



SLICK-C® Macro Programming Guide

How to Customize and Extend SlickEdit® Products
with the Slick-C® Macro Programming Language

slick|edit®

Slick-C[®] Macro Programming Guide

(Version 12.0.2)

Information in this documentation is subject to change without notice and does not represent a commitment on the part of SlickEdit Inc. The software described in this documentation is protected by U.S. and international copyright laws and by other applicable laws, and may be used or copied only in accordance with the terms of the license or nondisclosure agreement that accompanies the software. It is against the law to copy the software on any medium except as specifically allowed in the license or nondisclosure agreement. The licensee may make one copy of the software for backup purposes. No part of this documentation may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or information storage and retrieval systems, for any purpose other than the licensee's personal use, without the express written permission of SlickEdit Inc.

Copyright 1988-2007 SlickEdit Inc.
Cover design copyright by SlickEdit Inc.
Produced in the United States of America.

SlickEdit, Visual SlickEdit, Clipboard Inheritance, DIFFzilla, SmartPaste, Context Tagging, and Slick-C are registered trademarks of SlickEdit Inc. Code Quick | Think Slick is a trademark of SlickEdit Inc. All other products or company names are used for identification purposes only and may be trademarks of their respective owners. Protected by U.S. Patent 5,710,926.

Table of Contents

Table of Contents	3
Introduction	9
Working with the Slick-C® Source Code	9
Four Ways to Use Slick-C®	11
Recording Slick-C® Macros	11
Key Bindable Command	11
Event-Driven Dialog Boxes	12
Creating a Simple Event-Driven Dialog Box	12
Loading Code and Displaying Dialog Boxes	14
Binding Commands to Keys for Dialog Box Display	14
Batch Macros	14
Language Constructs	17
Identifiers	17
Reserved Words and Keywords	17
Comments	18
String Literals	18
Numeric Constants	19
Defining Constants	20
Types	21
Arrays	21
Differences from C++	21
Hash Tables	22
Struct	22
Differences from C++	23
Union	23
Anonymous Unions	24
Pointers to Variables	25
Pointers to Functions	25
Strings	27
Typeless Container Variables	27
Mathematical Operators	29
Declarations	33
Scoping and Declaring Variables	33
Simple Variable	33
Details about Variable Initializations	34
Type Casting	34
Differences from C++	34
Statements	37
Assignment Operator	37
if Statement	37
Block Statement	37
Loops	38

TABLE OF CONTENTS

while	38
for	38
do	38
break	38
continue	38
parse Statement	39
switch Statement	41
Functions	43
Defining a Procedure	43
Argument Declarations	43
Default Arguments	44
Defining a Command	44
name_info Attributes	45
OnUpdate Functions	47
Function Prototypes	48
Library Functions	48
Built-in Functions	48
Finding Functions	48
Differences between Commands, Built-ins, and Defs	49
defmain: Writing Slick-C® Batch Files	49
Preprocessing	51
#pragma	51
#region and #endregion	53
Including Header Files	53
Defining Controls	55
Defining Events and Event Tables	57
def Primitive	57
Event-Driven Dialog Boxes	59
Module Initializations	61
Compiling and Loading Macros	63
Debugging Macros	65
Finding Procedures	65
Finding Run-Time Errors	65
Performance Profiling	65
Error Handling and the rc Variable	67
Dialog Editor	69
Microsoft Visual Basic and Slick-C®	69
Creating Dialog Boxes	71
Dialog Editor Summary	71
Adding Controls	71
Deleting Controls	72
Setting Properties	72
Aligning Controls	72
Sizing Controls	72
Moving Controls	72

Miscellaneous Assignments When the Form is Active	73
Miscellaneous Menu Items	73
Creating a Form	74
Saving a Form	74
Adding Event Handlers	74
Inherited Code Found Dialog Box	74
Loading and Running the Form	74
Adding a Cancel Button	75
Adding an OK Button and Closing a Dialog Box	75
Displaying Dialog Boxes	76
Modal and Modeless Dialog Boxes	79
Dialog Box Parent Window	80
Remembering a Dialog Box's Previous Position	80
Clipboard Inheritance®	81
Open Dialog Box	83
Dialog Box Inheritance Order	84
Objects and Instances	85
Active Object	85
Active Form	85
Instance Expressions	86
Using Functions as Methods	87
Built-in Controls	89
Label Control	89
Spin Control	89
Text Box Control	90
Editor Control	91
Frame Control	91
Command Button Control	91
Radio Button Control	91
Check Box Control	92
Combo Box Control	92
List Box Control	95
Vscroll Bar Control	97
Hscroll Bar Control	98
Drive List Control	98
File List Box Control	99
Directory List Box Control	99
Picture Box Control	100
Gauge Control	100
Image Control	101
Adding a Bitmap Command Button or Check Box	101
Adding Dialog Box Retrieval	104
Menus	107
Menu Macro Programming	107
Menu Editor Dialog Box	107
Menu Item Alias Dialog Box	108

TABLE OF CONTENTS

Auto Enable Properties Dialog Box	108
Save Failed Dialog Box	109
Insert Literal Dialog Box	109
Creating and Editing Menu Resources	109
Common Macro Dialog Boxes	115
String Functions	117
Search Functions	121
Selection Functions	125
Writing Selection Filters	127
Unicode and SBCS or DBCS Macro Programming	129
Shelling Programs from a Slick-C® Macro	133
DLL Interface	135
Command Line Interface	137
Command Line Arguments	137
get_string Procedure	137
Single Argument Prompting with Support for Prompt Style	137
Hooking Exit and Other Events	139
Invoking a Macro on Startup	139
Invoking a Macro on Exit	139
State File Caching	141
Windows Data Structure	143
Window Properties	145
View Properties	145
Buffer Properties	145
Tutorials	147
Defining Stack Routines	147
Searching for a String Within a Current Function	149
Creating the Macro	149
Analyzing the Macro	151
Command Line Search Options	152
Reading and Modifying Buffers	152
Functions for Reading and Modifying Buffers	152
Common Functions for Navigating Buffers	153
Escape Backslashes Example	153
Comment Out Debug Print Lines Example	154
Turning on Line Numbers for all Files	154
Write a New Macro to Find a Command Name	154
Counting Lines of Code	155
Event Names	159
ASCII Characters	159
Function Keys	159
Extended Keys	159
Mouse Events	159

On Events	160
Miscellaneous Keys	160
Miscellaneous Events	160
Key Name Examples	161
Differences Between Slick-C® and C++	163
Structures	163
Arrays	163
Hash Tables	163
Assignment Statement	164
Comparison Operator	164
Preprocessing	164
switch Statement	164
Labeled Loops	164
Variable Argument Functions	164
Built-in Graphics Primitives	165
Clipboard Inheritance®	165
End of Statement Semicolon	165
Type Checking	165
Capability not Supported by Slick-C®	166

TABLE OF CONTENTS

Introduction

The Slick-C® programming language enables the customization of SlickEdit®, and enables the creation of new functionality. Many of the actions performed using SlickEdit are performed using Slick-C macros. Slick-C functions are mapped to menus, buttons, and keys, and perform the action behind an event. Use Slick-C to customize, modify, and bind functions to other shortcuts.

The Slick-C API is extensive, covering many actions normally performed in a code editor, including navigation and buffer modification. Much of the code in SlickEdit is written in Slick-C and this source is provided when SlickEdit is installed. This enables the customization of the product, and the use of the Slick-C source as an example for writing new macros. After SlickEdit is installed, the Slick-C macros are located in the `macros` subdirectory of your installation directory.

Working with the Slick-C® Source Code

Slick-C macros are stored in files ending in the `.e` extension. The Slick-C macro translator compiles these files to byte code which is saved in a corresponding file with the `.ex` extension.

Slick-C follows a C-style linking model with the distinction that macros can be loaded and reloaded dynamically. Compiled macros and dialog box templates are stored in the state file `vslick.sta` (UNIX®: `vslick.stu`), which is located in your configuration directory.

Slick-C uses the C preprocessor. Slick-C header files use the `.sh` extension. All Slick-C source files `#include slick.sh`.

Four Ways to Use Slick-C®

There are four ways to extend the SlickEdit® code editor using Slick-C. They are to record Slick-C macros, write a key bindable command, create an event-driven dialog box, or to write a batch macro.

Recording Slick-C® Macros

When using macro recording, Slick-C source code is created for a key bindable command. To create a recorded macro, complete the following steps:

1. From the main menu, select **Macro > Record Macro**.
2. Perform the actions that you want the macro to repeat.
3. When finished, select **Macro > Stop Recording**.

The macro is saved as Slick-C source code and you can edit the recorded macro through the user interface. Recorded macros are saved in the `vusrmacros.e` file in the user configuration directory.

Key Bindable Command

A key bindable command is the most common way to extend the editor. Command macros can be bound to keys or invoked from a menu. To create a Slick-C® command, complete the following steps:

1. From the main menu, select **Macro > List Macros**.
2. Select the macro to edit.
3. Click **Edit**.
4. Open a new file named `test.e`, then type:

```
command void hello()
{
    message("Hello World");
}
```

5. Invoke the **load** command to compile and load the macro.
6. Press **F12**. Or, from the main menu, select **Macro > Load Module**. A new command is created named **hello**.

The **hello** command can now be bound to a key or invoked from the command line.

1. Place the cursor on the command line by pressing **Esc**.
2. Type **hello** and press **Enter**.
3. The message **Hello World** is displayed.

To bind the **hello** command to **Alt+5**, complete the following steps:

1. Invoke the Key Bindings dialog box. From the main menu, select **Tools > Options > Key Bindings**.
2. In the **Search for Command** combo box, type **hello**.
3. Click **Bind**.
4. In the **Binding** column text box, press **Alt+5**.
5. Click **Finish Binding**.

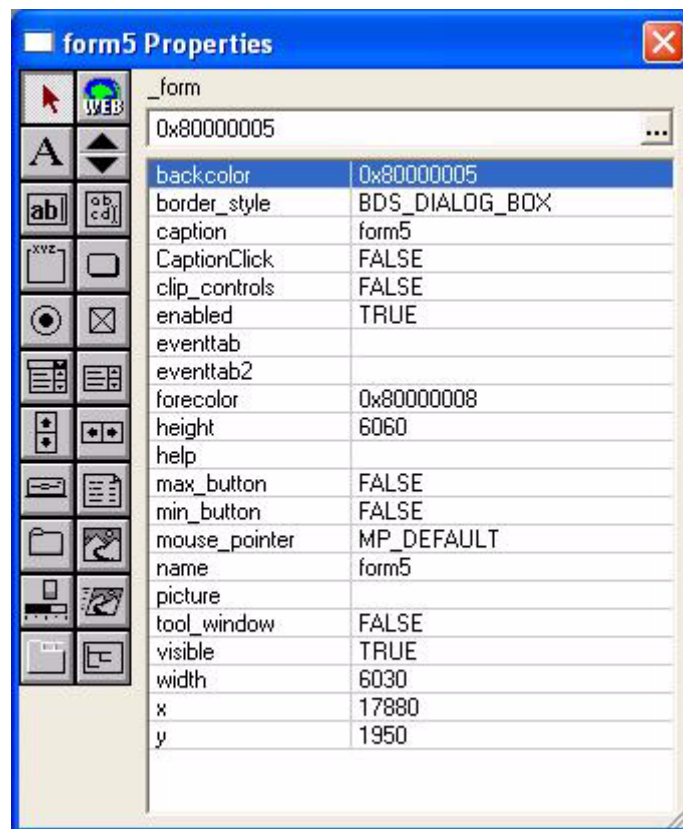
Event-Driven Dialog Boxes

Event-driven forms are ideal for creating dialog boxes. After creating the dialog box, use the **show** command to display the dialog box from the command line or from a menu item. To display a dialog box from a key binding, write a key bindable command that uses the **show** command to display the dialog box. For more information, see [Creating Dialog Boxes](#).

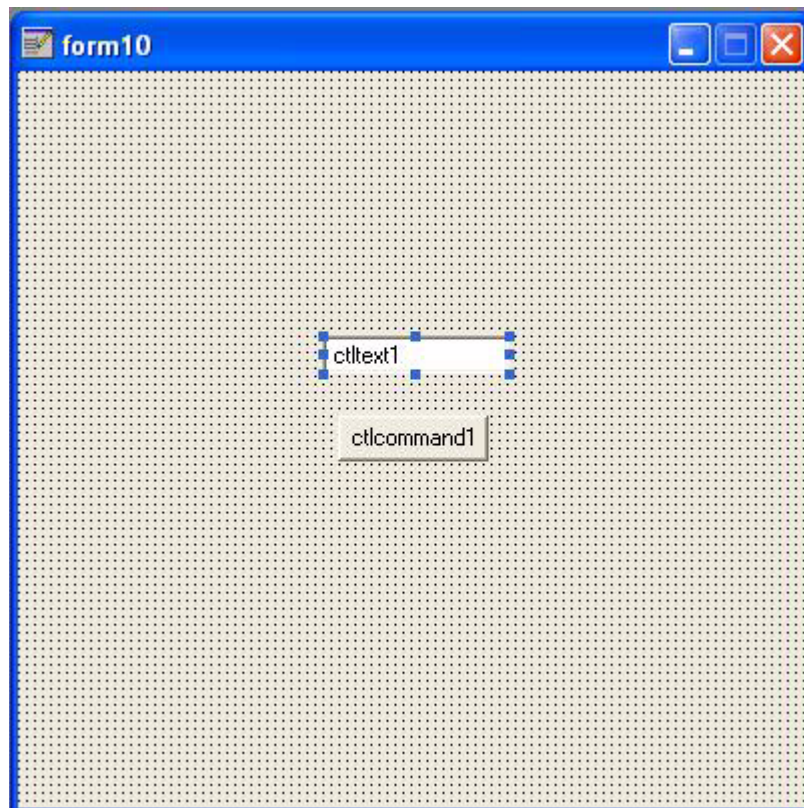
Creating a Simple Event-Driven Dialog Box

To create a simple event-driven dialog box, complete the following steps:

1. From the main menu, select **Macro > New Form**.
2. In the dialog editor Properties dialog box, double-click **Insert Button Control**.



3. Double-click **Insert Text Cox Control** in the dialog editor Properties dialog box.
4. Move the command button or the text box so that they do not overlap. Click on the object with the left mouse button, hold it, and drag to move the object.



5. Double-click on the command button that appears on the form (not the bitmap in the dialog editor Properties dialog box). The Select An Event dialog box appears with **lbutton_up** displayed in the combo box.
6. Press **Enter** to select the event.
7. The Open dialog box is displayed for a new file that is to contain the source code for this dialog box. Type **form1.e** and press **Enter**. A file is displayed named `form1.e` with the following lines of code:

```
#include "slick.sh"

defeventtab form1;
void ctlcommand1.lbutton_up()
{
}

```

8. If the previous lines of code are not displayed, then a `form1.e` file might already exist. If so, modify the existing `form1.e` file to contain the previous lines of code.
9. Modify the code to add the following statement: **ctltext1.p_text=Hello World;**

Example:

```
#include "slick.sh"

defeventtab form1;
void ctlcommand1.lbutton_up()
{
    // Set the p_text property of the text box control
    ctltext1.p_text="Hello World";
}
```

10. From the main menu, select **Macro > Load Module**.

Loading Code and Displaying Dialog Boxes

To load the dialog box, and then display it, complete the following steps:

1. Right-click on the form and select **Load and Run Form**.
2. Click **ctlcommand1. Hello World** is displayed in the text box.
3. To close the Form1 dialog box, press **Esc**.
4. Type **show form1** to display this dialog box from the command line.
5. To display the dialog box modally, type **show -modal form1** on the command line. The **show** command provides many options.

The dialog source is saved in the `vuserdefs.e` file in the user configuration directory, **My SlickEdit Config**. Press **Ctrl+Shift+Space** while any dialog box is running to enable editing (including the Properties dialog box).

Binding Commands to Keys for Dialog Box Display

To bind a command to a key that displays a dialog box, use the following example to write the necessary command:

```
#include "slick.sh"
command void run_form1()
{
    show("-modal form1");
}
```

Batch Macros

Use a batch macro when working with Slick-C® primitives that you want to share among multiple users. Batch macros cannot be bound to a key; however, you can execute a batch macro from the command line or a menu item.

1. From the main menu select **File > New**.
2. Type the following code:

```
#include "slick.sh"
void defmain()

{
    message("Hello World");
}
```

3. Save the macro as `hellow.e`, and press **Esc** to place the cursor on the command line.

4. Type **hellow**, and press **Enter**.
5. The message **Hello World** is displayed.

Batch programs must be saved before they are executed so that the macro can compile. Also, batch programs are automatically compiled if there is no corresponding `.ex` file, or if the date of the source file is newer than the date of the `.ex` file.

Language Constructs

The Slick-C® language is rooted in the C language. Slick-C contains some constructs from REXX and a dialog system usually found only in languages such as Microsoft® Visual Basic®.

Identifiers

A variable or identifier may contain any of the characters “A-Za-z\$_0-9” and must start with one of the characters “A-Za-z_\$”.

Reserved Words and Keywords

The table below shows the keywords that are reserved in the Slick-C® language.

arg	boolean	break	case	const	continue
default	defmain	definit	defexit	defload	do
double	else	false	for	if	int
intdiv	long	no_code_swapping	null	parse	return
short	static	struct	switch	true	typedef
typeless	union	var	void	with	while
_notinit	_reinit	_str	_command		

The table below shows the keywords that are reserved, but deprecated. Avoid using them.

_notinit	bigint	bigstring	bigfloat
----------	--------	-----------	----------

The table below shows the keywords that are reserved and used for event, dialog, and menu programming.

def	defeventtab	submenu	endsubmenu
_check_box	_combo_box	_control	_command_button
_editor	_form	_frame	_gauge
_hscroll_bar	_image	_inherit	_label
_list_box	_menu	_nocheck	_picture_box
_print_preview	_radio_button	_spin	_sstab
_sstab_container	_text_box	_tree_view	_vscroll_bar

All identifiers starting with **p_** are reserved to be used as Slick-C property names. SlickEdit® reserves all identifiers starting with “_” for internal use.

Comments

Slick-C® supports both of the C++ comment styles.

- Use `//` to declare that the rest of the line is a comment
- Use `/*` to open a block comment and `*/` to close a block comment.
- Block comments can be nested.

Example:

```
i=1; //this is a comment
/* this is a /* nested */ comment */
```

String Literals

Strings can be surrounded with single or double quotes. Double quoted strings are identical to C++ string literals.

A backslash followed by a character has special meaning, as outlined in the table below.

Characters	Meaning
<code>\a</code>	Bell character (7)
<code>\b</code>	Backspace character (8)
<code>\f</code>	Form feed character (12)
<code>\n</code>	New line character (10)
<code>\r</code>	Carriage return (13)
<code>\t</code>	Tab character (9)
<code>\v</code>	Vertical tab character (11)
<code>\?</code>	Question mark character
<code>\'</code>	Single quote character
<code>\"</code>	Double quote character
<code>\\</code>	Backslash character
<code>\xhh</code>	Hexadecimal character code
<code>\ooo</code>	Octal character code

If single quotes are used, two single quotes consecutively represent one single quote character. If double quotes are used, a backslash followed by a double quote represents one double quote character. The operator `==` used in the example below compares two strings for exact equality. The Slick-C® language does have an operator `==`. However, this operator strips leading and trailing spaces and tabs from both operands.

Example:

```
abc:=='abc'
"Can't find file:")== 'Can''t find file'
\t:==_chr(9)
```

A backslash (not inside quotation marks) followed by a character or a number has the special meaning, as shown in the table below.

Characters	Meaning
\a	Bell character (7)
\b	Backspace character (8)
\f	Form feed character(12)
\n	New line (10)
\r	Carriage return (13)
\t	tab character (9)
\v	Vertical tab character (11)
\xdd	Hexadecimal character code <i>dd</i>
\ddd	Decimal character code <i>ddd</i>

WARNING Using the above feature is not recommended. Use a quoted string.

Numeric Constants

The Slick-C® language supports integer constants in both decimal and hexadecimal formats. Hexadecimal numbers are defined as **0x** hexdigits. The Slick-C language supports floating point numbers. The mantissa is limited to 32 digits and the exponent is limited to nine digits. When precision is lost, the result is rounded. Overflow and underflow are detected. Floating point numbers have the following syntax:

```
[+|-] digits [.] [digits] [E[+|-] digits]
      or
[+|-] [.] [digits] [E[+|-] digits]
```

There may be blank spaces before and after the leading sign.

Examples:

```
4.04
4e2
4e2
4E-2
4E-2
```

Defining Constants

Slick-C® supports the **#define** preprocessor directive. The **#define** directive is for defining constants or in-line functions. Use the following syntax to define the constant or in-line function:

```
#define name[(param1,param2,...)] value
```

Use a backslash at the end of a line to indicate that the **value** text continues to the next line. Any occurrence of **name** is replaced with the text **value** before the source is compiled.

IMPORTANT When **value** represents an expression, place parentheses around it to make sure that there is not a problem with operator precedence.

Example:

```
#define MAXLINES 15
#define MAXLINESP1 MAXLINES+1
#define max(a,b) ( (a) >= (b) ) ? (a) : (b) )
#define min(a,b) ( (a) <= (b) ) \
    ? (a) : (b) )
defmain()
{
    x=MAXLINES;
    y=MAXLINESP1;
    a=max(x,y);
}
```

Types

Slick-C® types are similar to the types in C. The following types are available in Slick-C:

- Numeric types: The numeric types are **int**, **long**, and **double**. All numeric types are signed. Slick-C does not support **char**, **short**, or **float** types. The built-in Slick-C Boolean type is **boolean**.
- String types: Slick-C has a built-in string **type_str**.
- Array types: Array types are declared like C arrays, but cannot have a size limit. Array elements are always dynamically allocated.
- Void type: **void** is only permitted as the return type of a function.
- Typedefs: Slick-C supports C-style **typedef** type declaration statements.
- Pointers and reference types: Slick-C provides pointer and reference types in the same manner as in the C language.
- Hash tables: Slick-C provides a **:[]** hash table which is declared in the same manner as array types.
- Struct and union types: Slick-C supports C-style structs and unions.
- Typeless container types: Typeless variables are declared using the **typeless** type. A typeless variable can be assigned to or from any type, including structs, arrays, and hash tables.

Arrays

Use array variables to keep a list of items. To define an array variable, use the following syntax:

```
[static] TypeName variable1[] [= {e1,e2, ...}] , variable2[] [= {e1,e2, ...}] ...;
```

The first element of an array starts at 0. Use more than one set of brackets (**[]**) for multi-dimensional arrays. Do not define the maximum number of elements in the array, because array elements are allocated when you access them. The maximum number of elements that can be placed in an array is approximately 2 billion. Use the **_length()** method to determine the number of elements in an array. The syntax for using this method is `variable._length()`.

```
[static] TypeName (*variable1)([ArgDecl1, ArgDecl2,...]){=function_name};
```

Where *ArgDecl* has the almost the same syntax as a variable declarations, except **static** is not supported and the ampersand (**&**) operator is used to specify call by reference parameters. Call by reference array and hash table parameters require parentheses around the ampersand (**&**) and *id*.

To empty an array, use the following statements:

```
array._makeempty();           // Empty the array
array=null;                   // Empty the array. Same as above
```

Differences from C++

- Space for array elements is allocated when you index into the array.
- You cannot use pointer variables to traverse array elements.
- You cannot limit the number of elements that the array may contain.
- Specifying an array variable WITHOUT the **[]** operator does not return a pointer to the first element. Instead, it refers to the entire array. This allows you to copy one array to another, or define a function which returns a copy of an array.

- There is not a **sizeof** function which tells you the size of the array in bytes. There is a **_length** method which tells you the number of elements in the array.

Example:

```
int gai[]={1, 7, 12};
int gaai[][]={{1},{1,2},{1,2,3}}; //two dimensional array
_str    gastring1[]={"Value1", "Value2"};
typeless    gat[]={"String", 1, 2.4};

defmain()
{
    t=gai;                // Copy all the array elements into a local container
                        // variable
    t[t._length()]=45;    // Add another array element.
    for (i=0;i<t._length();++i ) {
        messageNwait("t["i"]="t[i]");
    }
}
```

Hash Tables

Slick-C® provides a **:[]** hash table, which is declared similar to array types.

Hash tables are indexed with a string **:[]** operator. Use the following syntax to define a hash table variable:

```
[static] TypeName variable1:[ ]
    [= {s1=>e1, s2=>e2, ...} , variable2:[ ] [= {s1=>e1,s2=>e2, ...} ...];
```

Struct

Slick-C® supports C-style structs and unions. Static structure members are not supported. Slick-C structs cannot have member functions. Structures are typically used to logically group data. For example, a record in a database might have a name, address, and phone number. This can be logically grouped into a structure to more easily group data for a complete record. Structures can also have the added effect of reducing the number of global variables. The following is the syntax for defining a structure:

```
[static] struct [StructName ] {
    member-variable-decl1;
    member-variable-decl2;
} [variable1[={e1,e2, ...}] , variable2[={e1,e2, ...}], ...];
```

The **struct** declaration provides the option of defining your own type called *StructName* and to declare one or more variables. The syntax of *member-variable-decls* is identical to declaring other variables, except that static structure members are not supported. Use the following syntax for accessing a member of a struct variable:

```
variable.member_name
```


Example:

```

struct PHONERECORD {      // Define a type called PHONERECORD
    _str Name;
    _str PhoneNumber;
} gPhoneRecord;          // Declare a variable of that type

PHONERECORD gPR={        // Declare a variable of type PHONERECORD.
    "Steve", "555-1346"
};

PHONERECORD gRecordArray[]; // See arrays below
struct PHONERECORD2 {      // Define a type called PHONERECORD2
    _str Name;
    _str PhoneNumber;
    _str FaxNumber;
};

defmain()
{
    messageNwait("Name="gPR.Name" PhoneNumber="gPR.PhoneNumber);
    t=gPR; // Copy phone record data into a local container variable
           //Container variables can access structure elements as an array.
    messageNwait("Name="t[0]" PhoneNumber="t[1]);
}

```

Differences from C++

- There is no **sizeof** operator like in C++. Since the Slick-C® interpreter stores all types as container variables, the **sizeof** operator has no meaning.
- Space for structure elements is allocated when you access the element.
- Structure data is not contiguous. The Slick-C interpreter stores all types as container variables, including the members of a struct.

Union

Slick-C® supports C-style unions. Unions are typically used in place of a struct in the case where you have mutually exclusive member variables. In this case, a union requires less memory than a struct. Memory is only allocated for one member variable at a time. The syntax for defining a union is shown in the following example:

```

[static] union [UnionName ] {
    member-variable-decl1;
    member-variable-decl2;
} [variable1[={e1}] , variable2[={e1}], ...];

```

The **union** declaration provides the option to define your own type named *UnionName* and to declare one or more variables. The syntax of *member-variable-decls* is identical to declaring other variables, except that static union members are not supported. The syntax for accessing a member union variable is *variable.member_name*.

Example:

```
union {
    int i;
    _str s;
    double d;
} gu={1};    // Type checking here is with first member variable.

#define KIND_INT 1
#define KIND_STRING 2
#define KIND_DOUBLE 3
defmain()
{
    struct {
        int kind;
        // Here we are nesting a union inside a struct.
        // This union only requires space for one of these
        // members at a time.
        union {
            int i;
            _str s;
            double d;
        }u;
    } x;

    x.kind=KIND_INT;x.u.i=1;
    ...
    switch (x.kind) {
    case KIND_INT:
        messageNwait("x.u.i="x.u.i);
        break;
    case KIND_STRING:
        messageNwait("x.u.s="x.u.s);
        break;
    case KIND_DOUBLE:
        messageNwait("x.u.d="x.u.d);
        break;
    }
}
```

Anonymous Unions

An anonymous union is a union member variable that is not named. This saves you from having to type the union member variable name.

Example:

```
defmain()
{
    struct {
        int kind;
        union {
            int i;
            _str s;
            double d;
        }; // No name for this union member variable
    } x;
    x.kind=1;x.i=1;
}
```

Pointers to Variables

Pointers are used in Slick-C® to resolve certain programming tasks. If you access an invalid pointer, the Slick-C macro stops. Pointer variables are declared using the following syntax:

```
[static] TypeName *variable1[=&v1] , *variable2[=&v2] ...;
```

The unary & operator is used to return the address of a variable. The unary * operator is used to dereference a pointer. Use the operator -> (for example, `p->member-variable`) to access members of a pointer to a structure.

IMPORTANT When a module is reloaded, static variable addresses change. Make sure you reinitialize global pointer variables which point to static (module scope) variables.

Pointers to Functions

Function pointer variables are useful for callback functions. If accessing an invalid function pointer, the Slick-C® macro stops.

IMPORTANT When a module is reloaded, static function addresses change. Make sure you reinitialize global function pointer variables which point to static (module scope) functions.

The syntax for calling a pointer to function variable is as follows:

```
(*pfn)([e1, e2, ...])
```


Strings

String variables are declared using the `_str` type. You can get the length of the string using the `length` built-in.

Slick-C® has additional string operators so that the compiler always knows whether to perform a string or numeric operation. The `+` operator always means add two numbers, and the concatenation operator `:+` always means concatenate two strings.

Typeless Container Variables

Typeless container variables can be declared using the `typeless` type. A typeless container can be passed to a function using the `var` type. Container variables are declared with the `typeless` keyword or are declared by an assignment to an undefined variable. The container variable can store the contents of any typed variable. This is easy for the interpreter since all typed variables are stored as container variables. At run time, the interpreter must check the current type of the container variable (and sometimes convert it) to perform an operation.

The compiler performs (double) floating point arithmetic on container variables. Currently, there is only a very small difference in speed between arithmetic operations on integer type variables and container variables, because the Slick-C® language has been optimized for string and container operations.

STRINGS

Example:

```
typeless t;
t=1; // Store an integer
// Convert the contents of the variable t to a
// floating pointer number (double type) and add 1.
// NOTE: The interpreter is smart and will only perform
// integer arithmetic here.
t=t+1;
// Since + always means addition, the compiler converts
// string constants to the smallest possible numeric type
t=t+"1";

// Declare string variable
_str s;
s=1; // Compiler will convert int to string
t=1.2;
// Must cast string type to int or compiler will complain.
t=(int)t+(int)s; // Result is 2 and not 2.2 because of the cast of t
                // to int.

//Destroy the integer and make an array.
//Also make the 0 element an integer.
t[0]=1;
t[1]=2; // Add another element;

t2=t; //Copy the array and all its elements
struct {
    int x;
    int y;
} st;
st.x=1;st.y=2;
t=st;
// Print out the elements of the structure
for (i=0;i<t._length();++i ) {
    messageNwait("t["i"]="t[i]");
}
```

Mathematical Operators

Slick-C® uses the operator precedence of C. The table below contains the unary operators that an expression can use.

Operator	Description
<code>!e1</code>	Logical NOT. Result is 1 if <code>e1</code> evaluates to 0. Otherwise the result is 0.
<code>~e1</code>	Bitwise complement.
<code>-e1</code>	Negation.
<code>+e1</code>	No change.
<code>++v1</code>	Increments the variable <code>v1</code> and returns the result.
<code>v1++</code>	Returns the value of <code>v1</code> and then increments the variable <code>v1</code> .
<code>--v1</code>	Decrements the variable <code>v1</code> and returns the result.
<code>v1--</code>	Returns the value of <code>v1</code> and then decrements the variable <code>v1</code> .

The binary and ternary operators for the Slick-C language are listed in the table below. In addition to the operators listed in the previous table, string concatenation is implied. If a binary operator does not exist between two unary expressions, concatenation is automatically performed.

All numeric operators, except bitwise operators, support floating point numbers. Bitwise operators support 32-bit integers for all platforms.

Operator	Description
<code>=</code>	Assign right operand to left operand.
<code>+=</code>	Add left operand to right operand and assign to left operand.
<code>-=</code>	Subtract right operand from left operand and assign to left operand.
<code>/=</code>	Divide left operand by right operand and assign to left operand.
<code>*=</code>	Multiply left operand with right operand and assign to left operand.
<code> =</code>	Bitwise OR left operand with right operand and assign to left operand.
<code>^=</code>	Bitwise XOR left operand with right operand and assign to left operand.
<code>&=</code>	Bitwise AND left operand with right operand and assign to left operand.

Operator	Description
<code>e1?e2:e3</code>	If expression <code>e1</code> is TRUE (not the string 0), expression <code>e2</code> is returned. Otherwise, expression <code>e3</code> is returned.
<code>&&</code>	Logical AND. If left hand expression is false, right hand expression is not evaluated.
<code> </code>	Logical OR. If left hand expression is true, right hand expression is not evaluated.
<code> </code>	Bitwise OR.
<code>^</code>	Bitwise XOR.
<code>&</code>	Bitwise AND.
<code>==</code>	Equal. Performs a numeric or string comparison depending on the operands. This function is NOT identical to the C strcmp function (see <code>:==</code> operator below). If both operands are numbers, a numeric comparison is performed. Otherwise a string comparison is performed. In any case, leading and trailing spaces and tabs are stripped before the comparison is performed.
<code>></code>	Greater than. Performs a numeric or string comparison depending on the operands. See <code>==</code> operator.
<code>>=</code>	Greater than or equal. Performs a numeric or string comparison depending on the operands. See <code>==</code> operator.
<code><</code>	Less than. Performs a numeric or string comparison depending on the operands. See <code>==</code> operator.
<code><=</code>	Less than or equal. Performs a numeric or string comparison depending on the operands. See <code>==</code> operator.
<code>!=</code>	Not equal. Performs a numeric or string comparison depending on the operands. See <code>==</code> operator.
<code>:==</code>	Exactly equal. Always performs string comparison. This is equivalent to the C expression <code>(strcmp(a,b)==0)</code> .
<code>:!=</code>	Not exactly equal. Always performs string comparison.
<code>:<=</code>	Exactly less than or equal. Always performs string comparison.
<code>:<</code>	Exactly less than. Always performs string comparison.
<code>:>=</code>	Exactly greater than or equal. Always performs string comparison.
<code>:></code>	Exactly greater than. Always performs string comparison.
<code>:+</code>	Concatenation.
<code><<</code>	Bitwise shift left.

Operator	Description
>>	Bitwise shift right.
+	Addition.
-	Subtraction.
/	Division with possible floating point result.
intdiv	Division with integer result.
*	Multiplication.
%	Remainder.

Two sets of comparison operators exist. The operators <, >, =, !=, <=, and >= perform a numeric comparison if both string expressions are valid numbers. The operators :<, :>, :==, :!=, :<=, and :>= always perform a string comparison.

Select the appropriate comparison operator for performing a string or numeric comparison. Expressions may extend across line boundaries if the line ends in a binary operator or if the line ends with a backslash.

The table below shows examples of math operators in Slick-C™.

Example	Operator
(1.0==1)	== true
(1e2==100)	== true
(1e2:=100)	== false
(" abc "=="abc")	== true
(" abc "==="abc")	== false
(" abc "!="abc")	== true
(" 1 "=="1)	== true
(" 1 "==="1)	== false
("abc":<"def")	== true
1 2	:=="12"
1 (2)	:=="12"
pow(4,2)	==16
5%2	==1
5/2	==2
5/2.0	==2.5
5 intdiv 2.0	==2
5&2	==0
5 2	==7

MATHEMATICAL OPERATORS

Example	Operator
<code>(10<7)</code>	<code>== false</code>
<code>(10:<7)</code>	<code>== true</code>

Declarations

Variables and functions are declared in the same way in Slick-C® as they are defined in C.

Scoping and Declaring Variables

The Slick-C® language supports global, static (module), and local scope variables. Global variables can be accessed by any module. The scope of static and local variables are limited to the module in which they are defined. Variables are declared the same way that they are defined in C++. The differences are defined in the following list:

- Classes are not supported.
- **void** is only allowed as the returned type of a function.
- **char**, **short**, and **float** types are not yet available. All numeric types are signed. The current Slick-C numeric types are **int**, **long**, and **double**.
- Slick-C has a built-in string type **_str**.
- Arrays, while declared similarly, cannot have a size limit. Arrays elements are always dynamically allocated.
- Slick-C provides a **:[]** hash table operator which is similar to the array operator **[]**, except that hash tables are indexed with a string type.
- There is no **sizeof** operator. Since the Slick-C interpreter currently stores all types as container variables, the **sizeof** operator has no meaning.
- Static structure members are not supported.

Declarations are broken up into categories.

Simple Variable

A simple variable is a variable that is not an array, hash table, structure, or function pointer. The syntax for defining a simple variable is:

```
[static] TypeName variable1[=expression1] , variable2[=expression2] ...;
```

The comma is used to declare more than one variable of the same type. Local variables do not have to be defined. Using a variable not already defined as global or constant declares the variable to be a local typeless variable. However, you may declare variables within the scope of a function to ensure that the variable will be local even if the name is declared elsewhere as a global or constant.

DECLARATIONS

Example:

```
// Declare a global integer
int gi=1;
//Declare a module scope integer
static int    si=2+4;
//Declare some global string variables
_str    gstring1="Value1", gstring2="Value2";
//Declare a global large floating point variable
double gd=1.4;
//Declare a global typeless variable
typeless gt="xyz";
defmain( )
{
    _str s="ess";
    // Declare a local string variable and initialize it to "ess"
    t=gi;
    // Copy gi into local container variable t.
    message("t=t"s="s");
}
```

Details about Variable Initializations

Global and static numeric variables, which include **boolean**, **int**, **long**, and **double**, are initialized to 0 when there is no specified value provided. Local variables of any type are not initialized.

Global and static variables declared as **typeless** or **_str** are initialized with "" (a zero length string) when there is no initialization value provided.

Example:

```
boolean    globalboolean=true;
int         globalint;
double     globaldouble;
defmain()
{
    // Will print message "globalboolean=1 globalint=0 globaldouble=0"
    message("boolean="globalboolean" "globalint" "globaldouble);
}
```

Type Casting

Slick-C® enforces strict type checking on typed non-container variables. There are times when you need to convert one type to another. Type casting helps communicate that the compile process is accurate. Also, type casting can change the value of an expression.

The syntax for converting one type of checking to another is the following:

```
(TypeName) expression
```

Some casts are not permitted in Slick-C. For example, you cannot cast a struct type to another struct type.

Differences from C++

- Slick-C® does not support the C++ function style cast mechanism.
- Slick-C does not permit pointer types to be cast.

Example:

```
defmain()  
{  
    int i;  
    double d;  
    d=1.2;  
    i=(int)d; // i gets the value 1, NOT 1.2  
    typeless t;  
    t=1.2;  
    i=t;      // Here i gets 1.2 BUT  
    boolean b;  
    b= i!=0;  // Can't use cast here.  
    i=(int)b; // Need cast here.  
}
```

DECLARATIONS

Statements

Slick-C® statements are constructed in the same manner as the statements in the C language.

Assignment Operator

The simple assignment statement has the syntax *variable=expression*. For example:

```
i=1;
i=i+1;
```

Assignment statements can be cascaded (**x=y=z**). Assignment statements within **if** statements are not allowed. The compiler flags assignments within **if** statements as an error.

if Statement

The syntax for an **if** statement is the following:

```
if (expression) statement [else statement]
```

statement can be a C-style statement block which contains multiple statements. For more information, see [Block Statement](#).

IMPORTANT The value 0 for all types is false. All other values are true. Like C++, Slick-C® uses the value 0 for null pointers. For the string type, only a one byte length string where the first character is an ASCII 0 is false. A 0 length string ("") is true when used in a boolean expression.

Example:

```
if (x<y) a=1;
if (x="a") {
    y=1;
} else if (x="b") {
    y=2;
} else if (x="c") {
    y=3;
} else if (x="d") {
    y=4;
}
```

Block Statement

A statement block is typically used to allow multiple statements within an **if** or loop construct. However, it can also be used to declare a new local scope. A statement block has the following syntax:

```
{
    statement
    statement
    ...
}
// statement may declare local variables.
```

STATEMENTS

Example:

```
if (i<1) {
    int x=1;{
    int x;
    // Can do the assignment here
    x=3;
}
// The variable x will be 1 here and not 3.
}
```

Loops

Slick-C® supports C-style **while**, **for**, and **do** loops.

while

The **while** loop evaluates *condition_exp* first and then executes *statement* if *condition_exp* is true (not the value 0). The *statement* will continue to be executed until *condition_exp* becomes false (0) or a break statement is reached.

Example:

```
[label:] while (condition_exp) statement
```

for

The C-style **for** loop is free-form. The expressions before the first semicolon of the **for** loop are executed before entering the loop. The *condition_exp* expression is checked before entering the **for** loop also. If *condition_exp* is true (not the value 0), the *statement* is executed. The *statement* continues to be executed until *condition_exp* becomes false (0) or a break statement is reached. When the bottom of the **for** loop is reached, but before *condition_exp* is checked again, the expressions after the second semicolon are executed.

Example:

```
[label:] for (b4e1 ,b4e2 ... .b4e3); [condition_exp] ; {cont_e1,cont_e2 ...
,cont-e3}) statement
```

do

The **do** loop executes *statement* first and then evaluates *condition_exp*. If *expression* is true (not the value 0), the *statement* continues to be executed until *expression* becomes false (0) or a break statement is reached.

```
[label:] do statement while ( condition_exp );
```

break

Loops might be exited with the **break** primitive. The **break** primitive supports an optional **label** argument (like Java™). If specified, the label must match the *label* of one of the loops that you are currently using.

continue

The **continue** primitive might be used to skip to the top of a loop. Using **continue** on a **for** loop causes the expressions after the second semicolon to be executed before *condition_exp* is checked. When

continue is used on a **do** statement, the *condition_exp* is not checked and execution resumes at the top of the loop.

Loops might be exited with the **continue** primitive. The **continue** primitive supports an optional **label** argument (like Java). If specified, the label must match the *label* of one of the loops that you are currently using.

Example:

```
outerloop:
  for(i=1;i<3;++i){
    for (j=1;++j) {
      if (j==3) break outerloop;//Exit both loops
      messageNwait("i="i);
    }
  }
  for(i=1;i<10;++i){
    messageNwait("i="i);
  }
  // The above for loop is equivalent to the loop below.
  i=1;
  for (;i<10;) {
    messageNwait("i="i);
    ++i;
  }
  for(i=1;i<10;++i) messageNwait("i="i);
  i=1;
  while (i <10) ++i;
  i=1;
  do {
    messageNwait("i="i);
  } while (i<10);
  status=search(":", "@" /* No message option. */);
  for(;;) {
    if ( status ) {
      break;
    }
    get_line(line);
    messageNwait("found match line="line);
    status=repeat_search();
  }
}
```

parse Statement

The syntax for **parse** is *parse string with template*. When we converted our REXX-style language to C-style, we had to leave the syntax of this statement alone because there was no reasonable way to represent this statement as a function. This statement parses *string* as specified by *template*.

The table below shows what *Template* may contain.

Item	Description
<i>variable_name</i>	Output variable.
.	Null output variable.

Item	Description
<i>nnn</i>	Number specifying new parse column.
<i>+nnn</i>	Amount to increment parse column relative to start of last string found or last column setting.
<i>-nnn</i>	Amount to decrement parse column relative to start of last string found or last column setting.
<code>'text'[,search_options]</code>	String constant to search for. If found, parse column becomes first character after <i>text</i> . Otherwise parse column becomes first character after length of string being parsed. <i>search_options</i> is an optional expression that may evaluate to a string of one or more of the option letters U , R , I , and Y . U specifies UNIX regular expressions. R specifies SlickEdit regular expressions. B specifies Brief regular expressions. I specifies a case insensitive search. Y specifies a binary which search allows positions in the middle of a DBCS character (only effects Japanese operating systems). See the topic on Regular Expressions in the Help system.
<code>(expression)[,search_options]</code>	String <i>expression</i> to search for. If found, parse column becomes first character after text. Otherwise parse column becomes first character after length of string being parsed. See above for a description of the search options.

The rules for parse column are:

- The parse column is initialized to column 1.
- If a column or column increment specifies a column greater than the length of the string being parsed, the parse column is set to the length of the string being parsed plus one.
- If a column decrement specifies a column less than the length of the string being parsed, the parse column is set to column 1.

The rules for setting output variables are:

- Output variables are set in groups. An output variable group is defined to be consecutive variables with no search or column specifiers between them.
- Before variables of an output variable group can be set, the end parse column within the source string must be found. In the case the end parse column is set by a search, the end parse column for this output variable group becomes the first character to the left of the text found. In the case the end parse column is set by a column or column increment the end parse column becomes the first character to the left of the column. The start parse column is the current parse column as specified by the template.
- A word parse of the text between the start and end columns is performed to set the variables in an output variable group if the group contains more that one variable. Otherwise the one output variable is set to the text between the start and end columns of the source string. Each variable set by a word parse will have no leading or trailing tabs/spaces except for the last output variable which is set to the rest of the sub-string.
- If the start column is greater than the end column the variables in the output group are set to null.

The following code shows examples of **parse**:

```

parse '1 2 3' with a b c;
// Results are  a=='1', b=='2', c=='3'

parse '1 '\t' 2 '\t' 3' with a b c;
// Results are  a=='1', b=='2', c=='3'.  Note that tab and space characters are
stripped.

parse '1 2 3' with a . b;
// Results are  a=='1', b=='3'

parse 'xxx1 2 3yyy 4 5' with 'xxx' a b c 'yyy' d e;
// Results are  a=='1', b=='2', c=='3', d=='4', e=='5'

parse 'xxx1 2 3yyy 4 5' with 'xxx' a 'yyy' b;
// Results are  a=='1 2 3', b==' 4 5'

parse 'xxx1 2 3yyy 4 5' with 'xxx' +0 a 'yyy' +0 b;
// Results are  a=='xxx1 2 3', b=='yyy 4 5'

parse 'c/x/y' with 1 delim +1 s1 (delim) s2 (delim) options;
// Results are delim=='/', s1=='x', s2=='y', options=='

```

switch Statement

Slick-C® supports the C **switch** statement. The Slick-C **switch** supports integers and string types. The **switch** statement uses the syntax shown in the following example:

```

switch (expression) {
    [ case expression:
        statements
    ]
    [ case expression:
        statements
    ]
    ...

    [ default:
        statements
    ]
}

```

The **switch** expression is evaluated and compared against all the case expressions. After a match is found, ALL statements below the case are executed, including those statements found in the next case and the default, until a break statement is reached.

STATEMENTS

Example:

```
outerloop:
    for (i=1; ++i) {
        switch (i) {
            case 1:
            case 2:
                messageNwait("i=1 or i=2");
                break;
        // Done with these cases
            case 3:
                break outerloop;
        }
    }
```

Functions

A function can be called from the macro language. Slick-C® has four kinds of functions: procedures, commands, library functions, and built-ins. The following sections describe each of the function types.

Defining a Procedure

There are several differences between defining a procedure and defining a command with the **_command** primitive. The scope of a procedure may be limited to a module. Command functions are invoked by typing the name on the SlickEdit® command line, from a menu item definition, by using the **execute** function, or by typing the command name followed by arguments in parentheses in a Slick-C® expression. Procedures can only be called by the latter method and cannot be bound to keys. A procedure name must be a valid Slick-C identifier (same as C identifier). The name of a command can be a string constant containing a single character such as “/” (SlickEdit uses the slash to define a search command). Use the following syntax to define a procedure:

```
[static] [TypeName] id (TypeName [&] id1, TypeName [&] id2, ...)
```

```
{
    statement
    statement
    ...
}
```

Argument Declarations

The syntax for an argument declaration is the same as for declaring a variable, except that the **static** keyword cannot be used. An ampersand (&) before the *id* declares a call by reference parameter. Call by reference array and hash table parameters require parentheses around the & and *id*.

The last argument in the declaration list may be an ellipsis to indicate that the function accepts more arguments of any type. Use the **arg** function to access these optional arguments.

TypeName specifies the return type of the function. For more information, see [Types](#). If the return type is not specified, the function will return typeless. When the **void** type is used, a value cannot be specified to the return statement. The return statement is used to specify the result of the function call and exit the function.

The optional **static** keyword is used to limit the scope of a procedure to the module in which it is defined. By default, procedures are global and can be accessed by any module. Procedures are called by specifying the name followed by comma delimited arguments, if any, in parentheses.

Example:

```
boolean proc(int &p1, _str p2, _str (&list)[], int (*&pfn)(int))
```

```
{
    return(true)
}
```

NOTE The **list**, **p1**, and **pfn** parameters are call by reference parameters. Like C++, the **list** parameter requires parentheses around the & reference operator and the name, because the **[]** operator would otherwise be processed first. This avoids deviating much from C++ syntax. The command **pfn** is a reference to a pointer to a function.

Procedures can have up to 15 arguments defined. The procedure can be called with more arguments than defined by the procedure declaration. These extra arguments and the arguments defined in the procedure

declaration can be retrieved by the **arg** function. Defining arguments with default values instead of using the **arg** function makes your code more understandable. Calling the **arg** function with no parameters returns the number of parameters with which the function was called. The minimum number of arguments with which the procedure may be called is defined by the procedure heading. A parameter of type **var** specifies a typeless variable passed by reference.

Default Arguments

The assignment operator has special meaning in an argument declaration. It defines a default value for an argument. The default value is used if the caller does not specify the parameter. Default arguments must always be specified in the function definition. Unlike C++, default arguments in prototypes do not have an effect on the compiled code.

Example:

```
static int proc2()
{
    return("before");
}
int proc(_str p1=proc2()+ "after", int p2=2)
{
    return(p1+p2);
}
defmain()
{
    proc();           // Use defaults ("beforeafter" ,2)
    proc("param1");   // Use the second default value
    proc("param1",3); // Specify both values
    proc(,3);         // This is not allowed
}
```

Defining a Command

The **_command** primitive is used to define a new command with argument completion. A command can be invoked by typing its name on the SlickEdit® command line, selecting it from a menu item definition, pressing a key, calling it in a Slick-C® function, or typing its name followed by arguments in parentheses in a Slick-C expression. Command procedures always have global scope and can be bound to a key with the Key Bindings dialog box (**Tools > Options > Key Bindings**).

The syntax for defining a command is shown in the following example:

```
_command [TypeName | void] name1[,name2 [,name3... ]( [ArgDecl1, ArgDecl2, ...] )
    [name_info(const_exp)]
{
    statements
}
```

TypeName specifies the return type of the command (see [Types](#)). If *TypeName* or **void** is not specified, the return type is typeless. When the **void** type is used, a value cannot be specified to the return statement. The return statement is used to specify the result of the function call and exit the function. If your command uses the **arg** function to access arguments, specify an ellipsis for the last argument.

The syntax for *ArgDecls* is the same as for declaring a variable, except that the **static** keyword may not be used. In addition, an **&** before the *id* declares a call by references parameter. Call by reference array and hash table parameters require parentheses around the **&** and the *id*. However, all typed or named arguments must have a default value.

The last argument in the declaration list can be an ellipsis to indicate that the function accepts more arguments of any type. Use the **arg** function to access these optional arguments.

The name of a command may be a valid Slick-C identifier, or a string constant of a length of one, such as `"/`". SlickEdit uses the slash to define a search command.

Example:

```
// Allow command in read only mode
// Use ellipsis because this accesses arguments
_command int goto_line(...) name_info('VSARG2_READ_ONLY|VSARG2_REQUIRES_EDITORCTL)
{
    param=arg(1);
    if (param==" " || ! isinteger(param)) {
        message('Please specify line number');
        return(1);
    }
    p_line=param;
    return(0);
}

_command void mycommand(_str filename="") name_info(FILE_ARG)
{
    if (filename=="") {
        _message_box("No filename specified");
    }
    message("filename="filename);
}
```

Commands receive unnamed command line arguments by calling the **arg** function. When a command is invoked from the command line, the expression **arg(1)** contains the rest of the command line after the name with leading spaces removed. For example, invoking the edit command **e file1 file2** calls the **e** command with **file1 file2** in **arg(1)**. The **parse** built-in is an excellent function for parsing a command line string (see the Help system for more information on parsing). When another macro calls a command, more than one argument string can be passed. Calling the **arg** function with no parameters returns the number of parameters with which the command or procedure was called.

name_info Attributes

The optional **name_info** expression is used to specify command argument completion rules and restricts when the command may be executed.

const_exp is a single constant expression. A comma (,) character in the string indicates the end of an argument.

The first argument in *const_exp* indicates the type of word arguments the command accepts and is used for argument completion purposes. For a list of already defined argument types, look in the `slick.sh` file for constants that end in **_ARG**. *const_exp* may contain one or more of the **_ARG** constants. Separate each **_ARG** constant with a space. An asterisk (*) character may be appended to the end of a completion constant to indicate that one or more of the arguments may be entered. The second argument (after the

FUNCTIONS

quoted comma) specifies when the command should be enabled or disabled. One or more of the flags in the table below can be specified and ORed together with the bitwise OR (|) operator.

Flag	Description
VSARG2_CMDLINE	Command supports the command line. VSARG2_CMDLINE allows a fundamental mode key binding to be inherited by the command line.
VSARG2_MARK	ON_SELECT event should pass control on to this command and not deselect text first. Ignored if command does not require an editor control.
VSARG2_QUOTE	Indicates that this command must be quoted when called during macro recording. Needed only if command name is an invalid identifier or keyword.
VSARG2_LASTKEY	Command requires last_event value to be set when called during macro recording.
VSARG2_MACRO	This is a recorded macro command. Used for completion.
VSARG2_TEXT_BOX	Command supports any text box control. VSARG2_TEXT_BOX allows a fundamental mode key binding to be inherited by a text box.
VSARG2_NOEXIT_SCROLL	Do not exit scroll caused by using scroll bars. Ignored if command does not require an editor control.
VSARG2_EDITORCTL	Command allowed in editor control. VSARG2_EDITORCTL allows a fundamental mode. Key binding to be inherited by a non-MDI editor control.
VSARG2_NOUNDOS	Do not automatically call _undo('s') . Require macro to call _undo('s') to start a new level of undo.
VSARG2_READ_ONLY	Command allowed when editor control is in strict read only mode. Ignored if command does not require an editor control
VSARG2_ICON	Command allowed when editor control window is iconized. Ignored if command does not require an editor control.
SARG2_REQUIRES_EDITORCTL	Command requires an editor control.
VSARG2_REQUIRES_MDI_EDITORCTL	Command requires MDI editor control.
VSARG2_REQUIRES_AB_SELECTION	Command requires selection in active buffer.
VSARG2_REQUIRES_BLOCK_SELECTION	Command requires block/column selection in any buffer.
VSARG2_REQUIRES_CLIPBOARD	Command requires editorctl clipboard.

Flag	Description
VSARG2_REQUIRES_FILEMAN_MODE	Command requires active buffer to be in fileman mode.
VSARG2_REQUIRES_TAGGING	Command requires <ext>_proc_search/find-tag support.
VSARG2_REQUIRES_SELECTION	Command requires a selection in any buffer.
VSARG2_REQUIRES_MDI	Command requires MDI interface maybe because it opens a new file or uses _mdi object. Commands with this attribute are removed from pop-up menus in which the MDI interface is not available (editor control OEMs).

Example:

```
#include "slick.sh"
// This command supports completion where the first argument
// is a filename and the second argument is an environment variable.
_command test1(...) name_info(FILE_ARG " ENV_ARG)
{
    parse arg(1) with file_name env_name;
    message("file_name="file_name" env_name="env_name);
}
// This command is enabled only when the target is an editor control
// which has a selection.
_command void gui_enumerate()
    name_info('VSARG2_REQUIRES_EDITORCTL|VSARG2_REQUIRES_AB_SELECTION)
{
    ...
}
// This command supports completion on multiple filenames
_command e,edit(...) name_info(FILE_ARG '*', 'VSARG2_CMDLINE|VSARG2_REQUIRES_MDI)
{
    ...
}
```

The **edit** command allows any number of file name arguments to be given. When the user is presented with a selection list of file names, many files may be selected with the spacebar key. If an asterisk (*) is appended to the end of a completion constant, that command must support a space-delimited list of strings. Double quotes are placed around arguments with embedded spaces.

The value of *const_exp* may be retrieved by the built-in function **name_info**.

OnUpdate Functions

A Slick-C® command can have a corresponding **_OnUpdate_<cmdname>** function. This function is used to provide more precise control over the enabling and disabling of a command than the **name_info** command can provide.

Example:

```
int _OnUpdate_linehex(CMDUI &cmdui,int target_wid,_str command)
{
    if ( !target_wid || !target_wid._isEditorCtl()) {
        return(MF_GRAYED);
    }
    if (p_UTF8) {
        return(MF_UNCHECKED|MF_GRAYED);
    }
    if (p_hex_mode==2) {
        return(MF_CHECKED|MF_ENABLED);
    }
    return(MF_UNCHECKED|MF_ENABLED);
}
```

Function Prototypes

Function prototypes provide the compiler with type information about a function without providing any code. Slick-C® reduces the need for prototypes by performing some argument checks at link time. When the linker finds an uninitialized variable error, it recommends that you add a function prototype to your source so the compiler can find your error. You might need a function prototype if you want to use the function address in an expression. Prototypes are not allowed for event functions.

The syntax for defining a function prototype is identical to defining a function except that a semicolon (;) is placed after the closing parentheses of the parameter list. Unlike C++, default arguments in prototypes have no effect on the compiled code. No code or **name_info** is given.

Example:

```
int proc(_str s,_str list[]); // Function prototype
int (*pfn)(_str s,_str list[])=proc; //Pointer to function
_command void command1(...); // Function prototype
_command void command1(...) { //Must have ... here to match prototype
    // Use arg function here to get or set arguments.
}
```

Library Functions

A library function is a function that was implemented in a dynamically loaded library and was not written in the Slick-C® language. A library function must follow Slick-C calling conventions and be registered with the interpreter.

Built-in Functions

A built-in function is a function that was implemented in the interpreter and was not written in the Slick-C® language.

Finding Functions

There are over 1200 documented functions and 200 properties. There are two ways to find the function that you seek. First, you can use the menu item **Help > Macro Functions by Category**, which displays smaller lists of these functions by category. Second, you can view source code for existing commands. If you do not know the name of the command but you do know the key that invokes the command, use the

what_is command or **Help > What is Key** to find the name of the command that is executed. Then, use the **find_proc** command or **Macro > Find Slick-C Proc** to display the macro source code.

Differences between Commands, Built-ins, and Defs

- A command definition looks like a procedure that starts with the **_command** primitive, and has an optional **name_info** construct after the arguments. Built-ins are not defined.
- Commands and built-ins always have global scope. Procedures may have the static (module) or global scope.
- Commands may be bound to keys. Built-ins and procedures may not.
- Commands may be invoked from the command line or the **execute** function. Built-ins and procedures may not.
- A command may be given the same name as a built-in. However, this limits how the command may be called within a macro (use the **execute** function). None of the commands have the same name as a built-in so you can call any command just like any other function.
- Only commands may be given non-alphanumeric single character names such as **+**, **=**, **!**, **@**, **#**, **\$**, etc. However, this limits how the command can be called within a macro (place the command in quotes or use the **execute** function).

defmain: Writing Slick-C® Batch Files

A batch macro contains a special function named **defmain**. Slick-C batch files have the extension **.e**. Batch macros can be invoked by typing the name (extension not required) followed by arguments on the SlickEdit® command line, quoting the name in a macro, or by using the **execute** function. If the batch macro needs to be recompiled, the Slick-C translator is invoked before the batch macro is executed. Do not use the **load** command to load a batch program, because **defmain** is not invoked and an error will result. If you load a batch program that you do not want, use the **unload** command to unload it. When a batch program is executed, the **defmain** procedure is called after the procedure **definit** is called. For more information, see [Module Initializations](#).

The syntax of the **defmain** function is shown in the following example:

```
[TypeName | void] defmain()
{
    statement
    statement
    ...
}
```

TypeName specifies the return type of the function. If *TypeName* or **void** is not specified, the return type is typeless. When the **void** type is used, a value cannot be specified to the return statement. The return value of **defmain** is placed in the predefined **rc** global variable.

NOTE The **execute** function only supports returning an **int** type. Check the global **rc** variable for other types.

The **arg** function is used to retrieve the command line arguments passed to the **defmain** procedure. All of the command line arguments will be in **arg(1)**. Use the **parse** statement to easily parse multiple space delimited arguments.

FUNCTIONS

The following example displays the arguments given to the macro on the SlickEdit message line. If you define a procedure in a batch program, use the **static** keyword to conserve memory. SlickEdit stores the names of global procedures and variables in a names table.

```
defmain()  
{  
    messageNwait("Arguments given: "arg(1));  
    parse arg(1) with word1 word2 .;  
    messageNwait("word1="word1" word2="word2);  
    return(0);  
}
```

Extending the editor with a batch macro has the advantage of conserving memory and reducing the size of the state file. Also, batch macros can be easily shared between multiple users. The editor keeps the batch macro loaded only while it is executing. External batch macro names and arguments are not supported by completion. To provide completion, you must define a command with the **_command** primitive and have it call the external batch program. If you name the command the same name as the batch program (without the extension), use the **xcom** command to bypass internal command searching. There are two ways to invoke a Slick-C batch macro:

- Type the name of the module followed by arguments on the SlickEdit command line.
- Type **vs -p *program*** at the shell prompt, where *program* is the name of the batch program and *vs* is the name of the editor. Alternatively, you may use the **-r** option to have SlickEdit remain resident after the batch program completes.

For the above methods, SlickEdit invokes the translator to compile the source code file if the source code file exists and its date is later than the date of the `.ex` file.

Preprocessing

Preprocessing in Slick-C® is identical to C/C++. Preprocessing allows you to conditionally compile source code or define textual replacements.

The syntax of the Slick-C language conditional **if** block is any of the following examples:

```
#if expression
    [statements]
#elif expression
    [statements]
[#else
    statements]]
#endif
```

There may be nothing more than space or tab characters preceding a **#**. Text on the same line following **#else** or **#endif** is not permitted. The expression specified **MUST** be valid. To display an error message and end the compile, use the **#error** directive: `#error expression`.

Usually, preprocessing is used to write macros that operate on multiple operating systems or environments. The table below shows the constants that are automatically defined by the Slick-C translator.

Constant	Description
__PCDOS__	Non-zero if the current operating system is Windows. Use machine() built-in function to determine at run time which of these operating systems you are running.
__UNIX__	Non-zero if current operating system is UNIX compatible.
__NT__	Non-zero if the current operating system is Microsoft Windows NT® compatible.
__VERSION__	Version number of SlickEdit®.

Use the Slick-C translator **-d** option to define a constant for use by preprocessing. To test if a constant has been defined, use the **defined()** function.

Example:

```
#if !defined(my_constant)
    #define my_constant "default value"
#endif
#if __PCDOS__
    name="c:\util\myprog"
#elif __UNIX__
    name="/usr/bin/myprog"
#else
    #error "Don't know what to do for this OS"
#endif
```

#pragma

The **#pragma** preprocessor directive is used to change various options during the course of a compile. Slick-C® offers the following options:

- **#pragma option(autodecl [, (on | off)])**

This option enables **autodeclvars** and **autodeclctls** on and off. If the second argument is not given, value is restored to the command line invocation value. The default value is on. See the additional information on **autodeclvars** and **autodeclctls**.

- **#pragma option(autodeclvars [, (on | off)])**

This option enables and disables **autodeclvars**. If the second argument is not given, the value is restored to the command line invocation value. The default value is on. When enabled, the compiler attempts to automatically declare typeless variables when an assignment is made.

- **#pragma option(autodeclctls [, (on | off)])**

This option enables and disables **autodeclctls**. If the second argument is not given, the value is restored to the command line invocation value. The default value is on. When enabled, the compiler attempts to automatically declare control variables.

- **#pragma option(redeclvars [, (on | off)])**

This option enables and disables **redeclvars**. If the second argument is not given, the value is restored to the command line invocation value. The default value is off. When this option is enabled, any variable can be redeclared as a typeless variable. This is used to generate code for variables without having the type information.

- **#pragma option(strictreturn [, (on | off)])**

This option turns **strictreturn** on and off. If the second argument is not given, the value is restored to the command line invocation value. The default value is on. When enabled, and an explicit return type is given to a function, the compiler will flag an error if a return statement is potentially missing.

- **#pragma option(strictellipsis [, (on | off)])**

This option turns **strictellipsis** on and off. If the second argument is not given, value is restored to the command line invocation value. The default value is on. When enabled, the ellipsis must be given as the last argument to a function or prototype for type checking to succeed when calling function with extra arguments.

- **#pragma option(strictsemicolons [, (on | off)])**

Turns **strictsemicolons** on/off. If the second argument is not given, value is restored to the command line invocation value. The default value is off. When enabled, semicolons must terminate all statements. Use this pragma so that smart editing features work, and to prevent compilation errors.

- **#pragma option(strictparens [, (on | off)])**

Turns **strictparens** on/off. If the second argument is not given, value is restored to the command line invocation value. The default value is off. When enabled, parentheses must be given on all built-in functions. Use this pragma for more readable code.

- **#pragma option (strict [, (on|off)])**

This option enables **autodeclvars**, **autodeclctls**, **strictsemicolons**, and **strictparens**. If the second argument is not given, the value is restored to the command line invocation value. The default is off. See additional information on **autodeclvars**, **autodeclctls**, **strictsemicolons**, and **strictpa-**

rens. Use this pragma to turn on the highest level of type checking and syntax enforcement in the Slick-C translator. This option might include further strictness tests in the future.

NOTE All **#pragma** options may be specified by command line compiler options. Run `vst.exe` (UNIX: `vst`) with no arguments to view compiler options. You can use the VST environment variable to specify compiler options for all of your macros.

#region and #endregion

The **#region** directive lets you specify a block of code that you can expand or collapse when using Selective Display. The **#endregion** directive marks the end of a **#region** block. A **#region** block must be terminated with **#endregion**. The syntax of these directives is:

```
#region [name]
#endregion [name]
```

The *name* parameter (optional) is used to indicate the name of the region. This name is displayed in the editor window when the region is collapsed.

Example:

```
#region Region_1
void Test() {}
void Test2() {}
void Test3() {}
#endregion Region_1

defmain()
{
}
```

Including Header Files

The syntax of the **include** statement is:

```
#include string_constant
```

This statement includes the file specified by *string_constant* for compiling. If *string_constant* does not specify a path, the Slick-C® translator will look in the same directory of the main source file. Otherwise, the path specified by *string_constant* is searched. If the file is not found, the Slick-C translator looks for the include file in the directories specified by the VSLICKINCLUDE and VSLICKPATH environment variables. Include files may be nested.

Defining Controls

Usually, you do not need to communicate with the compiler about a control to which you refer; however there are a couple of cases in which you must declare a control. This can happen when the compiler cannot safely assume that you are referring to a control, or when the compiler cannot find the location of the dialog box of the control that you are trying to access. The compiler needs to tell the linker which dialog box is supposed to contain your control. The syntax for declaring a control variable is:

```
[_nocheck] ObjectName ControlName;
```

OR

```
[_nocheck] _control ControlName;
```

ObjectName can be one of the following: **_text_box**, **_combo_box**, **_image**, **_picture_box**, **_command_button**, **_radio_button**, **_check_box**, **_label**, **_list_box**, **_gauge**, **_spin**, **_vscroll**, or **_hscroll**.

The **_nocheck** keyword tells the compiler not to check if the control exists on the current dialog box.

The `[_nocheck] ObjectName ControlName;` declaration is only permitted outside the scope of a function.

The `_nocheck _control ControlName;` declaration already supports local procedure scope.

Example:

```
// Create a form with a command button named ctlcancel, and
// gauge named ctlgauge1. Set the cancel and default
// properties of the command button to true.
//
#include "slick.sh"
static boolean gcancel;
_command test()
{
    // Need to tell compiler ctlgauge1 is a control because
    // the form1_wid.ctlgauge1 is too ambiguous.
    _control ctlgauge1;

    // Show the form modeless so there is no modal wait
    form1_wid=show("form1");
    // Disable all forms except form1_wid

    disabled_wid_list=_enable_non_modal_forms(0,form1_wid);
    gcancel=0;
    for (i=1;i<=100;++i) {
        // Read mouse, key, and all other events until none are left or
        // until the variable gcancel becomes true
        process_events(gcancel);
        if (gcancel) {
            break;
        }
        // Do work here. Replace the delay below with the operation
        // you want to do. The delay makes this example look more real.
        delay(10);

        form1_wid.ctlgauge1.p_value=i;
    }
    // Enable all forms that were disabled.
    _enable_non_modal_forms(1,0,disabled_wid_list);
    form1_wid._delete_window();
}
defeventtab form1;
ctlcancel.lbutton_up()
{
    gcancel=1;
}
```

Defining Events and Event Tables

Event tables are used for describing event or key bindings by source code, creating event-driven dialog boxes, and describing inheritance.

def Primitive

The **def** primitive is used to bind a key sequence or event to a command or procedure and is not typically used when creating event-driven dialog boxes. The **defeventtab** primitive selects the active event table that the **def** primitive sets the bindings to. If there is no **defeventtab** declaration before the first **def** primitive, the **default_keys** event table is used. The **default_keys** event table defines the event handlers for Fundamental mode. The source code representing the bindings is translated and then the event tables are loaded either by the **load** command or by executing the module as a batch program. For more information on batch programs, see [defmain: Writing Slick-C® Batch Files](#). Even though executing the module as a batch program unloads the module when the **defmain** function terminates, the event table changes remain present. The following syntax is used for defining a key:

```
def {prefix_key} event [- event] [, event [- event]] ... = [command];
```

command can be either a command (defined with **_command**) or global procedure. If *command* is not specified, the existing event is unbound. The words *prefix_key* and *event* may be any valid event name. Some event names do not need to be enclosed in quotes.

Example:

```
def "A-x"=safe_exit;
// Note that "A-a" is different than "A-A"
// which requires the Alt and Shift keys to
// be pressed.
  def "A-?"=help;
  def "C-X" "b"=list_buffers;
  def \0 - \255= nothing;
```

The **defeventtab** primitive is used to define a new event table. The syntax for defining a an event table is:

```
defeventtab name;
```

name may contain a period (.) character. The period is used to separate the form name from the control name. The **def** primitive changes the binding of events of the last event table defined. If no event table is defined, the **default_keys** event table is used.

Example:

```
defeventtab c_keys;
  def " " =c_space;
  def "ENTER"=c_enter;
```

Event tables are global in scope. When an event table is loaded by the **load** command or by executing the module as a batch program, the new bindings replace the event bindings of the existing event table. If the event table specified by **defeventtab** does not exist, a new one is created.

Event-Driven Dialog Boxes

Event tables are for creating event-driven dialog boxes and inheritance. The event table definition code is automatically inserted by the dialog editor. To begin working with event tables, see [Creating Dialog Boxes](#). To attach an event table to a form (dialog box outer window) or form control, define an event table with the same name (**p_name** property) as the form. Dot the form name with the control name if you want to specify inheritance for an event table that is attached to a control.

Example:

```
defeventtab form_name[.control_name] [_inherit [etab_name]];
```

Using the **_inherit** primitive, you can link one event table to another. This makes it possible to perform Clipboard Inheritance® (see [Clipboard Inheritance®](#)). If no name follows the **_inherit** keyword, the inheritance is unlinked. To add event handlers using the **def** primitive or by defining an event handler function, use the following code:

```
[ReturnType] ctl_name.event [- event] [, event [- event]] ... ( [ArgDecl1,
ArgDecl2,...])
{
    statements
}
```

If *ctl_name* is the same name as the last event table form name (name before dot), the event handler is attached to an event table named *form_name*. Otherwise, the event handler is attached to an event table named *form_name.ctl_name*.

The word *event* in the previous code can be any valid event name. Some event names do not need to be enclosed in quotes. It is a best practice to always enclose the event names in quotes.

The syntax for *ArgDecls* is the same as is the syntax for declaring a variable except that the **static** keyword may not be used. An ampersand (&) before the *id* declares a call by references parameter. Call by reference array and hash table parameters require parentheses around the ampersand and *id*.

The following is an example of a form with a text box and OK button:

```
#include "slick.sh"
// Define an event table for the dialog box window.
defeventtab form1;

// Since this is the first event handler defined for this control
// and the name of this control does not match the last defined event, the
// table, the Slick-C translator automatically defines the event table
// form1.ctlcommand1 and defines the lbutton_up event handler within
// this new event table
void ctlcommand1.lbutton_up()
{
    // Set the p_text property of the text box control
    ctltext1.p_text="Hello World";
}
```

When the above code is loaded with the **load** command (**Macro > Load Module**), the editor attaches the **form1.ctlcommand1** event table to a control named **ctlcommand1** on **form1**. A **form1** event table is not created because an event handler for this event table was defined. When you save the configuration, event tables that are not used are deleted.

Module Initializations

The Slick-C® language provides two module initialization functions called **definit** and **defload**. If the two are present, the procedures **definit** and **defload** are called when a module is loaded. The **definit** module is called before **defload**. When the module is saved by the **write_state** command, the **definit** procedure is invoked each time the editor is invoked. This gives your module an opportunity to perform initializations such as creating a temporary file, or allocating a selection, or bookmark. The following example shows the syntax used for defining the special functions **definit** and **defload**:

```
definit()
{
    statements
}
defload()
{
    statements
}
```

The return value of these functions is always **void**. You cannot specify an argument to the return statement. To enhance the performance of SlickEdit®, use the **defload** primitive instead of the **definit** primitive. The **definit** primitive forces a module to be loaded when the editor is invoked. When **definit** is called, the expression **arg(1)** indicates whether the module was loaded with the **load** command or when the editor initialized. When a module is loaded, **arg(1)** returns **L**. Otherwise **arg(1)** returns a null string ("").

Example:

```
int gmarkid= -1;
definit()
{
    // If this is an editor invocation
    if (arg(1)!="L") {
        gmarkid=-1;    // Indicate no mark is allocated.
    }
}
```

There are two subtle points to this example when assuming that the **gmarkid** variable is used to contain an allocated mark id (also called selection handle). First, the variable **gmarkid** is scoped as global and not static. This is because the mark needs to remain allocated when this module is reloaded. When the module is reloaded, an unload of the module occurs first and the **_free_selection** built-in is not called to free a mark already allocated (there is no **defunload** primitive). Modules with static variables (module scope) lose their value when reloaded. Second, the value of **arg(1)** is used to make sure that the variable **gmarkid** is initialized only when the editor is invoked and not when the module is loaded. Use this as a template for creating a temporary buffer in the hidden window.

Example:

```
#include "slick.sh"
definit()
{
    get_view_id(view_id);
    activate_view(HIDDEN_VIEW_ID);
    status=find_view(".bookmark");
    if ( status ) {
        /* Create a buffer and view in hidden window. */
        status=load_files("+c +t");
        if (status) {
            // The nls function may be used for national language support
            // in the future.
            _message_box(nls('Could not create bookmark buffer'))
            return;
        }
        p_buf_name=".bookmark";_delete_line();
        p_buf_flags= THROW_AWAY_CHANGES|HIDE_BUFFER|KEEP_ON_QUIT;
    }
    // Note: ELSE case cannot empty bookmark buffer unless mark ids
    // are freed. Might as well leave them.
    get_view_id(bookmark_view_id);
    activate_view(view_id);
}
```


Compiling and Loading Macros

The commands **st** and **load** are used to compile Slick-C® modules from within the editor. The **st** command translates the module specified into binary code. When a module is not specified, the current buffer is translated. The **load** command (**F12** or **Macro > Load Module**) translates the module specified if necessary, and loads the resulting byte code. When a module is not specified, the current buffer is saved, translated, and loaded. If a module is loaded that has already been loaded, it is replaced. Both the commands invoke the external program `vstw.exe` (UNIX: `vstw`) to translate the source module into byte code. DO NOT use the **load** command on batch programs. After doing so, you are no longer able to execute the batch program until you use the **unload** command (**Macro > Unload Module**).

A module that is loaded with the **load** command can be unloaded using the **unload** command (**Macro > Unload Module**). However, the symbol table or the names table still contains the names of globally scoped variables, procedures, and commands until you save the configuration. The configuration is automatically saved when you exit the editor. You can invoke the **save_config** command from the command line to save the configuration at any time.

Debugging Macros

The Slick-C® translator `vstw.exe` (UNIX: `vstw`) enables debug messages to be inserted into the code and compiled. Use the **messageNwait** function to display a message and wait until a key is pressed. The **_message_box** function can be used to display a dialog box with a message and wait until you press **Enter** to proceed. Two helpful features are described in the following sections that will help you debug and work on Slick-C macros: [Finding Procedures](#) and [Finding Run-Time Errors](#). Useful defs tab `.e` extension aliases are listed in the table below.

Alias Name	Value
m	<code>messageNwait(%\n: %\c);</code>
mb	<code>_message_box(%\n: %\c);</code>

Finding Procedures

The **find_proc** command (**Macro > Go to Slick-C Definition**) finds Slick-C® source code or help for a Slick-C symbol name that you specify. Use this function if you are browsing a macro and you want to find out more about a function. You can find the procedure at the cursor by pressing **Ctrl+Dot**. The syntax of the **find_proc** command is:

```
find_proc proc_name
```

The table below shows some examples of using **find_proc** on the command line.

Command	Description
<code>find_proc find_proc</code>	Finds the source code for find_proc .
<code>find_proc cursor_up</code>	Finds the source code for cursor_up .
<code>find_proc substr</code>	Displays help on substr built-in.

Finding Run-Time Errors

When a Slick-C® error occurs, a dialog box with a “Slick-C Error” title is displayed. Usually the Slick-C Stack tool window is displayed listing the call stack at the time of the error. Double-click in this tool window to view source for a call stack entry. The **find_error** command (**Macro > Find Slick-C Error**) finds the last Slick-C interpreter run-time error. The module with the error is loaded and the cursor is placed on the line causing the error.

Performance Profiling

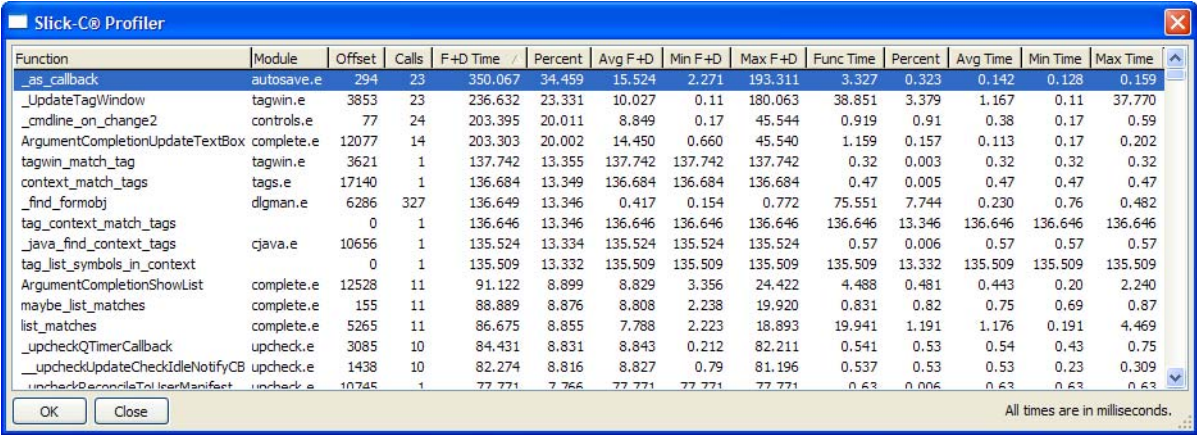
The Slick-C® interpreter supports performance profiling. This is useful to identify bottlenecks or other inefficiencies in Slick-C code. The profiler does not affect performance when it is inactive, and there is only a minimal effect on performance when it is collecting data.

To use this feature, invoke the **profile** command on the SlickEdit® command line with the following options:

- **profile on** - Starts profiling data collection (also resets counters).
- **profile off** - Stops profiling data collection.

- **profile view** - Displays profiling data (also stops collection).
- **profile command args** - Executes the specified Slick-C command with the specified arguments, then displays the profiling data. For example, to profile a CVS update, type **profile cvs-gui-mfupdate**.
- **profile save** - Saves the profiling data for loading/viewing at a later time.
- **profile load** - Loads previously saved profiling data for viewing.

Prior to displaying the profiling data, the applicable Slick-C source files are scanned in order to resolve the names of static functions. Then the Slick-C Profiler dialog is displayed showing the data in multi-column, non-modal tree format. Each line represents one function, which is either a Slick-C function or an exported DLL function, depending on what was called when the profiling data was collected. All times are displayed in milliseconds.



The screenshot shows the 'Slick-C Profiler' dialog box. It contains a table with 15 columns: Function, Module, Offset, Calls, F+D Time, Percent, Avg F+D, Min F+D, Max F+D, Func Time, Percent, Avg Time, Min Time, and Max Time. The table lists various functions such as _as_callback, _UpdateTagWindow, _cmdline_on_change2, etc. At the bottom, there are 'OK' and 'Close' buttons, and a note that 'All times are in milliseconds.'

Function	Module	Offset	Calls	F+D Time	Percent	Avg F+D	Min F+D	Max F+D	Func Time	Percent	Avg Time	Min Time	Max Time
_as_callback	autosave.e	294	23	350.067	34.459	15.524	2.271	193.311	3.327	0.323	0.142	0.128	0.159
_UpdateTagWindow	tagwin.e	3853	23	236.632	23.331	10.027	0.11	180.063	38.851	3.379	1.167	0.11	37.770
_cmdline_on_change2	controls.e	77	24	203.395	20.011	8.849	0.17	45.544	0.919	0.91	0.38	0.17	0.59
ArgumentCompletionUpdateTextBox	complete.e	12077	14	203.303	20.002	14.450	0.660	45.540	1.159	0.157	0.113	0.17	0.202
tagwin_match_tag	tagwin.e	3621	1	137.742	13.355	137.742	137.742	137.742	0.32	0.003	0.32	0.32	0.32
context_match_tags	tags.e	17140	1	136.684	13.349	136.684	136.684	136.684	0.47	0.005	0.47	0.47	0.47
_find_formobj	dlgman.e	6286	327	136.649	13.346	0.417	0.154	0.772	75.551	7.744	0.230	0.76	0.482
tag_context_match_tags		0	1	136.646	13.346	136.646	136.646	136.646	136.646	13.346	136.646	136.646	136.646
_java_find_context_tags	cjava.e	10656	1	135.524	13.334	135.524	135.524	135.524	0.57	0.006	0.57	0.57	0.57
tag_list_symbols_in_context		0	1	135.509	13.332	135.509	135.509	135.509	135.509	13.332	135.509	135.509	135.509
ArgumentCompletionShowList	complete.e	12528	11	91.122	8.899	8.829	3.356	24.422	4.488	0.481	0.443	0.20	2.240
maybe_list_matches	complete.e	155	11	88.889	8.876	8.808	2.238	19.920	0.831	0.82	0.75	0.69	0.87
list_matches	complete.e	5265	11	86.675	8.855	7.788	2.223	18.893	19.941	1.191	1.176	0.191	4.469
_upcheckQTimerCallback	upcheck.e	3085	10	84.431	8.831	8.843	0.212	82.211	0.541	0.53	0.54	0.43	0.75
_upcheckUpdateCheckIdleNotifyCB	upcheck.e	1438	10	82.274	8.816	8.827	0.79	81.196	0.537	0.53	0.53	0.23	0.309
_upcheckReconcileToolBarManifest	upcheck.e	10745	1	77.771	7.766	77.771	77.771	77.771	0.63	0.006	0.63	0.63	0.63

The profiling data can be sorted by clicking any sortable column. Double-click on any function to open the associated file in SlickEdit, with the cursor at the function location.

The Slick-C Profiler displays the following columns:

- **Function** - Name of the function called.
- **Module** - Name of the module from which the function comes.
- **Offset** - The P-code offset of the function within the module.
- **Calls** - Number of calls to the function.
- **F+D Time** - Total time spent in the function and its descendants.
- **Percent** - Percentage of the total time spent in the function and its descendants.
- **Avg F+D** - Average time spent in the function and its descendants.
- **Min F+D** - Minimum time spent in the function and its descendants.
- **Max F+D** - Maximum time spent in the function and its descendants.
- **Func Time** - Total time spent in the function only.
- **Percent** - Percentage of the total time spent in the function.
- **Avg Time** - Average time spent in the function.
- **Min Time** - Minimum time spent in the function.
- **Max Time** - Maximum time spent in the function.

Error Handling and the rc Variable

The **rc** variable is a predefined global variable that is accessible from all loaded modules. The following functions require that you use the **rc** variable for error handling: **buf_match** and **get_env**.

By convention, functions that use integer error codes return negative error codes that correspond to the error codes in `rc.sh`. For these functions, 0 means success and positive codes means the error code is not in `rc.sh`.

Some functions display an error message on the message line. Use the **clear_message** function to clear the message.

Example:

```
// Cause a message
_deselect();_copy_to_cursor();
// Clear the message
clear_message();
```


Dialog Editor

The dialog editor is used to create dialog boxes. There are some similarities and differences between Slick-C® and Microsoft Visual Basic. The dialog editor is where you build the additional text boxes, combo boxes, radio buttons, image controls, menu items, and forms for a dialog box.

Microsoft Visual Basic and Slick-C®

Creating event-driven dialog boxes in Slick-C is similar to Microsoft Visual Basic except that the language has C++-style syntax. The following list contains some of the differences between Slick-C and Microsoft Visual Basic:

- When an event is sent to a control or dialog box, the object receiving the event **MUST** be the active object (not necessarily the same as the system focus). This is a major difference between Slick-C and Microsoft Visual Basic. If a button control receives an event and executes a statement such as this: **p_caption=New button caption**, the caption on the button is changed and **NOT** the caption for the dialog box.
- Built-in properties all start with the prefix **p_** to avoid these keywords from conflicting with their own identifiers.
- A more general method of object instance referencing is used.
- Almost all properties that can be accessed at design time can also be accessed at run time. For example, the **p_name** property for a control or dialog box may be set after the dialog box is displayed.
- Event tables are used to group event handlers for controls. Event tables in Slick-C are used in a similar fashion to classes in C++.
- Slick-C has sophisticated and powerful Dialog Box Inheritance Order. For more information, see [Dialog Box Inheritance Order](#).
- Parent, child, next, and previous (**p_parent**, **p_child**, **p_next**, **p_prev**) creation order relationships are all maintained when dialog boxes are created.
- Event tables can be linked together. One event table can inherit the event handlers of another event table. The event table links can be changed at run time.
- The dialog editor allows event tables to be transferred through the clipboard. Controls from the same or different dialog boxes may reference the same event tables. There is no need for control arrays. For more information, see [Clipboard Inheritance®](#).
- Functions can be used as methods that operate on an instance of an object.

Creating Dialog Boxes

The following information is used to create dialog boxes.

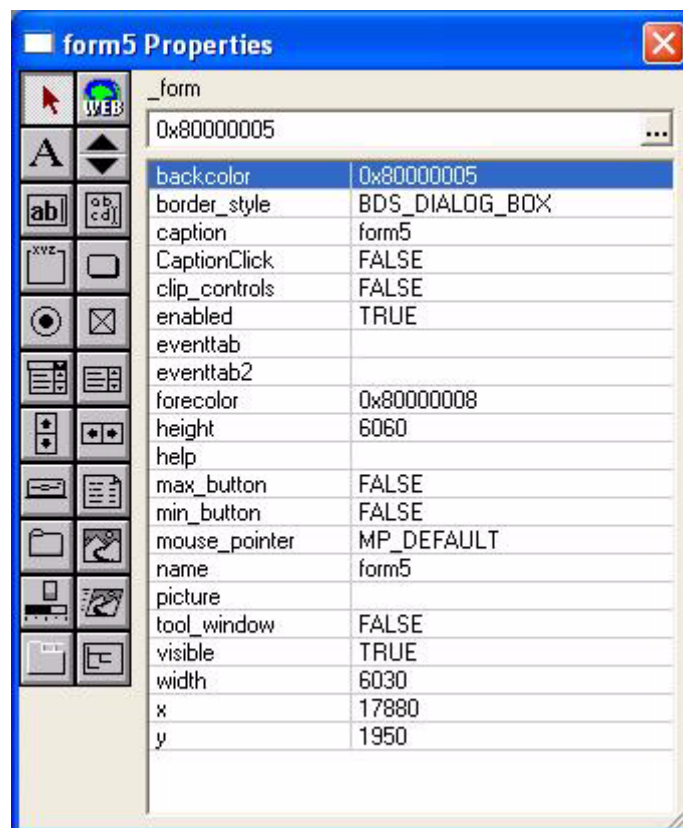
Dialog Editor Summary

Press **Ctrl+Shift+Space** to edit a dialog box that is being run. If you press Ctrl+Shift+Space while the Properties dialog box is active, you edit the Properties dialog box. Double-click the system menu to close the edited Properties form. Some UNIX window managers do not close windows when you double-click on the system menu.

Adding Controls

The bitmaps on the left of the Properties dialog box are used to create controls. Hover over a bitmap to display the function of a bitmap. There are two methods for creating a control. The first method is to double-click the left mouse button on the bitmap of the control that you want to create. This places a new control in the middle of the selected form.

The **Picture Box** and **Frame** controls enable you to place controls inside of them. To do so, select **Window > Properties** or, use the **show_properties** command.



To use the other method for creating a control, complete the following steps:

1. Single-click on the **Text Box** bitmap.

2. Move your mouse so that it appears on top of the form that you are editing. If you cannot see the form that you are editing, display it by selecting **Window > Selected Form**.
3. To create the text box control, click the left mouse button, and, while holding it down, move the mouse pointer to the right to create a dotted rectangle. When you release the mouse, the text box control is displayed within the rectangle.

Deleting Controls

Select the control(s), then press **Backspace** or **Delete**.

Setting Properties

To set properties, complete the following steps:

1. Select the control. Left-click the mouse button on the property in **Properties** list box.
2. Type the new value in **Properties** combo box. Press **Enter** when the **Properties** list box is active to set the property.
3. Select the control. Double-click the left mouse button on the property in the **Properties** list box to go to the next value of the property. For **color** and **picture** properties, a dialog box is displayed.

Aligning Controls

Select the control with which you want to align the other controls. Select the other controls with **Shift+LButton**. Double-click the left mouse button on one of the properties **x** or **y** to align the controls in the **x** or **y** direction. Press **Enter** on the value in the **Properties** combo box.

Sizing Controls

To size controls, use one of the following methods:

- To size a single control, select the control and click and drag one of the selection handles with the left mouse button.
- To size multiple controls, select the controls and set the **width** or **height** property.
- To size multiple controls, select the controls and press **Shift+Left**, **Shift+Right**, **Shift+Up**, or **Shift+Down** to move the lower right corner of the selected controls by one pixel.

Moving Controls

To move controls, use one of the following methods:

- Select the control, then click and drag with the left mouse button.
- Select the control, then set the **x** or **y** property.
- Select the control(s), then press the **Left**, **Right**, **Up**, or **Down** arrow key to move the selected controls by one pixel.

Miscellaneous Assignments When the Form is Active

The table below shows a list of miscellaneous button and key assignments that can be used when the form is active.

Assignment	Action
RButton	Displays menu with various dialog editor commands.
Ctrl+Shift+Space	Loads form and Slick-C® code. Runs dialog box. If you accidentally press Ctrl+Shift+Space when in the Properties dialog box, you will be editing the Properties dialog box. Double-click on the system menu to close the edited Properties form. Some UNIX window managers do not close windows when you double-click on the system menu.
Ctrl+S	Loads form and saves into state file. Under UNIX, this may just list source for the form that can be executed.
Ctrl+L	Loads form.
Ctrl+C	Copies selected controls.
Ctrl+V	Pastes controls from the clipboard.
Ctrl+X	Cuts selected controls.
Ctrl+A	Selects all controls with same parent as the already selected control(s).
Tab	Deselects all controls and selects next control in tab order (p_tab_index).
Shift+Tab	Deselects all controls and selects previous control in tab order.
LButton	Double-click (on control) displays Select an Event Function dialog box for adding or modifying event handlers.

Miscellaneous Menu Items

The table below shows the miscellaneous menu items.

Menu Item	Description
System Box of form, Show Properties	Display Properties dialog box.
Window > Properties	Display Properties dialog box.
Window > Selected Form	Display selected form (form being edited).
Macro > New Form	Creates a new dialog box with a default name.
Macro > Open Form	Open existing dialog box or create new dialog box.
Macro > Grid	Sets the distances between the dots on edited form.

Creating a Form

A form is the outer window of a dialog box. The objects within the dialog box are called controls. The form also refers to the entire dialog box. A new form can be created by using one of the following methods:

- Use the **New Form** menu item (**Macro > New Form**).

OR

- Use the **Open Form** menu item (**Macro > Open Form**) and specify the name of a new form.

Saving a Form

Click on the form being edited and press **Ctrl+S**.

Adding Event Handlers

Set the form name and the control names (**name** property in **Properties** list box) before adding code to the dialog box because these names are referenced in the code. Prefix your control names using the letters **ctl** so that they are easily recognizable. To add an event handler, complete the following steps:

1. Double-click on the control in the dialog box for which you want to add code (not the bitmap in the Properties dialog box). The Select An Event dialog box is displayed.
2. Select an event and click **OK**. If this is the first event handler for this dialog box, you will be prompted with an Open dialog box for a new file to contain the source code for this dialog box.
3. Type a unique file name. Usually this file name is derived from the name of the dialog box you are creating, such as **form1.e**.

After performing the above steps, the dialog editor inserts an event function definition into your source file and places your cursor in the function.

Inherited Code Found Dialog Box

This dialog box is displayed when there is no code for the event you have chosen and the control is using an inherited event table. You will see this dialog box if you copy a control with existing code, paste elsewhere and then double-click on the new instance of the control.

The following options are available on the dialog:

- **Inherit code** - When this option is selected, a statement which links a new event event table to an inherited event table (event table not belonging to the control and possibly copied through the clipboard). This affects user level 1 inheritance code (**p_eventtab**) only.
- **Go to inherited code** - When this option is selected, no code is inserted. The cursor is placed on the existing inherited event handling code.
- **Don't inherit code** - Select this option when you do not want to inherit the existing user level 1 inheritance code (**p_eventtab**). Sometimes when you copy a control with existing code to the clipboard, you will not want to inherit the existing event handlers.

Loading and Running the Form

Invoke the **run_selected** command (**Macro > Load and Run Form**) to run the current dialog box that you are editing. This loads the code, loads the dialog box, and runs the dialog box. To close the dialog box, double-click on the system menu (some UNIX window managers do not close windows when you double-click on the system menu) or press **Ctrl+Shift+Space** (in the running version of your dialog box and not

the edited copy). Press **Ctrl+Shift+Space** when any dialog box is running to edit it (this includes the Properties dialog box).

Display the dialog box from the command line by typing **show <FormName>**. To display the dialog box modally enter **show -modal <FormName>** on the command line. For more information about this command, see [Displaying Dialog Boxes](#). Dialog box templates and compiled macros are stored in the state file `vslick.sta` (UNIX: `vslick.stu`).

The example code below shows how to write a command that displays a dialog box. This is used when binding a command to a key that displays a dialog box.

```
#include "slick.sh"
_command void run_form1()
{
    // The -modal option displays other windows while the dialog box
    // is displayed.
    show("-modal form1");
}
```

Adding a Cancel Button

To add a **Cancel** button, complete the following steps:

1. Double-click **Insert Button Control**.
2. Set the **caption** property to **Cancel**, set the **cancel** property to **TRUE** and set the **name** property to **""**.
3. Set the **cancel** property to **TRUE** by double-clicking the left mouse button on the **cancel** property in the **Properties** list box. Set the **name** property of a control (never the form) to **""** if you are not going to reference the control by name.
4. Clicking **Cancel** (the command button with the **p_cancel** property set to **TRUE**) when your dialog box is running will close the dialog box even though you have not written any code. If you do add code to your **Cancel** button, you must close the dialog box by typing the following in the command line: **p_active_form._delete_window()**;

Adding an OK Button and Closing a Dialog Box

To add an **OK** button and close a dialog box, complete the following steps:

1. Create a command button control by double-clicking **Insert Button Control** in the Properties dialog box.
2. Set the **caption** property to **OK**, set the **default** property to **TRUE**, and set the **name** property to **ctlok**.
3. Double-click on the command button control in the dialog box for which you want to add code (not the bitmap in the Properties dialog box). The Select An Event dialog box is displayed.
4. Choose the **lbutton_up** event and click **OK**.
5. If this is the first event handler for this dialog box, the Open dialog box for a new file to contain the source code for this dialog box is displayed. Type a unique file name. Usually this file name is derived from the name of the dialog box that you are creating, such as **form1.e**.

CREATING DIALOG BOXES

After completing the previous steps, the dialog editor inserts an event function definition into your source file and places your cursor in the function. Add the code as shown in the following example:

```
#include "slick.sh"
defeventtab form1
// Code for OK button.
ctlok.lbutton_up()
{
    // Close the dialog box and return a value. The _delete_window
    // function allows modal dialog boxes to return a value. For
    // more information, see
    // "Displaying Dialog Boxes" below. Each object
    // in the dialog box will receive an on_destroy event.
    // NOTE: If "" is a valid return value. Return 1 here and store
    // your results in the global _param1 variable.
    p_active_form._delete_window("return value");
    // Statements after closing a dialog box are executed.
}
```

Before closing a dialog box, review the following information: First, if a modal dialog box returns a value, the value "" (zero length string) **MUST** be returned to indicate that the dialog box has been canceled. This convention is used so that when running a dialog box, you can press **Ctrl+Shift+Space** to safely cancel and edit the dialog box. Use the global container variables **_param1.._param10** to return multiple strings. Alternately, you can make an array or structure and place it in **_param1**. If you do place your string results in the global variables **_param1.._param10**, make sure your dialog box returns 1 (or any value other than "") to indicate that the dialog box was not canceled.

Displaying Dialog Boxes

The **show** command is called in function-style syntax from within a macro. It can also be invoked from the command line or a menu item.

The command line call syntax is:

```
show cmdline
```

The function call syntax is:

```
show(cmdline [,arg1 [,arg2 ... [argN]])
```

cmdline is a string in the format:

```
[option] form_name
```

option can be one of the options in the table below.

Option	Description
-mdi	Keep the form on top of the MDI window.
-app	Keep the form on top of the SlickEdit® application window. This allows the MDI window to be displayed on top of the form.
-xy	Restore the previous x,y position of the dialog box. If the old position cannot be found, the dialog box is centered. When the dialog box is closed, the x,y position is automatically saved (the dialog manager calls _save_form_xy).

Option	Description
-hidden	Do not make the form visible. Run the form modally. All other forms are disabled. Control returns to the caller when the form window is deleted with _delete_window .
-nocenter	Do not center the form.
-new	Normally, when a form is already displayed, the existing form is given focus. This option allows for multiple instances of a form to be displayed.
-reinit	(UNIX only) This option causes the _delete_window function to make the form invisible instead of deleting the form. The destroy events are dispatched even though no windows are actually destroyed. Next time show is called for the same dialog box, the invisible dialog box is made visible, some properties are reinitialized, and the create events are sent. Be careful when using this option. Not all dialog boxes can use this option without minor modifications. The form_parent() function does not work because the next time the form is used, the parent is not changed to the new parent specified.
-hideondel	(UNIX only) This option is the same as the -reinit option except no properties are reinitialized when the invisible dialog box is shown again.

form_name specifies a form or menu resource. If it is an integer, it must be a valid index into the names table of a form or menu. Otherwise, it should be the name of an existing form or menu that can be found in the names table.

on_create and on_load Events

When a dialog box and all its objects are created, each object receives an **on_create** event. The **on_create** event receives the *arg1, arg2, ..., argN* arguments given to the **show** function. After the **on_create** events are sent, the form receives an **on_load** event. You CANNOT set the final focus in an **on_create** event. Use the **_set_focus** function during the **on_load** event to set the initial focus to a control other than the control with lowest tab index (**p_tab_index**) that is enabled and visible.

Return Value of show

If the **-modal** option is given, the return value given to **_delete_window** is returned. "" is returned if the dialog box is edited or destroyed during an **on_create** event. Use the global variables **_param1.._param10** to return more than one string value. Alternately, you can make an array or structure and place it in **_param1**.

If the **-modal** option is not given, the form window *id* is returned if successful. Otherwise, a negative error code is returned.

CREATING DIALOG BOXES

Example:

```
// This example requires that you create a form called form1 with a
// command button and load this file.
#include "slick.sh"
_command mytest()
{
    result=show("-modal form1");
    if (result=="") {
        return(COMMAND_CANCELLED_RC);
    }
    message("_param1=_param1" _param2=_param2);
}

defeventtab form1
ctlcommand1.on_create()
{
    // Global variable _param1.._param10 are defined in "slick.sh" to
    // allow for multiple strings
    // to be returned in separate variables.  Alternatively, if the
    // return strings do not contain spaces, you could concatenate
    // them together with a space and use the parse built-in to easily
    // separate them.
    _param1="string1";
    _param2="string2";
    // Close the dialog box and indicate that the dialog box was not
    // canceled. Each object in the dialog box will receive an
    // on_destroy event.
    p_active_form._delete_window(1);
}
```

Example:

```
// This example requires that you create a form called form1 with a
// command button and load this file.
#include "slick.sh"
_command void mytest()
{
    show("-modal form1","param1 to on_create", "param2 to on_create");
}

defeventtab form1
ctlcommand1.on_create(_str arg1="", _str arg2="")
{
    //arg1=arg(1);  Could get the arguments with the arg built-in
    //arg2=arg(2);
    messageNwait("arg1=arg1" arg2="arg2");
}
```


Example:

```
#include "slick.sh"
defmain()
{
    index=find_index("form1",oi2type(OI_FORM));
    if (!index) {
        messageNwait("form1 not found");
        return(1);
    }
    // Can specify name table index instead of name
    // When show is called without the "-modal" option, the
    // positive window id (instance handle) of the form created
    // is returned.
    form_wid=show("-hidden -nocenter "index);
    if (form_wid<0) {
        return(1);
    }
    // Place the form at the top left corner of the display.
    form_wid.p_x=form_wid.p_y=0;
    // Make the form visible
    form_wid.p_visible=1;
    return(0);
}
```

Modal and Modeless Dialog Boxes

If you do not want the MDI window or any other form to get focus when your dialog box is displayed, specify the **-modal** option to the **show** command. When the **-modal** option is given, other forms, including the MDI window, are disabled (**p_enabled=0**) until the form is closed. In addition, the **_delete_window** function can be used to return a value (see the previous example).

Modeless example:

```
#include "slick.sh"
defmain()
{
    // When show is called without the "-modal" option, the positive
    // window id (instance handle) of
    // the form created is returned.
    form_wid=show("-hidden -nocenter form1");
    if (form_wid<0) {
        return(1);
    }
    // Place the form at the top left corner of the display.
    form_wid.p_x=form_wid.p_y=0;
    // Make the form visible
    form_wid.p_visible=1;
    return(0);
}
```

If you need to display a status dialog box during processing, you might require a modeless dialog box so control is returned to you. However, it is a best practice to disable all other dialog boxes including the MDI window during processing.

Advanced modeless example:

```
#include "slick.sh"
static typeless gcancel;
_command void test()
{
    // Show the form modeless so there is no modal wait
    form1_wid=show("form1");
    // Disable all forms by the one with p_window_id==form_wid. A
    // space delimited string of disabled form window ids is returned.
    disabled_wid_list=_enable_non_modal_forms(0,form_wid);
    gcancel=0;
    for (;;) {
        // Read mouse, key, and all other events until non are left
        // or until the variable gcancel becomes true.
        process_events(gcancel);
        if (gcancel) break;
        // Do your processing here
    }
    // Enable the forms that were disabled.
    enable_non_modal_forms(1,0,disabled_wid_list);
    form1_wid._delete_window();
}
defeventtab form1;
ctlcancel.lbutton_up()
{
    gcancel=1;
}
```

Dialog Box Parent Window

The parent window of a dialog box form has two uses. First, the dialog box remains on top of the parent window. Use the **show** command and specify the **-app** option if you want to allow a modeless dialog box be displayed behind the MDI window. The **-mdi** option of the **show** command can be used to make sure a dialog box stays on top of the MDI window.

Command line examples:

```
show -app _calc_form
show -mdi -new _calc_form
```

Second, the parent window is used by some dialog boxes (such as the Print and Spelling dialog boxes) to determine on which buffer to operate. This permits the dialog boxes to support the editor control. To do this, they call the **_form_parent** function during an **on_create** event to get the window *id* of the window which contains the buffer to be operated on. These dialog boxes only support certain parent windows. For example, the Print dialog box will not run correctly if the **-app** option of the **show** command is used.

Remembering a Dialog Box's Previous Position

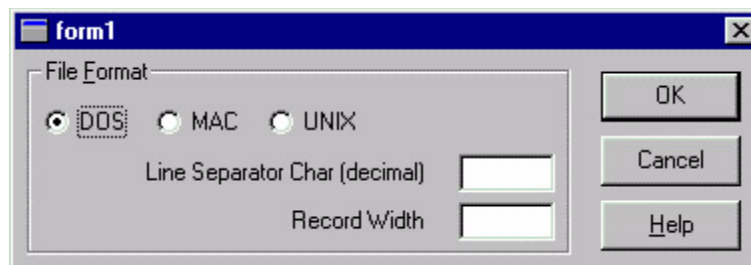
The **show** command centers the dialog box to the current form or MDI window. Usually this is fine, but sometimes it is helpful for a dialog box to reappear in the same position that it was in when the user closed the dialog box. To do this, specify the **-xy** option to the **show** command. This adds the **IS_SAVE_XY** flag to the **p_init_style** property. When the dialog box is closed, the **x** and **y** position of the dialog box is stored and later saved in the auto restore file (**vrestore.slk** by default) when you exit the editor. The form is centered if the old **x,y** position information cannot be found.

Clipboard Inheritance®

Clipboard Inheritance enables the transferring of objects from one place to another using the clipboard to create new instances that inherit the code of the original objects. Code for the new instances can be added that affects only the new instances, and code of the original instances can be modified, affecting both instances.

For example, you may want to create a group of controls that are needed by the SlickEdit® File Open dialog to allow the user to specify the various supported file formats. SlickEdit supports the following file formats:

- **DOS** - Each line is separated with a carriage return, followed by linefeed.
- **Macintosh®** - Each line is separated with a carriage return only.
- **UNIX** - Each line is separated with a linefeed only.
- **Record width** - A user-specified number of bytes placed in each line.
- A single user specified line separator character. The following partial dialog box can be used to handle the file formats of SlickEdit.



Example:

```
// The names of the controls do not need to be declared.

// The names of the radio button controls are ctlopendos, ctlopenmac, ctlopenunix,
// ctlopenauto

// The first text box is named ctlopenlinesep and the text box below it is named
// ctlopenwidth.

defeventtab form1;

// Define the lbutton_up event for the DOS radio button. This function will
// get called when any of the radio buttons get turned on. The event
// table automatically created here is called form1.ctlopendos.
ctlopendos.lbutton_up()
{
    // Set the text displayed in both text boxes to nothing so the users
    // knows that the radio button format has been chosen.
    ctlopenlinesep.p_text='';
    ctlopenwidth.p_text='';
}

static zap_radio_buttons()
{
    ctlopendos.p_value=0;
    ctlopenmac.p_value=0;
    ctlopenunix.p_value=0;
    ctlopenauto.p_value=0;
}

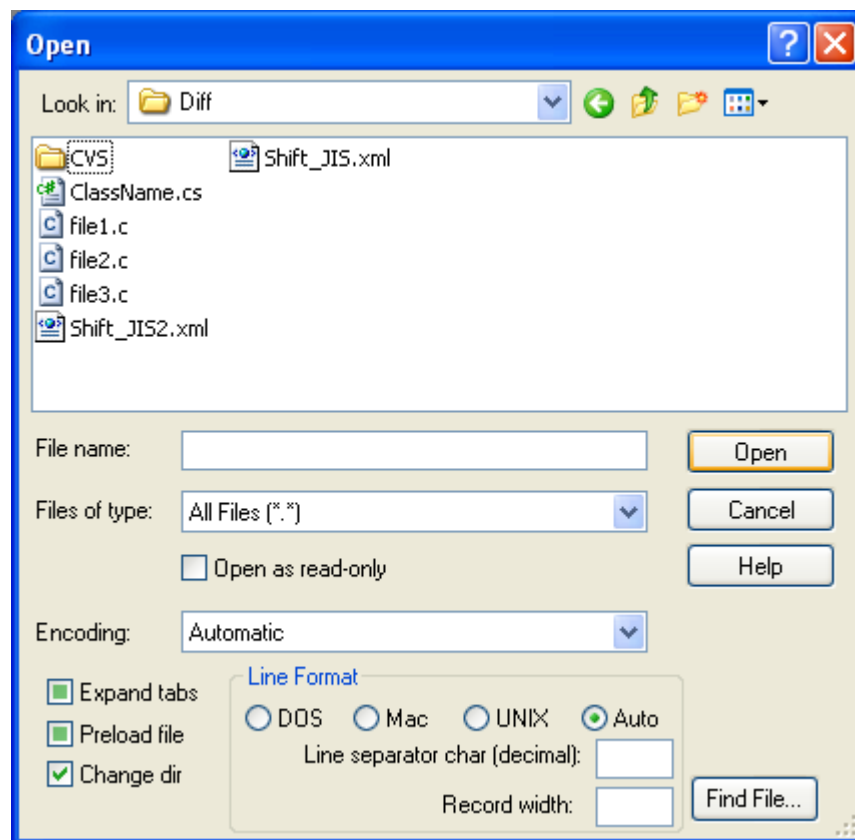
// Define the on_change event for the first text box. For a text box, the
// on_change event gets called when the users modifies the text in the text box.
// The event table automatically created here is form1.ctlopenlinesep.
ctlopenlinesep.on_change()
{
    if (p_text!='') {
        ctlopenwidth.p_text=''; // Clear out the other text boxes text.
        zap_radio_buttons();    // Turn off all the radio buttons
    }
}

// Define the on_change event for the second text box. The event table
// automatically created here is form1.ctlopenwidth.
ctlopenwidth.on_change()
{
    if (p_text!='') {
        ctlopenlinesep.p_text=''; // Clear out the other text boxes text.
        zap_radio_buttons();    // Turn off all the radio buttons
    }
}
```

Only the first radio button **ctlopandos** has an event handler defined. The other radio button use the **form1.ctlopandos** event table. This can be accomplished in the dialog editor using Clipboard Inheritance or, if the radio buttons are already created, you can set the **p_eventtab** property of the other radio buttons to **form1.ctlopandos**. To use Clipboard Inheritance, write the **lbutton_up** event code for the DOS radio button, copy the DOS radio button to the clipboard, paste it back onto the dialog box within the frame, and set the **p_caption** property for the new radio button to **MAC**. Either of these methods can be used to fill an event table. When the **ctlopandos.lbutton_up()** function gets called, it gets and sets the properties of controls that exist on this dialog box.

Open Dialog Box

Clipboard Inheritance® was created by copying controls to the clipboard and pasting them.



The Open File dialog box has the form name **_edit_form**. This dialog box is created by copying the **_open_form** dialog box (code links and all) to the clipboard, pasting it, and then adding the **Find File** button and the advanced controls. The **_open_file** form can be thought of as the base File Open dialog box class. It is used for all other File Open and Save As operations except for opening files for editing which requires additional controls. The inherited code from the base class File Open dialog required no changes except for the **OK** button. For this, the **OK** button code was replaced with new code. The Find File displays a dialog box which has all of the same advanced controls. The advanced controls were taken from the Open File dialog box (**_edit_form**) and all its related controls, and copied onto the Find File dialog box. The only additional code required was for the **OK** button, which was needed to return the results of the advanced options to the caller.

The following statement highlights the syntax for linking one event table to another:

```
defeventtab dlgbox2.textbox1 _inherit dlgbox1.textbox1.
```

Dialog Box Inheritance Order

Each control in Slick-C® has two properties, called **p_eventtab** and **p_eventtab2**. The **p_eventtab** property defines the user level 1 inheritance. User level 1 inheritance permits the modification of the event handlers for one specific instance of a control without affecting any other (except when Clipboard Inheritance® is used). The dialog editor automatically inserts the necessary function declaration code so that you need to only add statements within the function. After you write the event handler and load the new code, the **p_eventtab** property displayed in the **Properties** list box is updated to reflect that you have defined a user level 1 event table. The **p_eventtab2** property defines the user level 2 inheritance. User level 2 inheritance is typically used to affect all controls of a specific type. Normally, the dialog editor sets these properties for you when a control is created. For example, when you create a combo box control with the dialog editor, the **p_eventtab2** property is automatically set to **_ul2_combobox**. The **_ul2_combobox** event table defines the default processing used by every combo box. The user level 1 event handler receives an **on_change** event (sent from the user level 2 code) when the text in the combo box changes.

SlickEdit® uses a pre-defined inheritance order called Dialog Box Inheritance Order. When a control receives an event, the following search begins to determine which event handler should get control:

1. IF and ONLY if the event SlickEdit® is searching is a key event, check the dialog box user level 1 inheritance on the frame of the dialog box.
2. Check current control's user level 1 inheritance.
3. Check current control's user level 2 inheritance.
4. Check automatic inheritance. Only the text box, combo box, and editor window can have any automatic inheritance. This is how your emulation is supported in these controls.
5. Check the dialog box frame user level 1 inheritance.
6. Check the dialog box frame user level 2 inheritance.
7. Check dialog manager inheritance.

As soon as an event handler is found, the search stops and the event handler is executed. Each inheritance level can have up to 20 linked event tables. This limit is only to avoid infinite event table link loops. At run time it is possible, but unusual, to change all inheritance links and event tables for any object. The **eventtab_inherit** function can be used to get or set an event table inheritance link.

Objects and Instances

Every object instance can be uniquely identified by a window id (also called instance handle). Slick-C® treats objects and windows the same. However, some objects, such as image control, have a window id but *do not* allocate an operating system resource known as a window.

Active Object

When an object receives an event, that object is the active object. More specifically, the **p_window_id** property is set to the instance handle of that object. You can change the active object by setting the **p_window_id** property to the window id of another object. Accessing a property without specifying a control name or instance handle accesses the property of the active object and not the active form.

NOTE Changing the active object does NOT change the focus. Use the **_set_focus** method to change the focus.

Active Form

Slick-C® has a **p_active_form** property that returns an instance handle to the current form. The Slick-C interpreter actually does not keep track of what form is active. The active form is found by traversing through the parents (**p_parent**) of the active object until the form is reached.

Instance Expressions

The examples below display common instance expressions.

```

ctltext1.p_text="test"; // Assuming ctltext1 has been declared
                        // globally or locally, lookup the
                        // ctltext1 control of the active form to get
                        // the window id, and set the
                        // p_text property.
x=_control ctltext1;   // Put the window id of the "ctltext1" control
                        // of the active form in the variable x.
                        // The variable x does not have to
                        // be declared. There are cases where
                        // the control keyword is not needed. It
                        // is better to always use it so you
                        // don't have to worry.
x.p_text="test";       // Set the p_text property of the
                        // object referenced by the
                        // instance expression x.
(x+1-1).p_text="test"; // Same as previous statement. This
                        // shows that any valid Slick-C
                        // language expression may be
                        // used to get the window id.
x.(x+1-1).x.p_text="test";
                        // Same as the previous statement but wastes
                        // more code spaces. This shows that multiple
                        // periods ('.') may be used in an
                        // instance expressions.
form_wid=p_active_form; // Get the window id of the active form.

form_wid.ctltext1.p_text="test";
                        // Lookup ctltext1 as if the object
                        // referred to by the variable form_wid
                        // were the active object.
p_next.p_next.p_prev.p_prev.p_text="test";
                        // Waste some code space and access the p_text
                        // property of the active object.
p_window_id=_control ctltext1;
                        // Make the ctltext1 control the
                        // active object.
p_text="test";         // Access the p_text property of the
                        // active object.
_cmdline.p_text="test"; // _cmdline is a constant window id defined
                        // in "slick.sh". Set the command line p_text
                        // property to "test". Cool!!

```


Using Functions as Methods

A command or procedure can be called as a method without any additional declaration data. The sample Slick-C® source below is an example of this feature.

```
#include "slick.sh"
defmain()
{
    // Call the tbupcase function as a method to operate on Visual
    // command line. _cmdline is a constant instance
    // handle defined in slick.sh.
    _cmdline.tbupcase();
}

// This function uppercases the text in a text box or combo box
// input field and has been written to operate on the current object.
void tbupcase()
{
    // The p_text property is used to get and set the contents of a
    // text box or combo box input field
    p_text=upcase(p_text);
}
```

The **tbupcase** is not defined to be a method of a particular class. This feature permits macros written in SlickEdit® text mode to be converted into SlickEdit macros and used as methods. Also, most functions are written to operate on the current object, meaning you have access to many methods. Using functions as methods is useful when writing dialog box event handlers. If a function is called and a statement within the function is not valid for the current object, the macro is stopped, and a dialog box is displayed indicating the error. The **find_error** command (**Macro > Find Slick-C Error**) can then be used to locate the source of the error.

Built-in Controls

Label Control

Label control is for displaying text in any font. Labels can be aligned left, right, centered horizontally, centered vertically, or centered horizontally and vertically. If you do not need to align the label, set the **p_auto_size** property to TRUE to ensure that the text fits inside the window. A common use of a label control is to place it to the left of a text box to tell the user about what goes in the text box. Select the label control and use the **Up**, **Down**, **Left**, and **Right** arrow keys to center the label to the text box.

For a complete list of label control properties, methods, and events, from the main menu, select **Help > Macro Functions by Category**.

Spin Control

The most common use of a spin control is to increment or decrement a number displayed in a text box. This can be performed WITHOUT writing any code, by making the **tab_index** property of the text box one less than the **tab_index** property of the spin control. An error is displayed if there is no text box with a tab index one less than the spin control, unless the increment property of the spin control is set to zero. To create spin control, complete the following steps:

1. Create the text box and then create the spin control.
2. Turn off the **auto_size** property of the text box so you can make the height of the text box larger than the font.
3. Use the spin control to increment or decrement the value in a gauge or scroll bar control or increment or decrement a hexadecimal number displayed in a text box. The default increment is 1. Set the **increment** property of the spin control to zero and process the **on_spin_up** and **on_spin_down** events. The **on_change** event is called with a *reason* set to **CHANGE_NEW_FOCUS**, before an **on_spin_up** or **on_spin_down** event, to allow you to return the window id of the control you want to get focus, after spinning is completed. Return an empty string (") if you do not want to change the event.

Example:

```
#include "slick.sh"

// This example requires form name form1 with a text box and spin
// control. The spin control should be named ctlspin1 and the
// increment property should be zero. The tab index of the text
// box MUST be one less than the spin control. This code does not
// reference the name of the text box so that you can use Clipboard
// Inheritance(R) to create multiple working copies of a spin control
// capable of incrementing/decrementing the value in a text box control
// without writing any new code.
defeventtab form1;
ctlspin1.on_change(reason)
{
    if (reason==CHANGE_NEW_FOCUS) {
        return(p_prev);
    }
}
ctlspin1.on_spin_up()
{
    new_dec_value=hex2dec(p_prev.p_text)+1;
    p_prev.p_text=dec2hex(new_dec_value);
}
ctlspin1.on_spin_down()
{
    new_dec_value=hex2dec(p_prev.p_text)-1;
    p_prev.p_text=dec2hex(new_dec_value);
}
```

For a complete list of spin control properties, methods, and events, from the main menu, select **Help > Macro Functions by Category**.

Text Box Control

The text box control enables the user to enter a single line of text. Editor control determines the number of lines that can be entered. Text boxes support completion with the spacebar and question mark keys. Set the completion property of the text box. The FILE_ARG completion type is the most common. It provides completion on file names. New commands can be written that operate in all text boxes, edit windows, and editor controls.

Example:

```
#include "slick.sh"
_command void upcase_line() name_info('','VSARG2_TEXT_BOX|VSARG2_REQUIRES_EDITORCTL)
{
    init_command_op();
    get_line(line);
    replace_line(upcase(line));
    retrieve_command_results();
}
```

Bind the **upcase_line** command in the previous example to **Alt+F12**. This command works in all text boxes, edit windows, and editor controls. The key binding might not work in a text box if you bind the **upcase_line** to one of the CUA keys Alt+A, Alt+Z, Ctrl+X, Ctrl+C, or Ctrl+V. Use the Redefine Common

Keys dialog box (**Tools > Options > Redefine Common Keys**) to allow all key bindings to be inherited into text box controls.

For a complete list of text box control properties, methods, and events, from the main menu, select **Help > Macro Functions by Category**.

Editor Control

Editor control is used to enter multiple lines, view clipboards, to work with the calculator, and for version control comments. Almost all of the key bindings for an MDI edit window work in an editor control even when the emulation is changed. Use macro recording to write a new command that works in an edit window and editor control. Mark the **Allow in non-MDI editor control** check box when you finish recording the macro.

For a complete list of editor control properties, methods, and events, from the main menu, select **Help > Macro Functions by Category**.

Frame Control

Frame control is used to group a set of related controls. Radio buttons are placed inside of a frame control to indicate to the dialog manager that only one of the radio buttons in the group can be turned on at a time. There are two ways to place a control inside of a frame control:

- Click the left mouse button on the bitmap in the Properties dialog box of the control that you want to place inside the frame. Click and drag with the left mouse button inside the frame control to create the control with the size of the rectangle displayed.
- Copy or cut the control you want to place inside the frame to the clipboard. Select the frame control and press **Ctrl+V** to paste the control inside the frame control.

For a complete list of frame control properties, methods, and events, from the main menu, select **Help > Macro Functions by Category**.

Command Button Control

The command button control is most typically used to create an **OK**, **Cancel**, or **Help** button.

For a complete list of command button control properties, methods, and events, from the menu, select **Help > Macro Functions by Category**.

Radio Button Control

Radio buttons must be grouped. When one radio button is enabled, the other radio buttons in the same group are not available. Radio buttons are considered in the same group if they have the same parent. Usually, radio buttons are grouped by placing them inside a picture box or frame control. A picture box can have its **border_style** property set to **BDS_NONE** to display that the picture box control does not exist. Use one of the following methods to place a radio button inside a frame:

- Click the left mouse button on the radio button bitmap in the Properties dialog box. Click and drag with the left mouse button inside the frame or picture box control to create the control with the size of the rectangle displayed.
- Copy or cut the radio buttons that you want to place inside the frame or picture box to the clipboard. Select the frame or picture box and press **Ctrl+V** to paste the controls inside it.

For a complete list of radio button control properties, methods, and events, from the main menu, select **Help > Macro Functions by Category**.

Check Box Control

A check box is used to set up a true or false option. Check boxes can be displayed to the left or right of the caption.

For a complete list of check box control properties, methods, and events, from the main menu, select **Help > Macro Functions by Category**.

Combo Box Control

A combo box is used in place of a text box for combo box retrieval, when only a fixed set of responses is permitted, or when a common set of responses are known and a different response may be typed in. Combo box retrieval is a mechanism in that the combo list box displays the previous responses entered in the text box of the combo box. The combo box has two style properties:

- The **PSCBO_NOEDIT** style is used when only a fixed set of responses are allowed. Combo boxes support completion with the spacebar and question mark keys. Set the **completion** property of the combo box if there is an existing completion type that suits the needs.
- The **FILE_ARG** completion type is the most common. It provides completion on file names.

Example:

```
// This example illustrates combo box retrieval.
// This example requires a form named "form1", an OK button named
// "ctllok", and combo box named "ctlcombol".
defeventtab form1;
ctllok.lbutton_up()
{
    // When the OK button is pressed, you will want to save combo box
    // retrieve information.
    _append_retrieve(_control ctlcombol,ctlcombol.p_text);
}
ctllok.on_create()
{
    // Fill in the combo box list
    ctlcombol._retrieve_list();
}
```

A combo box consists of four controls: the root window, text box, picture box, and list box. The properties and methods of the sub-controls may be accessed individually with the **p_cb**, **p_cb_text_box**, **p_cb_picture**, **p_cb_list_box** instance handle properties. The **p_cb_picture** property is only available when the control is displayed.

Example:

```
defeventtab form1;
ctlcombol.on_create()
{
    // To make the loop a little more efficient, activate the list
    // box of the combo box control
    p_window_id=p_cb_list_box;
    for (i=1;i<=100;++i){
        // Add an item to the active list box.
        _lbadd_item("line="i);
    }
    // Activate the root window of the combo box.
    p_window_id=p_cb;
}
```

Example:

```
#include "slick.sh"
defeventtab form1;
ctlcombol.on_create()
{
    // Show a picture which indicates that clicking on the picture
    // box button displays a dialog box. _pic_cbdots is a global
    // variable defined in "slick.sh" which is a handle to a picture.
    vp_cb_picture.p_picture=_pic_cbdots;
}
ctlcombol.lbutton_down()
{
    // Check if the left mouse button was pressed inside the picture
    // box of the combo box
    if (p_cb_active==p_cb_picture) {
        result=show("-modal form2");
        // process result here
        return("");
    }
    // Skip user level 1 inheritance and execute the default event
    // handler defined by user level 2 inheritance.
    call_event(p_window_id,lbutton_down,2);
}
```

Example:

```
// This example requires a form named "form1", command button named
// "ctllok", a combo box named "ctlcombol", and another command button
// named "ctlcommand1"
#include "slick.sh"
defeventtab form1;
ctllok.lbutton_up()
{
    // Check if text in combo box text is valid. You might think you
    // could use a non-editable style combo box. However, many users
    // prefer typing in names using completion, to using the mouse to
    // select an item out of a list box.
    status=ctlcombol._cbl_search("", "$");
    if (status) {
        _message_box("Combo box contains invalid input");
        return("");
    }
    // have valid input
}
ctlcommand1.lbutton_up()
{
    // Add some items to the combo box list
    ctlcombol.p_cb_list_box._lbadd_item("Hello")
    ctlcombol.p_cb_list_box._lbadd_item("Open");
    ctlcombol.p_cb_list_box._lbadd_item("New");
    // Make the correct item in the combo box list current so combo
    // box retrieval works better. _cbl_search searches for p_text
    // in the combo list box. The "$" specifies that an exact match
    // should be found and not a prefix match.
    int status=ctlcombol._cbl_search("", "$");
    if (!status) {
        messageNwait("Found it!");
        // Select the line in the combo box so that an up or down arrow
        // selects the line above or below and not the current line.
        ctlcombol.p_cb_list_box._lbselect_line();
    }
}
```

A combo box receives an **on_change** event with a *reason* argument under the circumstances listed in the table below.

Reason	Description
CHANGE_OTHER	The p_text property changed, probably because of typing.
CHANGE_CLINE	The p_text property changed because selected line in list box changed and the list was visible.
CHANGE_CLINE_NOTVIS	The p_text property changed because a key was pressed which scrolls the list (Up , Down , PgUp , PgDn) while the list was invisible.

Reason	Description
CHANGE_CLINE_NOTVIS2	Same as CHANGE_CLINE_NOTVIS. Sent to user level 2 inheritance only. User level 2 inheritance will receive the CHANGE_CLINE_NOTVIS reason as well if the user level 1 inheritance does not catch the on_change event.

The **on_drop_down** event is sent to a combo box with a *reason* argument. The *reason* argument specifies one of the conditions listed in the table below.

Reason	Description
DROP_UP	After combo list box is made invisible.
DROP_DOWN	Before combo list box is made visible.
DROP_INIT	Before retrieve next/previous. Used to initialize list box before it is accessed.
DROP_UP_SELECTED	Mouse released while on valid selection in list box and list is visible.

Example:

```
#include "slick.sh"
defeventtab form1;
ctlcombol.on_drop_down(reason)
{
    if (reason==DROP_INIT) {
        if (p_user=="") {
            p_user=1;    // Indicate that the list box has been filled.
            // Insert a lot of items
            p_cb_list_box._insert_name_list(COMMAND_TYPE);
            p_cb_list_box._lbsort();
            p_cb_list_box._lbtop();
        }
    }
}
```

For a complete list of combo box control properties, methods, and events, from the main menu, select **Help > Macro Functions by Category**.

List Box Control

A list box provides a way to select from a fixed set of items. Multiple items from the list can be selected one time by setting the **multi_select** property to MS_SIMPLE_LIST or MS_EXTENDED (used by Open dialog box). A list box receives an **on_change** event, with a *reason* argument set to CHANGE_SELECTED, when items are selected or deselected because of a key press or mouse event. None of the **_lb???** functions cause an **on_change** event. Use the **_find_longest_line()** function to find the longest line in a list box.

Example:

```
// This example requires a form named "form1", a command button
// named "ok", and a list box named "ctl1list1".
#include "slick.sh"
defeventtab form1;
ctl1list1.on_change(reason)
{
    // Check the reason value.  In the future we may add more reason
    // values for the list box.
    if (reason==CHANGE_SELECTED) {
        // IF any items in the list box is selected
        if (p_Nofselected) {
            ctlok.p_enabled=1; // Enable the OK button
        } else if (!ctlok.p_enabled){
            ctlok.p_enabled=0; // Disable the OK button
        }
    }
}
```

Example:

```
// This example illustrates how to resize a dialog box based on the
// longest item in a list box.
#include "slick.sh"
defeventtab form1;
ctl1list1.on_create()
{
    _lbadd_item("Line1");
    _lbadd_item("This is a longer line2");
    _lbadd_item("This is the longest item in the list box");
    longest=_find_longest_line();

    // Add on a little to account for the left and right borders of the
    // list box.  Have to convert client width because its in pixels.
    list_width=longest+ p_width-_dx2lx(p_xyscale_mode,p_client_width);
    form_wid=p_active_form;

    // Again we have to account for the left and right borders.
    // Multiply p_x of list box by two to show equal amounts of
    // spacing on each side of the list box.
    form_width=2*p_x+ list_width+ form_wid.p_width-
        _dx2lx(form_wid.p_xyscale_mode,form_wid.p_client_width);

    p_width=list_width;
    form_wid.p_width=form_width;

    // Now make sure the whole dialog box can be seen on screen
    form_wid._show_entire_form();
}
```

Example:

```
//This example illustrates adding pictures to a list box.
#include "slick.sh"
#define PIC_LSPACE_Y 60    // Extra line spacing for list box.
#define PIC_LINDENT_X 60   // Indent before for list box bitmap.

defeventtab form1;
ctllist1.on_create()
{
    // Add some extra line height.
    p_pic_space_y=PIC_LSPACE_Y;
    // _pic_??? arguments are global variables defined in "slick.sh"
    // which are name table indexes to pictures.  You can create and
    // load your own pictures.  All the bitmaps are shipped with the
    // editor.  Use the bitmap file "_drremov.bmp" as a template for
    // creating your own bitmap for a list box.  You can load your
    // own bitmap files with the _update_picture function.
    _lbadd_item("a:",PIC_LINDENT_X,_pic_drremov);
    _lbadd_item("b:",PIC_LINDENT_X,_pic_drremov);
    _lbadd_item("c:",PIC_LINDENT_X,_pic_drfixed);
    // The p_picture property must be set to indicate that this list box
    // is displaying pictures and to provide a scaling picture for
    // the p_pic_point_scale property.  The p_pic_point_scale property
    // allows the picture to be resized for fonts larger or smaller than
    // the value of the p_pic_point_scale point size.  If
    // p_pic_point_scale is 0, the picture is not scaled.
    p_picture=picture;
    p_pic_point_scale=8;
}
```

Example:

```
// This example illustrates how to disable a list box and make the
// items in the list box appear grayed.
#include "slick.sh"
defeventtab form1;
ctllist1.on_create()
{
    _lbadd_item("item1");
    _lbadd_item("item2");
    p_no_select_color=1;
    p_enabled=0;
    p_forecolor=_rgb(80,80,80);
}
```

For a complete list of list box control properties, methods, and events, from the main menu, select **Help > Macro Functions by Category**.

Vscroll Bar Control

The scroll bar control is used to provide the user an avenue for selecting an integer that has a fixed range or a way for displaying the completion status of a process. Set the **min**, **max**, **small_change**, and **large_change** properties to define the minimum integer value, maximum integer value, increment/

decrement that occurs when arrows are pressed, and increment/decrement that occurs when you click the left button between the arrow and thumb box respectively.

The **on_change** event is sent after dragging the thumb box is completed. The **p_value** property contains the new scroll position and will be in the range **p_min..p_max**.

The **on_scroll** event is sent while you click and drag the thumb box of a scroll bar.

Example:

```
#include "slick.sh"
defeventtab form1;
ctlvscroll1.on_scroll()
{
    message("on_scroll p_value="p_value);
}
ctlvscroll1.on_change()
{
    message("on_change p_value="p_value);
}
```

For a complete list of scroll bar control properties, methods, and events, from the main menu, select **Help > Macro Functions by Category**.

Hscroll Bar Control

See [Vscroll Bar Control](#).

Drive List Control

The drive list is a combo box that allows selection of different disk drives. The Open dialog box uses this control.

The drive list control receives an **on_change** event with a *reason* argument of **CHANGE_DRIVE** when the drive is changed by selecting a different drive from the combo list box.

Example:

```
#include "slick.sh"
defeventtab form1;
ctlcombol.on_change(reason)
{
    if (reason==CHANGE_DRIVE) {
        message("Item selected from list. Current drive is now "_dvldrive());
    }
}
```

For a complete list of drive list control properties, methods, and events, from the main menu, select **Help > Macro Functions by Category**.

File List Box Control

The file list box control displays a list of files. Multiple files can be selected by setting the **multi_select** property to MS_SIMPLE_LIST or MS_EXTENDED used by Open dialog box. A file list box receives an **on_change** event with a *reason* argument under the circumstances listed in the table below.

Reason	Description
CHANGE_SELECTED	Occurs when items are selected or cleared because of a key press or mouse event. None of the _lb??? functions cause an on_change event.
CHANGE_FILENAME	The _flfilename() function was called which changed the file names listed.

Example:

```
#include "slick.sh"
defeventtab form1;
ctlcommand1.lbutton_up()
{
    ctllist1._flfilename("*.bat","c:\\");
}
ctllist1.on_change(reason)
{
    if (reason==CHANGE_FILENAME) {
        message("File list display directory "_flfilename());
    }
}
```

For a complete list of file list box control properties, methods, and events, from the main menu, select **Help > Macro Functions by Category**.

Directory List Box Control

The directory list box control displays a list of directories. A file list box receives an **on_change** event with one of the *reason* arguments listed in the table below.

Reason	Description
CHANGE_SELECTED	Occurs when items are selected or cleared because of a key press or mouse event. None of the _lb??? functions cause an on_change event.
CHANGE_PATH	The _dlfilename() function was called which changed the file names listed, the left mouse button was double-clicked, or Enter was pressed.

Example:

```
// This example requires a form named "form1", a text box named
// "ctltext1", and a directory list box named "ctlldir1".
#include "slick.sh"
defeventtab form1;
ctlldir1.on_change(reason)
{
    if (reason==CHANGE_PATH) {
        // Set the text in the text box to current directory. Changing
        // directories with the directory list box control changes the
        // editor's current directory.
        ctltext1.set_command(_dlpath(),1);
    }
}
```

For a complete list of directory list box control properties, methods, and events, from the main menu, select **Help > Macro Functions by Category**.

Picture Box Control

The picture box is used to place other controls inside of it, like the frame control. The picture box is capable of displaying bitmaps, displaying bitmap buttons, and all the features of the image control. To display bitmaps and bitmap buttons, use the image control feature described in the topic [Image Control](#).

For a complete list of picture box control properties, methods, and events, from the menu item select **Help > Macro Functions by Category**.

Gauge Control

Gauge control is typically used to indicate the completion status of a process.

Example:

```
// Create a form with a command button named ctlcancel, and
// gauge named ctlgauge1. Set the cancel and default
// properties of the command button to true.
//
#include "slick.sh"
static boolean gcancel;
_command test()
{
    // Need to tell compiler ctlgauge1 is a control because
    // the form1_wid.ctlgauge1 is too ambiguous.
    _control ctlgauge1;

    // Show the form modeless so there is no modal wait
    form1_wid=show("form1");
    // Disable all forms except form1_wid

    disabled_wid_list=_enable_non_modal_forms(0,form1_wid);
    gcancel=0;
    for (i=1;i<=100;++i) {
        // Read mouse, key, and all other events until none are left or
        // until the variable gcancel becomes true
        process_events(gcancel);
        if (gcancel) {
            break;
        }
        // Do work here. Replace the delay below with the operation
        // you want to do. The delay makes this example look more real.
        delay(10);

        form1_wid.ctlgauge1.p_value=i;
    }
    // Enable all forms that were disabled.
    _enable_non_modal_forms(1,0,disabled_wid_list);
    form1_wid._delete_window();
}
defeventtab form1;
ctlcancel.lbutton_up()
{
    gcancel=1;
}
```

For a complete list of gauge control properties, methods, and events, from the main menu, select **Help > Macro Functions by Category**.

Image Control

Image control is for creating bitmap buttons or toolbar buttons. The image control performs a subset of the features of the picture box control.

Adding a Bitmap Command Button or Check Box

Perform the following steps to add a bitmap button to a dialog box:

1. Create a new form for editing. From the main menu, select **Macro > New**.
2. Create an image control. Double-click the **Image Control** bitmap.
3. Set the **p_picture** property to `bbfind.bmp`. Make sure that you specify the full path (the default path used by the installation program is `c:\vslick\bitmaps` on Windows or `/usr/lib/vslick/bitmaps` on UNIX). In this step you enter the `bbfind.bmp` bitmap as an example. The **bb** prefix indicates that this is a bitmap that can be used by a toolbar. You can edit the `bbfind.bmp` file with Paintbrush (`pbrush.exe`). Use `bbblank.bmp` as a template for creating your own bitmap buttons.
4. Set the **p_command** property to `gui_find`. The Down arrow of the combo box displays all the editor commands.
5. Set the **p_message** property to **Searches for a string you specify**.
6. Set the **p_style** property to `PSPIC_FLAT_BUTTON` or `PSPIC_BUTTON`.

Example:

```
// This example illustrates how to load your own picture like
// a toolbar button.
#include "slick.sh"
defeventtab form1;
ctlimage1.on_create()
{
    index=_update_picture(-1,bitmap_path_search("bbfind.bmp"));
    if (index<0) {
        if (index==FILE_NOT_FOUND_RC) {
            _message_box("Picture bbfind.bmp was not found");
        } else {
            _message_box("Error loading picture bbfind.bmp\n\n"get_message(index));
        }
        return("");
    }
    p_picture=index;
    p_command="gui_find";
    p_message="Searches for a string you specify";
    p_style=PSPIC_FLAT_BUTTON;
}
```


Example:

```
// This example illustrates how to give the appearance of a button
// being pushed in. While you can do this by setting styles, here
// you can see how some other functions accomplish this task.
// For this example, create a form named form1 and an image control
// named ctlimage1.
#include "slick.sh"
defeventtab form1;
ctlimage1.on_create()
{
    index=_update_picture(-1,bitmap_path_search("bbfind.bmp"));
    if (index<0) {
        if (index==FILE_NOT_FOUND_RC) {
            _message_box("Picture bbfind.bmp was not found");
        } else {
            _message_box("Error loading picture bbfind.bmp\n\n"get_message(index));
        }
        return("");
    }
    p_picture=index;
    p_command="gui_find";
    p_message="Searches for a string you specify";
    p_style=PSPIC_BUTTON;
}
ctlimage1.button_down()
{
    // Reset the button counter so we don't get
    // double and triple click events.
    get_event('B');
    mou_mode(1)
    mou_capture();
    done=0;
}
```

Example continued:

```

event=MOUSE_MOVE;
for (;;) {
    switch (event) {
    case MOUSE_MOVE:
        mx=mou_last_x("m"); // "m" specifies mouse position in
                           // current scale mode
        my=mou_last_y("m");

        if (mx>=0 && my>=0 && mx<p_width && my<p_height) {
            if (!p_value) {
                p_value=1;    // Show the button pushed in.
            }
        } else {
            if (p_value) {
                p_value=0;    // Show the button up.
            }
        }
        break;
    case LBUTTON_UP:
    case ESC:
        p_value=0; // Restore the button state
        done=1;
    }
    if (done) break;
    event=get_event();
}
mou_mode(0);
mou_release();
say('out');
return("");
}

```

Adding Dialog Box Retrieval

Dialog box retrieval enables previous responses for check boxes, radio buttons, spin boxes, text boxes, and combo boxes to be retrieved. Press **F7** to retrieve the previous response, and **F8** to retrieve the next response. For example, the Insert Literal dialog box contains a spin box that is used to enter the character code of the character to insert. If you use it to enter a Hex value of 0xAE (to insert a registered trademark symbol), then later use it to enter a Hex value of 0x99 (to insert an unregistered trademark symbol), the next time you use the dialog you can press F7 to retrieve the previous entry of 0xAE, and then F8 to retrieve the next entry of 0x99.

The responses to dialog boxes are saved for the next session when you exit the editor and auto-restore is enabled.

Example:

```
// This example illustrates how to add dialog box retrieval to your own
// dialog boxes.  Create a form named "form1", a text box (any name), a
// check box (any name), and a command
// button named "ok".
#include "slick.sh"
defeventtab form1;
ctlok.on_create()
{
    // Retrieve the previous response to this dialog box.
    _retrieve_prev_form();
}
ctlok.lbutton_up()
{
    _save_form_response();
    p_active_form._delete_window(1);
}
```


Menus

You can change or add menu items by using the Menu Editor dialog box (**Macro > Menus**, select a menu). Or, to create a new menu, use the Open Menu dialog box (**Macro > Menus**) and click **New**. A quick way to bind a pop-up menu to a mouse click is to use the **Show** button on the Open Menu dialog box while recording a macro. When you are finished recording the macro, the Key Bindings dialog box is displayed which enables the binding of the new macro to a mouse click.

Menu Macro Programming

This section describes macro programming details about menus for advanced menu item enabling and for writing macros that manage menus.

Menu Editor Dialog Box

To access this dialog, select **Macro > Menus**, then click **Open** or **New**.

The Menu Editor is used for editing menu resources. Use the Menu Editor to modify the SlickEdit® MDI menu bar or an existing menu resource which can be displayed as a pop-up or menu bar. The **New** button on the Open Menu dialog box creates a new menu resource and places you in the Menu Editor so you can add menu items. After creating a menu, you can use the **Show** button on the Open Menu dialog box while macro recording to create a command which runs a menu by displaying it as a pop-up. If you bind the recorded command to a left or right button mouse event, the menu will be displayed at the cursor position. You DO NOT need to specify key bindings for menu items because our Menu Editor automatically determines the key bindings for you. Use the General Options dialog box (**Tools > Options > General**, select the **More** tab) to choose between short and long key names.

See [Creating and Editing Menu Resources](#) for information on creating forms with menu bars or advanced information.

The Menu Editor dialog contains the following options:

- **Menu name** - Renames the current menu resource to the name specified. You can define your own menu resource which is used instead of our menu bar WITHOUT changing the name of our default menu bar `_mdi_menu`. Use the `-m` invocation option (for example, `-m mymenu`) or set the `def_mdi_menu` macro variable to your menu name (**Macro > Set Macro Variable**).
- **Caption** - Title displayed for menu item. For menu items, set the caption to - if you want a line separating menu items which follow.
- **Command** - Macro command executed when the menu item is selected. This may be an internal macro command or a command line for running an external program.
- **Alias** - Displays the [Menu Item Alias Dialog Box](#).
- **Help Cmd** - Macro command executed when the user presses F1 when the menu item is selected. Usually it is a **help** or **popup_imessage** command. For example, if you specified **gui_open** as the menu item command, specify `help open dialog box` as the help item. If you don't know the name of the dialog box displayed, search for help on the command. The help for each command should indicate the name of the dialog box displayed. Some commands do not display dialog boxes. For these commands, specify `help command` where *command* is name of the command this menu item executes, or `help xxxx menu` where *xxxx* is the name of the drop-down menu this command is on.
- **Message** - Message text to be displayed when selection cursor is on this menu item. This message is currently only used when the menu is used as the SlickEdit menu bar. In the future, we will

supply a callback to allow other forms to display menu messages. Let us know if you need this feature.

- **Submenu** - Check this box if you want to create a menu which contains other menu items.
- **Auto Enable** - Displays the [Auto Enable Properties Dialog Box](#).
- **Up** - Moves the selected menu item above the previous menu item.
- **Down** - Moves the selected menu item below the next menu item.
- **Next** - Selects the menu item after the currently selected menu for editing. Use this button to insert a blank menu item after the last menu item in the list.
- **Insert** - Inserts a blank menu item before the selected menu item.
- **Delete** - Deletes the selected menu item.

Menu Item Alias Dialog Box

This dialog box allows you to define aliases (similar commands) for the command that is being executed. Enter each alias command on a separate line. If one of the alias commands is bound to a key, that key name will be displayed to the right of the menu item. For example, the **e** and **edit** commands are absolutely identically in function except that the **e** command requires fewer characters to type. The **gui_open** command is identical to the **edit** command except that it prompts the user with a dialog box, whereas the **edit** command prompts for files on the command line. These two examples illustrate the best reasons for using aliases.

Auto Enable Properties Dialog Box

For convenience, SlickEdit® has some predefined enable/disable attributes which you can specify for any command. When these predefined auto-enabling attributes are not enough, then you need to implement a callback which determines the enable/disable state of the command. See [Creating and Editing Menu Resources](#) for information on enabling and disabling menu items with your own callback.

The Auto Enable Properties dialog contains the following options:

- **Requires editor control** - Indicates that this command should be enabled only if operating on an editor control.
- **Allow in read only mode** - Indicates that this command should be enabled if the editor control is in strict read-only mode.
- **Allow when window is iconized** - Indicates that this command should be enabled if the editor control is an MDI child which is iconized.
- **Requires selection in active buffer** - Indicates that this command should be disabled if there is no selection in the active buffer.
- **Requires Context Tagging®** - Indicates that this command should be disabled if Context Tagging does not support the current buffer language type.
- **Requires fileman mode** - Indicates that this command should be disabled if the current buffer is not in fileman mode.
- **Allow in non-MDI editor control** - Indicates that this command should be allowed in a non-MDI editor control.
- **Requires block selection** - Indicates that this command should be disabled if there is no selection or the current selection is not of type block/column.

- **Requires a clipboard** - Indicates that this command should be disabled if there is no editor control clipboard available.
- **Requires a selection** - Indicates that this command should be disabled if there is no selection.

Save Failed Dialog Box

This dialog contains the following options:

- **Save as read only** - (UNIX only) This check box is enabled if your file does not have any **write** permissions (no **w** letter). Turn this check box on to have SlickEdit® temporarily change the permissions on the file to **read/write**. The resulting file will not have any write permissions (no **w** letter).
- **Save without creating a backup** - This check box is enabled if SlickEdit was unable to create a backup file when trying to save your file. This can happen when you don't have permissions to create the backup directory or when you are out of disk space. If you are editing files on a network drive, you may not have access rights for creating a backup directory on that drive.
- **Configure local backup directory** - (Non-UNIX platforms only) If you are editing files on a network drive, you may not have access rights for creating a backup directory on that drive. Configuring a local backup directory guarantees that you always have **write** access. If the directory you specify does not exist, SlickEdit will create one for you.

Insert Literal Dialog Box

To access this dialog, select **Edit > Insert Literal** or use the `insert_literal` command.

This dialog allows you to insert a character that you choose at the cursor location in the current buffer. The text box to the right of the **Character Code** label displays the character. The second text box displays the decimal character code, hex character code, or ASCII character depending on the option selected.

Creating and Editing Menu Resources

Modified menus are stored in the state file `vslick.sta` (UNIX: `vslick.stu`) file. The easiest way to create or change a menu is to use the Open Menu dialog box (**Macro > Menus**). After you select the menu, the Menu Editor dialog box is displayed and you can edit the menu resource. After the menu is created, use the `show`, `mou_show_menu`, or `_menu_show` function to run the menu by displaying it as a pop-up window. The `_menu_set` method may be used to create a menu bar on a form. Another way to create or change a menu is to define or modify a menu resource.

Defining a Menu Resource

Use the `insert_object` command to insert macro source code for a menu into the current buffer. Edit the resource properties and then run the macro to apply the resource changes. Ignore the message **No main entry point** if it is displayed. Changing a menu resource does not change any menu bars. Menu bars represent menu resources that have been loaded. A menu definition has the following format:

```
_menu menu_name {
    submenu menu_item, help_command, help_message, categories {
        menu_item, command, categories, help_command, help_message
    }
    submenu

    }endsubmenu
}
```

The table below contains the menu items and their definitions:

Menu item	Definition
menu_item	Menu item name in double quotes. Use & to choose selection character.
command	Any editor command. Places the cursor on the command line and press ? to list all editor commands.
help_command	Command to be executed when F1 is pressed. Usually it is a help or popup_imessage command.
categories	Specifies zero or more help categories in double quotes. Multiple help categories are separated with (pipe).
help_message	A single line message in double quotes displayed on message line.

Example of a menu definition:

```
_menu mymenu {
    submenu "&File", "Help file menu", "Displays File drop-down menu", "ncw" {
        "&New", "new", "ncw", "help new", "Creates a new file to edit";
        "&Open\tCtrl+O", "gui_open", "help gui_open", "Open a file";
    }
    submenu "&Edit", "Help edit menu", "Displays Edit drop-down menu", "ncw" {
        "Cu&t", "cut", "sel|nrndonly", "help cut", "Deletes the selection and copies it
to the clipboard";
    }
}
```

Predefined Attributes for Auto-Enabling Commands

Predefined enabling or disabling attributes can be specified for any command. Specify these attributes in the **name_info** of a command definition. Auto-enabling attributes affects the enable/disable state for a command placed in a menu or in a toolbar. The following command is disabled when there is no editor control on which to operate:

```
#include slick.sh
_command void top_of_buffer()
    name_info(' VSARG2_READ_ONLY| VSARG2_REQUIRES_EDITORCTL)
{
}
}
```

Macro Callbacks for Enabling Commands

If the auto-enable attributes do not provide the features that you want, you can define the enable and disable callback for the command. The name of the callback function you define is based on the name of the command as shown in the following example:


```

#include "slick.sh"
static boolean gSomeOtherState;
/*
    This function gets called if your command is used in a menu or toolbar.
    You must return a combination of the MF_ flags ORed together.

    BEWARE: If an _OnUpdate callback causes a Slick-C run-time error, you
    may not see the error. In addition, the timer used for toolbars,
    Context Tagging(R), AutoSave, and some other features may be
    automatically terminated. Exit and restart the editor to restart
    this timer. Use the "say" function to debug your _OnUpdate
    callback
*/
int _OnUpdate_mycommand(CMDUI &cmdui,int target_wid,_str command)
{
    //say('h1');
    // Lets assume this command requires the target to be an
    // editor control with a selection.
    // IF the target is not an editor control
    if ( !target_wid || !target_wid._isEditorCtl()) {
        //say('disabled at h2');
        return(MF_GRAYED);
    }
    //say('h3');
    // IF the editor control does not have a selection

    if (!target_wid.select_active2()) {
        //say('disabled at h4');
        return(MF_GRAYED);
    }
    //say('h5');
    if (gSomeOtherState) {
        //say('disabled at h6');
        return(MF_GRAYED);
    }
    //say('enabled at h7')
    return(MF_ENABLED);
}
_command void mycommand() name_info('','VSARG2_REQUIRES_EDITORCTL)
{
    // Some code here...
}
// This command effects the enable/disable of mycommand
_command void mycommand2(_str argument="0")
{
    gSomeOtherState=(argument)?1:0;

    // Indicate that the enable state of the toolbar buttons must be
    // updated. The _tbSetRefreshBy function is very fast.
    // Toolbars will be updated after the macro terminates and
    // the user stops typing fast.
    _tbSetRefreshBy(VSTBREFRESHBY_USER);
}

```

Placing a Menu Bar on a Form

The following sample code shows how to add a menu on a form as a menu bar:

```
#include slick.sh
// Create a form called form1 and set the border style to anything BUT
// BDS_DIALOG BOX.  Windows does not allow forms with a dialog
// box style border to have menu bars.
defeventtab form1;
form1.on_load()
{
    // Find index of MDI menu resource
    index=find_index(def_mdi_menu,oi2type(OI_MENU));
    // Load this menu resource
    menu_handle=p_active_form._menu_load(index);
    // _set_menu will fail if the form has a dialog box style border.
    // Put a menu bar on this form.
    _menu_set(menu_handle);
    // You DO NOT need to call _menu_destroy.  This menu is destroyed
    // when the form window is deleted.
}
form1.on_init_menu()
{
    // Gray out all menu items that are not allowed when there
    // no child windows.
    _menu_set_state(p_menu_handle,!ncw,MF_GRAYED,C);
}
```

Displaying a Menu as a Pop-Up

If the **show** or **mou_show_menu** function meets your needs, use one of them. The following sample code shows how to display a menu as a pop-up:

```
#include slick.sh
defmain()
{
    // Low-level code to display menu bar as pop-up.
    // Could just use show or mou_show_menu function.
    index=find_index(_mdi_menu,oi2type(OI_MENU))
    if (!index) {
        message(Can't find _mdi_menu);
    }
    menu_handle=_menu_load(index,P);
    // Display this menu in the menu of the screen.
    x=_screen_width()/2;y=_screen_height()/2;
    flags=VPM_CENTERALIGN|VPM_LEFTBUTTON;
    _menu_show(menu_handle,flags,x,y);
    _menu_destroy(menu_handle);
}
```


Common Macro Dialog Boxes

There are several important macro dialog box forms and functions that you can use in your own macros. The table below lists the general purpose forms and dialog box functions.

Form	Description
<code>_textbox_form</code>	General purpose form that displays a variable number of text boxes or combo boxes.
<code>_sellist_form</code>	General purpose form that displays a list box, an optional combo box, and a variable number of command buttons.
<code>_open_form</code>	General purpose form used to open and save files that does not have the advanced controls.
<code>_edit_form</code>	General purpose form used to open and save files that has the advanced controls used for the File > Open dialog box.
<code>_font_form</code>	General purpose form used to prompt for a font.
<code>_choose_font</code>	(Non-UNIX platforms only) General purpose dialog box built in to operating system used to prompt for a font.
<code>_printer_setup</code>	(Non-UNIX platforms only) General purpose dialog box built in to operating system used for printer setup.

If a key displays a dialog box, you can find out the command the key executes by using the Key Bindings dialog box (**Tools > Options > Key Bindings**).

String Functions

The table below describes commonly used string functions. See **Help > Macro Functions by Category > String Functions** for a complete list.

See also documentation for the [parse Statement](#).

Function	Description
<code>_str center (_str string,int width [,-str pad_ch])</code>	Returns <i>string</i> padded evenly on left and right with spaces or a character you choose with the optional argument <i>pad_ch</i> .
<code>_ dec2hex (long number [,int base])</code>	Returns <i>number</i> converted to <i>base</i> specified.
<code>_str expand_tabs (_str string [,int start [,int count [,_str option]]])</code>	Very similar to substr function except that this function supports tab characters very well.
<code>_str field(_str string,int width)</code>	Returns <i>string</i> padded with trailing spaces to <i>width</i> characters
<code>long hex2dec(_str number [,int base])</code>	Returns <i>number</i> converted to <i>base</i> specified.
<code>_str indent_string(int width)</code>	If indent with tabs is on, a string of tabs of length <i>width</i> is returned. Otherwise, a string of spaces of length <i>width</i> is returned.
<code>boolean isalnum(_str ch)</code>	Returns non-zero value if <i>ch</i> is a numeric or alphabetic character.
<code>boolean isalpha(_str ch)</code>	Returns non-zero value if <i>ch</i> is an alphabetic character.
<code>boolean isdigit(_str ch)</code>	Returns non-zero value if <i>ch</i> is a numeric character.
<code>boolean isinteger(_str string)</code>	Returns non-zero value if <i>string</i> is a valid int . If <i>string</i> is floating point number, 0 is returned.
<code>boolean isnumber(_str string)</code>	Returns non-zero value if <i>string</i> is a valid double (floating pointer number).
<code>_str last_char(_str string)</code>	Returns last character of <i>string</i> . If <i>string</i> is null, the space character is returned.
<code>int lastpos(_str needle [,_str haystack [,int start [,_str options]])</code>	Returns the position (1..length(haystack)) of the last occurrence of <i>needle</i> in <i>haystack</i> . If <i>needle</i> is not found, 0 is returned. Regular expressions are supported.
<code>int length(_str string)</code>	Returns the number of characters in <i>string</i> .
<code>_str lowercase(_str string)</code>	Returns <i>string</i> converted to lowercase.
<code>_str number2onoff(_str number)</code>	Returns off if number==0 . Otherwise on is returned.

STRING FUNCTIONS

Function	Description
<code>_str number2yesno(_str number)</code>	Returns N if <i>number</i> ==0. Otherwise Y is returned.
<code>parse <i>expr</i> with <i>template</i></code>	Breaks apart the expression <i>expr</i> given into variables that appear in <i>template</i> , and much more.
<code>boolean parseoption(_str &cmdline, _str option_ch)</code>	Strips + or - option from <i>cmdline</i> . Returns non-zero number if <i>option_ch</i> was found.
<code>int pos(_str needle [, _str haystack [, int start [, _str options]]])</code>	Returns the position (1..length(<i>haystack</i>)) of the first occurrence of <i>needle</i> in <i>haystack</i> . If <i>needle</i> is not found, 0 is returned. Regular expressions are supported.
<code>boolean setonoff(_str &name, _str value)</code>	Sets <i>name</i> to 1 or 0 corresponding to <i>value</i> =on or <i>value</i> =off. Returns 0 if input value is valid. Displays message if <i>value</i> is not on or off.
<code>boolean setyesno(int &name, _str value)</code>	Sets <i>name</i> to 1 or 0 corresponding to <i>value</i> =Y, Yes or <i>value</i> =N, No. Returns 0 if input value is valid. Displays message if <i>value</i> is not Y or Yes, N or No.
<code>_str stranslate(_str string, _str replace_string, _str search_string, _str search_options)</code>	Returns <i>string</i> with all occurrences of <i>search_string</i> replaced with <i>replace_string</i> .
<code>_str strieq(_str string1, _str string2)</code>	Returns true if <i>string1</i> matches <i>string2</i> when case is ignored.
<code>_str strip(_str string, _str ltb [, _str strip_char])</code>	Returns <i>string</i> stripped of leading and/or trailing <i>strip_char</i> .
<code>_str strip_filename(_str filename, 'P' 'D' 'E' 'N')</code>	Returns <i>filename</i> with part stripped. P=Path, D=Drive, E=Extension, N=Name.
<code>_str strip_last_word(_str &line)</code>	Returns the last space delimited word in <i>line</i> . The last word and trailing spaces are deleted from <i>line</i> .
<code>_str strip_options(_str cmdline, _str &options)</code>	Returns <i>cmdline</i> without words that start with the characters -, +, or [. <i>options</i> variable is set to stripped option words.
<code>_str substr(_str string, int start [, int length [, _str pad]])</code>	Returns <i>length</i> characters of <i>string</i> beginning at <i>start</i> . By default, <i>length</i> defaults to rest of <i>string</i> . If <i>length</i> is greater than length of <i>string</i> , the return string is padded with blanks or <i>pad</i> character if specified.
<code>_str translate(_str string [, _str output_table [, _str input_table [, _str pad]]])</code>	Returns <i>string</i> with characters translated according to arguments.
<code>_str upcase(_str string)</code>	Returns <i>string</i> converted to uppercase.

Function	Description
<code>int verify(_str string, _str reference [, M] [,int start])</code>	Returns the position (1..<code>length(string)</code>) of first character not matching or matching a character in <i>reference</i> . 0 is returned on failure.
<code>_str word(_str string,int Nth)</code>	Returns the <i>Nth</i> space or tab-delimited word in <i>string</i> . Is returned if the <i>Nth</i> word does not exist.

Search Functions

Two levels of search functions exist: high level functions that provide user interfacing and multiple file searching, and built-in functions that are used without affecting the high level search commands such as the **find_next** command. The built-in functions are not affected by the global editor search options.

The table below shows a list of commonly used search functions. For a complete list, see **Help > Macro Functions by Category > Search Functions**.

Function	Description
<code>gui_find</code>	Displays Find and Replace tool window open to the Find tab, and performs search using the find or _mffind functions.
<code>gui_replace</code>	Displays Find and Replace tool window open to the Replace tab, and performs search using gui_replace2 or _mfreplace functions.
<code>gui_replace2</code>	Performs a search and replace based on arguments given. This function is very similar to the replace function, except that this function uses a dialog box to prompt the user where to replace.
<code>find_next</code>	Searches for next occurrence of search string used by any of these high-level search functions. This function is not affected by previous searches done with low-level built-in functions.
<code>find</code>	Performs search based on arguments given.
<code>replace</code>	Performs a replace based on arguments given. The user is prompted where to replace through the message line.
<code>_mffind</code>	Performs a multiple file and buffer search based on the arguments given.
<code>_mfreplace</code>	Performs a multiple file and buffer search based on the arguments given.
<code>search</code>	Performs a search, or search and replace, based on arguments given. Does not support wrapping to top or bottom of file. When performing a replace, the user is not prompted at all.
<code>repeat_search</code>	Searches for the next occurrence of search string used by last call to the search built-in.

Example:

```
// This example searches for lines that contain a particular search
// string and places the lines in another window and buffer.
defmain()
{
    orig_wid=p_window_id;
    // The +w option forces a new window to be created. The +t options
    // force a new buffer to be created.
    status=edit("+w +t");
    if (status) {
        _message_box("Unable to create temp window and buffer\n\n":+
            get_message(status));
    }
    delete_line();           // Delete the blank line
    output_wid=p_window_id;

    p_window_id=orig_wid;
    top();                   // Place the cursor at the top in column 1.

    status=search("if","w@"); // Case insensitive word search for if
                             // @ specifies no string not found
                             // message.

    for (;;)
    {
        if (status) {
            break;
        }
        get_line(line);      // Place the cursor at the end of the line so no
                             // more occurrences can be found on this line
        _end_line();
        output_wid.insert_line(line);
        status=repeat_search();
    }
    // Make the output window active so we can see the results
    p_window_id=output_wid;
}
```

Example:

```
// This example is very similar to the example above except that the
// output data is placed in a view and buffer.  The only advantage in
// using a view and buffer is that the output can be displayed
// in a list box without the user having to see a new window created.
#include "slick.sh"
defmain()
{
    // Create a temporary view and buffer within the current window.
    // Each window can store multiple cursor positions (views) to
    // any buffer.
    orig_view_id=_create_temp_view(temp_view_id);

    if (orig_view_id=="") {
        return("");
    }

    activate_view(orig_view_id);
    top(); // Place the cursor at the top in column 1.
    status=search("if","w"); // Case sensitive word search for if
    for (;;) {
        if (status) {
            // clear the pending message caused by built-in search failing
            clear_message();
            break;
        }
        get_line(line);
        // Place the cursor at the end of the line so no more occurrences
        // can be found on this line.

        _end_line();
        activate_view(temp_view_id);
        insert_line(' 'line); // Insert a space at the beginning of the line
                               // because this will be inserted into a listbox
        activate_view(orig_view_id);
        status=repeat_search();
    }
}
```

Example continued:

```
// Display the buffer in a list box.
// The _sellist_form dialog box will delete the temp view and buffer
// The original view must be activated before showing
// the _sellist_form or
// the dialog box will operate strangely
activate_view(orig_view_id);
result=show("_sellist_form -mdi -modal",

           "Sample Selection List",
           // Indicate next argument is view_id
           SL_VIEWID|SL_SELECTCLINE,
           temp_view_id,
           "OK",
           "", // Help item
           "", // Use default font
           ""  // Call back function
           );
if (result) {
    message("Selection list cancelled");
} else {
    message("Item selected is "result);
}
}
```

Selection Functions

SlickEdit® supports multiple selections; however, only one selection can be active or visible. Selections are specified by handles. The **_alloc_selection** built-in returns a handle to a selection. The function **_free_selection** frees a selection associated with the selection handle given. Most selection functions accept a selection handle. A handle of "" specifies the active selection or selection showing, that is always available. The **_free_selection** function cannot free the active selection. To free the active selection, you must use the **_show_selection** function first to make another selection the active selection. The actual handle number of the active selection is returned by the expression **_duplicate_selection()**.

For a list of selection-related functions, from the main menu, select **Help > Macro Functions by Category > Selection Functions**.

Example:

```
// Duplicate the current line.
mark_id=_alloc_selection();
//
if (mark_id<0) {
    message(get_message(mark_id));
    return(rc);
}
_select_line(mark_id);
_copy_to_cursor(mark_id);
// This selection can be freed because it is not the active selection.
_free_selection(mark_id);

// This code copies selected text and keeps the resulting selection on
// the source text instead of the destination text.
if (_select_type()==""){
    message(get_message(TEXT_NOT_SELECTED_RC));
    return(1);
}
mark_id=_duplicate_selection()    // Make a copy of the active selection.
_copy_to_cursor();
// Save the selection id.
old_active_mark_id=
duplicate_selection();
// Must make another mark active before the old active mark can
// be freed.
show(selection(mark_id));        // Make copy of visible mark active
free(selection(old_active_mark_id));
```


Writing Selection Filters

The module `markfilt.e` provides the procedure **filter_selection** for filtering selected text. Define a global procedure that accepts a string and returns a string. Then pass the name of the procedure to the **filter_selection** procedure.

Example:

```
// This batch program converts the marked text into hexadecimal ascii
// codes. Each hexadecimal ascii code is separated by a comma. One
// possible use of this function could be to convert a binary font
// file into hexadecimal ascii codes to be compiled into a C program.
#include "slick.sh"

_str hex_filt(string);

defmain()
{
    if (_select_type()==" " ) {
        message(get_message(TEXT_NOT_SELECTED_RC));
        return(TEXT_NOT_SELECTED_RC);
    }
    // Under scores must be converted to dashes.
    return(filter_selection(hex_filt));
}

_str hex_filt(string)
{
    line="";
    for (i =1;i<=length(string);++i) {
        line=line:+dec2hex(_asc(substr(string,i,1))):+",";
    }
    return(line);
}
```


Unicode and SBCS or DBCS Macro Programming

The following information applies for Unicode users only. When the code editor is running in UTF-8 mode (by default, `vs.exe` for Windows runs in this mode), buffers can contain either SBCS/DBCS data or UTF-8 data depending on how a buffer is loaded. To make it easier for macros to support these two buffer data formats, almost all macro functions accept and return UTF-8 strings. This allows most macros to automatically work. Macros that use or set column positions often do not work correctly for both buffer data formats. The solution is to call raw functions.

Example:

```
// This will not work if the current buffer is an SBCS/DBCS buffer,
// word is a UTF-8 string (that this example assumes), and word
// contains characters above 127.
p_col=p_col+length(word);
// This will work
p_col=p_col+_rawLength(word);
// This works too
word=_rawText(word);
p_col=p_col+length(word);
```

Example:

```
This will not work if the current buffer is an SBCS/DBCS buffer and
The current line contains characters above 127.
get_line(line);

string=expand_tabs(line,p_col);
// This works
get_line_raw(line);

string=expand_tabs(line,p_col);
// This works too, but is less efficient if all operations on line
// can support raw data.
get_line(line);
string=expand_tabs(_rawText(line),p_col);
```

The **_UTF8()** macro function indicates if the code editor is in UTF-8 mode. The **p_UTF8** property tells you whether the current buffer contains UTF-8 data. The **p_encoding** property indicates what format the buffer will be saved in by default.

Like typical programming languages (Java, C++), Slick-C® source files are code page dependant. Strings are converted from the current code page to UTF-8. This is important if you enter characters above 127. All of the macro functions and properties accept and return UTF-8. The Slick-C functions in the table below DO NOT accept or return UTF-8 data.

Function	Definition
The function: <code>_default_option(VSOPTIONZ_SPECIAL_CHAR_XLAT_TAB).</code>	All other options for this function are UTF-8.
All seek functions: <code>goto_point()</code> , <code>_QROffset()</code> , <code>_GoToROffset</code> , <code>_nrseek()</code> , <code>point()</code> , and <code>seek()</code> .	All seeking is done on raw data. Buffers need to be loaded in the same raw format so that seek functions work.

Function	Definition
All <code>_rawXXX()</code> or <code>XXX_raw()</code> functions.	Unlike the C API, the Slick-C functions <code>get_text()</code> and <code>_expand_tabsc()</code> return UTF-8 data.

The **`p_display_xlat`** Slick-C property DOES NOT accept or return UTF-8 data.

The following are the Slick-C raw functions:

- `_expand_tabsc_raw()`
- `get_line_raw()`
- `get_text_raw()`
- `insert_line_raw()`
- `_insert_text_raw()`
- `replace_line_raw()`
- `_rawLength()`
- `_rawSubstr()`
- `_rawText()`

The table below shows the raw functions that optionally support raw data.

Function	Description
<code>pos()</code>	When <code>p_rawpos</code> appended to <i>options</i> argument.
<code>lastpos()</code>	When <code>p_rawpos</code> appended to <i>options</i> argument.
<code>upcase()</code>	When <code>p_UTF8</code> property given as second argument.
<code>lowcase()</code>	When <code>p_UTF8</code> property given as second argument.
<code>parse</code>	When <code>p_rawpos</code> appended to <i>options</i> of search argument.

The following are the Slick-C new UTF-8 functions:

- `_MultiByteToUTF8()`
- `_UTF8()`
- `_UTF8Asc()`
- `_UTF8Chr()`
- `_UTF8ToMultiByte()`

The following C API functions DO NOT accept or return UTF-8 data:

- The functions `vsGetText()`, `vsGetRText()`, `vsExpandTabsc()`, `vsQSelectedTextLength()`, `vsGetSelectedText()` - These function always return raw data. Use the **`vsUTF8()`** function or check the `VSP_XLAT` property to determine if you need to translate the buffer data. Since these API functions assume that the maximum buffer length is the same as the read length, it would be useless for these functions to return translated data.
- All seek functions (`vsQOffset`, `vsQROffset`, `vsGoToPoint`, and `vsGoToROffset`) - All seeking is done on raw data. Since the Context Tagging® database stores seek positions, buffers need to be loaded in the same raw format so that seek works.

- All `vsXXXRaw()` functions.

Shelling Programs from a Slick-C® Macro

To execute another program from a Slick-C macro, use the **shell** built-in, the **dos** command, or the **execute** built-in. The latter method is similar to executing a command on the command line, and enables the creation of expressions that execute Slick-C internal commands, Slick-C batch programs, or external programs. If you are only interested in executing an external program, use the **shell** built-in or the **dos** command.

Example:

```
// Capture the output of Slick GREP and process the error messages.
dos("-e grep DEBUG *.c");
// Redirect the output of grep to a file.
shell("grep DEBUG *.c >junk");
// Run the DOS dir command and wait for a key to be pressed before
// closing command shell window.
shell("dir *.c >junk","w");
// Display the Calculator dialog box. Show is an internal command.
execute("show _calc_form");
```


DLL Interface

SlickEdit products have a DLL interface for Windows. Use the Slick-C® macro language instead of the DLL interface except when you need an interface to the DLL in another program, when better speed is needed, or when the Slick-C macro language is missing a function that you want.

After a DLL function is added, call it from a Slick-C macro just like any other Slick-C function. DLL functions can be used for timer call backs and any place a Slick-C function is used.

To get started using the DLL interface, edit the `simple.c` file located in the `samples\simple` subdirectory of your installation directory. The VSAPI functions have the prefix **vs**.

Command Line Interface

This section describes how to write macros using the command line interface.

Command Line Arguments

When a command is invoked, the expression **arg(1)** contains the rest of the command line after the name with leading spaces removed. Alternatively, the command can declare a named argument whose value is the same as **arg(1)**. For example, invoking the edit command **e file1 file2** calls the **e** command with **file1 file2** in **arg(1)**. The **parse** built-in is an excellent function for parsing a command line string. When another macro calls a command, more than one argument string can be passed. Calling the **arg** function with no parameters returns the number of parameters with which the command or procedure was called.

Example:

```
#include "slick.sh"
// This command supports completion on a filename followed by an
// environment variable argument
_command test1() name_info(FILE_ARG,"ENV_ARG")
{
    parse arg(1) with file_name env_name;
    message("file_name="file_name" env_name="env_name);
}
```

The string constant expression given to the **name_info** keyword is used for argument completion, restricting when the command can be executed, and a few other options.

get_string Procedure

The **get_string** procedure reads a single argument from the user.

Example:

```
#include "slick.sh"
_command test2()
{
    if (get_string(file_name,"Filename: ",FILE_ARG;Help message")) {
        return(1); // Cancel key pressed.
    }
    if (get_string(env_name,"Environment variable name: ",
        ENV_ARG;Help message","PATH") ) {
        return(1); // Cancel key pressed.
    }
    message("file_name="file_name" env_name="env_name);
}
```

Single Argument Prompting with Support for Prompt Style

Use the **prompt** procedure to write a command that accepts one command line argument, or prompts for the argument if it is not given. If the user presses **Esc** while being prompted for the argument, file execution does not continue.

COMMAND LINE INTERFACE

Example:

```
// This command supports completion on an environment variable argument

#include "slick.sh"
_command test3() name_info(ENV_ARG)
{
    // If the user selects to abort, the prompt procedure
    // stops execution.
    env_name=prompt(arg(1),"Environment variable name: ");
    message("env_name="env_name);
}
```

Hooking Exit and Other Events

Invoking a Macro on Startup

To invoke any macro command defined by typing **_command** or an external program when the editor initializes, use the **-#** invocation option. For example, invoking the command **vs makefile -#bottom_of_buffer** loads the file `makefile` and executes the **bottom_of_buffer** command. To invoke a command with parameters, place the command and parameters inside double quotes. Another method for getting macro code to start without changing any invocation options is to create a module with a definite entry point.

Invoking a Macro on Exit

If you want a function to be invoked when the editor exits, create a macro procedure with a name that has the prefix **_exit_**. To automatically invoke a macro when exiting SlickEdit®, use the following code:

```
_exit_cleanup_stuff()  
{  
    messageNwait("Got here");  
}
```


State File Caching

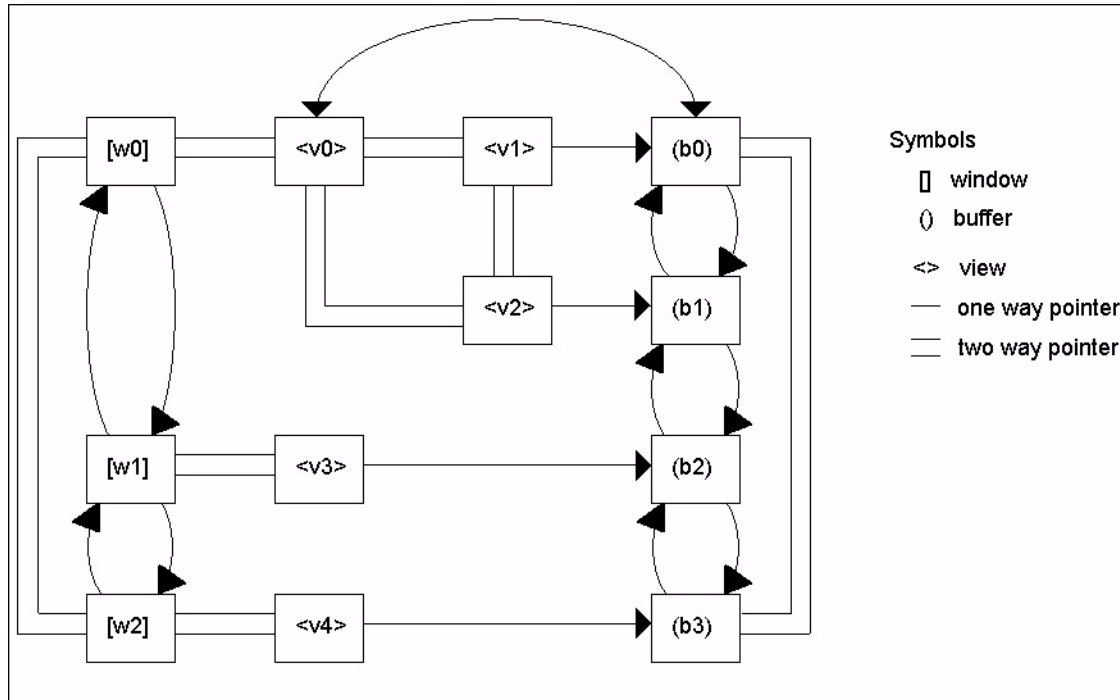
By default, a module, event, dialog box template, or picture from `vslick.sta` (UNIX: `vslick.stu`) is not loaded until it is referenced. Using the **definit** primitive forces a module to be loaded when the editor is invoked. The default state file cache is about 200 K. You can set this size with the **-st** invocation option or with the General Options dialog box (**Tools > Options > General**, select the Virtual Memory tab). When the state file cache becomes full, the least recently used module, dialog box template, event table, or picture is removed from memory to reduce the cache size.

You might have critical modules that you want permanently stored in memory. Place the **no_code_swapping** keyword at the top of the module to force the module to be loaded and permanently stored in memory on startup; then, if a critical disk failure occurs while reading the state file, the product is protected. A few modules that provide basic editing capabilities remain permanently in memory.

Windows Data Structure

This section describes how to write a Slick-C® macro that manipulates windows, views, and buffers.

The following diagram shows startup with two files loaded (buffers b2 and b3) and two windows (w1 and w2) viewing those files:



The extra window, w0, is a hidden window used to allow quick switching to system buffers such as **.command** and **.killed**. If you attempt to leave the hidden window active, another window is made active when the editor refreshes the screen. Since window w1 is active, you currently see window w1 of buffer b2. You might be able to see window w2 of buffer b3 if the window w1 does not overlap window w2.

A ring of buffers and a ring of windows are maintained, where each window may contain a ring of views. However, by convention, all windows except the hidden window contain one view. Some macros temporarily create extra views in other windows, but they delete them before they terminate. The built-ins **_next_buffer** and **_prev_buffer** activate the next and previous buffers. **_next_window** and **_prev_window** move around the window ring. **_next_view** and **_prev_view** move around the active view ring. The built-in function **load_files** inserts views, windows, and/or buffers. The command **_delete_buffer** removes the active buffer from the buffer ring and activates the previous non-hidden buffer. **_quit_view** removes the active view from the active windows view ring. The previous view becomes the new active view. When **_quit_view** is executed and only one view exists in the active window, the window is removed and the previous window becomes active. The hidden window cannot be deleted.

A view holds the information necessary for the editor to remember the location and scroll position in a buffer. A view also contains a window id and a buffer id. Activating a view with the **activate_view** built-in activates the window and buffer specified by the view as well as selecting the cursor location/scroll position.

Each buffer maintains a non-active view. When a buffer is activated by one of the built-ins **_next_buffer**, **_prev_buffer**, **_delete_buffer** or **load_files** (assuming you do not use an option that overrides this), the

WINDOWS DATA STRUCTURE

active view information is saved in the non-active view of the buffer, and the buffer's new non-active view information is copied into the active view.

The tables below describe the contents of each structure.

Window Properties

Window Property	Description
<code>p_window_x, p_window_y</code>	Top left coordinates of window.
<code>p_window_height, p_window_width</code>	Height and width of window.
<code>p_tile_id</code>	Indicates that windows are part of a tile window group and whether a window is zoomed. Windows of a tiled window group have the same tile_id . A zoomed window has a negative tile_id .
<code>p_x</code>	The top left x position of window.
<code>p_y</code>	The top left y position of window.
<code>p_height</code>	Window height.
<code>p_width</code>	Window width.
<code>p_view_id</code>	Pointer to active view.
<code>p_next (_next_window)</code>	Window id of next window.
<code>p_prev (_prev_window)</code>	Window id of previous window.
<code>p_child</code>	Window id of child window.

View Properties

View Property	Description
<code>block, line within block</code>	Accessible via point and goto_point .
<code>p_line</code>	Line number of current line.
<code>p_col</code>	Column position within current line (1..2 billion).
<code>p_left_edge</code>	Column scroll position.
<code>p_cursor_x</code>	Text cursor x position.
<code>p_cursor_y</code>	Text cursor y position.
<code>p_window_id</code>	Window id.
<code>p_buf_id</code>	Buffer id.

Buffer Properties

Buffer Property	Description
<code>p_buf_name</code>	Name of buffer.

Buffer Property	Description
p_buf_flags	Indicates whether a buffer is hidden and may specify other buffer options.
p_Noflines	Number of lines in file.
p_modify	Non-zero indicates buffer has been modified.
p_margins	String containing left, right, and new paragraph margins (1..2 billion).
p_tabs	String containing up to 2 billion tab stops.
p_mode_name	Name of current mode.

Tutorials

Defining Stack Routines

These examples show you what can be done in a language that supports typed variables and untyped container variables. The following example code shows how to define a set of stack routines in Slick-C® that support any type of element:

```
void stacknew(typeless &stack)
{
    stack._makeempty(); // Destroy current contents of stack.
    stack[0]=0;         // Make an array and use first element as top count.
}
void stackpush(typeless &stack, typeless &value)
{
    stack[++stack[0]]=value;
}
typeless stackpop(typeless &stack)
{
    if (stack[0]<=0) return('');
    // Make a copy of the element.
    result=stack[stack[0]--];
    // Free space allocated by value and delete array element. _deleteel is a
    // built-in method which operates on arrays and hash tables.
    stack._deleteel(stack[0]+1);
    return(result);
}
defmain()
{
    // The above routines can handle variables of any type, including
    // string constants.

    struct RECORD {
        int i;
        _str s;
    };
    // You can't make a limit on the number of elements in an array.
    // We will add support for initially allocating a specific number of elements.
    RECORD arecord[];
    arecord[0].i=4;arecord[0].s="element 0";
    RECORD symboltable[]; // Declare a hash table/associative array
    symboltable:["name1"].i=1;symboltable:["name1"].s="element 0";
    stacknew(stack);
    stackpush(stack,arecord); // Push an array onto the stack

    stackpush(stack,symboltable); // Push a hash table/associative array onto
    // the same stack

    stackpush(stack,"string"); // Push a string constant onto the same stack
}
```

The following example shows how a container variable can access structure members as an array:

```

/*
   Read lines of a file which contains tab delimited data into an array of
   structures.  Each line represents an array structure element.

   The tab delimited data on each line represents fields in the structure.
   We will assume the file contains valid data for filling this structure.
*/
int ReadTable(_str filename,typeless (&table)[])
{
    // Use an editor buffer to open and cache the file.  Data is read
    // in blocks from the file only.  We don't need this much power, but
    // Slick-C needs a few more non-editor file I/O functions.
    status=_open_temp_view(filename,temp_view_id,orig_view_id);

    if (status) return(status);
    top();up(); // Place cursor on line 0 before first line of file.
    for (j=0;++j) {
        if (down()) break;
        get_line(line);
        if (line=="") continue;
        rest=line;
        p= &table[j]; //Make p point to this structure element.
        // Here we access structure members as an array of elements
        p->[0]="";
        // Note that loop supports fields which are strings of length 0
        for (i=0;++i) {
            if (rest=="") break;

            // Parse is similar to REXX.  We unable to come up with a function syntax
            // that we were satisfied with so with went with a REXX style syntax.
            // Place text up to but not included tab character into value variable.
            // Place tab character and rest of data in rest variable.
            parse rest with value "\t" +0 rest;
            if (substr(rest,1,1)=="\t") {
                rest=substr(rest,2);
            }
            p->[i]=value;
        }

        }
    _delete_temp_view(temp_view_id);
    activate_view(orig_view_id);
    return(0);
}

struct TABLE_ENTRY {
    _str name;
    int value;
};

// defmain is the main entry pointer for a Slick-C batch/script macro
defmain()
{

```

Example continued:

```
TABLE_ENTRY table[];
// Table file should exist
// NOTE: (TABLE_ENTRY []) is type compatible with (typeless [])
status=ReadTable("table",table);
if (status) {
    _message_box("Failed to read table file");
    return(1);
}
_message_box("First record:  name=":+table[0].name:+ " value=":+table[0].value);
}
```

Searching for a String Within a Current Function

This macro can be used with many languages. It searches the current procedure or function for a specified string, with specified options. Use this macro in cases where references do not work, such as searching for a partial identifier name.

Several useful aspects of this macro, aspects that can be reused in other macros, are that it prompts the user for a string, it selects the current procedure, and it performs a search within the selection.

Creating the Macro

Complete the following steps:

1. Enter the macro code below into a file called `procsearch.e`.
2. To load the module, from the main menu, select **Macro > Load Module**.
3. Bind the command **proc_search** to a key. To use the macro, press the appropriate key.
4. In the **Search string** text box, enter the text to search for, and in the **Options** text box, enter the search options (see [Command Line Search Options](#)).

Contents of procsearch.e:

```

#include 'slick.sh'

_command int proc_search(...) name_info('VSARG2_READ_ONLY|
                                         VSARG2_REQUIRES_EDITORCTL|
                                         VSARG2_MARK)
{
    // Save the original cursor position to restore later
    typeless original_position;
    save_pos(original_position);

    // Prompt the user for a search string, and search options
    _str result = show('-modal _textbox_form',
                      'Search Function', // Dialog box caption
                      TB_RETRIEVE_INIT, // Flags
                      '',               // Use default text box width
                      '',               // Help item
                      '',               // Button list
                      'procsearch',     // Retrieve name
                      'Search string:', // First prompt
                      'Options:ixcs');  // Second prompt and default
    if ( result==' ' ) {
        // If the user clicked the cancel button, just return
        return(COMMAND_CANCELLED_RC);
    }

    // The results from the text boxes
    _str search_string=_param1;
    _str search_options=_param2;

    int status=select_proc(); // Select the current proc
    if ( status ) {
        // In rare cases select_proc can fail if a procedure is too complex.
        // If select_proc failed, show an error messages, return the cursor to the
        // original position, and return.
        _message_box(nls("select_proc failed"));
        restore_pos(original_position);
        message(get_message(status));
        return(status);
    }
    lock_selection(); // Lock the selection

    begin_select(); // Move the cursor to the
    // beginning of the selection

    status=find(search_string,'m'+search_options); // Find the text that the
    // user specified using the
    // options specified. We prepend
    // the 'm' option since we
    // know we are searching in
    // a selection.

```


Contents of `procsearch.e` continued:

```

if ( status ) {
    // If the search string was not found, deselect and return the cursor to
    // the original position.
    deselect();
    restore_pos(original_position);
}

// Just return the status. This will leave the proc selected so that
// find_next works
return(status);
}

```

Analyzing the Macro

The **save_pos()** call at the beginning of the macro saves the current cursor position information. This function places the cursor in its original position if necessary.

The **show()** function launches a dialog box. In this case, the **show()** function launches a general purpose dialog box named **_text box_form**. The dialog box **_text box_form** prompts the user for one or more strings. After the first argument, the remaining arguments to **show()** pass to the **on_create** dialog box. In this case, there are several arguments.

The second argument to **show()** is the caption for the **on_create** dialog box.

The next argument is a set of flags. In this case, the only flag specified is `TB_RETRIEVE_INIT`. The `TB_RETRIEVE_INIT` flag tells the dialog box to initialize itself by retrieving the last values filled in for this dialog box.

Use the next three arguments to specify text box width, help, and a button list. These particular arguments are unused in this example, which is why they are shown here as `"`.

The retrieve name is a unique name used to retrieve the values that were previously filled in for this dialog box. Any remaining arguments are interpreted as prompts for the user. Default values can be given by specifying the prompt as **prompt:defaultvalue**. The first prompt is the search string, and the second is for search options. The options have default **ixcs**, meaning case-insensitive, and exclude comments and strings. See the following section for a list of command line search options.

After the call to **show**, verify that the result is `"`. If so, then the user clicked the Cancel button, so we return. Otherwise, SlickEdit® must obtain the values that the user provides. These values are returned in global variables **_param1.._param N**. In this case, our search string is returned in **_param1**, and the search options are in **_param2**. These are saved in local variables.

SlickEdit calls **select_proc** to select the current function. If **select_proc** returns a non-zero status, then it failed, so it is returned. In rare cases, **select_proc** can fail if a function is too long, or has preprocessing that keeps it from correctly identifying the end of the function.

Next, **lock_selection()** is called, and then **begin_select()** is called to move to the beginning of the selection.

Now, we can call **find()** with the search string and the search options from the user. Insert **m** at the beginning of the options string to specify search only in the selection.

Finally, check the status from **find**. If the string is not found, clear the function and restore the original cursor position.

Command Line Search Options

Command line search options include the characters listed in the table below.

Option	Description
+	(Default) Forward search.
-	Reverse search.
<	(Default) Place cursor at beginning of string found.
>	Place cursor after end of string found.
E	(Default) Case-sensitive search.
I	Case-insensitive search.
M	Search within visible mark.
H	Find text in hidden lines.
R	Search for SlickEdit® regular expression.
U	Interpret string as a UNIX regular expression.
B	Interpret string as a Brief regular expression.
N	(Default) Do not interpret search string as a regular search string.
@	No error message.
W	Limits search to words such as variable names.
,	Delimiter to separate ambiguous options.

Reading and Modifying Buffers

Slick-C® includes the Slick-C API. The API covers many actions normally performed in a code editor, including navigating and modifying buffers.

The following example focuses on one particular category of the API, those functions that allow you to programmatically traverse and modify buffers. These powerful functions enable you to take tasks that you can do manually, and create a macro to perform the same tasks in seconds.

Functions for Reading and Modifying Buffers

The table below contains functions for reading and modifying buffers.

Function	Action
<code>_str cur_word(int &start_col [, _str from_cursor])</code>	Gets the current word at cursor.
<code>int delete_line()</code>	Deletes the current line.
<code>void _delete_text(int len)</code>	Delete len bytes starting from the cursor position.

Function	Action
<code>void get_line(_str &line)</code>	Retrieves current line.
<code>_str get_text([int count [,int seek_pos]])</code>	Gets a stream of text starting at current line.
<code>void keyin(_str string)</code>	Inserts string of characters as if typed from the keyboard.
<code>void insert_line(_str line)</code>	Inserts line after current line.
<code>void _insert_text(_str string)</code>	Inserts string at cursor position.
<code>void replace_line(_str line)</code>	Replaces current line.

Common Functions for Navigating Buffers

The table below contains functions that can be used for navigating buffers.

Function	Action
<code>int up([int num])</code>	Moves cursor up num lines, or one line if no value passed in.
<code>int down([int num])</code>	Moves cursor down num lines, or one line if no value passed in.
<code>void left()</code>	Moves cursor one position to the left.
<code>void right()</code>	Moves cursor one position to the right.
<code>void top()</code>	Places cursor at first line and first column of buffer.
<code>void bottom()</code>	Places cursor at end of last line of buffer.
<code>void _begin_line()</code>	Places cursor at the beginning of the current line.
<code>void _end_line()</code>	Places cursor after the end of the current line.

Escape Backslashes Example

Escape backslashes if, for every slash in a directory name, you actually need two for the compiler to handle the directory name or string properly.

Example:

```
_command escape_slash(){
    _str myLine;
    get_line(myLine); // Set string szLine to the current line
    myLine = stranslate(myLine, "\\\\", "\\"); // Replace slash with double slashes
    replace_line(myLine); // Replace the line in the buffer
}
```

The above command accepts the following line of code:

```
myDirectory = "C:\Data\Corporate\Internal";
```

and replaces it with:

```
myDirectory = "C:\\Data\\Corporate\\Internal";
```

Comment Out Debug Print Lines Example

Print or debug statements can be used to debug. These statements need to have supporting comments or they must be deleted. The following example shows a simple function that loops through your entire file and contains supporting comments for all of the lines that have a **printf** statement:

```
_command comment_printf(){
    _str curLine;
    top(); // Go to top of buffer
    up(); // Get to the top line
    while ( !down() ) {                // Loop until end of file
        get_line( curLine );           // Get the current line
        if( pos( "printf", curLine ) ){ // Search for a printf

            _begin_line();              // If printf exists, move cursor to the
                                      // first column

            _insert_text( // );         // add a comment
        }
    }
}
```

The function uses many of the buffer modifications and navigation macros. It loops line-by-line through the file, checks for a string, and adds a comment when necessary. Modify this macro to meet your needs. For example, if you want the lines deleted instead of commented, replace the **_insert_text()** call with **delete_line()**. Also, check to see if the comment characters already exist before you add the comment text. Instead of calling **_begin_line**, call **begin_line_text_toggle**—this places the cursor at the first non-space character of a line. Next, check if you are in a comment by calling **_in_comment()**.

Turning on Line Numbers for all Files

Every time you select a menu, click a button, or enter a key, a Slick-C® macro is called to perform an action. More than half of the code in SlickEdit products is written in Slick-C and this source is provided to you when you install, so you can tweak the product or use the Slick-C source as an example to help write your own macros. By default, the Slick-C source is located in the `macros` subdirectory of your SlickEdit® installation folder.

To make a macro change, or to recycle existing code, you need to know how to find a name to a particular command and how to find its location in the source code. This tutorial will walk you through these steps.

Write a New Macro to Find a Command Name

The SlickEdit® code editor includes a line number toggle option to turn line numbers on and off for each edit window. This is found on the main menu by selecting **View > Line Numbers**. By default, all files are displayed without line numbers. When you enable them, they are enabled throughout sessions, until you disable them. The SlickEdit code editor has the option to turn on line numbers for specific file types or extensions. Line numbers can be enabled for each extension by choosing **Tools > Options > File Extension Setup**. Select the **General** tab, then check **Display line numbers**.

Macro Description and Steps Required

Complete the following steps to write a macro to enable the line numbers for all files:

1. Find the macro that is associated by selecting **View > Line Numbers** from the main menu. To find a macro associated with a menu, view the menu in the Menu Editor dialog.
2. Close any open files.

3. From the main menu, select **Macro > Menus**. The dialog box contains a list of all menus. To view the main menu, select **_mdi_menu** and click **Open**. The Menu Editor dialog is displayed.
4. Navigate to **View > Line Numbers**. When you select **Line Numbers**, certain fields in the dialog box are populated. The **Command** field is populated with the Slick-C® command that is invoked when this menu item is selected. In this case, the command is **view-line-numbers-toggle**. Every time that you select **Line Numbers** from the menu, **view-line-numbers-toggle** is called.
5. View source code of **view-line-numbers-toggle**. To jump to the source code, go to **Macro > Go to Slick-C Definition**. Use this method to view any Slick-C command.
6. Start typing **view**. A drop-down menu is displayed. The command **view_line_numbers_toggle()** is in the list. Select this and click **OK**. By viewing the source, it is a simple if on then off, else on algorithm, using bitwise logic. Use the following to enable viewing line numbers:
`p_LCBufFlags|=VSLCBUFFLAG_LINENUMBERS`
7. Create a new macro to always turn on line numbers. The following information creates a new Slick-C file named `DisplayAllLines.e` that contains the code below. This file contains the function **_buffer_add_ViewLineNumbers**.

Example:

```
#include "slick.sh"

void _buffer_add_ViewLineNumbers()
{
    p_LCBufFlags|=VSLCBUFFLAG_LINENUMBERS;
}
```

Any Slick-C macro that starts with **_buffer_add_** is called when a new edit window is displayed. To enable the numbers for every file, use the logic from Step 6 in this function. This function does not start with an **_command**. This is because an internal function to be invoked during the open event of an edit window is being invoked. Commands are functions defined with the **_command** and can be bound to a key, menu, button, or invoked from the command line.

8. Load the macro. From the main menu, select **Macro > Load Module > ViewLineNumbers.e**. To ensure the module was loaded properly, the message "Modules loaded" must be displayed at the bottom left of the status bar. If an error message is displayed, then the module did not load, and the change did not take effect. Correct the error, and load the module again.

To unload `ViewLineNumbers.e` to remove the functionality that turns on line numbers for all files, from the main menu select **Macro > Unload Module**. Select `ViewLineNumbers.ex` from the list and click **OK**. The list shows a `.ex` extension on the module instead of an `.e` because you are actually compiling the source file into a binary file (`.ex`) and loading it, not the actual source code.

Every new file opened has line numbers. If any files are open at the beginning, close and reopen them and they will all have line numbers.

Counting Lines of Code

The number of lines of code that you have in your project, workspace, or file can be determined using the following macro. This information is used to measure performance analysis tasks.

Macro Description and Steps Required

This macro uses the current workspace and loops through all projects (a workspace can contain many projects) and files within the projects. It then displays a report in an editor window. A number of steps are required to accurately gather the required information. The following information describes the macro.

Gather Workspace, Project, and File Information

Get a list of all projects and files in the workspace. **_GetWorkspaceFiles()** (Line 88) gets the list of all projects in a workspace and places the list in a temporary buffer. The loop following (Lines 93-95), parses through the buffer and stores the information in a temporary array for later reporting. This array, defined in Line 67, is a three-dimensional array to store multiple projects, and multiple files per project.

Loop through each project, starting at Line 98, and fill the array with all file names for each project.

GetProjectFiles() does this by placing the list in a temporary buffer. Grab the names from the buffer and put them in the array (Lines 109-124).

Count

For each project, open up a temporary buffer for each file in the project. Think of it as an invisible buffer where you can move the cursor programmatically to check whether it is in a comment.

- **_open_temp_view** (Line 139) opens it.
- **up()** and **top()** (Line 158) places the cursor at the top to start.
- **down()** (Line 161) will move the cursor down one line at a time.

Loop through the file to read one line at a time, as mentioned above (Lines 161-202). This validates whether the current line is in a comment (Line 171), and if not, it increments the counter. If the current line is in a comment, the next step is to jump to the end of the comment or comment block (Line 168). Another check is made to see if the current line is in a comment and count it if it is not a comment.

Report

All of the information is now stored in an array, so the next task is to generate a report and loop thru the array to display the results. This is done in Lines 220-263.

The **displayResultsInBuffer** flag can be changed to false to only display the total lines in the entire workspace.

Now that you understand the macro, the next steps are to load and run it.

Load

You can obtain the `linecount.e` file from the SlickEdit Web site at www.slickedit.com: from the Web site's main menu, click **Support > Product Documentation > Slick-C Macro Programming Language**.

To load `linecount.e`, first save it to your local hard drive, then from the SlickEdit main menu, click **Macros > Load Module > linecount.e**. Click **Open**.

For more information on `linecount.e`, see [Counting Lines of Code](#).

Run from the Command Line

The command line is located at the bottom left of the code editor window.

1. Go to the command line by pressing **Esc** or by clicking on the space at the bottom left.
2. Type **linecount** and press **Enter**.
3. Associate the command with a menu. The Menu Editor dialog allows you to add items to all menus.
4. Select **Macro > Menus**, then select a menu. For example, the right-click menu in the editor is **_ext_menu_default**.
5. In the Menu Editor dialog, select **Insert** to create a new menu item.
6. Type a new **Caption**, set the **Command** to **linecount**.

Assigning a Key Binding Shortcut to the Command

1. First, find a key sequence that is not used. Do not bind keys that are bound. From the main menu, select **Help > What is Key**, and type a sequence that you want to use. If it is not defined, a message is displayed stating so, and it can be used.
2. From the main menu, select **Tools > Options > Key Bindings**.
3. Type **linecount** in the command field.
4. Click **Add Key or Mouse Click** and type the key sequence to bind.
5. Click **Bind**, then **Done**.
6. Load and run this macro, to determine the number of lines of code that are written.

Event Names

Event names are used as arguments to the **def** primitive. Event names are also used when comparing events returned by the **get_event** or **test_event** built-in functions or when defining an event handler function. An event name is a string literal of a length of one or more. An event name string of a length one specifies an ASCII character. To keep the macro source compatible, some event names do not have to be enclosed in quotes as long as the `_` (underscore) character is used instead of the `-` (dash) character. The following sections list the acceptable constants.

ASCII Characters

Acceptable ASCII characters are `\0..255`. Backslash is used for non-displayable keys.

You may also quote displayable characters such as `"a"` or `"4"`. The keys `\1..29` are also represented by the keys `C-A`, `C-B..C-Z`, `C-[C-\,C-]`, `C-^`, and `C-_`. The ASCII keys `\129..255` are the same key binding as `\128`.

Function Keys

Acceptable function keys are `F1`, `F2`, and `F12`.

Extended Keys

Acceptable extended keys are the following:

Enter	End	Left	Pad_Star
Tab	PageUp	Right	Pad_Plus
Escape	PageDown	Up	Pad_Minus
Backspace	Delete	Down	Pad_5
Home	Insert	Pad_Slash	

Mouse Events

The following are acceptable mouse events:

LButton-Down	RButton-double-click	MButton-triple-click
LButton-double-click	RButton-triple-click	LButton-Up
LButton-triple-click	MButton-Down	RButton-Up
RButton-Down	MButton-double-click	MButton-Up

On Events

The following are the **On** events:

On-Change	On-Got-Focus	On-Hsb-Top	On-Vsb-Bottom
On-Change2	On-Hsb-Bottom	On-Load	On-Vsb-Line-Down
On-Close	On-Hsb-Line-Down	On-Load-Focus	On-Vsb-Line-Up
On-Create	On-Hsb-Line-Up	On-Resize	On-Vsb-Page-Down
On-Create2	On-Hsb-Page-Down	On-Scroll	On-Vsb-Page-Up
On-Destroy	On-Hsb-Page-Up	On-Sscroll-Lock	On-Vsb-Thumb-Pos
On-Destroy2	On-Hsb-Thumb-Pos	On-Spin-Down	On-Vsb-Thumb-Track
On-Drop-Down	On-Hsb-Thumb-Track	On-Spin-Up	On-Vsb-Top

Miscellaneous Keys

Acceptable miscellaneous keys are: C-A-Enter, C-A-Tab, C-A-Esc, C-A-Backspace, C-Prts, C-Ctrl, and A-Alt.

Miscellaneous Events

On-Select is an acceptable miscellaneous event.

Key Name Examples

The following are examples of uses for key names in the Slick-C™ language:

```
def "A-x"=safe_exit; // Note that "A-a" is different than "A-A" which
                    // requires the Alt and Shift keys to be pressed.
def "A-?"=help;
def "C-X" "b"=list_buffers;
def \0 - \255= nothing;
ctlcombol.on_change()
{
}
ctlcombol."c-s-a"() // Define event handler for Ctrl+Shift+A
{
}
ctlcombol."a"-"z", "A-"Z"() // Define event handler for characters A-Z
                          // upper and lowercase
{
}
void p()
{
    for (;;) {
        key=get_event();
if (key==name2event("ESC") break;
        if (key==name2event("UP")) {
            ...
        } else if (key==name2event("DOWN") ) {
            ...
        }
    }
}
```


Differences Between Slick-C® and C++

Structures

- Space for structure member variables is allocated when you access the member.
- Structure data is not continuous. This is obvious for string, array, and hash table member variables that contain variable size data. However, even other types are sometimes stored elsewhere.
- There is not a **sizeof** function that tells you the size of a structure in bytes.

Arrays

- Space for array elements is allocated when you index into the array.
- You cannot use pointer variables to traverse array elements.
- You cannot limit the number of elements that the array may contain.
- Specifying an array variable without the `[]` operator does not return a pointer to the first element. Instead it refers to the entire array. This allows you to copy one array to another or define a function that returns a copy of an array.
- There is not a **sizeof** function that tells you the size of the array in bytes. There is a **_length** method that tells you the number of elements in the array.

Example:

```
struct PHONERECORD {
    _str name;
    _str PhoneNumber;
};

defmain()
{
    PHONERECORD list[]; // No size limit is allowed here.

    //Allocate space for 0 index and name member
    list[0].name=Joe;
    // Allocate space for PhoneNumber member.
    list[0].PhoneNumber=555-1234;

    PHONERECORD list2[];
    list2=list; // Copy the entire array into list2

    t=list2; // Now copy the entire array into a container variable.
}
```

Hash Tables

Slick-C® provides a `[:]` hash table operator that is similar to the array operator `[]` except that you index hash tables with a string type.

Assignment Statement

Assignment statements in Slick-C® are not as shallow as C++. Array, hash table, and structure types are recursively traversed. Pointers are not traversed.

Example:

```
struct {
    int a[];
} s1,s2;
s1.a[0]=1;
s2=s1; // Copy stucture and all elements of array.
```

Comparison Operator

The **==** and **!=** operators support comparing container types, arrays, hash tables, and structures. Complex types are traversed recursively, like the assignment statement. Strings within an array, hash table, or struct must match exactly (spaces matter).

Preprocessing

Preprocessing expressions can use string and floating point expressions.

switch Statement

The **switch** statement supports string expressions and integer expressions.

Labeled Loops

The **break** and **continue** statements accept an optional label parameter so that you can break a specific loop (like Java).

Example:

```
outerlabel:
    for (;;) {
        for (;;) {
            if () break outerlabel;
            if () continue outerlabel;
        }
    }
```

Variable Argument Functions

An **arg** function allows you to define functions that accept a variable number of arguments. The **arg** function can be used on the left side of an assignment statement.

Example:

```
defmain()
{
    p(Param1,2,x);
}
void p()
{
    messageNwait(Called with arg() arguments);
    for (i=1;i<=arg();++i) {
        messageNwait(arg(i)=arg(i));
    }
    //All undeclared variable parameters are passed by reference.
    //so when a variable is passed, we can change the contents of
    //the callers variable.
    arg(3)=New value for x;
}
```

Built-in Graphics Primitives

You can define dialog box resources and menu resources. There are primitives for defining event handlers for dialog boxes and declaring control types. This allows the Slick-C® linker to detect a reference to a control that does not exist on a dialog box before you execute the code.

Clipboard Inheritance®

Clipboard Inheritance provides inheritance specifically for dialog boxes. This feature enables the copying of parts of existing dialog boxes to the clipboard and pasting them elsewhere, and the original code still runs. New code can be attached to the new controls without affecting the original controls, and to affect both instances of the controls (inheritance). Creating inheritance for parts of dialog boxes is very natural because the Slick-C® language has been designed for this feature. See [Clipboard Inheritance®](#) for more information.

End of Statement Semicolon

Slick-C® assumes that the end of line is a semicolon except under a few conditions. Expressions may extend across line boundaries if the line ends in a binary operator or if the line ends with a backslash, and expressions in parentheses may extend across line boundaries.

Type Checking

Type checking in Slick-C® is identical to C++ except for the following:

- The **typeless** type is compatible with ALL other types.
- String constants are automatically converted to numeric types where necessary.
- Integer types are automatically converted to string types.
- Functions do not require prototypes. However, when a prototype is given, strict type checking is enforced like you would expect. A **#pragma** option to require prototypes will eventually be added.

Capability not Supported by Slick-C®

- You cannot define your own classes, methods, or inheritance. The important **class**, **public**, **private**, and **new** keywords are not supported. Classes in Slick-C will not require a delete to free objects.
- Only one syntax is currently supported for making a call with a pointer to a function variable. The `pfn(p1, p2,)` syntax is not supported. This limitation is necessary for container variables because the compiler does not know the type of the variable.
- **char** and **short** types are not available.
- Template classes are not supported. Container variables are sometimes a more powerful mechanism for accomplishing much of what is done with template classes. However, users will likely want the speed and additional type checking of template classes, and many programmers are already used to template classes.
- Function overloading is not supported.
- **enum** is not supported.
- Slick-C only supports the less ambiguous C-style type casting.
- Because Slick-C does not allow low level manipulation of memory, you cannot do things like type cast an **int *** to a **long ***.
- There are no character constants defined using single quote characters. Slick-C currently allows the use of single quotes to define strings. Single quoted strings are much more readable for file names or regular expressions that require the use of backslashes.
- **goto** is not supported. (Slick-C supports labeled loops.)

Index

A

- Active form 85
- Active object 85
- Adding completion to command 45
- Aligning Controls 72
- AND operator 30
- Anonymous Unions 34
- arg built-in 44
- Argument Declarations 43
- Array length method 21
- Array variables 25
- Arrays 25
- Assignment Operator 37
- Associative array variables 25

B

- Batch Macro 14
- bigfloat 33
- bitmaps (adding to image control) 102
- bitmaps (adding to list box) 97
- Bitwise and 30
- Bitwise or 30
- Bitwise xor 30
- boolean 33, 37
- break 38
- Built-in Function 48
- Built-ins 49

C

- Cancel button (adding) 75
- case 41
- case statement 41
- casting types 34
- Check Box control 92
- Clipboard Inheritance 81
- Closing a dialog box 75
- Combo Box control 92
- Command Button control 91
- commands
 - defining with Slick-C 44
- Commands, Built-ins, Defs (Differences) 49
- comment styles (C++ language constructs) 18
- Compiling macros 63
- completion
 - Adding to command 45
- Concatenation 30
- constants (defining) 20
- constants (numeric) 19
- continue 38

D

- Data Set Utilities dialog box 107
- def 57
- def primitive 57
- Default Arguments 44
- defeventtab 57
- defeventtab primitive 57
- defining constants 20
- defining event tables 59

- defining functions 43
- defining keys 57
- defining procedures 43
- definit 61
- defload 61
- defmain 49
- Defs 49
- Dialog Box Retrieval 104
- dialog boxes (forms for macros) 115
- Dialog Editor 71
- Dialog Editor Properties dialog box 71
- Directory List Box control 99
- double 33
- Drive List control 98

E

- Editor Control 91
- environment variables
 - VSLICKINCLUDE 53
 - VSLICKPATH 53
 - VST 53
- Event driven dialog box 12
- Event driven dialog (event tables) 59
- Event tables 57
- events and event tables 57
- Executing programs from macro 133
- expressions (unary operators in) 29

F

- File List Box control 99
- Filtering marked text 127
- filter_selection procedure 127
- Find Slick-C Error menu item 65
- find_error command 65
- find_proc command 65
- Floating point numbers 19
- Frame control 91
- Function Pointers 25
- Function Prototypes 48
- Functions 48
- functions (defining) 43

G

- Gauge control 100
- get_string procedure 137

H

- Hash Tables 25
- Header Files, Including 53
- Hex character code 19
- hexadecimal numbers 19
- Hscroll Bar control 98

I

- If Statement 37
- Image control 101
- Including macro header files 53
- Inheritance 59, 81, 84
- Inherited Code Found dialog box 74
- initializing macro modules 61

Instances 85

L

Label control 89
Language Constructs 17
List Box control 95
load command 57, 63
Loading macros 63
Loops 38

M

Mac OS X
 writing selection filters 127
macro prompting 137
Mathematical Operators 29
Menu Item Alias dialog box 108
Menu Macro Programming 107
menus (changing or adding) 107
Methods 87
Modal and modeless dialog boxes 79
Module Initializations 61
Moving Controls 72
Multiple Instances 85

N

name_info Attributes 45
no_code_swapping 141
null 21
Numeric constants 19

O

Object Instances 85
OK button (adding) 75
Open Form dialog 74
operator 25
Operators 29
OR operator 30

P

parse 39
Picture Box control 100
pictures (adding to image control) 102
pictures (adding to list box) 97
Pointer Variables 25
Pointers to Functions 25
procedures (defining) 43
profiling (Slick-C) 65
prompt procedure 137
prompting from macros 137
Properties dialog box 71
prototypes 48
p_window_id 85

R

Radio Button Control 91
rc variable 67
return statement 43, 44
Run-time error finding 65

S

Scoping and Declaring Variables 33
Selecting Controls 72
Setting Properties 72
Shelling from macro 133
Simple Variables 33
Sizing Controls 72
Slick-C Batch Files, Writing 49
Slick-C Profiler dialog 65
Slick-C profiling 65
Spin control 89
st command 63
string concatenation 30
String literals 18
string operators 27
struct 23
switch 41

T

Text Box control 90
Type Casting 34
typeless 27
Typeless Container Variables, How They Work 34
typeless variable declaration 33
TypeName 33

U

union 34

V

Variable Initializations, Details 34
Vscroll Bar control 97
VSLICKINCLUDE 53
VSLICKPATH 53
vslick.sta 141
VST 53
vstw program 63
vstw.exe program 63

W

wid_expression 86

X

xcom command 50

Symbols

#endregion 53
#pragma 51
#region 53

Numerics

_command 44
_control 86
_inherit 59
_str center 117