|  |  |
| --- | --- |
|  | **2008** |
|  | Julien TEMPLIER |

|  |
| --- |
| **[DesktopX PLugin DOcumentation]** |
| Description of a DesktopX plugin and how to implement it. |

Table of Contents

[Overview 4](#_Toc209266782)

[SDMessage 9](#_Toc209266783)

[SD\_INITIALIZE\_MODULE 9](#_Toc209266784)

[SD\_TERMINATE\_MODULE 9](#_Toc209266785)

[SD\_QUERY\_PLUGIN\_INFO 10](#_Toc209266786)

[SD\_QUERY\_CUSTOM\_STATES 10](#_Toc209266787)

[SD\_GET\_OBJECT\_SIZE 11](#_Toc209266788)

[SD\_CREATE\_PLUGIN 11](#_Toc209266789)

[SD\_DESTROY\_PLUGIN 11](#_Toc209266790)

[SD\_DUPLICATE\_PLUGIN 11](#_Toc209266791)

[SD\_INITIALIZE\_PLUGIN 12](#_Toc209266792)

[SD\_TERMINATE\_PLUGIN 12](#_Toc209266793)

[SD\_LOAD\_DATA 12](#_Toc209266794)

[SD\_SAVE\_DATA 12](#_Toc209266795)

[SD\_CONFIGURE 13](#_Toc209266796)

[SD\_STATE\_CHANGING 13](#_Toc209266797)

[SD\_DRAW 13](#_Toc209266798)

[SD\_COMMAND 14](#_Toc209266799)

[SD\_WINDOW\_MESSAGE 14](#_Toc209266800)

[SD\_MODE\_CHANGED 14](#_Toc209266801)

[SDHostMessage 18](#_Toc209266802)

[SD\_GET\_OBJECT\_DIRECTORY 18](#_Toc209266803)

[SD\_GET\_PLUGINS\_DIRECTORY 18](#_Toc209266804)

[SD\_LOAD\_IMAGE 19](#_Toc209266805)

[SD\_REDRAW 19](#_Toc209266806)

[SD\_GET\_STATE 19](#_Toc209266807)

[SD\_SET\_STATE 19](#_Toc209266808)

[SD\_GET\_CONTAINER\_RECT 19](#_Toc209266809)

[SD\_GET\_OBJECT\_RELATIVE\_RECT 20](#_Toc209266810)

[SD\_SET\_TOOLTIP\_TEXT 20](#_Toc209266811)

[SD\_REGISTER\_FILE 20](#_Toc209266812)

[SD\_SET\_TOOLTIP\_TEXT 20](#_Toc209266813)

[SD\_CREATE 20](#_Toc209266814)

[SD\_PACK\_IMAGE 21](#_Toc209266815)

[SD\_SET\_OBJECT\_POS 21](#_Toc209266816)

[SD\_ENUMERATE\_OBJECTS 21](#_Toc209266817)

[SD\_DELETE\_OBJECT 22](#_Toc209266818)

[SD\_OBJID\_FROM\_GUIID 22](#_Toc209266819)

[SD\_SET\_OBJECT\_GUIID 22](#_Toc209266820)

[SD\_USER\_MODE 22](#_Toc209266821)

# Overview

SDPlugins are designed to allow you to extend the functionality of applications that support them such as DesktopX and ObjectBar. Examples of such Plugins would be virtual desktops, MP3 players, analog clocks, email checkers, Recycle bin checkers, printer status monitors, games, disk space monitors, CPU graphs, and anything else that needs to make calls to the system or the Internet and talk back to the object it is attached to or other objects running on the system. An SDPlugin will always attach itself to the object (or bar) that it is part of. And when exported for use by others, it will be included. This is done to decrease support.  
  
This guide is designed to document how SDPlugins are created. It is designed for software developers who are already familiar with a programming language and how to create DLLs.

# General information

Plugins must be compiled with **stdcall** calling convention.  
  
If you are compiling with MS Visual C++ you should also use “Multithread DLL” run-time library for Release version and “Debug Multithread DLL” run-time library for the Debug version, since the required modules are distributed with DX anyway and this way the plug-in will be a lot smaller.  
  
The constants mentioned in this document are declared in the [SDPlugin.h (click to download)](http://www.stardock.com/products/desktopx/documentation/SDPLugin/SDPlugin.h) header files.

# Plugin Lifetime

This chapter discusses the differences between the plugin lifetime and the instance (of the plugin) lifetime.  
  
The plugin dll is loaded when an object needs the plugin. The SD\_INITIALIZE\_MODULE is sent through the SDMessage callback (see below).  
  
The plugin dll is unloaded when the last object using the plugin is deleted or the plugin removed, so no object needs the plugin. SD\_INITIALIZE\_MODULE is sent through the SDMessage.  
  
When a plugin is selected to be used in an object an instance of the plugin is created and associated to the object. More instances can be added to an object (i.e. an EmailCheck plugin can be added more times to handle more email accounts).  
  
When an instance of the plugin is created the plugin receive SD\_CREATE\_PLUGIN. Note that at this point the plugin should not start its action (i.e. a clock plugin should not start displaying the hour), i.e. the DX object is not yet created. This message is sent for the plugin to initialize instance data, allocate variables and such.  
  
The plugin instance actually starts working when it received the SD\_INITIALIZE\_PLUGIN message. In fact now it receives the object HWND and the rect size of the object. It means that the object has been created and the plugin should start “running”.  
  
After the instance is created (SD\_CREATE\_PLUGIN ) but before it is running (SD\_INITIALIZE\_PLUGIN), the plugin can be configured. This can happen in three ways:

* Directly by the user: the user just clicked “Configure” in the plugin section of the object properties. At this point SD\_CONFIGURED is sent to the plugin and it would tipically open a config dialog to configure the instance settings.
* When an object is loaded from the disk (i.e. the user loads a theme or package with objects using your plugin). Now the instance config data is typically loaded config data files that the plugin previously saved in the theme directory and the host saved them in themes or packages. The plugin knows that it should retrieve the saved instance config data when it receives the SD\_LOAD\_DATA message.
* By the host through SD\_DUPLICATE\_PLUGIN message. This is probably the only hard part of the whole specs. Basically SD\_DUPLICATE\_PLUGIN is used overall by DesktopX to support special operations, when an object is duplicated and when an object is configured. On cloning an object, a new object is created and so, new plugin instances are created. These new instances need to be configured the same as the original object/instances. In fact SD\_DUPLICATE\_PLUGIN contains references of the original plugin instance to copy (in the sense of “configuring” it). Why is SD\_DUPLICATE\_PLUGIN needed on configuring an object?? When you open the object properties panel, you can still use all existing objects and so the object you are configuring. You can change its graphics, settings, add, remove, configure plugins and the configured object is still working the same way. All changes are applied when you actually click “Apply” or “Ok”. If you click “Cancel” the changes are ignored. To support this mechanism DesktopX performs these tasks:
  + When the user opens the object properties panel, a new “dummy” plugin instance is created. SD\_DUPLICATE\_PLUGIN is sent to configure this new instance as the original instance.
  + This “dummy” instance is used to be configured, but will never actually receive SD\_INITIALIZE\_PLUGIN.
  + When the user click Apply, the original object is destroyed with all his plugin instances. A new object is created with new settings (graphics, options, and new added plugins, removed plugins or differently configured plugin instances). New plugin instances are now created for the object. These are the final “real” instances and they are configured with SD\_DUPLICATE\_PLUGIN: the new plugin instance need to be configured the same as the “dummy” instances used in the properties panel.
  + The host destroys the “dummy” instances, while the new instances receive SD\_INITIALIZE\_PLUGIN and start working.

This could sounds difficult, but it is just to explain how the host works with plugins, you can simply ignore all of the above explanations and just follow and respect the plugin messages, just keep in mind that SD\_CREATE\_PLUGIN initializes the plugin config data (you would set this to default settings), SD\_CONFIG configures this data through the user’s input, SD\_DUPLICATE\_PLUGIN configures a plugin instance the same way as another plugin instance, SD\_LOAD\_DATA configures the plugin instance from disk saved data. SD\_INITIALIZE\_PLUGIN finally “starts” the plugin (i.e. a timer starts running, the email check plugin starts checking for emails, etc.).  
  
When the plugin instance is removed by the object, or the object is deleted, SD\_TERMINATE\_PLUGIN is sent for each instance.  
  
Note that there are plugin’s own configuration data and config data needed by the host. I.e. if the plugin is a drawing plugin, that is, it draws the object graphics (i.e. analog clock plugin), the host needs to know it to correctly allocate the graphic buffers to be used by the plugin. So, the host needs to know if the plugin will sublcass the object. Plugins pass this information to the host in two cases:

* When the plugin instance is created (SD\_CREATE\_PLUGIN). The plugin should typically set default flags and/or flags that won’t change.
* When the plugin is configured (SD\_CONFIG). The plugin can here change the current passed flags (saved in SD\_CREATE\_PLUGIN).

This way plugin options are “per instance” rather than “per plugin”. So, an instance could be set as drawing plugin, another instance of the same plugin could be not a drawing plugin (same for the subclass option and the other ones).  
  
As mentioned earlier, plugins can store data in external files (config files, graphics, etc). These files should be stored in the directory returned by SD\_GET\_THEME\_DIRECTORY.

## SDMessage

*BOOL SDMessage(DWORD objID, DWORD \*pluginIndex, UINT messageID, DWORD param1, DWORD param2)*

|  |  |
| --- | --- |
| Parameter | Description |
| objID | Unique object identifier. It must be passed to the host when using callbacks |
| pluginIndex | Object identifier used by the plugin. It is modified by the plugin itself and is passed by the host on all messages. The plugin can use it to directly reference the plugin local data and settings |
| param1 | message-specific |
| param2 | message-specific |

SDMessage is used by the host to send to the plugin callback messages.  
  
Return *TRUE* if the message is supported and is processed.  
Return *FALSE* if the message is unsupported or not processed.

### SD\_INITIALIZE\_MODULE

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | BOOL (\_\_stdcall \*)(UINT, DWORD, DWORD) | SDHostMessage function pointer |
| param2 | HINSTANCE | dll instance |

This is called once when the plugin module is loaded by the host (i.e. should be called when a plugin is going to use this module and no other plugins already exist that use this module.  
  
The *param1* contains a pointer to the SDHostMessage function the plugin will use to communicate with the host.

### SD\_TERMINATE\_MODULE

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | DWORD | N/A |
| param2 | DWORD | N/A |

This is called once when the plugin module is unloaded by the host (i.e. when the last plugin that uses this module is destroyed).

### SD\_QUERY\_PLUGIN\_INFO

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | SD\_PLUGIN\_INFO\* | Pointer to SD\_PLUGIN\_INFO structure |
| param2 | DWORD | N/A |

This is called when the host needs to obtain information about the plugin. The info is returned in the structure which is passed into *param1*. This structure is allocated by the host.  
  
Returns *TRUE* if the plugin is host compatible. Returns *FALSE* if the plugin isn't compatible with the host.  
  
Note: this message can be received before *SD\_INITIALIZE\_MODULE*, so if the plugin absolutely needs to initialize before even being queried for info, it should use *DllMain* to initialize itself and not *SD\_INITIALIZE\_MODULE*.

### SD\_QUERY\_CUSTOM\_STATES

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | char\* | Name of the custom state |
| param2 | int | Index of the custom state |

This is called by the host to enumerate the custom states supported by the plugin.

The name of the custom state should be copied into *param1*. Return TRUE to continue the enumeration, *FALSE* to cancel or stop the enumeration (*param1* will be ignored in this case).  
  
*example:  
  
if(iCustomStateIndex == 0)  
 strcpy(szCustomState, “No Mail, Default”);  
else if(iCustomStateIndex == 1)  
 strcpy(szCustomState, “No Mail, Mouseover”);  
else if(iCustomStateIndex == 2)  
 strcpy(szCustomState, “You’ve Got Mail, Default”);  
 else if(iCustomStateIndex == 3)  
strcpy(szCustomState, “You’ve Got Mail, Mouseover”);  
else  
 return FALSE;  
return TRUE;*

### SD\_GET\_OBJECT\_SIZE

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | SIZE\* | Pointer to a size structure |
| param2 | DWORD | N/A |

This message is sent if the plugin is a drawing plugin. The plugin has been already associated to the object and it is already configured. The host must now know the desired object size.

### SD\_CREATE\_PLUGIN

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | DWORD\* | Plugin flags |
| param2 | DWORD | N/A |

*pluginIndex* should be initialized here, allocating memory and storing pointers to that memory as needed.

### SD\_DESTROY\_PLUGIN

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | DWORD | N/A |
| param2 | DWORD | N/A |

*pluginIndex* should be de-initialized here; any memory it associated with it should be freed, along with any other local data that was allocated.

### SD\_DUPLICATE\_PLUGIN

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | DWORD | ObjectID of the original plugin |
| param2 | DWORD\* | pluginIndex of the original plugin |

example:  
  
CopyMemory((\*pluginIndex), (\*pluginIndex\_OriginalPlugin), sizeof(MYSTRUCT)); //copies over the memory from the old plugin’s PIN data to the new plugin’s PIN data, so they’re identical  
//Note further processing might be needed if more pointers/handles are located within the memory pointed to by pluginIndex!  
//OR, if all that’s stored in pluginIndex is a number or something that has nothing to do with memory:  
(\*pluginIndex) = (\*pluginIndex\_OriginalPlugin);

### SD\_INITIALIZE\_PLUGIN

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | SD\_PLUGIN\_INIT\* | N/A |
| param2 | DWORD | N/A |

The plugin should initialize plugin local data, timers, etc.

### SD\_TERMINATE\_PLUGIN

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | DWORD | N/A |
| param2 | DWORD | N/A |

This message should just be used to notify the plugin that it is being uninitialized. The plugin must here destroy timers, etc.

### SD\_LOAD\_DATA

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | char\* | Instance ID of the plugin |
| param2 | DWORD\* | Plugin flags |

This message is sent when the plugin should load the configuration data for the object.

In param1 the host writes the identifier of the plugin object instance that the plugin itself wrote in *SD\_SAVE\_DATA*. This is needed to identify plugin instances into different objects.

### SD\_SAVE\_DATA

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | char\* | Instance ID of the plugin |
| param2 | BOOL | TRUE if the user is exporting, FALSE otherwise |

This message is sent when the plugin should save the configuration data, i.e. before unloading or before exporting.

The plugin will need to send *SD\_REGISTER\_FILE* messages to the host for each file that needs to be associated with the object.  
  
The plugin should write in *param1* an instance identifier that the plugin will use on *SD\_LOAD\_DATA* time to identify the different instances if the plugin is working for more objects.  
  
If the user is exporting, the plugin could choose to save config data differently, i.e. an email notify plugin won't save the email login/pass information, since the objects could be distributed around the web.

### SD\_CONFIGURE

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | DWORD\* | Plugin flags |
| param2 | HWND | Handle of the parent window |

as defined in SD\_CREATE\_PLUGIN, the plugin can here modify its flags depending on the configuration.  
  
The plugin needs to popup a dialog to be configured.

### SD\_STATE\_CHANGING

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | char\* | the state the plugin is changing to |
| param2 | DWORD | N/A |

Here the developer can either return FALSE which would allow the host change the item/object’s state to the new state, or the developer can return TRUE and prevent the host from changing the item’s state.  
  
If the developer desires to return TRUE to prevent the host from changing the item’s state, the plugin will probably want to set it manually to one its custom states or another standard state. To do this the plugin must copy the new state string identifier in param1. If an empty string is set, the plugin will simply cancel the state changing request, nor the original state nor another modified one is applied.  
  
Some standard state identifiers are defined in SDPlugin.h.

### SD\_DRAW

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | SD\_DRAW\_INFO \* | Pointer to a SD\_DRAW\_INFO structure |
| param2 | DWORD | N/A |

The plugin must draw into the passed HDC. Note that the plugin shouldn’t consider what is already in the HDC. It should draw a complete frame, eventually using pink area (255, 0, 255) when it needs transparency. After the plugin returns the host will actually draw in the background, but this is not plugin task.  
  
If the host is DesktopX, the HBITMAP will always be a 32 bit memory DIB. In this case the plugin can directly modify the color arrays and if needed set \*bPerPixelAlpha TRUE to enable the host to render in per-pixel alpha blending mode. The bitmap so drawn is then passed to the DesktopX renderer, that is, the plugin will never draw to the HDC directly.  
  
If the plugin draws in opaque mode (not using per pixel information), it can use pink regions (RGB = 255, 0, 0) to indicate transparent regions.

### SD\_COMMAND

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | DWORD | N/A |
| param2 | DWORD | N/A |

This message is sent to plugins registered to perform a custom action with the SD\_FLAG\_CUSTOM\_ACTION flag.

### SD\_WINDOW\_MESSAGE

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | MSG\* | Pointer to a MSG structure |
| param2 | DWORD\* | Return value |

The plugin receives this message from the host if it has registered itself to hook window messages.  
  
Note that in the current implementation the time and pt members are not used.  
  
The message is not yet processed by the host. The plugin can simply retrieve the info or modify the message parameters.  
  
Returns TRUE if the message must be not processed by the host. The plugin must write in param2 the desired value to return to the window proc.  
Returns FALSE if the message must be processed by the host.

### SD\_MODE\_CHANGED

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | BOOL | TRUE if the host is in User Mode, FALSE if in Edit/Design mode |
| param2 | DWORD | N/A |

The host sends this message to all plugins when the current mode is changed.

# Plugin flags

The flags are used to configure the plugin instance. The default flag is set on SD\_CREATE, then it can be configured on SD\_CONFIGURE and SD\_LOAD\_DATA.  
  
These are the possible values:

SD\_FLAG\_DRAW = The plugin will draw directly the object appearance, it is a drawing plugin (i.e. analog clock). If a drawing plugin is selected for an object, the user won't be able to directly setup the messages/states in the GUI. The appearance will be only configurable through the plugin configuration dialog.  
SD\_FLAG\_SET\_SIZE = This flag can be used by drawing plugins, if the plugin needs to determine the initial object size, instead of using the one specified by the user in the object size settings (that will be disabled). In this case the plugin will receive SD\_GET\_OBJECT\_SIZE when the host needs to know those values to create or modify the window or do other internal adjustments.  
SD\_FLAG\_CUSTOM\_ACTION = The plugin will perform a custom action (i.e. control Winamp). It is a command plugin. If a command plugin is selected for an object, the command class will switch automatically to "Custom class" and the "Configure" button will open the plugin configuration panel. See SD\_COMMAND below for more details.  
SD\_FLAG\_CUSTOM\_STATES = The plugin has custom states that it wants host to observe\*/  
SD\_FLAG\_ONLY\_CUSTOM\_STATES = Items/objects using this plugin should only show the custom states when the users is editing the item's appearance.  
SD\_FLAG\_FILTER\_STATE\_CHANGES = The plugin receives SD\_STATE\_CHANGING SD\_STATE\_CHANGED that inform the plugin about state messages. The plugin can modify and/or remove them as they get sent.  
SD\_FLAG\_SUBCLASS = The plugin should set this flag to indicate if it wants to get and set messages from the window procedure sent from the host through SD\_WINDOW\_MESSAGE. Note that the plugin MUST NOT subclass the object window. This hooking service should be used instead.  
SD\_FLAG\_DRAW\_PPALPHA = The plugin informs the host that it will use the 4th byte of the 32bit HBITMAP to supply per-pixel alpha blending information.  
SD\_FLAG\_MORE\_INSTANCES = By default the host will avoid that more instances of the same plugin are added to an object. This flag will make it possible to add more than one instance.

**Typical order of messages**

|  |  |
| --- | --- |
| **Message** | **What it does** |
| SD\_QUERY\_PLUGIN\_INFO | Can be called only for querying purpose, in this case the other messages won’t be called. |
| SD\_INITIALIZE\_MODULE | Only called if this is the first plugin from this DLL being loaded. |
| SD\_CREATE\_PLUGIN | Setup pluginIndex, allocate memory |
| SD\_LOAD\_DATA |  |
| SD\_CONFIGURE | These are the two ways a plugin instance can be configured, infact the plugin can here modify the plugin flags. |
| SD\_QUERY\_CUSTOM\_STATES | Custom states actually depend on the plugin configuration, so these should be set when the plugin is first time created and when it is configured. |
| SD\_INITALIZE\_PLUGIN |  |
| SD\_QUERY\_CUSTOM\_STATES | Custom states actually depend on the plugin configuration, so these should be set when the plugin is first time created and when it is configured. |
| SD\_SAVE\_DATA | Save plugin data to object directory or plugin directory. Can be called at any time between SD\_INITIALIZE and SD\_TERMINATE. |
| SD\_TERMINATE\_PLUGIN | Uninit plugin, stoping any times it has, etc.. |
| SD\_DESTROY\_PLUGIN | Destroy pluginIndex, deallocate memory. |
| SD\_DEINIALIZE\_MODULE | Only called if this is the last plugin from this DLL being unloaded. |

In order to support Apply/Cancel/Ok in the object configuration panel the plugin is requested to duplicate its plugin data through SD\_DUPLICATE\_PLUGIN.

## SDHostMessage

SDHostMessage is used by the plugin to communicate with the host or request certain tasks (i.e. move an object, get the theme directory, etc.).  
  
The host will export this function as:  
  
DWORD SDHostMessage (UINT messageID, DWORD param1, DWORD param2)  
  
The pointer to this function is passed to the plugin in the SD\_INITIALIZE\_MODULE message.  
  
Here is a list of supported SDHostMessage messages:

### SD\_GET\_OBJECT\_DIRECTORY

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | char\* | char buffer to receive the object directory |
| param2 | DWORD | Object ID |

The object directory is the directory that contains the plugin custom files if needed, the files that will be registered through SD\_REGISTER\_FILE.  
This folders should be part of the theme, and should be the folder where the object/theme is being saved to. E.g. something like DesktopX\CurrentTheme\ etc etc.. This is there to save object/theme-specific “settings” which change from object to object, theme to theme, like the color of a clock or something like that.

Example:

char folder[MAX\_PATH];  
SDHostMessage(SD\_GET\_OBJECT\_DIRECTORY, &folder, objID);

### SD\_GET\_PLUGINS\_DIRECTORY

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | char\* | char buffer to receive the plugins directory |
| param2 | DWORD | Object ID |

Host would fill szDir with the path of the common plugins folder for DX or for OB… like DesktopX\SDPlugins\ or ObjectBar\Plugins\. The plugin could then create subfolders for itself as needed, so it can store data common to everything that uses this plugin.  
  
Again this is here so the plugin knows where to save general “data”, which should be common no matter what theme/object uses it, such as for the command line plugin’s history, etc.

### SD\_LOAD\_IMAGE

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | SD\_IMAGE\_INFO\* | SD\_IMAGE\_INFO structure |
| param2 | DWORD | N/A |

Loads a .ico, .bmp, .jpg or .png (all formats supported) file to a 32 bit bitmap.

You can path a width to extract a specific icon size if the file is an icon.

### SD\_REDRAW

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | DWORD | Object ID |
| param2 | DWORD | N/A |

Sends a request to update/redraw the object (i.e. the clock plugin would call this each second).  
  
The plugin will receive back a SD\_DRAW message when the host is ready to have the object redrawn.  
  
Depending by the implementation of the host, the plugin could receive SD\_DRAW back immediately or not.

### SD\_GET\_STATE

Param1 = objID;  
Param2 = (DWORD) szCurState;  
  
The host copies in szCurState the current state identifier.

### SD\_SET\_STATE

Would pass a string to the host to set the current state to that custom state if it exists.  
  
Param1 = objID;  
Param2 = (DWORD) szNewState;

### SD\_GET\_ABSOLUTE\_RECT

Param1 = objID;  
Param2 = (RECT\*) lprcContainerRect;  
  
The plugin should pass a pointer to a RECT struct (lprcContainerRect) in Param2, which the host will fill with the rectangular area of the screen occupied by the container HWND, the object/bar that owns the plugin whose ID is passed in Param1.

### SD\_GET\_\_RELATIVE\_RECT

Param1 = objID;  
Param2 = (RECT\*) lprcObjectRect;  
  
The plugin should pass a pointer to a RECT struct (lprcObjectRect) in Param2, which the host will fill the rectangular area occupied by the object in the HWND (the container). This is could not be the same rectangle passed on SD\_DRAW, because that rectangle refers to the area occupied in the passed HDC.  
  
Allows the plugin to process mouse handling and such if it subclasses, and could also be used for some other purposes as well.

### SD\_SET\_TOOLTIP\_TEXT

Param1 = objID;  
Param2 = (char\*) newText;

### SD\_REGISTER\_FILE

char szFilename[MAX\_PATH];  
Param1 = (DWORD) szFilename;  
  
The plugin sends this message to the host while processing the SD\_SAVE\_DATA message. This message will need to be sent once for every file that needs to be registered. The plugin will need to register all files that will need to be saved on every SD\_SAVE\_DATA message.

### SD\_SET\_TOOLTIP\_TEXT

Param1 = objID;  
Param2 = (char\*) newText;

### SD\_CREATE

param1 = (DWORD) (char\*) szObjectDefinition;  
param2 = (DWORD) IOversion;  
  
The plugin creates an object. It must make sure all required graphic and sound files are present in the theme directory.  
  
szObjectDefinition contains the multiline (\r separated) object string from “OBJECT” to “!OBJECT”.  
  
IOversion is the version of the IO syntax used in szObjectDefinition. You should check what version is used by the latest DX version and use the same number and latest syntax. You can find the version number in the first line of theme.ini.

### SD\_PACK\_IMAGE

HBITMAP \*hb = (HBITMAP \*) param1;  
  
Converts a double height 32 bit bitmap to a single height 32 bit bitmap with packed alpha.

### SD\_SET\_OBJECT\_POS

typedef struct SD\_SOP\_INFO\_  
{  
int x;  
int y;  
int width;  
int height;  
DWORD flags; // any combo of these flags:  
// SD\_SOP\_MOVE: x and y are used  
// SD\_SOP\_SIZE: width and height are used (drawing plugins only)  
// SD\_SOP\_SHOW: shows the object  
// SD\_SOP\_HIDE: hides the object  
// SD\_SOP\_ONTOP: sets the object on top of its zorder class, i.e. if it is a desktop level object, it will be placed on top of all other desktop level objects  
} SD\_SOP\_INFO;  
  
Param1 = objID; // object ID  
Param2 = (SD\_SOP\_INFO\*) &sop; // destination

### SD\_ENUMERATE\_OBJECTS

Enumerates all the loaded objects.  
  
BOOL (CALLBACK \*EnumObjectsProc) (DWORD ObjID, DWORD lParam);  
{  
To continue enumeration, the callback function must return TRUE; to stop enumeration, it must return FALSE.  
}  
param1 = (DWORD) EnumObjectsProc;  
param2 = (DWORD) appDefinedValue;

### SD\_DELETE\_OBJECT

Param1 = objID; // object ID  
Param2 = (DWORD) (BOOL) bDeleteChildren; // set TRUE to delete also all children of the object referred by objID  
  
Returns TRUE if no errors occurred, returns FALSE if: - the objID was invalid or – the object was being configured.

### SD\_OBJID\_FROM\_GUIID

### SD\_GUIID\_FROM\_OBJID

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| param1 | DWORD | Object ID |
| param2 | char[100] | The Object GUIID (out) |

The host copies in szObjectGUIID the object ID defined by the user in the GUI.

### SD\_SET\_OBJECT\_GUIID

char szObjectGUIID[100];  
Param1 = (DWORD) objID;  
Param2 = (DWORD) szObjectGUIID; // inThe host sets the object ID defined in the GUI as szObjectGUIID. This message is the opposite of SD\_OBJID\_FROM\_GUIID.

### SD\_USER\_MODE

Returns TRUE if DesktopX is in User mode, FALSE if it is in Design/Edit mode.

Mail Notification Plugin SDK Sample

This plugin is a complete SDPlugin sample that features config load/save, messaging and configuration.  
  
You can download the [plugin here](http://www.stardock.com/products/desktopx/documentation/SDPLugin/SDMailNotify.zip) (.ZIP).

Scriptable Mail Notification Plugin SDK Sample

This plugin is a scriptable version of the old SDMailNotify plugin. For an user perspective discussion please refer to the [Developer's Guide: Scripting - Scriptable plugins](http://www.stardock.com/products/desktopx/help/dev_guide_3a_scripting.htm#Scriptable_Plug-ins) section. For a real sample you can import the widget MailCheck.exe into DesktopX Builder and check how its script works with regards to plugin interoperability.  
  
You can download the [plugin source code here](http://www.stardock.com/products/desktopx/documentation/SDPLugin/SDMailNotify2.zip) (.ZIP).  
  
There are a few things to note while you look at the source code:

* There is no configuration management. It is left to the actual script.
* It uses the special SD\_SCRIPTABLE\_PLUGIN\_EVENT message to fire script events. This is supposed to solve threading issues as compared to actual native event mechanisms. With this regard, DesktopX is serving as an event proxy.
* You should compile using the Debug or Release MinSize configuration.
* There is a custom implementation of DllRegisterServer all scriptable plugins should comply with. It is basically registering the plugin information using HKEY\_CURRENT\_USER rather than HKEY\_LOCAL\_MACHINE. This allows auto-setup when running a gadget under limited user privileges.