Programmer's Guide for Doodle's Screen Saver v1.8

Table of Contents

1. General overview

Another screen saver. Why? Well, the main goal was to have a screen saver for OS/2 with a clean API, and to be able to control the screen saver from third party applications, like WarpVision, so the screen saver will not start while watching porn. ;)

The core of the saver is a DLL, called SSCore.DLL. This DLL contains all the screen saver functions. To use the screen saver, there must be a process loading this DLL, and calling its initialization function (see the API later). It will start a new thread in the caller process, and start screen saving, as long as the process is alive.

Right now, this application is the PM, because the SSCore.DLL is loaded and initialized from a WPS class, called WPSSDesktop. This WPS class replaces the old WPDesktop class, and removes its Lockup pages, replaces them with Screen Saver pages, and starts the screen saver if somebody asks for locking up the desktop.

2. Screen Saver modules

2.1. About the modules

The screen saver uses external modules for the actual screen saving. Once the user doesn't touch the keyboard or mouse for a given time, the screen saver core will load a screen saver module (according to the current settings), and tell it to start the saving. It's completely left to the module how it does it. It can use sound effects, draw to screen, notify an external device, or do anything it likes.

When the screen saving is in progress, and the user touches the mouse or keyboard, the screen saver core either tells to module to stop saving, or, if the password protection is turned on, tells the module to ask the user for password, and only stops the saving if the password matches.

One could ask why is it this way, that the module asks the password from the user. Well, it was thought that this way we can have several screen saver themes (actually, different modules), where the asking of password „fits” into the look of the screen saving.

Now, let's see the API of screen saver modules, let's see the functions one has to implement in a DLL in order to create a module for this screen saver! Please note, that you can always look at the example module to see an example of the implementation.

2.2. The screen saver module API

As a general guide, we can say that every function should return an int value as a status report. Please see the SSMODULE_* defines in SSModule.h file about valid return codes!

There are mandatory functions to implement, and there are optional ones. First, here is the description of the mandatory functions, which must be implemented in order to have a valid Screen Saver module.

2.2.1. The mandatory functions

*int SSModule_GetModuleDesc(SSModuleDesc_p pModuleDesc, char *pchHomeDirectory);*

This is the function which is called by SSCore to get information about the screen saver module. The function should fill the structure pointed by pModuleDesc with information about itself, and return SSMODULE_NOERROR.
The structure contains the module version, module name and module description. All these are for informational purposes. The structure also contains two fields, the bConfigurable and the bSupportsPasswordProtection. These are also informational fields only for the application configuring the screen saver. From this fields, it will know if this screen saver module can be configured or not, and if it will be able to support password protection or not, if the enabled password protection will be in effect for this module.

The module version number reported in this structure is built from two numbers, iVersionMajor and iVersionMinor. The minor version number will be extended with leading zero if it's smaller than 10, so make sure you give numbers 1 and 50 if you want to see v1.50, and not 1 and 5, because the later will result in v1.05.

The pchHomeDirectory contains the full path (with a trailing backslash) to the directory where the screen saver was installed. This can be used if, for example, the module wants to load NLS strings before returning from this function.

If the module exports the *SSModule_SetNLSText()* function (see description later), that will be called before this function (actually, before any other function), so the module should already know the current language of the screen saver core. It can then look for a NLS file for that language in its private folder (usually <*pchHomeDirectory*>\Modules\<*ModuleName*>*.msg), and if found, it can return the module description in the selected language.

*int SSModule_Configure(HWND hwndOwner, char *pchHomeDirectory);*

This function is called when the user wants the module to be configured. If the module cannot be configured it should return SSMODULE_ERROR_NOTSUPPORTED. Otherwise it should show a dialog window and let the user do its configuration. The current settings of the module should be loaded from/saved to its own configuration file in the %HOME%\.dssaver directory, if possible. If that directory does not exist, it can use the DSSaver Global Home folder (pchHomeDirectory), or optionally it can use the OS/2 INI fiels too.

The configuration window should be made so that its owner will be hwndOwner.

*int SSModule_StartSaving(HWND hwndParent, char *pchHomeDirectory, int bPreviewMode);*

This is called when the module should start screen saving. The thread which calls this function is initialized for PM, but you have to return from this function once you've started your module, so please don't create windows from this thead. Treat it as it would have not been initialized for PM. What you should do is to check if you're already running (return error code if so), start a new thread which will do the job, and return SSMODULE_NOERROR if all went ok.

The screen saver window itself should be done so that its parent will be hwndParent, and it should read the current configuration of the module before starting up. The current configuration should be loaded from OS2.INI, or from a private config file in %HOME%\.dssaver. If that directory does not exist, it can fall back to load the configuration file from the DSSaver Global Home folder, which is passed in the pchHomeDirectory parameter. The bPreviewMode tells if the module is started only for preview mode, in a small window, or if it's started for real screen saving.

Of course, if it's started for preview mode, the module should not hide the mouse, should not initialize itself to be system modal, and should not turn off the monitor. :)

If the module is started for non-preview mode, the hwndParent will be HWND_DESKTOP.

If you're asked to start for preview mode, the best way to do so is to subclass the hwndParent window, and do your job in there. See the example modules for how to do it.

If you're asked to start the real saving, you should create a system-modal window, with the size of screen (*not* the size of the desktop window, because that might be resized by xCenter and other applications), and do whatever you want in that window. Again, see the example modules for how to do it.

int SSModule_StopSaving(void);

This function has to stop the previously started screen saving, and return SSMODULE_NOERROR.

int SSModule_AskUserForPassword(int iPwdBuffSize, char *pchPwdBuff);

This function is called by SSCore if the module is already running, and the current screen saver settings have password protection turned on. (If the module is currently not running, this function should return error.)

If your module does not support password protected screen saving, return SSMODULE_ERROR_NOTSUPPORTED. Otherwise, this module should notify the saver window that it should ask for password, and return the entered password in pchPwdBuff.

If this function returns SSMODULE_NOERROR, the SSCore will compare it with the password set by the user, and either call SSModule_StopSaving() if the password matches, or call SSModule_ShowWrongPasswordNotification() if the password doesn't match. If this function returns an error, the SSCore will call SSModule_StopSaving() so it won't happen that the user cannot exit from screensaver.

int SSModule_ShowWrongPasswordNotification(void);

This function is called when the user enters wrong password. This should show a window to the user telling that a wrong password has been entered, or do it in any other way, which fits into the look of the screensaver.

2.2.2. The optional functions

The following functions are optional. One can decide if implements them or not.

Currently, there are two group of optional functions. The first one is about DPMS support. Implementing these functions has the advantage that when the system will switch to a DPMS state, where the monitor is turned off, the module will not eat the CPU anymore.

The other „group” of optional functions consists of two functions.

The first is to support different national languages. If it's implemented, the screen saver core will tell the module some common texts in the current language, that can be used at asking for the password and

showing wrong password notification. It will also tell the current locale string (like „de_DE_EURO” or „en_US”), which can be used by the module to get additional strings in the current language from its database, if supported.

The second one is a general function to tell the module some common settings and preferences that would be good if the module would honor. More about it later.

int SSModule_PauseSaving(void);

This is the function which is called by SSCore when it switches into a DPMS state in which the monitor is blanked. This notifies the module that it should pause all of its CPU-consuming processing, as the result will not be visible anyway.
This helps saving the battery life of laptops.

int SSModule_ResumeSaving(void);

This is the function which is called by SSCore when it switches back from a DPMS state in which the monitor was blanked. This notifies the module that it should resume all of its CPU-consuming processing.
If SSCore calls PauseSaving, it will call ResumeSaving too, even if the very next call will be StopSaving!

*int SSModule_SetNLSText(int iNLSTextID, char *pchNLSText);*

If this function is implemented by a module, SSCore calls it after loading the module, and calls it before unloading it.

Using this function, SSCore tells some common strings translated to the current language, so these strings can be used by the modules to show messages in the current language.

The first parameter tells what kind of string it is, and the second is a pointer to the string itself.

If the second parameter is not NULL, the module should save a copy of the given string in its own data area for later use. If the second parameter is NULL, then the module should free that copy. (See notes about it later!)

The SSCore will call this function (if exported) before starting the saving and also before calling SSModule_Configure() or SSModule_GetModuleDesc(). It will call this function for every

SSMODULE_NLSTEXT_* constant (see *SSModule.h* for **SSMODULE_NLSTEXT_*** defines!), and tell all the known NLS strings to the module.

Also, it will call this function after the saving has been stopped (and when the SSModule_Configure() call has returned), but with a second parameter of NULL for every SSMODULE_NLSTEXT_* constant. This way, the module should free all the resources it allocated for NLS texts. (Again, see notes about it later!)

The string ID of 0 is a special one (SSMODULE_NLSTEXT_LANGUAGECODE), which tells the current locale. It will be something like „en_US" or „de_DE_EURO". If the module wants to display some messages in the current language, but that string is not a common one, the module can use this string to look up the translation of the text in its own database for the current language.

There is one thing that you should note. The SSModule_Configure() can be called while the module is running (in preview mode), and this function will be called before and after SSModule_Configure(), too. This gives us two problems:

- It is possible that the NLS texts will be changed while the module is running, and while the module wants to access them. This makes it necessary for the modules to serialize the access to their local copy of NLS texts with a mutex semaphore. See the source code of the modules for examples of how to do it (implementation of SSModule_SetNLSText() API, and at every access of the local *apchNLSTextArray*)!
- Also, it can be that the SSCore tells the NLS strings to the module before starting the preview mode, and then the user presses the Configure button in the GUI, so SSCore will tell again the current NLS strings, and call the SSModule_Configure(). When the configuration is over, and the SSModule_Configure() returns, the SSCore will call this API again with NULLs to free the resources it gave before calling SSModule_Configure(). This would destroy the module's local copy of NLS strings, so there would be no NLS strings left for the running preview mode.
  This is not a real problem, if your module doesn't use extra NLS strings in preview mode, because the current NLS strings are for password protection only, which is not used in preview mode. However, if your module uses extra NLS string based on the current language code told here, you should ***not*** free your resources when this call is called with NULL, but leave that to the DLL uninitialization code, when it's sure that the NLS strings are not needed anymore!

*int SSModule_SetCommonSettings(int iSettingID, void *pSettingValue);*

This is the function which is called by SSCore after the SSModule_SetNLSText() series of calls, to tell the module some common module settings and preferences. These common settings are things that are set by the user at the main screen saver configuration pages in a hope that the active screen saver module will honor and support them.

This function is called (if exported by the module) for all the available common module settings with a setting-dependent kind of parameter casted to the type (*void **) and passed in pSettingValue.

The list of common module settings and their description is the following:

- Setting for treating first keypress of screen saving:
  iSettingID is SSMODULE_COMMONSETTING_FIRSTKEYGOESTOPWDWINDOW
  pSettingValue is an (*int*) casted to (*void **), and its value is **1** if the screen saver module should send the keypresses (WM_CHAR messages) it receives in its main window to its password asker window. This value is **0** if these messages don't have to be recorded and sent to the password asker window. See an example of how to do it in the example modules!

There are no more common module settings yet.

If this function receives unknown settings, it should return SSMODULE_ERROR_NOTSUPPORTED, otherwise it should return SSMODULE_NOERROR.

3. Controlling the Screen Saver

3.1. Using SSCore

Processes can load SSCore.DLL and use its functions. There are three purposes to use this DLL:
- A process can load it because it wants to start/initialize the screen saving.
- A process can load it because it wants to get/set the current settings of the screen saver, configure screen saver modules.
- A process can load it because it wants to temporary disable the screen saving for some reason (watching a movie, for example).
- A process can load it because it wants to get information about user activity. (Event about going into saving mode, or event about getting out of it, info about the time the user took without activity.)

Of course, nothing can be done with the DLL until it has been initialized, but if all goes well, the screen saver is already initialized at system initialization time.

Now, let's see the exported functions of SSCore!

3.2. The SSCore API

int SSCore_GetInfo(SSCoreVersionInfo_p pVersionInfo, int iBufSize);

This function returns version information about the current SSCore module. Right now this info only contains major and minor version numbers, and information about DPMS states supported by the current hardware.
Starting from v1.4, the function also returns a flag telling if Security/2 was found on this system or not.

The function will only fill *iBufSize* bytes of the buffer *pVersionInfo*, so it should be backward compatible.

*int SSCore_Initialize(HAB habCaller, char *pchHomeDirectory);*

This is the function which has to be called in order to start up the screen saver. Calling this function will result in starting a new thread in the calling process. This new thread (the worker thread) will load and start the screen saver modules when the user doesn't touch the input devices for a while. This also means that the caller process should not terminate, otherwise the screen saving will be turned off.

The pchHomeDirectory will be used as the *DSSaver Global Home directory*. The screen saver core will look for valid screen saver module DLLs in the ***Modules*** subdirectory of this Global home directory. You should pass a directory here in which you want to store all screen saver related stuffs, and in which you have a ***Modules*** subdirectory with all the installed screen saver module DLL files. This Global Home directory will be passed to the saver modules too, so if they require additional binary files, they will know the place from where they were loaded.
However, the screen saver configuration file (SSCore.CFG) and all the configuration files will be stored in the „.dssaver” subdirectory of the home directory of the current user, and if that's not possible for some reason, both the screen saver core and the saver modules will fall back to use this directory to store the config files.
So, in short, this is the DSSaver Global Home directory, which stores the binaries, and, if it's not possible to store them in the „.dssaver” subdirectory of the current %HOME%, then it also has the config files.

int SSCore_Uninitialize(void);

This is called to undo everything done in SSCore_Initialize(). This stops all ongoing screensaver modules, stops the worker thread, removes the input hook, and returns.

It should be called if the screensaver process is about to be terminated.
This function returns error if called from a process other that the one initializing the screen saver.

*int SSCore_GetListOfModules(SSCoreModuleList_p *ppModuleList);*

This function will return a list of valid screen saver modules found. It will return the full path of the screen saver module DLL, and also return the description of the module it got by calling the SSModule_GetModuleDesc() of that DLL.

Using this list, the caller application can build a list of available screensavers, and can also configure them using the full path of the screen saver module DLL, by calling the SSCore_ConfigureModule() function.

The list has to be freed up later using the SSCore_FreeListOfModules() call.

int SSCore_FreeListOfModules(SSCoreModuleList_p pModuleList);

This function frees the linked list allocated by SSCore_GetListOfModules().

int SSCore_GetCurrentSettings(SSCoreSettings_p pCurrentSettings);

This function returns the current settings of the screen saver core, by filling the structure pointed by pCurrentSettings.
Please note that the screen saver password is never returned in its clean form, only in an encrypted form, but the length of the encrypted form is always identical to the length of the original string.

*int SSCore_EncryptPassword(char *pchPassword);*

This function encrypts the given string, so that string can be put into the achEncryptedPassword field of the SSCoreSettings_t structure.
This can be used when the user enters a new password. That new password has to be encrypted, and then stored in the current settings.

int SSCore_SetCurrentSettings(SSCoreSettings_p pNewSettings);

This function stores the given new settings (pNewSettings) as the current settings of the screen saver, and saves it in the home directory into the file SSCore.CFG.

Make sure that you have the password encrypted in the structure, or be prepared for not being able to turn off the password protected screen saver with your password!

int SSCore_StartSavingNow(int bDontDelayPasswordProtection);

This function can be used to start the screen saver with the current settings right now. It can be used by the configuration application, to test current screen saver settings, or to start the screen saving instantly.

The only one parameter of this function tells if the password delaying setting should be honored or not. If the parameter is TRUE, then the password asking will not be delayed (if turned on), even if it was told to do so. This is good for cases when the user wants to start the screen saving at once. In those cases it has no sense to delay password protection.
If the parameter is FALSE, then the password asking (if it's turned on) will be delayed if it was told to do so in the settings.

*int SSCore_StartModulePreview(char *pchModuleFileName, HWND hwndParent);*

Once the configuration application wants to show a preview window of a given module, it should call this function. This will start up the given screen saver module DLL in preview mode, in the window hwndParent. The screen saver module will subclass hwndParent, and show its things in that window.

The function returns immediately, and the worker thread will post a WM_SSCORE_PREVIEWSTARTED message to the parent of hwndParent when the preview has been started.

int SSCore_StopModulePreview(HWND hwndParent);

If the user deselects every screen saver module in the configuration application, this function can be called to stop the preview of the last screen saver module.

The function returns immediately, and the worker thread will post a WM_SSCORE_PREVIEWSTOPPED message to the parent of hwndParent when the preview has been stopped.

*int SSCore_ConfigureModule(char *pchModuleFileName, HWND hwndOwner);*

Using this function, the selected screen saver module (pchModuleFileName) can be told to be configured. It can be that the module does not support configuration, in this case the function will return immediately.

int SSCore_TempDisable(void);

This function can be called by third party applications to temporary disable the screensaver for some reason. This function will increment a per-process variable, and also increment a global variable. If the global variable is not zero, the screen saver will not start.

The calling application should always undo the TempDisable by calling TempEnable the same times it has called TempDisable, altought the per-process counter is maintained by SSCore, and if the process forgets to decrease the counter before terminating, SSCore will do it for the application.

int SSCore_TempEnable(void);

This function will undo the effect of SSCore_TempDisable(). You should always call SSCore_TempEnable() the same times you've called SSCore_TempDisable() before!

int SSCore_GetCurrentState(void);

This function will return either SSCORE_STATE_NORMAL or SSCORE_STATE_SAVING, reflecting the current state of the screen saver core. If the screen saving is currently in progress, then it returns SSCORE_STATE_SAVING, otherwise it returns SSCORE_STATE_NORMAL.

*int SSCore_WaitForStateChange(int iTimeOut, int *piNewState);*

This function can be used to wait for the change of the current state. The function itself is basically a waiting on a shared event semaphore. The *iTimeOut* parameter can be used to specify the maximum amount of milliseconds to wait for the state change or use -1 to specify infinite wait time. (See the Control Programmer's Guide for valid timeout values for semaphores!)

If the *piNewState* parameter is not NULL, then it should point to an int variable, which will contain the new current state value (SSCORE_STATE_NORMAL or SSCORE_STATE_SAVING) on return.

The function return SSCORE_NOERROR if the state has been changed, or returns SSCORE_ERROR_TIMEOUT, if there was no state change in the given time interval. It can also return SSCORE_ERROR_INTERNALERROR, if some other error occured.

*int SSCore_SetNLSText(int iNLSTextID, char *pchNLSText);*

This is the function which should be called by the process which initialized the screen saver core, to tell the core some common strings in the current language. If this is called from a different process, the

function will return SSCORE_ERROR_ACCESSDENIED.

The strings given here will be passed to the screen saver module when it is started. This way, the screen saver modules can use the language set for the screen saver system automatically.

For valid string IDs, see the SSCORE_NLSTEXT_* defines in *SSCore.h* file!

Please note that the first ID (SSCORE_NLSTEXT_LANGUAGECODE) is a special one, this should be the current locale name, like „en_US" or „de_DE_EURO", so if a given saver module requires extra strings in the current language, which is not a common one, then it can use this string to get the current locale, and it can look up the translated version of the required string in its own database.

*int SSCore_GetInactivityTime(unsigned long *pulMouseInactivityTime, unsigned long *pulKeyboardInactivityTime);*

This function can be used to see how much time the user took being inactive. The function check the last timestamps when the user has a keyboard or mouse event, and based on the current timestamp it sets the variables pointed by *pulMouseInactivityTime* and *pulKeyboardInactivityTime* with the differences. So, they will contain the time in milliseconds which elapsed without mouse or keyboard activity.

If any of the parameters is NULL, then the function returns SSCORE_INVALID_PARAMETER.

The function return SSCORE_NOERROR if the inactivity times were reported. Returns SSCORE_INVALID_PARAMETER if any of the parameters is NULL. It can also return SSCORE_ERROR_INTERNALERROR, if some other error occured.