                {
                    pEvent->dz = Deltaz;
                }
            }

            if (Deltax || Deltay)
            {
                pEvent->type |= HIGV_MOUSE_MOVE;
                pEvent->dx = Deltax;
                pEvent->dy = Deltay;
            }

            if ( LastButtons != Buttons )
            {
                ChangedButtons = LastButtons ^ Buttons;

                UpdateMouseEvent(pEvent, ChangedButtons, Buttons);
```

```
                LastButtons = Buttons;
            }
        }
    }

    if ( pEvent->type & HIGV_MOUSE_DOWN )
    {
        gettimeofday(&CurButtonDownTime, NULL);
        interval = CurButtonDownTime.tv_usec + CurButtonDownTime.tv_sec
* 1000000;
        interval -= s_LastButtonDownTime.tv_usec +
s_LastButtonDownTime.tv_sec * 1000000;
        interval /= 1000;
        if (interval <= 300 && s_LastButtonDownTime.tv_sec
&&  !(s_LastButton ^ pEvent->Buttons))
        {
            pEvent->type = HIGV_MOUSE_DOUBLE;
            s_LastButtonDownTime.tv_sec = 0;
        }
        else
        {
            s_LastButtonDownTime = CurButtonDownTime;
        }

        s_LastButton = pEvent->Buttons;
    }

    return HI_SUCCESS;
  }

  return HI_FAILURE;
}

static HI_S32 GpmEvent2MouseEvent(const HIGV_MOUSE_EVENT_S* pGpmEvent,
HIGV_INPUTEVENT_S* pEvent)
{
  if (pGpmEvent->type & HIGV_MOUSE_DOUBLE)
  {
    pEvent->msg = HIGV_MSG_MOUSEDBCLICK;
  }
  else if (pGpmEvent->type & HIGV_MOUSE_DOWN )
  {
    pEvent->msg = HIGV_MSG_MOUSEDOWN;
```

```
    }
    else if ((pGpmEvent->type & HIGV_MOUSE_MOVE))
    {
        pEvent->msg = HIGV_MSG_MOUSEMOVE;
    }
    else  if (pGpmEvent->type & HIGV_MOUSE_UP)
    {
        pEvent->msg = HIGV_MSG_MOUSEUP;
    }

    pEvent->dx = pGpmEvent->dx;
    pEvent->dy = pGpmEvent->dy;
    pEvent->step = pGpmEvent->dz;

    if (pEvent->step)
    {
        pEvent->msg = HIGV_MSG_MOUSEWHEEL;
    }

    if (pGpmEvent->Buttons & HIGV_MOUSE_B_LEFT)
    {
        pEvent->value |= HIGV_MOUSEBUTTON_L;
    }

    if (pGpmEvent->Buttons & HIGV_MOUSE_B_MIDDLE)
    {
        pEvent->value |= HIGV_MOUSEBUTTON_M;
    }

    if (pGpmEvent->Buttons & HIGV_MOUSE_B_RIGHT)
    {
        pEvent->value |= HIGV_MOUSEBUTTON_R;
    }

    return HI_SUCCESS;
}

static HI_S32 GetMouseEvent(HIGV_INPUTEVENT_S *pMouseEvent, HI_U32
TimeOut)
{
    HI_S32 ret = HI_FAILURE;
    fd_set readset;
    struct timeval timeout = {0};
    HIGV_INPUTEVENT_S s_Event = {0};
```

```
HI_BOOL bHaveMsg = HI_FALSE;

if (s_MouseData.fd <= 0)
{
    usleep(TimeOut);
    return OpenMouseDevice();
}

FD_ZERO(&readset);
FD_SET(s_MouseData.fd, &readset);

timeout.tv_sec = 0;
timeout.tv_usec = TimeOut*1000;
while (timeout.tv_usec)
{
    ret = select(s_MouseData.fd+1, &readset, 0, 0, &timeout);
    if (ret == 0)
    {
       ret = HI_EEMPTY;
       break;
    }
    else if (ret < 0)
    {
        close(s_MouseData.fd);
        s_MouseData.fd = -1;
        ret = HI_EUNKNOWN;
        break;
    }

    if (FD_ISSET(s_MouseData.fd, &readset))
    {
       HIGV_MOUSE_EVENT_S evt;
       ret = GetEvent(&evt);
       if (ret == 0)
       {
          GpmEvent2MouseEvent(&evt, pMouseEvent);
          s_Event.dx += pMouseEvent->dx;
          s_Event.dy += pMouseEvent->dy;
          s_Event.msg = pMouseEvent->msg;
          s_Event.step += pMouseEvent->step;
          s_Event.value = pMouseEvent->value;
          bHaveMsg = HI_TRUE;

          if (s_Event.msg != HIGV_MSG_MOUSEMOVE)
```

```
            {
                break;
            }
        }
        else
        {
            close(s_MouseData.fd);
            s_MouseData.fd = -1;
        }
    }
}

if (bHaveMsg)
{
    *pMouseEvent = s_Event;
    return HI_SUCCESS;
}

return ret;
}

HI_S32 HI_GV_InitInputSuite(HIGV_INPUT_S *pInputDevice)
{
    pInputDevice->InitInputDevice = InitInputDevice;
    pInputDevice->DeinitInputDevice = DeinitInputDevice;
    pInputDevice->GetKeyEvent = GetKeyEvent;
    pInputDevice->GetMouseEvent = GetMouseEvent;

    return HI_SUCCESS;
}
```

## 5.2.4 Touchscreen Adaptation Reference Code

```
#define   ONE_POINTER    0
#define   TWO_POINTER    1

#define   GESTURE_SCROLL_FIRST_TRIGGER   50
#define   GESTURE_SCROLL_TRIGGER           5
#define   GESTURE_FLING_TRIGGER            500

static struct tsdev s_TsDevice;

static HI_BOOL s_AlwaysInTapReginon; //is always in down reginon
```

```
static HIGV_TOUCH_POINT_S  s_LatestDownEvent;
static HIGV_TOUCH_POINT_S  s_LatestMoveEvent;
static HIGV_TOUCH_POINT_S  s_LatestPointerEvent;
static HIGV_TOUCH_POINT_S  s_LatestTouchEvent;

static HI_S32 s_LastFocusX;
static HI_S32 s_LastFocusY;
static HI_S32 s_DownFocusX;
static HI_S32 s_DownFocusY;

static HI_S32 s_PointerMode;
static HI_BOOL s_PointerMove = HI_FALSE;

static HIGV_TOUCH_E GetTouchEventType(struct ts_sample* event)
{
    struct ts_sample currentInfo;
    HIGV_TOUCH_E type;

    /*save last one pointer pressure status*/
    static unsigned int lastOnePressure = 0;

    /*save last two pointer pressure status*/
    static unsigned int lastTwoPressure = 0;

    HIGV_MemSet(&currentInfo, 0x0, sizeof(struct ts_sample));
    HIGV_MemCopy(&currentInfo, event, sizeof(struct ts_sample));

    if (ONE_POINTER == currentInfo.id)
    {
        if (0 == lastOnePressure)
        {
            lastOnePressure = currentInfo.pressure;
            type = HIGV_TOUCH_START;
        }
        else if ( (1 == lastOnePressure) && (1 == currentInfo.pressure))
        {
            type = HIGV_TOUCH_MOVE;
        }
        else if ( (1 == lastOnePressure) && (0 == currentInfo.pressure))
        {
            lastOnePressure = currentInfo.pressure;
            type = HIGV_TOUCH_END;
        }
    }
```

```
    if (TWO_POINTER == currentInfo.id)
    {
        if (0 == lastTwoPressure)
        {
            lastTwoPressure = currentInfo.pressure;
            type = HIGV_TOUCH_POINTER_START;
        }
        else if ( (1 == lastTwoPressure) && (1 == currentInfo.pressure))
        {
            type = HIGV_TOUCH_MOVE;
        }
        else if ( (1 == lastTwoPressure) && (0 == currentInfo.pressure))
        {
            lastTwoPressure = currentInfo.pressure;
            type = HIGV_TOUCH_POINTER_END;
        }
    }

    return type;
}

static HI_S32 CalculateVelocity(HIGV_TOUCH_POINT_S* latestDownEvent,
HIGV_TOUCH_POINT_S* currEvent,
                            HI_S32* velocityX, HI_S32* velocityY)
{
    HI_S32 timeInternal;

    if ((NULL == latestDownEvent) || (NULL == currEvent))
    {
        return HI_FAILURE;
    }

    HI_U32 latesttime = latestDownEvent->timeStamp;
    HI_U32 currtime = currEvent->timeStamp;

    timeInternal = currtime - latesttime;

    if (0 >= timeInternal)
    {
        return HI_FAILURE;
    }

    HI_S32 deltaX = abs(currEvent->x - latestDownEvent->x);
```

```
    HI_S32 deltaY = abs(currEvent->y - latestDownEvent->y);

    *velocityX = (deltaX * 1000) / timeInternal;
    *velocityY = (deltaY * 1000) / timeInternal;

    return HI_SUCCESS;
}

static HI_S32 CalculateDistance(HIGV_TOUCH_POINT_S* Event1,
HIGV_TOUCH_POINT_S* Event2,
                            HI_S32* intervalX, HI_S32* intervalY)
{
    if ((NULL == Event1) || (NULL == Event2))
    {
        return HI_FAILURE;
    }

    *intervalX = abs(Event1->x - Event2->x);
    *intervalY = abs(Event1->y - Event2->y);

    return HI_SUCCESS;
}

static HI_S32 GestureDetector(HIGV_TOUCH_POINT_S* event,
HIGV_TOUCH_GESTURE_EVENT_S* callbackevent)
{
    int ret;
    HIGV_TOUCH_POINT_S currentEvent;
    HI_S32 focusX, focusY;
    HIGV_TOUCH_GESTURE_EVENT_S touch_gesture_event;
    HIGV_TOUCH_E type;

    if (NULL == event)
    {
        return HI_FAILURE;
    }

    HIGV_MemSet(&currentEvent, 0x0, sizeof(currentEvent));
    HIGV_MemCopy(&currentEvent, event, sizeof(currentEvent));

    HIGV_MemSet(&touch_gesture_event, 0x0,
sizeof(HIGV_TOUCH_GESTURE_EVENT_S));

    focusX = currentEvent.x;
```

```
    focusY = currentEvent.y;
    type = currentEvent.type;

    switch (type)
    {
        case HIGV_TOUCH_POINTER_START:
        {
            s_PointerMode += 1;
            HIGV_MemCopy(&s_LatestPointerEvent, &currentEvent,
sizeof(currentEvent));
            break;
        }

        case HIGV_TOUCH_POINTER_END:
        {
            s_PointerMode -= 1;
            break;
        }

        case HIGV_TOUCH_START:
        {
            s_DownFocusX = s_LastFocusX = focusX;
            s_DownFocusY = s_LastFocusY = focusY;

            s_AlwaysInTapReginon = HI_TRUE;
            s_PointerMode = 1;
            HIGV_MemCopy(&s_LatestDownEvent, &currentEvent,
sizeof(currentEvent));
            break;
        }

        case HIGV_TOUCH_MOVE:
        {
            HI_S32 scrollX = s_LastFocusX - focusX;
            HI_S32 scrollY = s_LastFocusY - focusY;

            if (s_PointerMode >= 2)
            {
                HI_S32 intervalX = 0;
                HI_S32 intervalY = 0;

                /**handler pinch gesture*/
                if (TWO_POINTER == currentEvent.id)
                {
```

```
                ret = CalculateDistance(&s_LatestTouchEvent,
&currentEvent, &intervalX, &intervalY);
                HIGV_MemCopy(&s_LatestPointerEvent, &currentEvent,
sizeof(currentEvent));
                s_PointerMove = HI_TRUE;
                touch_gesture_event.gestureevent.gesture.pinch.pointer1
= s_LatestTouchEvent;
                touch_gesture_event.gestureevent.gesture.pinch.pointer2
= currentEvent;
            }
            else if (ONE_POINTER == currentEvent.id)
            {
                ret = CalculateDistance(&s_LatestPointerEvent,
&currentEvent, &intervalX, &intervalY);
                s_PointerMove = HI_FALSE;
                touch_gesture_event.gestureevent.gesture.pinch.pointer1
= currentEvent;
                touch_gesture_event.gestureevent.gesture.pinch.pointer2
= s_LatestPointerEvent;
            }

            if (HI_SUCCESS != ret)
            {
                HIGV_Printf("[Func: %s, Line: %d]\n", __FUNCTION__,
__LINE__);
                return HI_FAILURE;
            }

            touch_gesture_event.isgesture = HI_TRUE;
            touch_gesture_event.gesturetype = HIGV_GESTURE_PINCH;
            touch_gesture_event.gestureevent.gesture.pinch.intervalX =
intervalX;
            touch_gesture_event.gestureevent.gesture.pinch.intervalY =
intervalY;
        }

        if (HI_TRUE == s_AlwaysInTapReginon)
        {
            HI_S32 deltaX = focusX - s_DownFocusX;
            HI_S32 deltaY = focusY - s_DownFocusY;

            HI_S32 distance = (deltaX * deltaX) + (deltaY * deltaY);

            /**handler scroll gesture*/
```

```
            if (GESTURE_SCROLL_FIRST_TRIGGER < distance)
            {
                s_LastFocusX = focusX;
                s_LastFocusY = focusY;
                s_AlwaysInTapReginon = HI_FALSE;

                HIGV_MemCopy(&s_LatestMoveEvent, event, sizeof(struct
ts_sample));

                touch_gesture_event.isgesture = HI_TRUE;
                touch_gesture_event.gesturetype = HIGV_GESTURE_SCROLL;
                touch_gesture_event.gestureevent.gesture.scroll.start =
s_LatestDownEvent;
                touch_gesture_event.gestureevent.gesture.scroll.end =
currentEvent;

touch_gesture_event.gestureevent.gesture.scroll.distanceX = abs(scrollX);

touch_gesture_event.gestureevent.gesture.scroll.distanceY = abs(scrollY);
            }
        }
        else if (abs(scrollX) > GESTURE_SCROLL_TRIGGER ||
abs(scrollY) > GESTURE_SCROLL_TRIGGER)
        {
            s_LastFocusX = focusX;
            s_LastFocusY = focusY;

            touch_gesture_event.isgesture = HI_TRUE;
            touch_gesture_event.gesturetype = HIGV_GESTURE_SCROLL;
            touch_gesture_event.gestureevent.gesture.scroll.start =
s_LatestMoveEvent;
            touch_gesture_event.gestureevent.gesture.scroll.end =
currentEvent;
            touch_gesture_event.gestureevent.gesture.scroll.distanceX =
abs(scrollX);
            touch_gesture_event.gestureevent.gesture.scroll.distanceY =
abs(scrollY);

            HIGV_MemCopy(&s_LatestMoveEvent, &currentEvent,
sizeof(currentEvent));
        }

        break;
    }
```

```
        case HIGV_TOUCH_END:
        {
            s_PointerMode = 0;

            /**handler tap gesture*/
            if (HI_TRUE == s_AlwaysInTapReginon)
            {
                touch_gesture_event.isgesture = HI_TRUE;
                touch_gesture_event.gesturetype = HIGV_GESTURE_TAP;
                touch_gesture_event.gestureevent.gesture.tap.pointer =
currentEvent;
            }
            else
            {
                HI_S32 velocityX = 0;
                HI_S32 velocityY = 0;
                ret = CalculateVelocity(&s_LatestDownEvent, &currentEvent,
&velocityX, &velocityY);

                /**handler fling gesture*/
                if (GESTURE_FLING_TRIGGER < velocityX ||
GESTURE_FLING_TRIGGER < velocityY)
                {
                    touch_gesture_event.isgesture = HI_TRUE;
                    touch_gesture_event.gesturetype = HIGV_GESTURE_FLING;
                    touch_gesture_event.gestureevent.gesture.fling.start =
s_LatestDownEvent;
                    touch_gesture_event.gestureevent.gesture.fling.end  =
currentEvent;
                    touch_gesture_event.gestureevent.gesture.fling.velocityX
= velocityX;
                    touch_gesture_event.gestureevent.gesture.fling.velocityY
= velocityY;
                }
            }

            break;
        }

    }

    if (HI_FALSE == s_PointerMove)
    {
```

```
        HIGV_MemCopy(&s_LatestTouchEvent, &currentEvent,
sizeof(currentEvent));
    }

    touch_gesture_event.touchtype = type;
    touch_gesture_event.touchevent.last = currentEvent;

    *callbackevent = touch_gesture_event;

    return HI_SUCCESS;
}

HI_S32 GetTouchEvent(HIGV_TOUCH_GESTURE_EVENT_S* pTouchEvent, HI_U32
TimeOut)
{
    HI_S32 ret = HI_FAILURE;
    struct ts_sample samp = {0};
    fd_set readset;
    struct timeval timeout = {0};
    HIGV_TOUCH_GESTURE_EVENT_S currentEvent;
    HIGV_TOUCH_POINT_S pointEvent;

    HIGV_MemSet(&samp, 0x0, sizeof(struct ts_sample));
    HIGV_MemSet(&currentEvent, 0x0, sizeof(HIGV_TOUCH_GESTURE_EVENT_S));
    HIGV_MemSet(&pointEvent, 0x0, sizeof(HIGV_TOUCH_POINT_S));

    if (s_TsDevice.fd <= 0)
    {
        usleep(300 * 1000);
        return OpenTouchScreenDevice();
    }

    timeout.tv_sec = 0;
    timeout.tv_usec = TimeOut * 1000;

    FD_ZERO(&readset);
    FD_SET(s_TsDevice.fd, &readset);

    ret = select(s_TsDevice.fd + 1, &readset, 0, 0, &timeout);

    if (ret == 0)
    {
        ret = HI_EEMPTY;
        return ret;
```

```
    }
    else if (ret < 0)
    {
        HIGV_Printf("[Func: %s, Line: %d]  ret: %d\n", __FUNCTION__,
__LINE__, ret);
        ret = HI_ERR_COMM_UNKNOWN;
        ts_close(&s_TsDevice);
        return ret;
    }

    if (FD_ISSET(s_TsDevice.fd, &readset))
    {

        ret = ts_read(&s_TsDevice, &samp, 1);

        if (ret < 0)
        {
            HIGV_Printf("[Func: %s, Line: %d] \n", __FUNCTION__, __LINE__);
            return HI_FAILURE;
        }

        if ((0 == samp.tv.tv_sec) && (0 == samp.tv.tv_usec))
        {
            ret = HI_EEMPTY;
            return ret;
        }

        HIGV_TOUCH_E type = GetTouchEventType(&samp);

        pointEvent.id = samp.id;
        pointEvent.x = samp.x;
        pointEvent.y = samp.y;
        pointEvent.pressure = samp.pressure;
        pointEvent.type = type;
        pointEvent.timeStamp = (samp.tv.tv_sec) * 1000 + (samp.tv.tv_usec)
/ 1000;

        ret = GestureDetector(&pointEvent, &currentEvent);

        if (-1 == ret)
        {
            HIGV_Printf("[Func: %s, Line: %d] \n", __FUNCTION__, __LINE__);
            return ret;
        }
```

```
        *pTouchEvent = currentEvent;

        return HI_SUCCESS;
    }

    return ret;
}
```

# 5.3 Multiple Graphics Layers

The HiGV allows you to create system windows on multiple graphics layers. It supports the following multi-graphics-layer features:

- Windows can be created based on graphics layers.
- The input method window can be switched among graphics layers.
- The default graphics layer of windows can be configured.
- The graphics layer to which a window belongs can be queried.
- Graphics layers can be created and destroyed.
- Graphics layers can be displayed and hidden.
- The active window at a graphic layer can be obtained.
- The graphics layers can share the source.

## 5.3.1 Development Application

The development application process is as follows:

**Step 1** Create a graphics layer by calling HI_GV_Layer_CreateEx. This function requires the input of the initialization parameter.

**Step 2** Perform operations by calling the graphics layer operation functions. You can display or hide the graphics layer, configure or obtain the default graphics layer, or specify the graphics layer to which a window belongs.

**Step 3** Destroy the graphics layer by calling HI_GV_Layer_Destroy.

**----End**

## 5.3.2 Parameters for Creating a Graphics Layer

```
typedef struct
{
    HI_S32    ScreenWidth;
    HI_S32    ScreenHeight;
    HI_S32    CanvasWidth;
    HI_S32    CanvasHeight;
    HI_S32    DisplayWidth;
    HI_S32    DisplayHeight;
```

```
    HIGO_LAYER_FLUSHTYPE_E    LayerFlushType;
    HIGO_LAYER_DEFLICKER_E    AntiLevel;
    HIGO_PF_E    PixelFormat;
    HIGO_LAYER_E    LayerID;
} HIGO_LAYER_INFO_S;
```

- **ScreenWidth** and **ScreenHeight**: width and height of the screen for display
- **CanvasWidth** and **CanvasHeight**: width and height of the graphics drawing region, that is, size of the created full-screen window
- **DisplayWidth** and **DisplayHeight**: width and height of the graphics mapping region, that is, size of the mapped full-screen window on the screen
- **LayerFlushType**: refresh mode of the graphics layer, including the single-buffer, dual-buffer, and triple-buffer modes
- **AntiLevel**: anti-flicker level, ranging from low to high. A higher level indicates better anti-flicker effect but more blurred picture.
- **PixelFormat**: pixel format of the graphics layer
- **LayerID**: hardware ID of the graphics layer. The chip platform determines whether the graphics layer is supported.

## 5.3.3 Reference Code

```
#define SCREEN_WIDTH  1280
#define SCREEN_HEIGHT 720

HI_S32 main(HI_S32 argc, HI_CHAR *argv[])
{
    HI_S32 Ret;
    HI_S32 hApp;
    HI_S32 Num;
    HI_HANDLE hDDB = INVALID_HANDLE;
    HI_HANDLE Layer;
    HIGO_LAYER_INFO_S LayerInfo = {SCREEN_WIDTH, SCREEN_HEIGHT,
SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_WIDTH, SCREEN_HEIGHT,
                            (HIGO_LAYER_FLUSHTYPE_E)(HIGO_LAYER_FLUSH_FLIP),
                            HIGO_LAYER_DEFLICKER_AUTO,
                            HIGO_PF_8888, HIGO_LAYER_HD_0};
    HIGV_WINCREATE_S WinCreate;

    if (argc < 2)
    {
        printf("Please input higvbin file.\n");
        return -1;
    }

    Ret = HI_GV_Init();
    if (HI_SUCCESS != Ret)
```

```
    {
        printf("HI_GV_Init fail Ret:%x\n", Ret);
        return -1;
    }
    Ret = HI_GV_PARSER_Init();
    if (HI_SUCCESS != Ret)
    {
        printf("HI_GV_PARSER_Init fail Ret:%x\n", Ret);
        return -1;
    }


    (HI_VOID)HI_GV_Font_SetSystemDefault(hFont);
    s_hLanTestFont = hFont;

    Ret = HI_GV_App_Create("Step2MainApp", (HI_HANDLE *)&hApp);
    if (HI_SUCCESS != Ret)
    {
        HI_GV_Deinit();
        HI_GV_PARSER_Deinit();
        return NULL;
    }
    HI_GV_PARSER_SetWidgetEventFunc(g_pfunHIGVAppEventFunc,
sizeof(g_pfunHIGVAppEventFunc)/sizeof(HI_CHAR *));
    printf("step6\n");

    Ret = HI_GV_PARSER_LoadFile(argv[1]);
    if (HI_SUCCESS != Ret)
    {
        printf("HI_GV_PARSER_LoadFile fail Ret:%x\n", Ret);
        HI_GV_Deinit();
        HI_GV_PARSER_Deinit();
        return -1;
    }
    Ret = HI_GV_Layer_CreateEx(&LayerInfo, LAYER_0);
    if (HI_SUCCESS != Ret)
    {
        printf("failed to HI_GV_Layer_CreateEx! ret:0x%x\n", Ret);
        return Ret;
    }

    Ret = HI_GV_PARSER_LoadViewById(allwidget);
    if (HI_SUCCESS != Ret)
    {
```

```
        printf("HI_GV_PARSER_LoadViewById fail Ret:%x\n", Ret);
        HI_GV_Deinit();
        HI_GV_PARSER_Deinit();
        return -1;
    }

    HI_GV_Widget_Show(allwidget);
    HI_GV_Widget_Active(allwidget);

    Ret = HI_GV_App_Start(hApp);
    if (HI_SUCCESS != Ret)
    {
        HI_GV_PARSER_Deinit();
        (HI_VOID)HI_GV_App_Destroy(hApp);
        HI_GV_Deinit();
        return NULL;
    }

    HI_GV_PARSER_Deinit();
    (HI_VOID)HI_GV_App_Destroy(hApp);
    HI_GV_Deinit();

    return 0;
}
```

# 5.4 MXML

The multi-XML (MXML) function applies to the scenario in which controls are separately controlled in windows of the graphics layers on different output devices. By using a set of XML GUI descriptions, this function clones the same GUI on different output devices and allows you to separately operate the GUIs on different output devices.

Controls on a specified graphics layer are cloned so that clone controls share the same event processing functions and DB. In this way, the GUI based on the same XML description can be independently displayed and operated on different output devices. Because all clone controls share the event processing functions, you can store the operating status of clone controls in their private data to facilitate clone control-related operations.

## 5.4.1 Development Application

The development application process is as follows:

**Step 1** Set the global graphics layer by configuring the layer to which the windows belong in xml2bin options. For example:

**xml2bin -L "LAYER_0;LAYER_1;LAYER_2"**

It indicates that there are three graphics layers: LAYER_0, LAYER_1, and LAYER_2.

**Step 2** Set the graphics layer to which the window belongs in the layer attribute of the window XML description.

**Step 3** Create the graphics layer and display windows at corresponding graphics layers during application initialization.

**Step 4** Query, obtain, and operate controls at corresponding graphics layers in the application.

**----End**

Calling HI_GV_Widget_GetPrivate obtains the private attributes of clone controls, and calling HI_GV_Widget_GetLayer queries the information about the graphics layer to which the current clone control belongs. You can query the clone controls that are associated with the same graphics layer based on the graphics layer information.

## 5.4.2 Reference Code

```
HI_VOID main()
{
    HI_S32 Ret;
    HI_S32 Num;
    HI_HANDLE Layer;
    HIGV_WINCREATE_S WinCreate;
    HIGO_LAYER_INFO_S LayerInfo1= {SCREEN_WIDTH, SCREEN_HEIGHT,
                            SCREEN_WIDTH,SCREEN_HEIGHT,
                            SCREEN_WIDTH, SCREEN_HEIGHT,
(HIGO_LAYER_FLUSHTYPE_E)(HIGO_LAYER_FLUSH_DOUBBUFER),
                           HIGO_LAYER_DEFLICKER_AUTO,
                           HIGO_PF_1555, HIGO_LAYER_SD_0};

HIGO_LAYER_INFO_S LayerInfo2 = {SCREEN_WIDTHHD, SCREEN_HEIGHTHD,
SCREEN_WIDTHHD, SCREEN_HEIGHTHD, SCREEN_WIDTHHD, SCREEN_HEIGHTHD,
(HIGO_LAYER_FLUSHTYPE_E)(HIGO_LAYER_FLUSH_DOUBBUFER),
                           HIGO_LAYER_DEFLICKER_AUTO,
                           HIGO_PF_1555, HIGO_LAYER_HD_0};

    Ret = HI_GV_Init();
    assert (Ret == HI_SUCCESS);
    Ret = HI_GV_PARSER_Init();
    assert(Ret == HI_SUCCESS);
    Ret = AppCreateSysFont(&hFont);
    assert(Ret == HI_SUCCESS);

    Ret = HI_GV_App_Create("Step2MainApp", (HI_HANDLE *)&g_hMXMLApp);
    assert(Ret == HI_SUCCESS);

    Num = sizeof(g_pfunHIGVAppEventFunc);
    HI_GV_PARSER_SetWidgetEventFunc(g_pfunHIGVAppEventFunc,
```

```
sizeof(g_pfunHIGVAppEventFunc)/sizeof(HI_CHAR *));

    Ret = HI_GV_PARSER_LoadFile("./higv.bin");
    assert(Ret == HI_SUCCESS);

    Ret = HI_GV_Layer_CreateEx(&LayerInfo, LAYER_0);
    assert(Ret == HI_SUCCESS);

    Ret = HI_GV_Layer_CreateEx(&LayerInfo2, LAYER_1);
    assert(Ret == HI_SUCCESS);
…

#if  SHOW_CLONE_LAYER_WIN
    HI_HANDLE hCloneWin = 0, hCloneWidget = 0;
    HI_GV_Widget_GetCloneHandle(allwidget, LAYER_1, &hCloneWin);

    Ret = HI_GV_Widget_Show(hCloneWin);
    assert(Ret == HI_SUCCESS);
#endif

    Ret = HI_GV_Widget_Show(allwidget);
   assert(Ret == HI_SUCCESS);
    Ret = HI_GV_Widget_Active(allwidget);
   assert(Ret == HI_SUCCESS);

…
}
```

# 5.5 CLUT8 Format

In the HiGV, the output image format after decoding can be set to CLUT8. The palette must be configured to support the CLUT8 format. To reduce memory usage, a public palette is used for image drawing and graphics layer display.

## 5.5.1 Development Application

The development application process is as follows:

**Step 1** Create a graphics layer and set the pixel format.

**Step 2** Set the information about the decoding graphics layer.

**Step 3** Create controls.

**----End**

## 5.5.2 Precautions

CLUT8 can be used after the graphics layer pixel format and public palette are configured. Note the following:

- Translucent windows are not supported.
- All the images and elements use the same CLUT8 pixel format.
- The scaling function is not supported.
- The palette format needs to be determined at the beginning of the GUI design, including colors of image elements and transparency.

## 5.5.3 Reference Code

```
    HI_S32 Ret;
    HIGV_WCREATE_S WinInfo = {HIGV_WIDGET_WINDOW, {80, 50, 600, 450},
INVALID_HANDLE, 0, 0};
    HI_HANDLE hWin0, hWidget = INVALID_HANDLE;
    HI_HANDLE hWinSkin;
    HIGV_WCREATE_S info;
    HIGV_DEC_SUFINFO_S DecSufinfo;
    HIGV_WINCREATE_S WinCreate;
    HIGV_STYLE_S WinStyle;
    HI_HANDLE hSkin;

    /* set layer info */
    HIGO_LAYER_INFO_S LayerInfo= {SCREEN_WIDTH, SCREEN_HEIGHT,
SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_WIDTH, SCREEN_HEIGHT,

(HIGO_LAYER_FLUSHTYPE_E)(HIGO_LAYER_FLUSH_DOUBBUFER),
                         HIGO_LAYER_DEFLICKER_NONE,
                         HIGO_PF_CLUT8, HIGO_LAYER_HD_0};

    HI_GV_Deinit();

    Ret = HI_GV_Init();
    assert(Ret == HI_SUCCESS);


   /* create layer */
    Ret = HI_GV_Layer_Create(&LayerInfo, &g_StLayer);
    if (HI_SUCCESS != Ret)
    {
        printf("failed to HI_GV_Layer_Create! ret:0x%x\n", Ret);
        assert(0);
    }
```

```
    Ret = HI_GV_App_Create("test_drawclip", &s_hDrawClipApp);
    assert(Ret == HI_SUCCESS);

    Ret = HI_GO_SetWindowMode(HIGO_WNDMEM_SHARED);
    assert(Ret == HI_SUCCESS);

    /* set decode surface pixel format is clut8 */
    DecSufinfo.PixFormat = HIGO_PF_CLUT8;
    DecSufinfo.MemType = HIGO_MEMTYPE_MMZ;
    DecSufinfo.IsPubPalette = HI_TRUE;
    Ret = HI_GV_Resm_SetDecSurfInfo(&DecSufinfo);
    assert(Ret == HI_SUCCESS);

    /* create window */
    WinCreate.hLayer = g_StLayer;
    WinCreate.PixelFormat = HIGO_PF_BUTT;
    WinInfo.pPrivate = &WinCreate;
    Ret = HI_GV_Widget_Create(&WinInfo, &hWin0);
    assert(Ret == HI_SUCCESS);

    memset(&WinStyle, 0x0, sizeof(WinStyle));
    WinStyle.StyleType = HIGV_STYLETYPE_COLOR;
    WinStyle.BackGround.Color = 0xff;
    WinStyle.FontColor = 0x00;
    Ret = HI_GV_Res_CreateStyle(&WinStyle,&hSkin);
    Ret = HI_GV_Widget_SetSkin(hWin0, HIGV_SKIN_NORMAL, hSkin);
   assert(Ret == HI_SUCCESS);

    memset(&info, 0x0, sizeof(info));
    info.rect.x = 50;
    info.rect.y = 50;
    info.rect.w = 260;
    info.rect.h = 260;
    info.hParent = hWin0;
    info.type = HIGV_WIDGET_IMAGE;

    Ret = HI_GV_Widget_Create(&info,  &hWidget);
    assert(Ret == HI_SUCCESS);

    /* set gif on image */
    HI_RESID ResID;
    Ret = HI_GV_Res_CreateID("./res/test.gif", HIGV_RESTYPE_IMG, &ResID);
    assert(Ret == HI_SUCCESS);
    HI_GV_Image_SetImage(hWidget, ResID);
```

```
    memset(&WinStyle, 0x0, sizeof(WinStyle));
    WinStyle.StyleType = HIGV_STYLETYPE_COLOR;
    WinStyle.BackGround.Color = 0x55;
    Ret = HI_GV_Res_CreateStyle(&WinStyle,&hSkin);
    Ret = HI_GV_Widget_SetSkin(hWidget, HIGV_SKIN_NORMAL, hSkin);

    s_QuitHandle = hWin0;
    Ret = HI_GV_Widget_SetMsgProc(hWin0, HIGV_MSG_KEYDOWN,
DrawClip_TestQuit, HIGV_PROCORDER_AFTER);

    Ret = HI_GV_Widget_Show(hWin0);
    HI_GV_Widget_Active(hWin0);

    Ret = HI_GV_App_Start(s_hDrawClipApp);

    HI_GV_App_Destroy(s_hDrawClipApp);
    HI_GV_Deinit();
```

# 5.6 RLE Format

Run-length encoding (RLE) is a lossless image encoding format. It not only reduces the memory space, but also loses no image data.

The RLE is a CLUT8 format of lossless image compression. GIF images can be converted into RLF images. If the RLE format is used, the RLE resource format file and palette must be generated.

RLF images are generated by converting GIF images using the RLE tool. The RLE tool can also be used to extract the palette.

## 5.6.1 Development Application

The development application process is as follows:

**Step 1** Generate the palette PAL file and RLE file by using the RLE tool.

**Step 2** Create a graphics layer and palette.

**Step 3** Set the information about the decoding graphics layer.

**Step 4** Load the view and display the window.

**----End**

## 5.6.2 Reference Code

```
HI_S32 main(HI_S32 argc, HI_CHAR *argv[])
{
```

```
HI_S32 Ret, hApp;
HI_HANDLE hFont;
HIGV_DEC_SUFINFO_S DecSufinfo;

HIGO_LAYER_INFO_S LayerInfo = {SCREEN_WIDTH,
                               SCREEN_HEIGHT,
                               SCREEN_WIDTH,
                               SCREEN_HEIGHT,
                               SCREEN_WIDTH,
                               SCREEN_HEIGHT,
              (HIGO_LAYER_FLUSHTYPE_E)(HIGO_LAYER_FLUSH_FLIP),
                           HIGO_LAYER_DEFLICKER_NONE,
                           HIGO_PF_CLUT8,
                           HIGO_LAYER_HD_0};

/* check args */
if (argc < 2)
{
    printf("Please input higvbin file.\n");
    return -1;
}

/* init higv module */
Ret = HI_GV_Init();
if (HI_SUCCESS != Ret)
{
    printf("HI_GV_Init fail Ret:%x\n", Ret);
    return -1;
}

Ret = HI_GV_PARSER_Init();
if (HI_SUCCESS != Ret)
{
    printf("HI_GV_Init fail Ret:%x\n", Ret);
    return -1;
}

/* set window memory mode */
Ret = HI_GO_SetWindowMode(HIGO_WNDMEM_SHARED);
assert(Ret == HI_SUCCESS);

/* load higv.bin file for parser */
Ret = HI_GV_PARSER_LoadFile(argv[1]);
if (HI_SUCCESS != Ret)
```

```
{
    printf("HI_GV_PARSER_LoadFile fail Ret:%x\n", Ret);
    HI_GV_Deinit();
    HI_GV_PARSER_Deinit();
    return -1;
}

HI_HANDLE HD_0 = HI_NULL;
/* create layer */
Ret = HI_GV_Layer_Create(&LayerInfo, &HD_0);
if (HI_SUCCESS != Ret)
{
    printf("failed to HI_GV_Layer_CreateEx! ret:0x%x\n", Ret);
    return Ret;
}

/* set public palette, once clut to rgb */
Ret = HI_GO_SetPubPalette(gs_Palette);
assert(Ret == HI_SUCCESS);

/* set layer palette */
HI_HANDLE  HigoLayer;
HI_PALETTE Palette={0};

Ret = RLE_CreatePalette("./res/pub.pal", Palette);
assert(Ret == HI_SUCCESS);

HI_GV_Layer_GetHigoLayer(HD_0, &HigoLayer);
Ret = HI_GO_SetLayerPalette(HigoLayer, Palette);
assert(Ret == HI_SUCCESS);

/* set decode surface info in resource manager */
DecSufinfo.PixFormat = HIGO_PF_CLUT8;
DecSufinfo.MemType = HIGO_MEMTYPE_OS;
/* clut to rgb active using public palette, after decode */
DecSufinfo.IsPubPalette = HI_TRUE;
Ret = HI_GV_Resm_SetDecSurfInfo(&DecSufinfo);
assert(Ret == HI_SUCCESS);

Ret = HI_GV_PARSER_LoadViewById(allwidget);
if (HI_SUCCESS != Ret)
{
    printf("HI_GV_PARSER_LoadViewById fail Ret:%x\n", Ret);
    HI_GV_Deinit();
```

```
        HI_GV_PARSER_Deinit();
        return -1;
    }

    Ret = HI_GV_App_Create("MainApp", (HI_HANDLE *)&hApp);
    if (HI_SUCCESS != Ret)
    {
        HI_GV_Deinit();
        HI_GV_PARSER_Deinit();
        return 0;
    }

    HI_GV_Widget_Show(allwidget);
    HI_GV_Widget_Active(allwidget);

    Ret = HI_GV_App_Start(hApp);
    if (HI_SUCCESS != Ret)
    {
        HI_GV_PARSER_Deinit();
        (HI_VOID)HI_GV_App_Destroy(hApp);
        HI_GV_Deinit();
        return 0;
    }

    printf("step 5:Destroy!\n");
    HI_GV_PARSER_Deinit();
    (HI_VOID)HI_GV_App_Destroy(hApp);
    HI_GV_Deinit();

    return 0;
}
```

# 5.7 Input Method

The HiGV implements various input methods including pinyin, English letters (uppercase and lowercase letters), digits, punctuations, and customized characters. You can enable or disable these input methods as required.

The input method further provides an input panel, which is mainly used to provide a selection operation for a complex input method. Especially when the pinyin input method is used, two level candidate codes are required for input. The panel structure of pinyin input method is shown in Figure 5-1.

**Figure 5-1** Pinyin input method

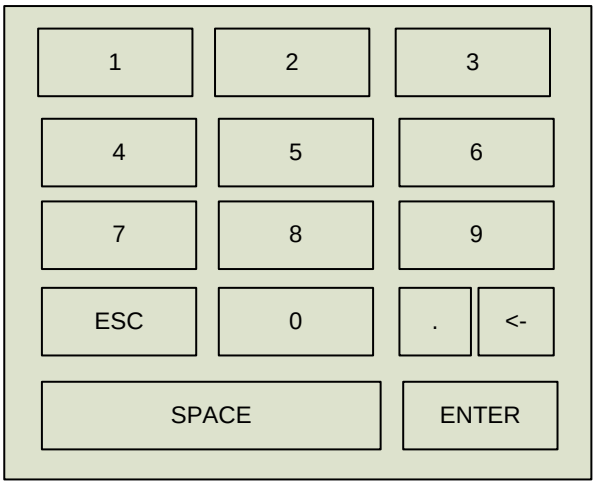

where

- The level-1 candidate codes are the direct mapping of the input key combination, for example, the candidate pinyin corresponding to the input keys in pinyin input method.
- The level-2 candidate codes are the mapping of the level-1 candidate codes, for example, the characters corresponding to the selected pinyin.

The HiGV also provides the soft keyboard to support the input by using the mouse or touch devices. You can define the soft keyboard and input method switch window in XML description or APIs. The generation process is the same as that for common GUIs. Currently you can use the virtual key values defined in **hi_gv_msg.h**.

The display of the soft keyboard relies on the current input method, which supports pinyin, uppercase letters, lowercase letters, digits, punctuations, and customized characters.

**Figure 5-2** Digital soft keyboard




# 5.7.1 Development Application

## Creating an Input Method

To create an input method, perform the following steps:

**Step 1** Create an input method by calling HI_GV_IMEWindow_Create() during program initialization. Set the number of candidate codes and the image resource for the input method window.

**Step 2** Configure the skin for the input method window.

**----End**

### Creating a Soft Keyboard

To create a soft keyboard, perform the following steps:

**Step 1** Define the controls (such as the window and buttons) of the soft keyboard and the input method switch window in the XML file or API.

**Step 2** Register the soft keyboard and input method by calling HI_GV_IMEWindow_SetSoftKB().

**----End**

## 5.7.2 Reference Code

```
HI_S32 switch_button_onclick(HI_HANDLE hWidget,HI_U32 wParam, HI_U32
lParam)
{
    HI_U32 i;
    static HI_U32 ime = 0;
    HI_GV_IMEWindow_E temp_ime;

    HI_HANDLE hActiveWidget;
    HI_U32 SupportIMEType;
    HI_GV_IMEWindow_E IMName;

    HI_GV_IMEWindow_GetIMEActiveWidet(g_hIMEWinHandle, &hActiveWidget);

    HI_GV_Widget_IsNeedIMEWindow(hActiveWidget, &SupportIMEType);
    HI_GV_IMEWindow_GetCurrentIME(g_hIMEWinHandle, &temp_ime);

    ime = temp_ime;
    if (0 == SupportIMEType)
    {

SupportIMEType=(HI_U32)pow((HI_DOUBLE)2,(HI_DOUBLE)HI_GV_IMEWindow_BUTT)
- 1;
    }



    i = 0;
    while(i < HI_GV_IMEWindow_BUTT)
    {
     ime = (++ime)%(HI_U32)HI_GV_IMEWindow_BUTT;
     if (SupportIMEType>>ime & 0x1)
     {
         HI_GV_IMEWindow_Change(g_hIMEWinHandle, (HI_GV_IMEWindow_E)ime);
```

```
        break;
    }
    i++;
    }

    return HIGV_PROC_GOON;
}

extern HI_HANDLE imswitch_skin[5];
HI_S32 im_switch_callback(HI_HANDLE hWidget,HI_U32 wParam, HI_U32 lParam)
{
    HI_GV_IMEWindow_E temp_ime;

    HI_GV_IMEWindow_GetCurrentIME(g_hIMEWinHandle, &temp_ime);
    HI_GV_Widget_SetSkin(switch_button,HIGV_SKIN_NORMAL,imswitch_skin[tem
p_ime])
    HI_GV_Widget_SetSkin(switch_button,HIGV_SKIN_MOUSEDOWN,
imswitch_skin[temp_ime]);
    HI_GV_Widget_Paint(im_switch, 0);
}

HI_S32 CreateIMWin()
{
   HI_RESID BgPic;
   HI_GV_IMEWINInit_S Initdata;
   HI_HANDLE g_hIMEWinHandle;

   memset(&Initdata, 0x00, sizeof(Initdata));
   Initdata. PixelFormat = HIGO_PF_BUTT;
   Initdata.CandidateNum = 5;
   Initdata.CapEnglishLogoIndex = (HI_U32)"./res/capenglishlogo.PNG";
   Initdata.EnglishLogoIndex = (HI_U32)"./res/englishlogo.PNG";
   Initdata.LeftArrowPicIndex = (HI_U32)"./res/leftarrowlogo.PNG";
   Initdata.NumberLogoIndex = (HI_U32)"./res/numberlogo.PNG";
   Initdata.PinyinLogoIndex = (HI_U32)"./res/pinyinlogo.PNG";
   Initdata.RightArrowPicIndex = (HI_U32)"./res/rightarrowlogo.PNG";
   Initdata.SymbolLogoIndex = (HI_U32)"./res/symbollogo.PNG";
   Initdata.UnActiveLeftArrowPicIndex = (HI_U32)"./res/unactiveleft.PNG";
   Initdata.UnActiveRightArrowPicIndex =
(HI_U32)"./res/unactiveright.PNG";

   ret = HI_GV_IMEWindow_Create(&Initdata, &hIMEWinHandle);//Create the
input method window
   if (HI_SUCCESS != ret)
```

```
    {
     printf("failed to create the window!\n");
     return ret;
    }

    ret = HI_GV_Res_CreateID("res/bgimage.PNG", HIGV_RESTYPE_IMG, &BgPic);
    if (HI_SUCCESS != ret)
    {
      printf ("Create IMEWindow EnglishLogoPic ResID Failure!\n");
    }

     memset(&WinStyle, 0x00, sizeof(HIGV_STYLE_S));
     WinStyle.StyleType = HIGV_STYLETYPE_PIC;
     WinStyle.BackGround.ResId = BgPic;
     WinStyle.FontColor = 0xff0000ff;
     WinStyle.bNoDrawBg = HI_FALSE;
     WinStyle.LineWidth = 5;

    ret = HI_GV_Res_CreateStyle(&WinStyle, &hNormalSkin);
    {
      printf ("Create hNormalSkin Failure!\n");
    }

    HI_GV_Widget_SetSkin(hIMEWinHandle, HIGV_SKIN_NORMAL, hNormalSkin);

#if SUPPORT_SOFTKB

    #define NUM_KEY_NUM 15
    HIGV_IMEWIN_KEYVALUE_PAIR_S NUM_KeyValueMap[NUM_KEY_NUM];
    HIGV_IMEWIN_SOFTKB_S SoftKBVec[HI_GV_IMEWindow_BUTT];

    HI_GV_IMEWindow_GetCurrentIME(g_hIMEWinHandle, &temp_ime);
    HI_GV_Widget_SetSkin(switch_button, HIGV_SKIN_NORMAL,
imswitch_skin[temp_ime]);

HI_GV_Widget_SetMsgProc(g_hIMEWinHandle,HIGV_MSG_IME_SWITCH,im_switch_cal
lback,HIGV_PROCORDER_AFTER);
     assert(Ret == HI_SUCCESS);

     NUM_KeyValueMap[0].KeyWidget = num_softkey_0;
     NUM_KeyValueMap[0].KeyValue = HIGV_KEY_0;
     NUM_KeyValueMap[1].KeyWidget = num_softkey_1;
     NUM_KeyValueMap[1].KeyValue = HIGV_KEY_1;
     NUM_KeyValueMap[2].KeyWidget = num_softkey_2;
```

```
NUM_KeyValueMap[2].KeyValue = HIGV_KEY_2;
NUM_KeyValueMap[3].KeyWidget = num_softkey_3;
NUM_KeyValueMap[3].KeyValue = HIGV_KEY_3;
NUM_KeyValueMap[4].KeyWidget = num_softkey_4;
NUM_KeyValueMap[4].KeyValue = HIGV_KEY_4;
NUM_KeyValueMap[5].KeyWidget = num_softkey_5;
NUM_KeyValueMap[5].KeyValue = HIGV_KEY_5;
NUM_KeyValueMap[6].KeyWidget = num_softkey_6;
NUM_KeyValueMap[6].KeyValue = HIGV_KEY_6;
NUM_KeyValueMap[7].KeyWidget = num_softkey_7;
NUM_KeyValueMap[7].KeyValue = HIGV_KEY_7;
NUM_KeyValueMap[8].KeyWidget = num_softkey_8;
NUM_KeyValueMap[8].KeyValue = HIGV_KEY_8;
NUM_KeyValueMap[9].KeyWidget = num_softkey_9;
NUM_KeyValueMap[9].KeyValue = HIGV_KEY_9;
NUM_KeyValueMap[10].KeyWidget = num_softkey_esc;
NUM_KeyValueMap[10].KeyValue = HIGV_KEY_ESC;
NUM_KeyValueMap[11].KeyWidget = num_softkey_dot;
NUM_KeyValueMap[11].KeyValue = HIGV_KEY_DOT;
NUM_KeyValueMap[12].KeyWidget = num_softkey_del;
NUM_KeyValueMap[12].KeyValue = HIGV_KEY_DEL;
NUM_KeyValueMap[13].KeyWidget = num_softkey_space;
NUM_KeyValueMap[13].KeyValue = HIGV_KEY_SPACE;
NUM_KeyValueMap[14].KeyWidget = num_softkey_enter;
NUM_KeyValueMap[14].KeyValue = HIGV_KEY_ENTER;

memset(Txt, 0, sizeof(Txt));
for (i = 0; i < 10; i++)
{
    Txt[0] = '0' + i;
    HI_GV_Widget_SetText(NUM_KeyValueMap[i].KeyWidget, Txt);
}

HI_GV_Widget_SetText(NUM_KeyValueMap[10].KeyWidget, "Esc");
HI_GV_Widget_SetText(NUM_KeyValueMap[11].KeyWidget, ".");
HI_GV_Widget_SetText(NUM_KeyValueMap[12].KeyWidget, "<-");
HI_GV_Widget_SetText(NUM_KeyValueMap[13].KeyWidget, "Space");
HI_GV_Widget_SetText(NUM_KeyValueMap[14].KeyWidget, "Enter");

memset(SoftKBVec, 0x0, sizeof(SoftKBVec));
SoftKBVec[HI_GV_IMEWindow_NUMBER].bNoDispIMW = HI_TRUE;
SoftKBVec[HI_GV_IMEWindow_NUMBER].hSoftKB = im_softkb_num;
SoftKBVec[HI_GV_IMEWindow_NUMBER].bDispSoftKB = HI_TRUE;
SoftKBVec[HI_GV_IMEWindow_NUMBER].pKeyValueMap = NUM_KeyValueMap;
```

```
    SoftKBVec[HI_GV_IMEWindow_NUMBER].KeyNum = NUM_KEY_NUM;


HI_GV_IMEWindow_SetSoftKB(g_hIMEWinHandle,SoftKBVec,HI_GV_IMEWindow_BUTT);
#endif
}
```

# 5.8 Thread Security

When there are multiple threads running at the same time in the project, changing control data by scheduling the HiGV APIs in other threads is insecure. In this case, you need to change the HiGV behavior in other threads by sending messages.

**Step 1** Register the message callback function.

HiGV messages can be registered in either of the following modes:

- Encapsulate service implementation into a function and register it as the control message event callback function by calling HI_GV_Widget_SetMsgProc.
- Register the message callback function in the .xml layout file. For details about the message callback function type, see the *HiGV Labels User Guide*.

**Step 2** Send a message to the HiGV in other threads.

**Step 3** The HiGV executes the registered callback function when it receives the message.

**----End**

**NOTE**

For details about the message transmission API, see the header file **hi_gv_msg.h**; for details about the message event, see the header file **hi_gv_widget.h**.

# 5.9 Animation

## 5.9.1 Method 1 for Creating an Animation

The animation component is an API that implements the animation effect of the UI. HiGV supports only Tween animation currently. The Tween animation effect is achieved by constantly changing the animation attributes of the animation object (including controls or images). The running speed of the animation at different time points is controlled by the Easing function. For details about the Easing function, see the website http://easings.net/zh-cn.

Currently, some HiGV controls support the animation effect. If users need to customize the animation effect, the interfaces of the animation component can be called to implement control-level animation.

**Step 1** Call HI_GV_TweenAnimCreate to create an animation instance.

**Step 2** Call HI_GV_TweenAnimSetDuration to configure the running duration of the animation.

**Step 3** Call HI_GV_TweenAnimAddTween to configure the Easing type (Transition or Ease) of the animation instance as well as the start and end ranges of the animation attributes.

**Step 4** Register the animation listening callback function, including the animation, start, running, and end. The animation effect is achieved by running the callback function. This callback function requires users to update the attributes of the animation object and refresh the UI to achieve the animation effect. The attribute values calculated by the Easing function can be obtained by calling HI_GV_TweenAnimGetTweenValue.

**Step 5** Call HI_GV_TweenAnimStart to start the animation.

**----End**

## 5.9.2 Method 2 for Creating an Animation

The HiGV provides some default tween animation effects (such as translation, alpha gradient, and rolling), which can be implemented through XML configuration. You can also dynamically create animations by calling APIs. For details, see Animation Information in section 3.5.2 "Using Resources."

**Step 1** Define the animation information in **res/xml/anim.xml**. The following shows an example of defining a translation animation:

```
<translate
    id="ANIM_BTN_MOVE"
    duration_ms=500
    repeatcount=3
    delaystart=20
    animtype=0
    fromx=20
    fromy=6
    tox=252
    toy=260
/>
```

**Step 2** Associate the animation with a control.

```
<button
id="ANIM_BUTTON"
…
Anim="ANIM_BTN_MOVE"
/>
```

When the ANIM_BUTTON button is displayed, the animation ANIM_BTN_MOVE that is associated with the button is displayed.

**----End**

## 5.9.3 Rebound Effects of List Controls

The HiGV provides default rebound effects for the list box, scroll view, scroll grid, and wheel view controls.

The creation procedure is as follows:

When defining XML resources of the list box, add the **reboundmax** attribute and set the maximum value of rebound.

# 5.10 Performance Optimization

The HiGV development process involves the configurations of many parameters (for resources and controls), which affect the functions and performance of the entire graphics system. This section describes the impact of each configuration on the UI performance and functions.

## 5.10.1 Control/Resource Instance

- The time required for creating and loading controls and resource instances increases as the number of controls and resource instances increases. Therefore, you need to control the number of controls and resource instances and reuse the controls with similar attributes (such as the displayed dialog box) and resources (multiple controls can share the same resource).
- Typically the fixed and frequently used controls are created by using the XML file. In some special scenarios where the control attributes change in real time, you can dynamically create a control by calling an API and destroy it immediately after it is used.
- If a control is frequently displayed and refreshed, the system performance is affected. Therefore, do not refresh the control unless necessary.
- The resource instances that are created by calling an API must be destroyed when they are not used. Creating a resource instance repeatedly does not result in memory leaks because the HiGV will check whether the resource to be created already exists. The HiGV releases all resources when it is deinitialized.

## 5.10.2 Optimization of Image Skins

- Based on the features of the nine-square skins, image resources can be split into multiple small images and then combined flexibly for use.
- When drawing images, the HiGV tiles images in non-scaling mode. Therefore, large single-color images can be clipped into small images with the size of 1 x 1 pixel, and images with no color change in the horizontal direction can be clipped into line images with the width of 1 pixel.

## 5.10.3 Data Models

- Data is synchronized when the control is bound to a new data model or the data changes. The time required for data synchronization depends on the data amount.
- HI_GV_Widget_SyncDB can be used to synchronize data for most cases. HI_GV_ADM_Sync is used only when all data that is bound to a data model needs to be synchronized in some specific scenarios.
- The number of allocated buffer rows for a UserDB is specified by **BufferRows**, corresponding to **cacherows** in the XML attribute. A larger value indicates that more data can be obtained at a time, reducing the times of calling the callback function for obtaining data. However, in this case, more memory is required for creating the data model.

## 5.10.4 Hide-Release Style and Window Switch Performance

- The skin resources of all HiGV GUIs are loaded when the controls are displayed for the first time to accelerate loading during startup. Memories are requested when image resources are loaded.
- If the control style is the hide-release style (isrelease attribute in the XML), resources such as the images and internal timer are released after the control is hidden and are reloaded when the control is displayed next time.
- If the resources are not released when the control is hidden, resources do not need to be reloaded when the control is displayed again. You can load these resources by calling HI_GV_Win_LoadSkin() during startup to facilitate memory management. This API is valid only for controls with resources that are not released when the controls are hidden.
- When the UI is being switched, you need to display (HI_GV_Widget_Show) and activate (HI_GV_Widget_Active) the next UI before hiding (HI_GV_Widget_Hide) the previous UI because the HiGV does not allow a hidden control to be focused. If the focus control is directly hidden, the HiGV will search for an active control, which is time-consuming. Besides, the image resources for displaying controls are not released, and therefore the used resources do not need to be loaded again.
-