StdWidget Overview

StdWidget is a lightweight framework for rendering graphical user interface elements (widgets) and managing events that occur when the user interacts with them. It should be noted that StdWidget is not a full-featured, high level UI library. Merely the aim is to simplify the process of creating VSOS applications that require a graphical user interface.

StdWidget architecture is divided into three main components:

1. the VSOS application itself
2. Framework
3. Style Manager

An application describes the user interface by providing a list of objects of the type StdWidget. The StdWidget data structure describes an abstract element of the user interface. The structure is very minimalistic containing only the bare minimum required to define the properties of the widget. See below for the definition:

```
typedef struct StdWidget {
	u_int16 flags; // one of the WL* | one of the WT_* | a combination of WF_*
	u_int16 symbol; // symbol is used together with (sometimes instead of) the caption
	WidgetContent content; // content defines what is shown on the widget
	UserEvent userEvent; // system calls this function whenever an event occurs
	RenderCallback renderCallback; // function responsible for drawing the widget
	u_int16 msgType; // UIMSG
	WidgetData data; // the internal state of the widget
} StdWidget;
```

Each widget is associated with a type. Currently available types are:

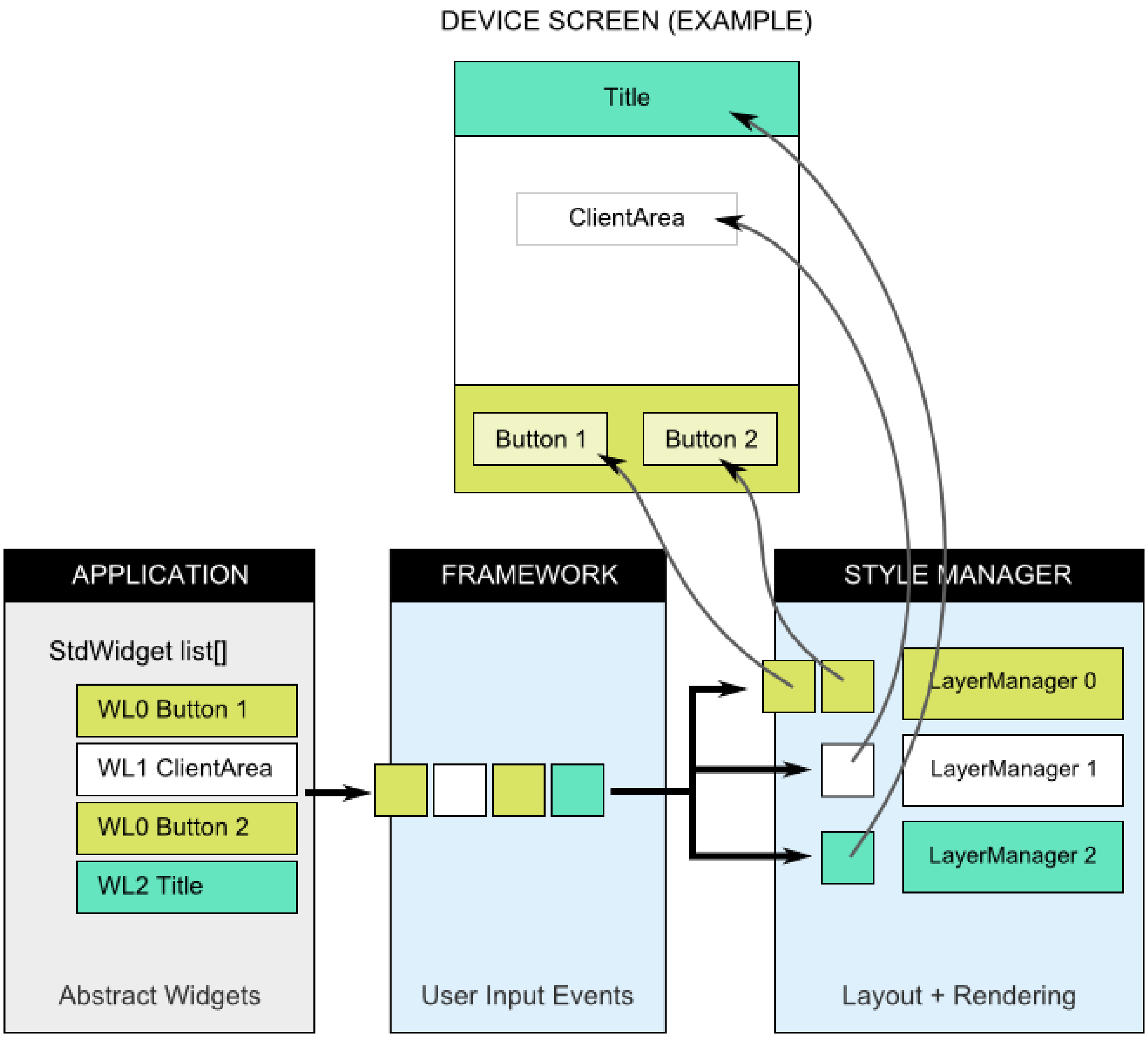
- buttons (WT_BUTTON)
- value sliders (WT_SLIDER)
- widget lists (WT_DLIST)
- custom widget (WT_CUSTOM)

Button is the most common generic widget type. A button without an associated function is regarded as a text label. A list widget is a 'virtual' widget which itself is not drawn but can spawn any number of (sub)widgets in its place.

Each widget is also associated with a layer. A layer is a conceptual group of widgets. It provides a mechanism to treat multiple widgets in a uniform way. For example, the contents of a pop-up window can be assinged to a specific layer so that the layer can be shown or hidden using a single control.

A widget's **userEvent** is a function which is called when the user interacts with the widget. For example, if the widget represents a play/stop button of an mp3 player, the application should provide a function which signals the mp3 model to perform the appropriate action.

A notable property of the StdWidget structure is the absense of positional information (i.e. screen coordinates). Most of the widgets are also sizeless by default. This way it is possible to define widgets without making any assumptions on how they will be presented to the user. In other words the UI elements are defined in a very abstract way, hopefully making it easier to port the application to different environments.



Framework and Style Manager

The Framework module manages the widget list provided by the application. It sends each widget to the Style Manager module which takes the level of abstraction down to match the underlying hardware. The Framework relies mainly on two services provided by the VSOS kernel: **LdcFilledRectangle** (for sending pixels to the display) and **GetTouchLocation** (for receiving user input). The standard Framework is intended to be used in association with a touch panel. If alternative forms of user input are needed, the Framework must be modified to reflect this requirement.

The Style Manager is responsible for lowering the level of abstraction so that the widgets can be displayed on the screen. The process involves determining the size and screen position for all of them. If a widget does not specify a render callback, Style Manager should assign a render function suitable for the widget type. The Style Manager module contains basic rendering functionality for standard widget types. The default implementation relies on a static library called **LibGfx4** which contains some graphic routines mostly dealing with 4-bit pixel data.

Widgets are treated differently depending on the layer they are assigned to. Currently up to 8 layers can be used. The basic idea is that the Style Manager provides a Layout Manager for each layer. The corresponding Layout Manager will determine a layout and visual style for all widgets contained in that layer. Layout Manager maintains its internal state and knows how much screen estate is still available and how much has been used. When there is not enough space for the widgets anymore, it can issue a signal so that the Framework will no longer send new widgets to it. A Layout Manager is also responsible for updating any screen regions that belong to it but are not covered by widgets.

To summarize: an application can define the user interface elements independently from the underlying hardware. The Framework is also largely independent from the screen resolution etc but is expected to track input device and provide user events. The Style Manager is the lowest level component. Style Manager's implementation should take into account the physical characteristics of the target platform.

Initialization and Use

From the application's point of view, using StdWidget is quite straightforward. The first step is to load the style manager.

```
libSWStyle=LoadLibrary("MP3STYLE");
mp3Style=SWSGet(libSWStyle);
```

The next task is to load and initialize the Framework. A reference to the style manager is passed to the framework.

```
libStdWidget=LoadLibrary("STDWDGT");
SWInit(libStdWidget,mp3Style,0);
```

Calling **SWUpdate**() periodically will update the display and poll for user-generated events. The second parameter to SWUpdate() is a layer mask. The least significant bits control the visiblity of the corresponding UI layers. Passing a value zero will display all of them. The third parameter is an array of StdWidget objects. The list must be terminated by a widget whose type is **WT_END_OF_LIST**.

It is important to note that the first call to SWUpdate() will draw and update the screen automatically. After that, widgets are drawn only per request. The application can set the flag **WF_PAINT** to request the rendering of a certain widget. The system will clear the flag once the widget has been drawn.

```
SWUpdate(libStdWidget, 0, widgetList/*remember to terminate the list!*/);
```

Whenever the user interacts with any of the widgets, the Framework will invoke the associated event. Below is an example initializer for a single button UI.:

```
StdWidget myUI[]={
        {WL0|WT_BUTTON,0/*no symbol*/,{"Caption"},
        ButtonEventCallback,
        0/*use default render callback*/,
        0/*UIMSG*/,{0/*ptr param*/,0/*int param*/}},
        {WT_END_OF_LIST},
};
```

There are a couple of special widget types. A widget declared with the **UIMSG_BUT_NEXTPAGE** value in the uiMessage field will be automatically assigned to an event which, once invoked, will reveal more widgets if there was not room for everything in the screen. **UIMSG_BUT_PREVPAGE** will behave in a symmetrical manner.

Finally the Framework imports a general purpose function called **SWCtrl**() which can be used to communicate with the library. The default framework supports two functions.

SWCtrl(libStdWidget,swcRepaint,0,0) requests a full repaint of the display. Setting the **WF_PAINT** flag for a large group of widgets is not always practical so one can request a full repaint instead.

The application can request and provide information about graphic symbols (icons) used by the widgets by calling **SWCtrl**(libStdWidget,swcGetIconInfo,0,0). Icon data is unfortunately hardware dependent. For this reason the application should take note of the format expected by the graphics library. If the application can provide icon data in the requested format, widget definitions can include a non-zero 'symbol' value to suggest rendering a graphic symbol instead of (or in addition to) the actual caption. Remember to update the **iconData** member of the **IconInfo** structure to define the icon data to use.

Custom Rendering

The Style Manager (or the application itself) can enforce a visual style for a widget by providing a suitable render callback function. In order keep the memory footprint as small as possible, the pixel data of a widget is not buffered as a whole. Instead, rendering takes place one vertical line at a time. For this reason, render callback function must provide the system with another callback function (called a "rasterizer") which, upon request, must be able to produce any given scan line of the widget. This means producing an array of pixel values accepted by **LcdFilledRectangle**(). Currently pixels are expected in the 16-bit "RGB565" format.

The default rendering functions behave in the following manner: a common graphics buffer is allocated that is large enough to hold the maximum-sized widget in a “compressed“ format (4 bits per pixel). All rendering functions draw widgets into the common buffer and return a rasterizer which extracts RGB565 values by performing a look-up in a color palette containing 16 color entries.