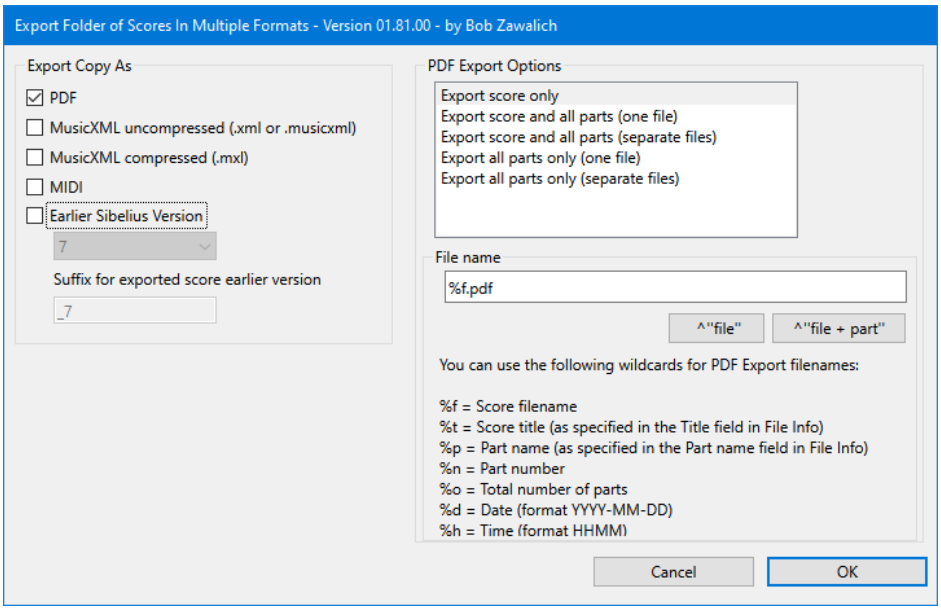
Plugins, Saving dialog settings, and the Add Plugin Preferences plugin

Bob Zawalich December 29, 2020 Updated July 13, 2025

Plugins and saving dialog settings

Many of the more complicated plugins present dialogs with settings, typically based on the dialog controls. In the dialog below, there are checkboxes, radio, an edit control, and 2 list boxes, which could be on or off, or contain text.



By their nature, dialog controls have default values which are stored in the .plg file as global variables. If you run the plugin and change the controls in the dialog, the plugin will usually "remember" the last settings you made until you close Sibelius and restart it. If you run the plugin again, the dialog settings will revert to those stored in the plugin file.

I have written about how plugins save settings and how you can change the settings in the file at

https://www.rpmseattle.com/of_note/how-sibelius-plugins-store-their-settings-for-user-data/

and

https://www.scoringnotes.com/tutorials/changing-default-dialog-settings-in-sibelius-plug-ins/

so I will not discuss this further in detail.

The Preferences plugin

Around the time of Sibelius 4, I was interested in being able to save plugin preferences across Sibelius sessions. In the example above, one might want to set up the dialog to always export

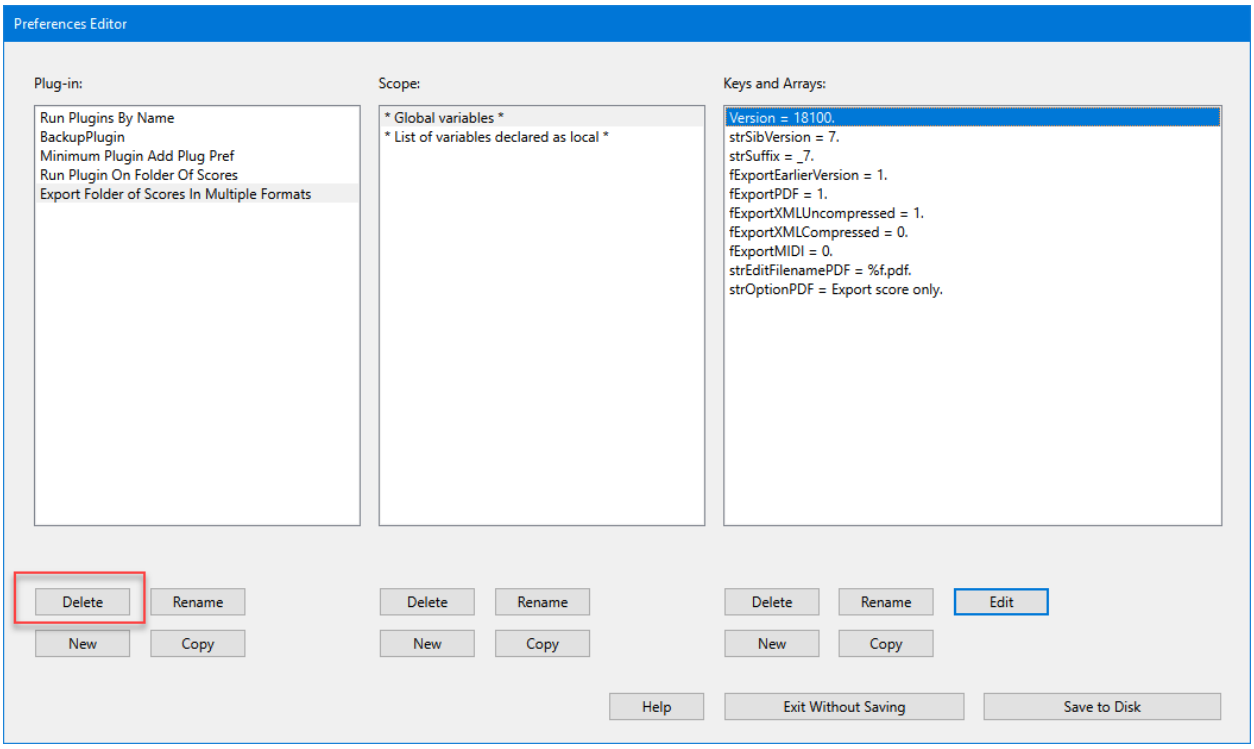
PDF, MusicXML, and Sibelius7 format, for example, and to have a specific file name format for PDFs, rather than having to reset these every Sibelius session.

At the time, we could not create text files, so I created Sibelius scores, and wrote out the data I wanted to save into those scores, and if such a score was available, I would read the settings from the score. One needed a separate score for each plugin, and these cluttered up Scores folders (even though I named the scores to suggest what they were for). When we were able to create text files, I switched to putting the data in text files, which were easier to deal with, but it still left text files lying around.

Hans-Christoph Wirth, who was also an active plugin developer at the time, wanted to do something better, and he took some of what I was doing, and wrote the **Preferences** plugin, which he convinced Daniel Spreadbury to ship it with Sibelius. **Preferences** provided an interface for plugin writers to add and read settings to a single Preferences Database file, which is a text file (*SibeliusPluginPreferences.dat*) stored in the user Plugins folder. The interface for **Preferences** is documented in the ManuScript Language Reference.

The **Preferences** plugin can be run by users to provide an editing interface for the data stored in plugins. It is not for the faint of heart, but it can be useful for programmers to check that the preferences were set properly.

Since *SibeliusPluginPreferences.dat* is a text file, and many plugins write to it, it can slow down plugins that need to read and write to the file. Often you might want to save the settings for a plugin for a short time while you are using a plugin regularly, but 2 years later you probably don't remember what was set, and the settings are mostly clutter.



I write about cleaning up the **Preferences** database in
https://www.scoringnotes.com/tips/speeding-up-some-sibelius-plug-ins/

so I refer you there if you are interested.

It was a lot of work…

Over the years I have refined how I use **Preferences** to minimize how much work is required to save and restore preferences, but it has always been enough work that I have to think about it for a while before I add the required code. I created a plugin template (which is published as **Minimum Plugin Preferences** in category Developers' Tools) that contains code that can be copied into another plugin using a text editor, and wrote about how to use the template in a PDF file that is included in the plugin zip file, and also available here: http://www.bobzawalich.com/wp-content/uploads/2020/06/Preferences-plugin-to-save-plugin-preferences.pdf

An attempt to simplify the process

I have used that approach for years, but it is still pretty tedious, so I was looking for ways to streamline it.

What you usually need to do is Save preferences after successfully running a dialog, and Get any preferences for the plugin before you put up the dialog. It occurred to me that the same data items were both Saved and Gotten, even though the code to do Save and Get was different. The code also required you to Open the preferences file before calling Get Preferences, and to be sure to Close preferences before you exit the plugin, which required some care. Preferences was not happy if a plugin quit while the Preferences database was opened.

So I decided to try this:

1. I was only going to deal with global variables, typically those that are dialog controls.

2. For each plugin, I would make a sparse array containing all the data required to Save or Get a global variable.

3. I would make a separate library plugin that would contain routines to Get and Save a sparse array of variable descriptions. These routines would handle both opening and closing the database so that it would be safe to exit a plugin after calling the library routine **API_GetPreferences** without explicitly closing the Preferences database.

The plugin **Add Plugin Preferences Lib**

The library plugin is called **Add Plugin Preferences Lib** (AddPluginPreferencesLib.plg in category **Developers' Tools**.). It is not added to a plugin menu, as it is only intended to be called by plugins. There is also a template plugin **Minimum Plugin Add Plug Pref** (**MinimumPluginAddPlugPref.plg**, in category **Developers' Tools**), and also in the public domain), which serves as an example of how a plugin calls **Add Plugin Preferences Lib,** and contains some code that can be copied into your own plugin.

I have set up my plugins that use this to check if **Add Plugin Preferences Lib** is available. If it is not installed, the caller will give a warning saying it is not installed and that the plugin will run but not save preferences over a Sibelius session. I am only warning once per Sibelius session to avoid antagonizing the user. If **Add Plugin Preferences Lib** is installed there will

be no warning. In your calling code make sure you deal correctly with the case where **Add Plugin Preferences Lib** is not available.

Using **Add Plugin Preferences Lib**

Add Plugin Preferences Lib controls the use of the preferences database, opening, closing, reading, and writing to it. It contains 4 routines intended to be called by other plugins, and these all have an **API_** prefix in their names, indicating that they are suitable to be called by other plugins. Of these, **API_SavePreferences** and **API_GetPreferences** are likely to be the only routines that will be called.

API_SavePreferences - a form of SavePreferences
API_GetPreferences – a form of GetPreferences

The next 2 routines are available if you need specific access to the database open and close, but Save and Get should handle this

API_OpenPreferences
API_CleanupPreferences

API_TraceSparseArrayPrefData – a routine that traces the sparse array containing data descriptions for data to be written and read.

There are 2 routines you will need to write for each plugin that calls **Add Plugin Preferences Lib.** Samples of these are available at the end of the **MinimumPluginAddPlugPref.plg** text file, and you can copy them into your plugin if you wish.**:**

- **GetAddPluginPreferencesLib()**, which returns a pointer to **Add Plugin Preferences Lib.** This code will never change, but it cannot be included in the library, because you need the pointer to access the library. You can simplify it if you know that **Add Plugin Preferences Lib** is always available in your environment, but I think it is easy enough to copy and call it that it is worth not worrying if it is not found.
- **SetUpPreferencesData ()**, which you must write to specify which global variables are used as preferences, and provide the properties that **Save** and **Get** will need.

I will discuss these in more detail shortly.

Calling **Add Plugin Preferences Lib**

Typically you will need to do these steps. You will not know which variables will be saved and restored until you have the dialog finished, so I recommend not doing any of this before the dialog is in final shape.

1. Get a pointer capable of calling **Add Plugin Preferences Lib** that checks that the plugin is installed**.** A copy of **GetAddPluginPreferencesLib (see below)** should handle this nicely. To call routines, you would have something like
    - pPlugin = **GetAddPluginPreferencesLib**(fWarnIfPluginMissing, fFirstTime);

    - ...
    - okGetPrefs = **pPlugin.API_GetPreferences**(("" & _PluginMenuName), valVersion, arsParametersPreferences, fTracePreferences);
2. Define the variables you want to read and write in your routine **SetUpPreferencesData()**. Copying the method out of **MinimumPluginAddPlugPref.plg** will save you some time here.
3. Do the following only if you have successfully gotten a pointer to **Add Plugin Preferences Lib.**
4. Before bringing up a dialog whose settings are to be preserved:
    - Call **SetUpPreferencesData()** in your **Run()** method
    - If that call succeeds, call **pPlugin.API_GetPreferences** to read any saved preferences from the Preferences database. There may be no such saved preferences, but that is OK. The dialog will use the defaults set up in the plugin file. Continue on regardless of what **API_GetPreferences** returns.
    - Bring up your dialog, which will use any settings that were acquired by **API_GetPreferences.** If the dialog returns False, indicating it was canceled by the user, you can safely exit.
    - If the dialog returned True, you will likely want to call **pPlugin.API_SavePreferences** to save the dialog settings from the dialog you just ran.
5. That is all you need to do. The hardest part is setting up **SetUpPreferencesData(),** and that is not very hard once you understand the structure of the data.

An annotated example of the calls to **Add Plugin Preferences Lib**

Here is the dialog and code used in **Run()** in the template plugin **MinimumPluginAddPlugPref.plg**

```
/ ******* The calls to preferences code starts here
// AddPluginPreferencesLib lets you set up plugin preferences with no code needed

*** I only want to show a warning message the first time the plugin is run in a Sibelius session. I set the global variable g_fFirstTime True in Initialize() and clear it here.

fFirstTime = 0 + g_fFirstTime;
if (g_fFirstTime)
{
   g_fFirstTime = False; // set true in Initialize
}

*** I get the pointer to the library plugin, It will put up a message if fWarnIfPluginMissing is True and fFirstTime is true So you can control any messages. pPlugin will be null if the plugin is not installed

fWarnIfPluginMissing = True; // will report error is plugin is not installed
pPlugin = GetAddPluginPreferencesLib(fWarnIfPluginMissing, fFirstTime);

fPreferencesAvailable = (pPlugin != null); // plugin will continue but without Preferences

*** I have debugging trace() calls set up in several of the library routines. fTracePreferences is normally off, but I leave it on in the template so you can see what it does. Get and Save need a plugin identifier, typically _PluginMenuName, and a plugin version number, which for me is always the global variable zg_VersionNumber. You can use what you like, but some valid strings are required.

fTracePreferences = True; // debug only
valVersion = "" & zg_VersionNumber;
```

```
*** only do this if we have a plugin pointer

if (fPreferencesAvailable)
{
   arsParametersPreferences = CreateSparseArray();

   SetUpPreferencesData(pPlugin, arsParametersPreferences, fTracePreferences); // on error,
arsParametersPreferences.Length will be 0

   fPreferencesAvailable = (arsParametersPreferences.Length != 0);

   if (fPreferencesAvailable)
   {
   okGetPrefs = pPlugin.API_GetPreferences(("" & _PluginMenuName), valVersion, arsParametersPreferences,
fTracePreferences);
   }
}

*** Normal code to bring up a dialog and test its return code

ok = DoDialog(DisplayDialog);
if (ok = False)
{
   return False;
}

*** The dialog returned true. I assume some settings have changed, so I write them to the Preferences database.
if (fPreferencesAvailable)
{
   okSavePrefs = pPlugin.API_SavePreferences(("" & _PluginMenuName), valVersion, arsParametersPreferences,
fTracePreferences);
}
```

You may note that **API_SavePreferences** and **API_GetPreferences** have return values that I do not check. Usually I do not care if they succeed or fail, and the variable is there mostly as a reminder to myself that they could fail, in case I decide to care about it. I tend to care as much about future maintenance as efficiency at the level of individual lines of code, but feel free to clean up anything you do not like.

API_SavePreferences and **API_GetPreferences** are straightforward to call. They need the calling plugin's _PluginMenuName and zg_VersionNumber variables to identify the plugin. These are global variables, which are addresses rather than values. In some cases passing such a variable directly has caused a plugin to fail, so as a rule of thumb I always force globals and array entries to be a string or a number before using it. Thus `("" & _PluginMenuName)` and `valVersion = "" & zg_VersionNumber;`.

Otherwise, you just pass in the sparse array or variable descriptions and **fTracePreferences**. If **fTracePreferences = True**, the called routines will dump out some data as they go along, so in a production plugin this would be set to false.

This mostly leaves us with **SetUpPreferencesData().**

Writing GetAddPluginPreferencesLib

This can be copied from here or from Minimum Plugin Add Plug Pref. Here it is in the form used when saved as a text file, so you copy copy it, open your plugin in a text file, and append the code to the end, or in the plugin editor you can paste this code into a new Method,, extract the name and parameters , and remove the closing "};"

You will need these 2 global variables. If MyMessageBox is not available you can use Sibelius.MessageBox instead.

```
_strAddPluginPreferencesLib "AddPluginPreferencesLib"
  _msgPluginNotInstalled "The plugin %s is not installed, and without it, this plugin will still run but its plugin preferences will not be saved across Sibelius sessions. Install the most recent version of %s to save preferences."
```

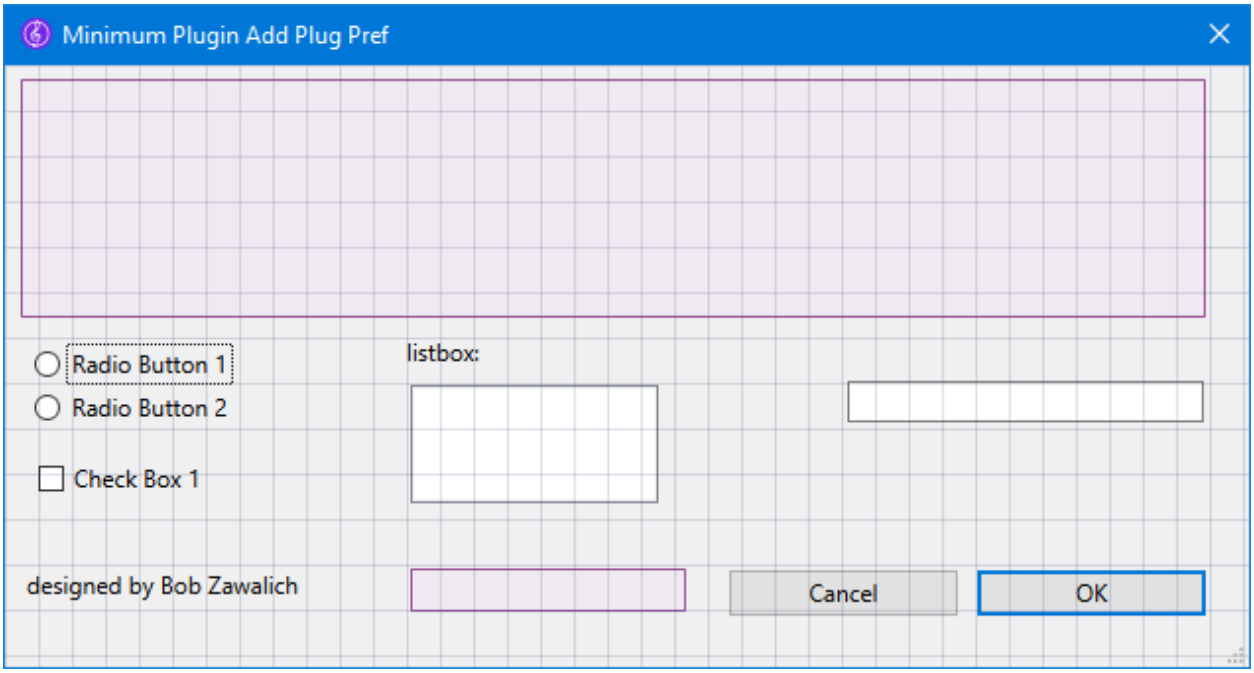
Here is the code

```
GetAddPluginPreferencesLib "(fWarnIfMissing, fFirstTimeInSession) {
strPlugin = '' & _strAddPluginPreferencesLib;

pList = Sibelius.Plugins;
fAvailable = pList.Contains(strPlugin);

if (fAvailable = False)
{
	if (fWarnIfMissing and fFirstTimeInSession)  // only warn once per session
	{
		msg = utils.Format(_msgPluginNotInstalled,  strPlugin, strPlugin);
		trace(msg);
		MyMessageBox(msg);
	}

	return null;
}

// the plugin object you get from searching the pList does not work for calls, but it works for getting properties

ptrPreferences = @strPlugin; // appropriate pointer for calling the plugin

return ptrPreferences;
}"
```

Writing SetUpPreferencesData()

By the time you do this, you need to know which variables are to be saved and restored. I am assuming that these will all be variables representing dialog controls, but any global variables will do. They need to be global variables, because such a variable is really an address, not a value, and the plugin can change the value of such a variable when nit is passed as an entry in a sparse array.

Here is the dialog used in **MinimumPluginAddPlugPref.plg.** It has radio buttons, a checkbox, a listbox and an edit control. The listbox has an underlying array, which would normally not be saved, but I will save it as an example.



Here are the global variables associated with the controls, which we want to save and restore:

dlg_fCheckbox1
dlg_fRadio1
dlg_fRadio2
dlg_lstAlphas (the listbox array)
dlg_strAlphaSelect (the current selection text for the listbox)

In the old way of doing this kind of code, I might save off only 1 of the radio buttons for a 2-button set, and derive the second as not (dlg_fRadio1). In this model, all the radios buttons are separately defined and stored.

Here is the **SetUpPreferencesData**() routine I wrote to declare these variables.

Update: the original doc called for 3 fields, the last of which was a key, typically the variable name minus its prefix. In my implementation of the library I only use the first 2 fields, and use the complete variable name for the key. It is a bit more space in the Preferences lib, but simplified the implementation for adding the code.

Note that for each variable there are 2 fields. This routine is meant to be self-documenting so we can see how well that works. The 2 fields are:

~~Note that for each variable there are 3 fields. This routine is meant to be self-documenting so we can see how well that works. The 3 fields are:~~

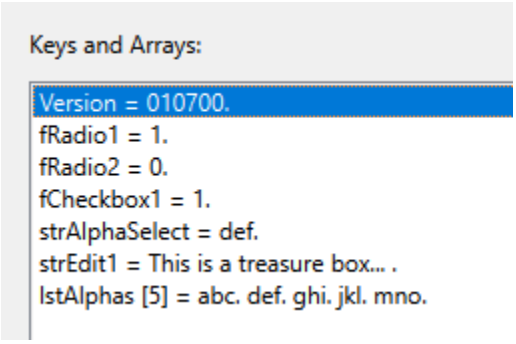
// the first is the variable itself, which must be a global variable, such as those used in a dialog.

// the second is a type name for the variable, which must be a literal string in lower case in English. (see below)

~~// the 3rd is the variable key (also a string) to be stored in Preferences.~~

Here we go. A few things I want to note:

- All the stored variables must be strings, numbers (which are declared as strings), Booleans, which return True or False, or (old-style) arrays, all of which can be defined as a plugin's global variables.
- The first entry is the actual variable, not escaped with ""& or 0+ . It is an address passed to the library so the library code can actually change the variable and the caller does not need to do that explicitly.
- ~~The "key" is what Preferences uses to identify the variable. You can use the variable name cast to a string if you like. Since I often use prefixes like **dlg**_ or **g**_ for globals, I often omit the prefix to save a little space in the Preferences database.~~
- Here is what Preferences stores for these variables, which you can compare to the variables described above.



- You need to pass in **pPlugin** so the plugin Trace routine can be called
- There is an **fTrace** parameter, set False in a production plugin, that will display debugging information.

Again I will add some annotations in red.

SetUpPreferencesData "(pPlugin, pPluginCalling, arsParametersPreferences, fTrace) {

```
MyMessageBox('SetUpPreferencesData must be changed with the data you want to save and restore for your plugin.
There are examples for your use in the code below that work for the dialog in this sample plugin only.');

// **** You will probably not be able to set this up until at least all the variables in the dialog are set up.

// each object to be saved or restored has 2 entries in the sparse array.

// the first is the name of the global variable, such as those used in a dialog, as a string variable.
// IT MUST MATCH THE NAME OF THE GLOBAL VARIABLE EXACTLY, INCLUDING CASE.
// The name is used both as a key in Preferences and is indirected on to access the global variable itself.

// the second is a type name for the variable, which must be a literal string in lower case in English.

// the allowable type names are 'boolean' (or 'b') (true/false), 'string' (or 's') (text or numbers)
// or 'array' or ('a'), a nonsparse array created with CreateArray().

// GetPreferences and  SavePreferences will use this data to set and get the values of these variables.

// for the variable keys, you must use the internal name of the variable as a string (in double quotes)
// so dlg_fRadio1 would use 'dlg_fRadio1'.

// in this sample, I am using the variables used in the controls for the sample dialog in this template

//the variables to be used in this example are:

// dlg_fRadio1 (boolean)
// dlg_fRadio2 (boolean)
// dlg_fCheckbox1 (boolean)

// dlg_strAlphaSelect (string from List Box control)
// dlg_strEdit1 (string from Edit control)

// dlg_lstAlphas (array)

// the following entries are all examples and will be changed in a real plugin

// BE VERY CAREFUL TO SPELL THE VARIABLE NAMES CORRECTLY!!!

arsParametersPreferences.Push('dlg_fRadio1');
arsParametersPreferences.Push('boolean');

arsParametersPreferences.Push('dlg_fRadio2');
arsParametersPreferences.Push('b');  // you can use just the first letter if you wish

arsParametersPreferences.Push('dlg_fCheckbox1');
arsParametersPreferences.Push('boolean');

arsParametersPreferences.Push('dlg_strAlphaSelect');
arsParametersPreferences.Push('string');

arsParametersPreferences.Push('dlg_strEdit1');
arsParametersPreferences.Push('s');  // again, a single letter shortcut

arsParametersPreferences.Push('dlg_lstAlphas');
arsParametersPreferences.Push('array');


if (fTrace)
{
        trace('SetUpPreferencesData filled array arsParametersPreferences');
        pPlugin.API_TraceSparseArrayPrefData(pPluginCalling, arsParametersPreferences);
}
```

```
return arsParametersPreferences.Length;
}"
```

That is all the code you actually need to write. It is really only the variable declarations that change from plugin to plugin. If you have written Preferences out "the old way", I trust you will find this to be easier.

If you have more elaborate use of **Preferences**, this may not help, but I find that this should handle the vast number of cases.

I first used this in a published plugin in **Run Plugins On Folder of Scores**.