Slick-C Macro Conventions and Best Practices for End Users

Table of Contents

Introduction

This article is intended to provide guidance as you begin to write macros for SlickEdit products. We will discuss such practical matters as suggested conventions and best practices to use when writing, storing and running your macros.

If you have not familiarized yourself with Slick-C® language basics (i.e. syntax, etc.), then see the *Slick-C® Macro Programming Guide* that ships with each SlickEdit product before continuing. We assume that you are familiar with the Slick-C macro language, since details of the language (e.g. syntax, constructs, etc.) will not be discussed in this article.

We make frequent use of some terms and features throughout this article, so here they are up front:

Configuration directory
: Directory where the user's state file, along with options, settings, and recorded macros are stored. You can see your Configuration Directory from the **Help > About SlickEdit** menu in the SlickEdit® product, and **Tools > Plug-In Information** in the SlickEdit® Plug-In for Eclipse™.

SlickEdit command line
: Text box anchored at the bottom of the SlickEdit application frame (or floating in the SlickEdit Plug-In). Click with mouse at the bottom of the application frame or hit ESC in most emulations to activate the command line.

State file
: Binary file that stores all loadable, compiled macro source code, resources (dialogs, toolbars, menus, images, key bindings), and settings. SlickEdit products ship with a state file (the system state file), and each user's configuration directory stores a user state file (Windows®: vslick.sta; UNIX®: vslick.stu).

Slick-C command
: A Slick-C function prefixed with the _command keyword. A Slick-C command can be bound to a key, menu, toolbar button, or executed from the SlickEdit command line.

Macro modules

When we refer to "macros" we are usually talking about a macro module/file. Keep this in mind as you read further. Macro files with a .e extension are source code, files with a .sh extension are header files, and files with a .ex extension are compiled macro source (byte code) that has been compiled with the Slick-C macro compiler. All compiled macro source files, with the exception of batch macros, are loaded and stored in the state file.

System macros

System macros, macros that ship with each SlickEdit product, are stored under the macros subdirectory. Feel free to use snippets of system macro source when writing your own macros—we ship the macro source for this reason.

Modifying macros that ship with SlickEdit

Don't do it! Take your hands away from the keyboard. Unless you have an OEM support contract with SlickEdit Inc., and sometimes not even then, you should not be modifying system macros...ever. Feel free to use snippets from them all you like in your own macros, but don't modify the ones we ship.

If you modify a system macro, you are guaranteeing that:

- The next patch or upgrade you do will wipe out your changes
- Support will not be able to help you

But I really, really, really want to change the way a command works

That's fine (no it's not), but keep in mind that wholesaling an entire module with all the functions and commands it contains could potentially (most probably) break future upgrades and patches that rely on the functions and commands that were in that module. If you must rewrite functionality that ships with the product, then at least do the following:

- Put the changed command inside a new macro module and keep it separate from the product
- Change the name of the copy of the command you are making
- Rebind your keys, menus, and toolbar buttons that used the command to your new command

Loadable vs. batch macros

There is very important distinction to make between macro modules that are loaded into the state file vs. batch macros that are not.

Loadable macros

These are the most common type of macros. Macro modules that you load are stored in your state file in your Configuration Directory. Loaded modules contain Slick-C commands, functions, and event tables. SlickEdit cannot make use of your custom macro until you load it. To load a module, use the **Macro > Load Module** menu item.

Loadable macros are good for:

- Defining Slick-C commands that can be bound to keys, buttons, menus
- Defining Slick-C event tables used for dialog and tool window interaction

Batch macros

Batch macros are macros that are temporarily loaded for the duration that they run and unloaded after completion. Batch macros are not stored in the state file. You can always tell a batch macro because it starts off with a `defmain`. Batch macros are useful as a way to share custom macros with other users.

They do not have to be loaded, and can be run from the SlickEdit command line. Batch macros can also be run from a menu or toolbar button. Batch macros cannot be bound to a key.

Batch macros are good for:

- Standalone, single-task macros. The Auto Tag dialog that tags run-time libraries is a batch macro.
- Facilitator macros that exist only to get key bindings, dialogs, tool windows, toolbars, or menus loaded into the state file. Your vusrdefs.e (UNIX: vunxdefs.e) in your configuration directory is a batch macro that migrates your options and key bindings after performing a patch or upgrade.

Recorded macros

You can record an operation or action in SlickEdit and the Slick-C source code to replay that operation is recorded in a macro file called vusrmacs.e in your configuration directory. vusrmacs.e is automatically saved and loaded into the state file each time you record a new macro. Recorded macros are also a good way to explore the Slick-C macro code and get ideas when writing your own macros.

You can start, stop, run, save, and manage recorded macros from the Macro menu.

Where do I store macros that I write?

User macros, macros created by you, can be stored anywhere. However, we suggest that you do NOT store them under the SlickEdit product directory. SlickEdit will automatically remember your loaded macros after performing a patch or upgrade.

If you are writing batch macros, then it would be a good idea to add an entry to the VSLICKPATH environment variable in a file called vslick.ini in your Configuration Directory. Doing so will allow you to execute batch macros without having to specify the full path.

Example vslick.ini with modified VSLICKPATH:

```
[Environment]
; IMPORTANT: Append to existing value of VSLICKPATH or else you will have BIG problems!
VSLICKPATH=%VSLICKPATH%;<my-custom-macro-directory>
```

Finding help and examples

Help system

The obvious place to start looking for help when writing macros is the SlickEdit Help system. All SlickEdit APIs are listed under the help topic "Macro Functions by Category" in the Table of Contents. Many of the API help pages include examples of their use.

Tracing macro code

For the case when the Help system is not enough, and you know of a feature that does something similar, you have some options:

1. Find a similar menu item and trace it
2. Find a similar toolbar button and trace it
3. Find a similar key binding and trace it
4. Record a macro command and trace it

An indispensable tool when tracing macros is the command fp (short for find_proc) which you run from the SlickEdit command line and give it a command name to trace. You can also use the **Macro > Go to Slick-C Definition** menu item. If we say: "Do a find-proc on xyz", then we mean:

1. Go to the SlickEdit command line
2. Type 'fp xyz' and hit ENTER

which will cause SlickEdit to jump to the definition of xyz. You can pop back to where you were by pressing Ctrl+Comma.

If your cursor is parked on a Slick-C command or function, you can go to the definition of that function by pressing Ctrl+Dot. Ctrl+Comma will pop you back. Using Ctrl+Dot, you can "push" down as many levels deep into the code as you need and use Ctrl+Comma to "pop" back.

Find a similar menu item and trace it

Let's say you wanted to write a macro that sorted lines in a buffer as a part of its funtionality. You know that **Tools > Sort** dialog has an option for sorting a buffer. You can use the menu editor (**Macros > Menus**) to edit the main menu named _mdi_menu and navigate down to the Sort menu item. From there you get the command **gui_sort** which you can trace to find the call to the **sort_buffer** command. That sure looks like what we need. Looking at the help for sort_buffer gets us help on all the options.

Find a similar toolbar button and trace it

Let's say you wanted to know when an edit window has some selected text. You know that the Cut toolbar button on the Standard toolbar behaves differently when there is not a selection (it cuts the current line), so you reason that it must be able to check for a selection. If we right-click on the Cut button and choose Properties we see the command is cut. If we do a find-proc on 'cut' and drill down into the function cut2, we see near the top that there is a call to select_active():

```
if ( ! select_active() ) {
   ...
```

That sure looks like what we need. A quick look at the help for select_active verifies this. Mission accomplished.

Find a similar key binding and trace it

Let's say you wanted to know the current word under the cursor so you could perform some operation on it. You know that Ctrl+Shift+U upcases the current word so you surmise that it must know how to get the current word. Good guess! If we run the **Help > What Is Key** dialog, or run what-is from the SlickEdit command line, we find out that Ctrl+Shift+U executes the upcase_word command. Tracing upcase_word (by doing a find-proc on 'upcase_word' of course), we see the following line of code:

```
   int start_col=0;
   _str word=cur_word(start_col,def_from_cursor,false,def_word_continue);
```

Looks like what we need, but what are all those arguments getting passed? Looking at the help, we determine that the only argument we really need to worry about is the first argument (the rest are all default parameters that we are not required to provide). So our call to cur_word would be much simpler:

```
   int start_col=0;
   _str word=cur_word(start_col);
```

Record a macro command and trace it

Let's say, for example, that you want to know how to find and highlight a search string in the current edit window. If you look at the command on the **Search > Find menu** using the menu editor, then you see it runs the **gui_find** command. gui_find will display the Find dialog, but that only tells you how to display the Find dialog, not how to actually perform the search. Recording a macro of the operation is what you want. After recording the operation, and giving the macro a name, we can edit the macro to find something like:

```
#include "slick.sh"
_command test1() name_info(','VSARG2_MARK|VSARG2_REQUIRES_EDITORCTL)
{
   _macro('R',1);
   find('some_search_string','I');
}
```

where the operation we performed was a case-insensitive search for the string 'some_search_string'.

Debugging/tracing macros

Although there is not a Slick-C debugger, there are several tools you can use to debug/trace a running macro.

Slick-C stacks

When your macro misbehaves badly (e.g. accesses an invalid property, operates on an uninitialized variable, etc.), the Slick-C interpreter stops and the current execution stack is dumped into the Slick-C Stack tool window. A vsstack file is also written to your TEMP directory (UNIX: /tmp).

A stack looks like:

```
 Stack trace written to file: C:\DOCUME~1\username\LOCALS~1\Temp\vsstack
 Invalid argument
test.ex 1438 my_test_macro()   p_window_id: 75   p_object: OI_FORM   p_name:
stdcmds.ex 6170 command_execute()   p_window_id: 75   p_object: OI_FORM   p_name:
stdcmds.ex 5447 nosplit_insert_line()   p_window_id: 4   p_object: OI_TEXT_BOX   p_name:
stdprocs.ex 6441 try_calling(967)   p_window_id: 4   p_object: OI_TEXT_BOX   p_name:
stdprocs.ex 8949 call_root_key(" ")   p_window_id: 4   p_object: OI_TEXT_BOX   p_name:
slickc.ex 1243 slick_enter()   p_window_id: 4   p_object: OI_TEXT_BOX   p_name:
```

There is a lot of useful information here, including the offset in the bytecode where the error occurred (somewhere in my_test_macro). You can use this byte offset to find the source code line by:

1. Opening the offending source code module (if the compiled module is test.ex, then the source code module is test.e).
2. Typing `st -f <offset>` on the SlickEdit command line and pressing ENTER. Example: `st -f 1438` with the test.e module open would put our cursor at the offending line in the my_test_macro function.

say function

Use the **say** function to print trace messages out to a trace window (UNIX will print trace messages out to the console).

Example:

```
#include "slick.sh"
...
_command test1()
{
   int i;
   for( i=0; i<10; ++i ) {
      say("i=":+i);
   }
}
```

messageNwait function

Sometimes a bunch of trace messages are not enough. You need to pause to see what is going on in SlickEdit during your macro's execution. Use the **messageNwait** function to print a message on the message line and pause for a keypress.

Example:

```
   ...
   messageNwait("Debugging...press any key to continue");
   ...
```

_StackDump function

The **_StackDump** function allows you to dump the current execution stack to a trace window and the Slick-C Stack tool window. You call _StackDump in your macro code when you need to know about exceptional conditions that occur.

Example:

```
...
// wid is a window id. We do not want to operate on a window that is no longer valid,
// but we should never even be pressing this code, so dump the stack when it happens.
if( wid>0 && !_iswindow_valid(wid) ) {
   say('my_test_macro: !!!!!! _iswindow_valid('wid') FAILED!');
   _StackDump(1);
}
...
```

Include the slick.sh header file

You must start every macro module you write with the include:

```
#include "slick.sh"
```

Every variable, structure, flag, or setting you could possibly be interested in is included by this header file.

Command context

Command context describes what types of objects a command can or cannot operate on. It also describes what types of information your command requires to operate correctly.

Command context is important when putting your command on a menu or toolbar button because SlickEdit will enable/disable the menu/button based on the current object context (e.g. if a command requires a selection to operate, then a button that runs that command will be disabled unless there is an active selection).

name_info

Most of the Slick-C commands you see in macro source will have a name_info line:

```
_command upcase_word() name_info(','VSARG2_TEXT_BOX|VSARG2_REQUIRES_EDITORCTL)
...
```

The first part of the name_info line (before the ',') is specifying completion information for the command. In most cases, as is the case in the example above, it will be blank. Completion information governs what types of items will be listed when an auto completion list is requested for the command

from the SlickEdit command line. An example of a command that uses the completion part of the name_info line is edit:

```
_command edit(....) name_info(FILE_ARG'*,'VSARG2_CMDLINE|VSARG2_REQUIRES_MDI)
...
```

FILE_ARG specifies that the edit command should get only file names when requesting a completion list from the SlickEdit command line. To see this in action, type the following from the SlickEdit command line:

```
edit ?
```

The second part of the name_info line (after the ',') is specifying that the command will work in a text box (and also the SlickEdit command line which is a text box) and that it requires an edit window to act upon. Some of the more common VSARG2_* constants (in slick.sh) that you can use to restrict/allow where your commands can operate are:

Constant	Description
VSARG2_MARK	Command operates on a selection. This is necessary when you bind your command to a key that would normally cause text to be deselected or deleted/replaced.
VSARG2_READ_ONLY	Command operates on read-only edit windows.
VSARG2_LASTKEY	Command requires knowledge of the last key the user pressed. Use last_event to retrieve the last key the user pressed.
VSARG2_TEXT_BOX	Command operates on any text box including the SlickEdit command line.
VSARG2_CMDLINE	Command operates on the SlickEdit command line.

OnUpdate functions

Sometimes you need more sophistication when determining whether a command is valid in a particular context. One example of this is the requirement that a command only operate on certain types of files. For example, you have a command that can only operate on C++ header files (.h). There is no name_info VSARG2_ constant you can use for this, so you have to write an _OnUpdate_<command-name> function to check the extension on the current buffer and enable/disable accordingly. The _OnUpdate_ function would look like:

```
// The name of the command that this function operates on is 'my_command'
int _OnUpdate_my_command(CMDUI &cmdui, int target_wid, _str command)
{
   return ( p_extension!="h" )? MF_GRAYED|MF_ENABLED;
}
```

Determining the platform at compile time

Sometimes, hopefully rarely, you need to know if you are on Windows or a UNIX-like platform in order to conditionally compile macro code. The classic example is with file separators (Windows: backslash; UNIX: slash) and path separators (Windows: semicolon; UNIX: colon). You can use the special

preprocesser macros __NT__ and __UNIX__ to conditionally compile for Windows and UNIX-like platforms respectively.

Example:

```
#if __NT__
// Do something Windows-specific
#endif
...
#if __UNIX__
// Do something UNIX-specific
#endif
```

Note: UNIX-like platforms include: AIX®, IRIX®, Linux®, Mac OS® X, HP-UX, Solaris™.

Determining the platform at run time

You can determine the exact platform at run-time by calling the **machine** builtin function. See the help for a complete list of platform strings returned.

Example:

```
   // If we are running on HP-UX or Solaris ...
   if ( machine()=='HP9000' || machine()=='SPARCSOLARIS' ) {
   ...
```

Determining if a file is open

When your macro needs to know whether a particular file is open in SlickEdit already, use **buf_match**.

Example:

```
boolean file_is_already_open = ( buf_match(filename,1,'e') != "" );
```

Determining if a file exists

When your macro needs to know whether a particular file exists in the file system, use **file_exists**.

Example:

```
boolean exists = file_exists(filename);
```

Selected text

There are three types of selections in SlickEdit:

LINE
: Entire line or lines are selected. Use the **_select_line** builtin when selecting lines.

COLUMN
: A vertical column/block of text is selected. Use the **_select_block** builtin when selecting columns.

STREAM/CHAR
: Arbitrary extent of text is selected. Use the **_select_char** builtin when selecting streams.

Filtering a selection (a.k.a. How do I stuff selected text into a variable?)

To iterate over the lines of selected text, use the filter_init, filter_get_string, filter_put_string, and filter_restore_pos APIs.

Example:

```
...
// Display each selected portion of each line in selection in a message box
filter_init();
_str text = "";
while( filter_get_string(text) == 0 ) {
   _message_box("Text: ":+text);
}
filter_restore_pos();
...
```

Knowing when an open file is modified

If the current file is modified, then the **p_modify** property will be true.

Example:

```
...
if( p_modify ) {
   _message_box("Current open file is modified!");
}
```

Slick-C language notes

Implicit variable declarations

Don't use them! Slick-C supports implicit variable declarations inside functions, but please don't use them.

Reasons **not** to use implicit declarations:

- If variables are always explicitly declared, there is never any confusion about the scope of a variable.
- When mixing implicit and explicit declarations, it is easy to have uninitialized variables. This cannot happen when only explicit declarations are used.

Put `#pragma option(strict,on)` at the top of all your modules to disallow implicit declarations:

```
#pragma option(strict,on)
...
#include "slick.sh"
```

Dynamic typing

Don't use it! Slick-C supports dynamic typing for historical reasons, but we beg you not to use it.

Reasons **not** to use dynamic typing:

- Harder to make semantic mistakes
- All type checks happen at compile time - fewer errors hopefully
- Functions are called with correctly typed arguments - no surprises

Put `#pragma option(strict,on)` at the top of all your modules to disallow dynamic typing:

```
#pragma option(strict,on)
...
#include "slick.sh"
```

Standard dialogs

There are some standard dialogs that are useful for performing common operations (e.g. opening a file, etc.).

Selection list dialog

When you want to present the user with a list of items to choose from, show the **_sellist_form** dialog. See the help for `_sellist_form` for examples.

Text box dialog

When you want to prompt the user to fill in one or more fields using text boxes, combo boxes, or radio buttons, use the **textBoxDialog** function. See the help for textBoxDialog for examples.

Open dialog

For a standard Open dialog, use the _OpenDialog function.

Example:

```
...
_str result=_OpenDialog("-modal "_stdform("_open_form"),
                        "Choose File", // Title
                        "", // Initial wildcards
                        def_file_types,
                        OFN_NOCHANGEDIR|OFN_FILEMUSTEXIST,
                        "", // Default extension
                        "", // Initial filename
                        "", // Initial directory
                        "",
                        "");
result=strip(result,'B','"');
if( result=="" ) {
   // User cancelled the dialog
   return;
}
// result now holds the file name
...
```

Directory chooser dialog

For a directory chooser dialog, show the _cd_form dialog.

Example that prompts for a folder location:

```
...
_str result = show("-modal "_stdform("_cd_form"),"Choose Location",true,true,true);
result=strip(result,'B','"');
if( result=="" ) {
   // User cancelled the dialog
   return;
}
// result now holds the location
...
```

Print dialog

For a standard print dialog that prints the current file, use the **gui_print** command.