

Plug-in Description

The AppController plug-in permits a Chipmunk script to control other Windows applications running on the same computer. It does this by issuing Win32 window-message commands. This permits you to do things like load applications such as Winamp or PowerDVD and control playback (play, stop, pause, fast-forward, and so on) from within Chipmunk. You can even control some operating system functions, or do things like simulate the user pressing the Sleep button on the keyboard.

At the most basic level, you can think of this as providing access to the SendMessage Windows API call. However, the plug-in actually exposes a number of other Win32 API calls which help support the overall goal. For instance, you can opt to use PostMessage instead of SendMessage, there are features to bring the target application to the foreground, or even load the application if it isn't already running. There are also two alternative ways to use the plug-in that greatly simplify issuing keyboard commands compared to handling the SendMessage calls manually.

It is assumed that the user has a basic understanding of the Win32 window messaging APIs. Although a complete discussion of this topic is far beyond the scope of these instructions, a simple description of the basic concepts is provided at the end of the document for those who are unfamiliar with the way Windows works internally.

Important: Use caution with this plug-in. Windows messages lie at the heart of every windows application. Sending unexpected or incorrectly formatted messages could cause the target application to crash or hang.

Contact and Legal Disclaimer

This plug-in is supplied as-is and without warranty of suitability for any purpose. You agree to use it entirely at your own risk. It must be supplied free of charge, and only Home Cinema Solutions is granted distribution rights. All other rights are reserved by Mindmagma.com, copyright ©2006. You may contact the author at the following e-mail address with questions or comments about the plug-in: jon@mindmagma.com

Please do not write asking for help with controlling a particular application or debugging the calls you're issuing to the plug-in. Windows messaging is potentially very complicated, and each application is different – I am unlikely to have your particular software available for testing on my end. Some applications may not be controllable. I do not maintain information about any application. You might try checking the Chipmunk website's download area to see whether someone else has exported a folder for the application you're interested in.

Files and Folders

Chipmunk.plugins.AppController.dll

This is the plug-in component itself. It resides in the Chipmunk plugin directory.

AppController folder

This folder should be created inside the Chipmunk plugin directory. This document and all the files listed below should be stored in the AppController folder.

winamp.xml

This is an exported folder used to control the popular Nullsoft Winamp application. You can import this folder into your Chipmunk tree. The commands are pretty basic – they call utility functions found in winamp.lua. Note that the first command, Initialize Library, should be associated with the Chipmunk OnStartup event. It calls the Lua dofile() command to load the utility functions into the Lua global namespace.

winamp.lua

This file contains many utility functions called by the tree exported as winamp.xml. Winamp is particularly interesting since it publicly supports control messages which are specifically intended to be used for remote control.

powerdvd.xml

This is an exported folder used to control the Cyberlink PowerDVD application, which is commonly shipped with many personal computer DVD players. You can import this folder into your Chipmunk tree. Like winamp.xml, these generally just refer to more complete commands stored in powerdvd.lua. The first command, Initialize Library, should be associated with the Chipmunk OnStartup event.

powerdvd.lua

This is another file of scripted utility functions. It is similar to the Winamp functions except that PowerDVD is primarily controlled through keystrokes. Both character-based and virtual-key-based keystrokes are demonstrated. (What that means is explained later in this document.)

Payload and Variables

The payload ID is always 10. Several payload events are supported and are described in the following sections. All available variables are listed below, although they are not all required for every event.

Variable	Content	Used by which events?
TargetWindowClass	text	(used by all events)
TargetWindowCaption	text	(used by all events)
ExePath	text	(used by all events)
ExeArgs	text	(used by all events)
WaitForLoad	milliseconds	(used by all events)
BringAppToForeground	Blank=false non-blank=true	(used by all events)
Character	text	KeyChar
VirtualKey	hex (0x0000)	KeyVirt
ShiftState	text flags (SCA)	KeyChar, KeyVirt
Msg	hex (0x0000)	SendMessage, PostMessage
wParam	hex (0x0000)	SendMessage, PostMessage
lParam	hex(0x0000)	SendMessage, PostMessage

For the variables which are used by all events, how they're used is the same across the board. Inside the plug-in, the same initial sequence takes place: the application tries to find the target window, and if that succeeds, it brings it to the foreground if necessary, then issues the various window messages related to the payload event.

If the target window is not found but an ExePath was specified, the plug-in attempts to load the application. If arguments are provided, they are supplied to the application being loaded. If the load succeeds, the plug-in waits for a short period of time to allow the target application to get started. Finally, it tries again to find the window. If it fails to do so this second time, the event is aborted, otherwise the window messages related to the payload event are then processed.

Finding the target window uses the Win32 FindWindow API call. I recommend only using one of the two target variables – either TargetWindowClass, or TargetWindowCaption. However, it is perfectly valid to specify both of them if you know them both. You must specify at least one of these two variables, or the plug-in aborts the event immediately. The brief discussion of Windows Messaging at the end of this document explains what the values mean.

Loading an application with ExePath uses a very flexible Windows function to actually start the application. It is the same function that is behind the “Run...” command on the Windows Start Menu, so you can use that to test the value you wish to use with ExePath. You can, of course, supply a simple path and filename to the executable, and it will work fine. However, you can also supply the path to a document that Windows recognizes. For example, supplying a URL would cause your web browser to be loaded, or supplying the path to a playlist would cause Media Player or Winamp to be loaded. On top of that, some applications register “short names” with Windows, and those short names can be used. For example, just the word “Notepad” is enough to allow Windows to locate and start the Notepad executable.

If you do specify a directory or a value such as a URL in ExePath, remember that Lua's syntax requires you to “escape” special characters used in strings – backslashes in particular. On my system this is what must be specified to load winamp:

```
c:\\program files\\winamp\\winamp.exe
```

WaitForLoad is specified in milliseconds (1,000 milliseconds is equivalent to 1 second). Note that the plug-in might not actually wait as long as you specify. It uses a function that monitors the launched application's window message queue. If that queue becomes idle, which signifies the application is ready to receive messages, the wait ends early and processing continues. This is good – the whole purpose of this plug-in is to send messages, so minimizing the wait-time makes sense. This also implies that you can set relatively large wait-times (10 or 15 seconds or more) to allow especially large or slow applications to load, or to accommodate programs that must do things like access slow physical hardware (for example, mount and read a DVD), without requiring you to “fine

tune” your delay time too much, since the plug-in won’t waste time waiting when the target application finally becomes available.

The `BringAppToForeground` variable uses the `SetForeground` API call. This is similar to using the “Restore” system menu command for a given application. If you don’t create this variable at all, or if you create it but leave it blank, the call will not be executed. If you put any value into this variable (such as “1”), the function will be called before messages are sent.

Payload Event: SendMessage

This payload event is used to execute the Win32 `SendMessage` API call. There are three parameters required by `SendMessage`, which match the three variables of the same name: `Msg`, `wParam`, and `lParam`.

The variables are specified as hexadecimal values in the standard C format, since that’s how you’re likely to see the definitions you’ll need to use this function properly. The “0x” is just a flag indicating that a hex value follows, and the subsequent digits are the hex value itself. For example, decimal 75 is expressed as 4B in hexadecimal, and would be specified in one of these variables as 0x4B.

The brief discussion of Windows Messaging at the end of this document explains a little bit about these values, but a detailed explanation is far beyond the scope of these instructions. `SendMessage` is part of the basic underlying foundation of Windows. Even bumping the mouse typically generates a stream of many hundreds of internal windows messages.

Payload Event: PostMessage

This works exactly like `SendMessage`, above, except that the Payload Event should be “PostMessage”... the difference is that `SendMessage` puts the message directly into the target window’s message handler function, whereas `PostMessage` stores the message in the target window’s message queue. Using the queue might be important in sequential operations, such as sending keyboard input (although the plug-in provides more useful ways to handle keyboard input).

Payload Event: KeyChar

The plug-in provides special support for simulating keyboard input because it is a commonly-required activity, and simulating it involves a series of messages that would be time-consuming to create by hand. Also, as you’ll see, the plug-in can take advantage of some Windows helper functions which are not available to your Lua scripts.

The `KeyChar` payload event uses the `Character` variable to simulate a keystroke that maps back to a readable character. Letters, numbers, and symbols such as question marks and dollar signs are examples of these. That’s all you really need to know in order to use this payload event – just set `Character` to whatever you want, say “X” or “7” or “w” – and this one is case-sensitive by the way – and set up the other variables and make the call.

You can use the ShiftState payload variable to indicate that the Shift, Control, or Alt key should be simulated – or any combination of those three. Do this by putting the letters S, C, and A into the variable. For example, “SC” indicates both the Shift and Control keys should be simulated when the specified character is pressed. Some of these will be implied by the character you supply (for example, you can’t type a dollar sign on the physical keyboard without pressing Shift). The plug-in will properly resolve these characters regardless of whether you specify the related shift key.

Payload Event: KeyVirt

As you might have guessed, this payload event lets you send keystrokes which do not map to a readable character, such as the Enter key, or the arrow keys, or even the multimedia keys on the newer enhanced keyboards. The VirtualKey variable is a hex value which corresponds to one of the standard VK_ constants. These are defined at the following Microsoft Developer Network website:

<http://msdn.microsoft.com/library/en-us/winui/winui/WindowsUserInterface/UserInput/VirtualKeyCodes.asp>

Note that not all VK_ constants listed on that web page will necessarily be usable. For example, some require extra flags that you can’t access through the plug-in, which are used for things such as discerning between the left Control and the right Control key. However, the more routine keys should work just fine, which should address any typical application you’re likely to encounter.

The ShiftState payload variable works exactly as described for the KeyChar payload event, above. However, using this payload event never automatically implies the use of any of the shift state keys.

Notice that a lot of characters aren’t listed on the web page listed above. For example, there isn’t a VK_ defined for the dollar sign. That’s because this function maps to the *physical* keyboard. To get a dollar sign using the KeyVirt payload event, you’d have to send the “4” key (hex 0x34) combined with the Shift key. In such cases, it’s usually easier to just use the KeyChar payload event.

Examples

The best examples are in the Winamp and PowerDVD support files provided with the plug-in, but here are some greatly simplified examples for users just browsing the help.

The following example sends the “Play” command to Winamp. If Winamp is not running, it will be loaded, then the command will be sent. The value 0x0111 in the Msg variable is the WM_COMMAND constant mentioned earlier in the documentation.

```
payload = Payload(10, "SendMessage")
payload.Variables: Add(Variable("TargetWindowClass", "Winamp v1. x"))
payload.Variables: Add(Variable("ExePath", "c:\\program files\\winamp\\winamp.exe"))
payload.Variables: Add(Variable("WaitForLoad", "10000"))
payload.Variables: Add(Variable("Msg", "0x0111"))
payload.Variables: Add(Variable("wParam", "0x9C6D"))
payload.Variables: Add(Variable("lParam", "0x0"))
SendCommand(payload)
```

The next example closes the Winamp application. If Winamp is not running, nothing happens. The value 0x0112 in the Msg variable is the WM_SYSCOMMAND constant mentioned earlier in the documentation, and the value 0xF060 stored in the wParam variable is the SC_CLOSE command.

```
payload = Payload(10, "SendMessage")
payload.Variables: Add(Variable("TargetWindowClass", "Winamp v1.x"))
payload.Variables: Add(Variable("Msg", "0x0112"))
payload.Variables: Add(Variable("wParam", "0xF060"))
payload.Variables: Add(Variable("lParam", "0x0"))
SendCommand(payload)
```

Understanding the Win32 Window Messaging System

In Windows, almost everything on the screen is made up of other windows, and we mean *everything* almost literally – icons, text labels, individual buttons, text-input areas, the individual parts of the more elaborate controls like drop-down lists, menus and each separate menu item, and so on. Even when there aren't any user applications running, there are hundreds of windows in use. The Windows XP desktop consists of three windows. Although it doesn't look like anything you'd normally associate with the word "window," a standard scrollbar is actually four windows – the arrow buttons at each end are little windows, the gray area in the middle is a window, and the scroll "thumb" (the thing that moves back and forth) is a window. There is a good reason for this, and it's directly related to the way this plug-in works: when a window is created, part of that process involves telling the operating system the location of a special function called the WndProc (which is an awkward abbreviation for "window procedure"). That function is how the window receives communications from the outside world. The WndProc is designed to receive and respond to Window Messages.

There are nuances which are mostly not relevant (such as parenting, which lets the main application window see the messages sent to child windows, such as the windows that make up a scroll bar). Generally all you have to realize is that most things on the screen are windows, they receive Window Messages, and the reason this plug-in is useful is because it can send Window Messages from your Chipmunk Lua script to those windows for processing. (The operating system actually stores window messages in queues. A number of messages can "stack up" and await processing while the application is busy doing something else.)

The next logical question is probably this: What is a Window Message, exactly? The answer is pretty simple: it's a number. Way back in the 80's when the earliest versions of Windows were released, Microsoft defined a very large list of C constants, each of which represented a Window Message. Most of those are still used today, 20 years later, and hundreds more have been defined since then. Typically the C constant name is used to identify the message. A common one is WM_SYSCOMMAND, which is defined as the hex value 0x0112.

A message is accompanied by a pair of parameters, called wParam and lParam, for historical reasons not important here. These are also just numbers. When an application receives a message such as WM_SYSCOMMAND, these parameters provide the details. For example, if wParam is set to the C constant SC_CLOSE, which is defined as the

number 0xF060, the application knows it has received the Close command from the System Menu. (The System Menu is what you see when you click the top left corner of a program's main window. In Windows XP this is represented by a small version of the application icon.)

Window messages can do things like push on-screen buttons, send keystrokes, minimize windows to the Taskbar, issue direct commands if the application is designed to receive them (for example, Nullsoft, the makers of Winamp, publish a list of WM_COMMAND messages for executing basic control of their application), and many other useful things.

It is important to note that the plug-in does not allow you to send *any* Window Message – far from it. Many messages use wParam and lParam as memory pointers to complicated data structures, and supporting those is far beyond the scope of this plug-in (not to mention Lua's capabilities, and probably most people's ability to accurately and safely simulate from an external application). This really isn't anything to worry about – most Window Messages aren't that interesting from the standpoint of controlling a program from Chipmunk. After all, YOU control programs with nothing more than the keyboard and “pushing” on-screen buttons, right?

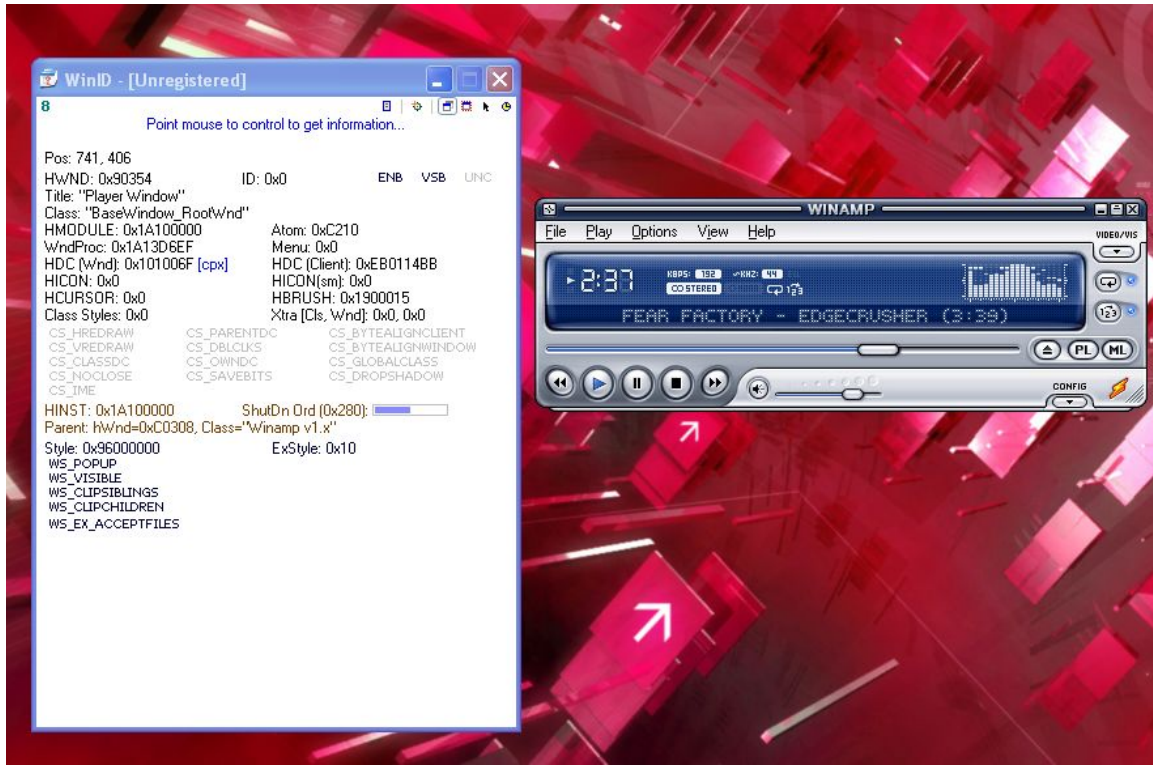
So now you know the basics about how Windows Messaging works, and why it works that way, and why it is interesting to us. That was the easy part. The hard part is putting this information to work.

The first thing you have to do is tell the plug-in how to find the window you're interested in. As you might imagine, since almost everything on the screen is a window, this is potentially complex. As if that isn't bad enough, some applications make things even more tricky. For example, the window that Winamp uses to receive messages isn't even visible on-screen. There isn't much anybody can do about situations like this – fortunately Nullsoft documents this kind of thing in the case of Winamp, but that isn't always true. Fortunately, with *most* applications, the main window is also the window which receives messages.

Windows are located by looking up something called their “ClassName” (which is an internal name supplied to the operating system when the program creates a window), or by searching for the window caption (the titlebar text). This uses the FindWindow API call, which goes back to the earliest versions of Windows. Back in those days, the titlebar of a window rarely changed. Unfortunately, starting with Windows 95, Microsoft devised the standard of putting a document name in the titlebar and instantly complicated the whole process. For that reason, I normally try to use the ClassName only.

There are tools which help you determine these values, to a degree. If you have a version of the Microsoft Visual Studio software development environment on your system or available to you, look for a tool called Spy or Spy++ but if Visual Studio isn't available to you, there are many freeware tools which do the same thing (and sometimes work better). The one I prefer is WinID, available at no charge from www.dennisbabkin.com.

The following screenshot shows WinID's display when the mouse cursor is over the Winamp main window. You can see near the top left that it shows "BaseWindow_RootWnd" as the ClassName. Unfortunately this is not the window which processes messages, and unfortunately this is the only ClassName you'll see in some utilities (particularly Spy). The correct ClassName is "Winamp v1.x", which WinID shows as the window parent (highlighted in brown near the middle of the window). In this case, trying to use "BaseWindow_RootWnd" with the plug-in doesn't cause any harm (Winamp is a fairly robust application), but it also won't produce any results. You have to set the payload variable to "Winamp v1.x" for everything to work properly.



Once you know how to find the window, the next trick is figuring out what to send. If you can control the application from the keyboard, that's probably your best option. The plug-in KeyChar and KeyVirt payload events make this relatively easy.

As described earlier in this document, you'll need to know the numeric constants assigned to keys, messages, and sometimes you'll need to know what values to supply in the wParam and lParam parameters. The authoritative location for those values is a file used by the C programming language called a header file, named WinUser.h, but that is a copyrighted file shipped with Visual Studio. However, Microsoft publishes the virtual key list on their developer website at the URL below:

<http://msdn.microsoft.com/library/en-us/winui/winui/WindowsUserInterface/UserInput/VirtualKeyCodes.asp>