S-Lang Library C Programmer's Guide (v2.3.0)

John E. Davis <www.jedsoft.org>

Sep 14, 2014

Preface

S-Lang is an interpreted language that was designed from the start to be easily embedded into a program to provide it with a powerful extension language. Examples of programs that use **S-Lang** as an extension language include the **jed** text editor and the **slrn** newsreader. Although **S-Lang** does not exist as a separate application, it is distributed with a quite capable program called **slsh** ("slang-shell") that embeds the interpreter and allows one to execute **S-Lang** scripts, or simply experiment with **S-Lang** at an interactive prompt. Many of the the examples in this document are presented in the context of one of the above applications.

S-Lang is also a programmer's library that permits a programmer to develop sophisticated platform-independent software. In addition to providing the **S-Lang** interpreter, the library provides facilities for screen management, keymaps, low-level terminal I/O, etc.

A Brief History of S-Lang

I first began working on **S-Lang** sometime during the fall of 1992. At that time I was writing a text editor (**jed**), which I wanted to endow with a macro language. It occurred to me that an application-independent language that could be embedded into the editor would prove more useful because I could envision embedding it into other programs. As a result, **S-Lang** was born.

S-Lang was originally a stack language that supported a postscript-like syntax. For that reason, I named it **S-Lang**, where the *S* was supposed to emphasize its stack-based nature. About a year later, I began to work on a preparser that would allow one unfamiliar with stack based languages to make use of a more traditional infix syntax. Currently, the syntax of the language resembles C, nevertheless some postscript-like features still remain, e.g., the '%' character is still used as a comment delimiter.

Acknowledgements

Since I first released **S-Lang**, I have received a lot feedback about the library and the language from many people. This has given me the opportunity and pleasure to interact with a number of people to make the library portable and easy to use. In particular, I would like to thank the following individuals:

Luchesar Ionkov for his comments and criticisms of the syntax of the language. He was the person who made me realize that the low-level byte-code engine should be totally type-independent. He also improved the tokenizer and preparser and impressed upon me that the language needed a grammar.

Mark Olesen for his many patches to various aspects of the library and his support on AIX. He also contributed a lot to the pre-processing (`SLprep`) routines.

John Burnell for the OS/2 port of the video and keyboard routines. He also made value suggestions regarding the interpreter interface.

Darrel Hankerson for cleaning up and unifying some of the code and the makefiles.

Dominik Wujastyk who was always willing to test new releases of the library.

Michael Elkins for his work on the curses emulation.

Hunter Goatley, Andy Harper, Martin P.J. Zinser, and Jouk Jansen for their VMS support.

Dave Sims and Chin Huang for Windows 95 and Windows NT support, and Dino Sangoi for the Windows DLL support.

I am also grateful to many other people who send in bug-reports, bug-fixes, little enhancements, and suggestions, and so on. Without such community involvement, **S-Lang** would not be as well-tested and stable as it is. Finally, I would like to thank my wife for her support and understanding while I spent long weekend hours developing the library.

Contents

Chapter 1

Introduction

S-Lang is a C programmer's library that includes routines for the rapid development of sophisticated, user friendly, multi-platform applications. The **S-Lang** library includes the following:

- Low level tty input routines for reading single characters at a time.
- Keymap routines for defining keys and manipulating multiple keymaps.
- A high-level keyprocessing interface (`SLkp`) for handling function and arrow keys.
- High level screen management routines for manipulating both monochrome and color terminals. These routines are *very* efficient. (`SLsmg`)
- Low level terminal-independent routines for manipulating the display of a terminal. (`SLtt`)
- Routines for reading single line input with line editing and recall capabilities. (`SLrline`)
- Searching functions: both ordinary searches and regular expression searches. (`SLsearch`)
- An embedded stack-based language interpreter with a C-like syntax.

The library is currently available for OS/2, MSDOS, Unix, and VMS systems. For the most part, the interface to library routines has been implemented in such a way that it appears to be platform independent from the point of view of the application. In addition, care has been taken to ensure that the routines are "independent" of one another as much as possible. For example, although the keymap routines require keyboard input, they are not tied to **S-Lang**'s keyboard input routines—one can use a different keyboard `getkey` routine if one desires. This also means that linking to only part of the **S-Lang** library does not pull the whole library into the application. Thus, **S-Lang** applications tend to be relatively small in comparison to programs that use libraries with similar capabilities.

Chapter 2

Error Handling

Many of the **S-Lang** functions return 0 upon success or -1 to signify failure. Other functions may return `NULL` to indicate failure. In addition, upon failure, many will set the error state of the library to a value that indicates the nature of the error. The value of this state may be queried via the `SLang_get_error` function. This function will return 0 to indicate that there is no error, or a non-zero value such as one of the following constants:

```
SL_Any_Error                      SL_Index_Error
SL_OS_Error                       SL_Parse_Error
SL_Malloc_Error                   SL_Syntax_Error
SL_IO_Error                       SL_DuplicateDefinition_Error
SL_Write_Error                    SL_UndefinedName_Error
SL_Read_Error                     SL_Usage_Error
SL_Open_Error                     SL_Application_Error
SL_RunTime_Error                  SL_Internal_Error
SL_InvalidParm_Error              SL_NotImplemented_Error
SL_TypeMismatch_Error             SL_LimitExceeded_Error
SL_UserBreak_Error                SL_Forbidden_Error
SL_Stack_Error                    SL_Math_Error
SL_StackOverflow_Error            SL_DivideByZero_Error
SL_StackUnderflow_Error           SL_ArithOverflow_Error
SL_ReadOnly_Error                 SL_ArithUnderflow_Error
SL_VariableUninitialized_Error    SL_Domain_Error
SL_NumArgs_Error                  SL_Data_Error
SL_Unknown_Error                  SL_Unicode_Error
SL_Import_Error                   SL_InvalidUTF8_Error
```

For example, if a function tries to allocate memory but fails, then `SLang_get_error` will return `SL_Malloc_Error`.

If the application makes use of the interpreter, then it is important that application-specific functions called from the interpreter set the error state of the library in order for exception handling to work. This may be accomplished using the `SLang_set_error` function, e.g.,

```
if (NULL == (fp = fopen (file, "r")))
  SLang_set_error (SL_Open_Error);
```

Often it is desirable to give error message that contains more information about the error. The `SLang_verror` function may be used for this purpose:

```
    if (NULL == (fp = fopen (file, "r")))
      SLang_verror (SL_Open_Error, "Failed to open %s: errno=%d",
                    file, errno);
```

By default, `SLang_verror` will write the error message to `stderr`. For applications that make use of the **SLsmg** routines it is probably better for the error message to be printed to a specific area of the display. The `SLang_Error_Hook` variable may be used to redirect error messages to an application defined function, e.g.,

```
    static void write_error (char *err)
    {
       SLsmg_gotorc (0, 0);
       SLsmg_set_color (ERROR_COLOR);
       SLsmg_write_string (err);
    }
    int main (int argc, char **argv)
    {
       /* Redirect error messages to write_error */
       SLang_Error_Hook = write_error;
             .
             .
    }
```

Under extremely rare circumstances the library will call the C `exit` function causing the application to exit. This will happen if the `SLtt_get_terminfo` is called but the terminal is not sufficiently powerful. If this behavior is undesirable, then another function exists (`SLtt_initialize`) that returns an error code. The other times the library will exit are when the interpreter is called upon to do something but has not been properly initialized by the application. Such a condition is regarded as misuse of the libary and should be caught by routine testing of the application during development. In any case, when the library does call the exit function, it will call an application-defined exit hook specified by the SLang_Exit_Error_Hook variable:

```
    static int exit_error_hook (char *fmt, va_list ap)
    {
       fprintf (stderr, "Fatal Error.  Reason:");
       vfprintf (stderr, fmt, va_list);
    }
    int main (int argc, char **argv)
    {
       SLang_Exit_Error_Hook = exit_error_hook;
         .
         .
    }
```

The idea is that the hook can be used to perform some cleanup, free resources, and other tasks that the application needs to do for a clean exit.

Chapter 3

Unicode Support

S-Lang has native support for the UTF-8 encoding of unicode in a number of its interfaces including the the **SLsmg** screen mangement interface as well as the interpreter. UTF-8 is a variable length multibyte encoding where unicode characters are represented by one to six bytes. A technical description of the UTF-encoding is beyond the scope of this document, and as such the reader is advised to look elsewhere for a more detailed specification of the encoding.

By default, the library's handling of UTF-8 is turned off. It may be enabled by a call to the `SLutf8_enable` function:

```
int SLutf8_enable (int mode)
```

If the value of `mode` is 1, then the library will be put in UTF-8 mode. If the value of `mode` is 0, then the library will be initialized with UTF-8 support disabled. If the value is -1, then the mode will determined through an OS-dependent manner, e.g., for Unix, the standard locale mechanism will be used. The return value of this function will be 1 if UTF-8 support was activated, or 0 if not.

The above function determines the UTF-8 state of the library as a whole. For some purposes it may be desirable to have more fine-grained control of the UTF-8 support. For example, one might be using the **jed** editor to view a UTF-8 encoded file but the terminal associated with the editor may not support UTF-8. In such a case, one would want the **SLsmg** interface to be in UTF-8 mode but lower-level **SLtt** interface to not be in UTF-8 mode. Hence, the following activation functions are also provided:

```
int SLsmg_utf8_enable (int mode);
int SLtt_utf8_enable (int mode);
int SLinterp_utf8_enable (int mode);
```

Note that once one of these interface specific functions has been called, any further calls to the umbrella function `SLutf8_enable` will have no effect on that interface. For this reason, it is best to call `SLutf8_enable` first before the calling one of the interface-specific functions.

Until support for Unicode is more widespread among users, it is expected that most users will still be using a national character set such as ASCII or iso-8869-1. For example, iso-8869-1 is a very widespread character set used on Usenet. As a result, applications will still have to provide support for such character sets. Unfortunately there appears to be no best way to do this.

For the most part, the UTF-8 support should be largely transparent to the user. For example, the interpreter treats all multibyte characters as a single character which means that the user does not have to be concerned about the internal representation of a character. Rather one must keep in mind the distinction between a character and a byte.

Chapter 4

Interpreter Interface

The **S-Lang** library provides an interpreter that when embedded into an application, makes the application extensible. Examples of programs that embed the interpreter include the **jed** editor and the **slrn** newsreader.

Embedding the interpreter is easy. The hard part is to decide what application specific built-in or intrinsic functions should be provided by the application. The **S-Lang** library provides some predefined intrinsic functions, such as string processing functions, and simple file input-output routines. However, the basic philosophy behind the interpreter is that it is not a standalone program and it derives much of its power from the application that embeds it.

4.1 Embedding the Interpreter

Only one function needs to be called to embed the **S-Lang** interpreter into an application: `SLang_init_slang`. This function initializes the interpreter's data structures and adds some intrinsic functions:

```
 if (-1 == SLang_init_slang ())
   exit (EXIT_FAILURE);
```

This function does not provide file input output intrinsic nor does it provide mathematical functions. To make these as well as some posix system calls available use

```
if ((-1 == SLang_init_slang ())    /* basic interpreter functions */
    || (-1 == SLang_init_slmath ()) /* sin, cos, etc... */
    || (-1 == SLang_init_array ()) /* sum, min, max, transpose... */
    || (-1 == SLang_init_stdio ()) /* stdio file I/O */
    || (-1 == SLang_init_ospath ()) /* path_concat, etc... */
    || (-1 == SLang_init_posix_dir ()) /* mkdir, stat, etc. */
    || (-1 == SLang_init_posix_process ()) /* getpid, umask, etc. */
    || (-1 == SLang_init_posix_io ()) /* open, close, read, ... */
    || (-1 == SLang_init_signal ()) /* signal, alarm, ... */
   )
  exit (EXIT_FAILURE);
```

If you intend to enable all intrinsic functions, then it is simpler to initialize the interpreter via

```
    if (-1 == SLang_init_all ())
      exit (EXIT_FAILURE);
```

See the **S-Lang Library Intrinsic Function Reference** for more information about the intrinsic functions.

4.2 Calling the Interpreter

There are several ways of calling the interpreter. The two most common method is to load a file containing **S-Lang** code, or to load a string.

4.2.1 Loading Files

The `SLang_load_file` and `SLns_load_file` functions may be used to interpret a file. Both these functions return zero if successful, or `-1` upon failure. If either of these functions fail, the interpreter will accept no more code unless the error state is cleared. This is done by calling `SLang_restart` function to set the interpreter to its default state:

```
    if (-1 == SLang_load_file ("site.sl"))
      {
         /* Clear the error and reset the interpreter */
         SLang_restart (1);
      }
```

When a file is loaded via `SLang_load_file`, any non-public variables and functions defined in the file will be placed into a namespace that is local to the file itself. The `SLns_load_file` function may be used to load a file using a specified namespace, e.g.,

```
    if (-1 == SLns_load_file ("site.sl", "NS"))
      {
        SLang_restart (1);
        SLang_set_error (0);
      }
```

will load `site.sl` into a namespace called `NS`. If such a namespace does not exist, then it will be created.

Both the `SLang_load_file` and `SLns_load_file` functions search for files along an application-specified search path. This path may be set using the `SLpath_set_load_path` function, as well as from interpeted code via the `set_slang_load_path` function. By default, no search path is defined.

NOTE: It is highly recommended that an application embedding the interpreter include the slsh lib directory in the search path. The `.sl` files that are part of slsh are both useful and and should work with any application embedding the interpreter. Moreover, if the application permits dynamically loaded modules, then there are a growing number of excellent quality modules for slsh that can be utilized by it. Applications that follow this recommendation are said to be conforming.

Files are searched as follows: If the name begins with the equivalent of `"./"` or `"../"`, then it is searched for with respect to the current directory, and not along the load-path. If no such file exists, then an error will be generated. Otherwise, the file is searched for in each of the directories of the load-path by concatenating the path element with the specified file name. The first such file found to exist by this process will be loaded. If a matching file still has not been found, and the file name lacks an extension, then the path is searched with `".sl"` and `".slc"` appended to the filename. If two such files are found (one ending with `".sl"` and the other with `".slc"`), then the more recent of the two will be used. If no matching file has been found by this process, then the search will cease and an error generated.

The search path is a delimiter separated list of directories that specify where the interpreter looks for files. By default, the value of the delimiter is OS-dependent following the convention of the underlying OS. For example, on Unix the delimiter is represented by a colon, on DOS/Windows it is a semi-colon, and on VMS it is a space. The `SLpath_set_delimiter` and `SLpath_get_delimiter` may be used to set and query the delimiter's value, respectively.

4.2.2 Loading Strings

There are several other mechanisms for interacting with the interpreter. For example, the `SLang_load_string` function loads a string into the interpreter and interprets it:

```
if (-1 == SLang_load_string ("message (\"hello\");"))
  return;
```

Similarly, the `SLns_load_string` function may be used to load a string into a specified namespace.

Typically, an interactive application will load a file via `SLang_load_file` and then go into a loop that consists of reading lines of input and sending them to the interpreter, e.g.,

```
while (EOF != fgets (buf, sizeof (buf), stdin))
  {
     if (-1 == SLang_load_string (buf))
       {
          SLang_restart (1);
       }
  }
```

Finally, some applications such as **jed** and **slrn** use another method of interacting with the interpreter. They read key sequences from the keyboard and map those key sequences to interpreter functions via the **S-Lang** keymap interface.

4.3 Intrinsic Functions

An intrinsic function is simply a function that is written in C and is made available to the interpreter as a built-in function. For this reason, the words 'intrinsic' and 'built-in' are often used interchangeably.

Applications are expected to add application specific functions to the interpreter. For example, **jed** adds nearly 300 editor-specific intrinsic functions. The application designer should think carefully about what intrinsic functions to add to the interpreter.

4.3.1 Restrictions on Intrinsic Functions

When implementing intrinsic functions, it is necessary to follow a few rules to cooperate with the interpreter.

The C version of an intrinsic function takes only pointer arguments. This is because when the interpreter calls an intrinsic function, it passes values to the function by reference and *not* by value. For example, intrinsic with the declarations:

```
int intrinsic_0 (void);
int intrinsic_1 (char *s);
void intrinsic_2 (char *s, int *i);
void intrinsic_3 (int *i, double *d, double *e);
```

are all valid. However,

```
int invalid_1 (char *s, int len);
```

is not valid since the `len` parameter is not a pointer.

The return value of an intrinsic function must be one of the following types: `void`, `char`, `short`, `int`, `long`, `double`, `char *`, as well as unsigned versions of the integer types. A function such as

```
int *invalid (void);
```

is not permitted since `int*` is not a valid return-type for an intrinsic function. Any other type of value can be passed back to the interpreter by explicitly pushing the object onto the interpreter's stack via the appropriate "push" function.

The current implementation limits the number of arguments of an intrinsic function to `7`. The "pop" functions can be used to allow the function to take an arbitrary number as seen from an interpreter script.

Another restriction is that the intrinsic function should regard all its parameters as pointers to constant objects and make no attempt to modify the value to which they point. For example,

```
void truncate (char *s)
{
   s[0] = 0;
}
```

is illegal since the function modifies the string `s`.

4.3.2 Adding a New Intrinsic

There are two basic mechanisms for adding an intrinsic function to the interpreter: `SLadd_intrinsic_function` and `SLadd_intrin_fun_table`. Functions may be added to a specified namespace via `SLns_add_intrinsic_function` and `SLns_add_intrin_fun_table` functions.

As an specific example, consider a function that will cause the program to exit via the `exit` C library function. It is not possible to make this function an intrinsic because it does not meet the specifications for an intrinsic function that were described earlier. However, one can call `exit` from a function that is suitable, e.g.,

```
void intrin_exit (int *code)
{
   exit (*code);
}
```

This function may be made available to the interpreter as an intrinsic via the `SLadd_intrinsic_function` routine:

```
if (-1 == SLadd_intrinsic_function ("exit", (FVOID_STAR) intrin_exit,
                                    SLANG_VOID_TYPE, 1,
                                    SLANG_INT_TYPE))
  exit (EXIT_FAILURE);
```

This statement basically tells the interpreter that `intrin_exit` is a function that returns nothing and takes a single argument: a pointer to an integer (`SLANG_INT_TYPE`). A user can call this function from within the interpreter via

```
message ("Calling the exit function");
exit (0);
```

After printing a message, this will cause the `intrin_exit` function to execute, which in turn calls `exit`.

The most convenient mechanism for adding new intrinsic functions is to create a table of `SLang_Intrin_Fun_Type` objects and add the table via the `SLadd_intrin_fun_table` function. The table will look like:

```
SLang_Intrin_Fun_Type My_Intrinsics [] =
{
 /* table entries */
  MAKE_INTRINSIC_N(...),
  MAKE_INTRINSIC_N(...),
       .
       .
  MAKE_INTRINSIC_N(...),
  SLANG_END_INTRIN_FUN_TABLE
};
```

Construction of the table entries may be facilitated using a set of `MAKE_INTRINSIC` macros defined in `slang.h`. The main macro is called `MAKE_INTRINSIC_N` and takes 11 arguments:

```
MAKE_INTRINSIC_N(name, funct-ptr, return-type, num-args,
                 arg-1-type, arg-2-type, ... arg-7-type)
```

Here `name` is the name of the intrinsic function that the interpreter is to give to the function. `func-ptr` is a pointer to the intrinsic function taking `num-args` and returning `ret-type`. The final

7 arguments specify the argument types. For example, the `intrin_exit` intrinsic described above may be added to the table using

```
MAKE_INTRINSIC_N("exit", intrin_exit, SLANG_VOID_TYPE, 1,
                 SLANG_INT_TYPE, 0,0,0,0,0,0)
```

While `MAKE_INTRINSIC_N` is the main macro for constructing table entries, `slang.h` defines other macros that may prove useful. In particular, an entry for the `intrin_exit` function may also be created using any of the following forms:

```
MAKE_INTRINSIC_1("exit", intrin_exit, SLANG_VOID_TYPE, SLANG_INT_TYPE)
MAKE_INTRINSIC_I("exit", intrin_exit, SLANG_VOID_TYPE)
```

See `slang.h` for related macros. You are also encouraged to look at, e.g., `slang/src/slstd.c` for a more extensive examples.

The table may be added via the `SLadd_intrin_fun_table` function, e.g.,

```
if (-1 == SLadd_intrin_fun_table (My_Intrinsics, NULL))
  {
     /* an error occurred */
  }
```

Please note that there is no need to load a given table more than once, and it is considered to be an error on the part of the application it adds the same table multiple times. For performance reasons, no checking is performed by the library to see if a table has already been added.

Earlier it was mentioned that intrinsics may be added to a specified namespace. To this end, one must first get a pointer to the namespace via the `SLns_create_namespace` function. The following example illustrates how this function is used to add the `My_Intrinsics` table to a namespace called `my`:

```
SLang_NameSpace_Type *ns = SLns_create_namespace ("my");
if (ns == NULL)
  return -1;

return SLns_add_intrin_fun_table (ns, My_Intrinsics, "__MY__"));
```

4.3.3 More Complicated Intrinsics

The intrinsic functions described in the previous example were functions that took a fixed number of arguments. In this section we explore more complex intrinsics such as those that take a variable number of arguments.

Consider a function that takes two double precision numbers and returns the lesser:

```
double intrin_min (double *a, double *b)
{
   if (*a < *b) return *a;
   return *b;
}
```

This function may be added to a table of intrinsics using

```
MAKE_INTRINSIC_2("vmin", intrin_min, SLANG_DOUBLE_TYPE,
                 SLANG_DOUBLE_TYPE, SLANG_DOUBLE_TYPE)
```

It is useful to extend this function to take an arbitray number of arguments and return the lesser. Consider the following variant:

```
double intrin_min_n (int *num_ptr)
{
   double min_value, x;
   unsigned int num = (unsigned int) *num_ptr;

   if (-1 == SLang_pop_double (&min_value))
     return 0.0;
   num--;

   while (num > 0)
     {
        num--;
        if (-1 == SLang_pop_double (&x))
          return 0.0;
        if (x < min_value) min_value = x;
     }
   return min_value;
}
```

Here the number to compare is passed to the function and the actual numbers are removed from the stack via the `SLang_pop_double` function. A suitable table entry for it is

```
MAKE_INTRINSIC_I("vmin", intrin_min_n, SLANG_DOUBLE_TYPE)
```

This function would be used in an interpreter script via a statement such as

```
  variable xmin = vmin (x0, x1, x2, x3, x4, 5);
```

which computes the smallest of 5 values.

The problem with this intrinsic function is that the user must explicitly specify how many numbers to compare. It would be more convenient to simply use

```
  variable xmin = vmin (x0, x1, x2, x3, x4);
```

An intrinsic function can query the value of the variable `SLang_Num_Function_Args` to obtain the necessary information:

```
double intrin_min (void)
{
   double min_value, x;

   unsigned int num = SLang_Num_Function_Args;
```

```
     if (-1 == SLang_pop_double (&min_value, NULL, NULL))
       return 0.0;
     num--;

     while (num > 0)
       {
          num--;
          if (-1 == SLang_pop_double (&x, NULL, NULL))
            return 0.0;
          if (x < min_value) min_value = x;
       }
     return min_value;
  }
```

This may be declared as an intrinsic using:

```
  MAKE_INTRINSIC_0("vmin", intrin_min, SLANG_DOUBLE_TYPE)
```

4.4 Intrinsic Variables

It is possible to access an application's global variables from within the interpreter. The current implementation supports the access of variables of type `int`, `char *`, and `double`.

There are two basic methods of making an intrinsic variable available to the interpreter. The most straight forward method is to use the function `SLadd_intrinsic_variable`:

```
   int SLadd_intrinsic_variable (char *name, VOID_STAR addr,
                                 SLtype data_type,
                                 int read_only);
```

For example, suppose that `I` is an integer variable, e.g.,

```
   int I;
```

One can make it known to the interpreter as `I_Variable` via a statement such as

```
   if (-1 == SLadd_intrinsic_variable ("I_Variable", &I,
                                        SLANG_INT_TYPE, 0))
     exit (EXIT_FAILURE);
```

Similarly, if `S` is declared as

```
  char *S;
```

then

```
   if (-1 == SLadd_intrinsic_variable ("S_Variable", &S,
                                        SLANG_STRING_TYPE, 1))
     exit (EXIT_FAILURE);
```

makes `S` available as a *read-only* variable with the name `S_Variable`. Note that if a pointer variable is made available to the interpreter, it should be declared as being *read-only* to prevent the interpreter from changing the pointer's value.

It is important to note that if `S` were declared as an array of characters, e.g.,

```
char S[256];
```

then it would not be possible to make it directly available to the interpreter. However, one could create a pointer to it, i.e.,

```
char *S_Ptr = S;
```

and make `S_Ptr` available as a read-only variable.

One should not make the mistake of trying to use the same address for different variables as the following example illustrates:

```
int do_not_try_this (void)
{
   static char *names[3] = {"larry", "curly", "moe"};
   unsigned int i;

   for (i = 0; i < 3; i++)
     {
        int value;
        if (-1 == SLadd_intrinsic_variable (names[i], (VOID_STAR) &value,
                                            SLANG_INT_TYPE, 1))
          return -1;
     }
   return 0;
}
```

Not only does this piece of code create intrinsic variables that use the same address, it also uses the address of a local variable that will go out of scope.

The most convenient method for adding many intrinsic variables to the interpreter is to create an array of `SLang_Intrin_Var_Type` objects and then add the array via `SLadd_intrin_var_table`. For example, the array

```
static SLang_Intrin_Var_Type Intrin_Vars [] =
{
   MAKE_VARIABLE("I_Variable", &I, SLANG_INT_TYPE, 0),
   MAKE_VARIABLE("S_Variable", &S_Ptr, SLANG_STRING_TYPE, 1),
   SLANG_END_TABLE
};
```

may be added via

```
if (-1 == SLadd_intrin_var_table (Intrin_Vars, NULL))
  exit (EXIT_FAILURE);
```

It should be rather obvious that the arguments to the `MAKE_VARIABLE` macro correspond to the parameters of the `SLadd_intrinsic_variable` function.

Finally, variables may be added to a specific namespace via the SLns_add_intrin_var_table and SLns_add_intrinsic_variable functions.

4.5 Aggregate Data Objects

An aggregate data object is an object that can contain more than one data value. The **S-Lang** interpreter supports several such objects: arrays, structure, and associative arrays. In the following sections, information about interacting with these objects is given.

4.5.1 Arrays

An intrinsic function may interact with an array in several different ways. For example, an intrinsic may create an array and return it. The basic functions for manipulating arrays include:

```
SLang_create_array
SLang_pop_array_of_type
SLang_push_array
SLang_free_array
SLang_get_array_element
SLang_set_array_element
```

The use of these functions will be illustrated via a few simple examples.

The first example shows how to create an return an array of strings to the interpreter. In particular, the names of the four seasons of the year will be returned:

```
void months_of_the_year (void)
{
   static char *seasons[4] =
     {
        "Spring", "Summer", "Autumn", "Winter"
     };
   SLang_Array_Type *at;
   SLindex_Type i, four;

   four = 4;
   at = SLang_create_array (SLANG_STRING_TYPE, 0, NULL, &four, 1);
   if (at == NULL)
     return;

   /* Now set the elements of the array */
   for (i = 0; i < 4; i++)
     {
       if (-1 == SLang_set_array_element (at, &i, &seasons[i]))
         {
            SLang_free_array (at);
```

```
                    return;
                 }
              }

           (void) SLang_push_array (at, 0);
           SLang_free_array (at);
         }
```

This example illustrates several points:

First of all, the `SLang_create_array` function was used to create a 1 dimensional array of 4 strings. Since this function could fail, its return value was checked. Also `SLindex_Type` was used for the array size and index types. In **S-Lang** version 2, `SLindex_Type` is typedefed to be an `int`. However, as this will change in a future version of the library, `SLindex_Type` should be used.

The `SLang_set_array_element` function was used to set the elements of the newly created array. Note that the address containing the value of the array element was passed and not the value of the array element itself. That is,

```
         SLang_set_array_element (at, &i, seasons[i])
```

was not used. The return value from this function was also checked because it too could also fail.

Finally, the array was pushed onto the interpreter's stack and then it was freed. It is important to understand why it was freed. This is because arrays are reference-counted. When the array was created, it was returned with a reference count of `1`. When it was pushed, the reference count was bumped up to `2`. Then since it was nolonger needed by the function, `SLang_free_array` was called to decrement the reference count back to `1`. For convenience, the second argument to `SLang_push_array` determines whether or not it is to also free the array. So, instead of the two function calls:

```
        (void) SLang_push_array (at, 0);
        SLang_free_array (at);
```

it is preferable to combine them as

```
        (void) SLang_push_array (at, 1);
```

The second example returns a diagonal array of a specified size to the stack. A diagonal array is a 2-d array with all elements zero except for those along the diagonal, which have a value of one:

```
        void make_diagonal_array (SLindex_Type n)
        {
           SLang_Array_Type *at;
           SLindex_Type dims[2];
           SLindex_Type i, one;

           dims[0] = dims[1] = n;
           at = SLang_create_array (SLANG_INT_TYPE, 0, NULL, dims, 2);
           if (at == NULL)
             return;
```

```
   one = 1;
   for (i = 0; i < n; i++)
     {
        dims[0] = dims[1] = i;
        if (-1 == SLang_set_array_element (at, dims, &one))
          {
             SLang_free_array (at);
             return;
          }
     }

   (void) SLang_push_array (at, 1);
}
```

In this example, only the diagonal elements of the array were set. This is bacause when the array was created, all its elements were set to zero.

Now consider an example that acts upon an existing array. In particular, consider one that computes the trace of a 2-d matrix, i.e., the sum of the diagonal elements:

```
double compute_trace (void)
{
   SLang_Array_Type *at;
   double trace;
   SLindex_Type dims[2];

   if (-1 == SLang_pop_array_of_type (&at, SLANG_DOUBLE_TYPE))
     return 0.0;

   /* We want a 2-d square matrix.  If the matrix is 1-d and has only one
      element, then return that element. */
   trace = 0.0;
   if (((at->num_dims == 1) && (at->dims[0] == 1))
       || ((at->num_dims == 2) && (at->dims[0] == at->dims[1])))
     {
        double dtrace;
        SLindex_Type n = at->dims[0];

        for (i = 0; i < n; i++)
          {
             dims[0] = dims[1] = i;
             (void) SLang_get_array_element (at, &dims, &dtrace);
             trace += dtrace;
          }
     }
  else SLang_verror (SL_TYPE_MISMATCH, "Expecting a square matrix");

  SLang_free_array (at);
  return trace;
}
```

In this example, SLang_pop_array_of_type was used to pop an array of doubles from the stack. This function will make implicit typecasts in order to return an array of the requested type.

4.5.2 Structures

For the purposes of this section, we shall differentiate structures according to whether or not they correspond to an application defined C structure. Those that do are called intrinsic structures, and those do not are called **S-Lang** interpreter structures.

Interpreter Structures

The following simple example shows one method that may be used to create and return a structure with a string and integer field to the interpreter's stack:

```
int push_struct_example (char *string_value, int int_value)
{
   char *field_names[2];
   SLtype field_types[2];
   VOID_STAR field_values[2];

   field_names[0] = "string_field";
   field_types[0] = SLANG_STRING_TYPE;
   field_values[0] = &string_value;

   field_names[1] = "int_field";
   field_types[1] = SLANG_INT_TYPE;
   field_values[1] = &int_value;

   if (-1 == SLstruct_create_struct (2, field_names,
                                     field_types, field_values))
     return -1;
   return 0;
}
```

Here, SLstruct_create_struct is used to push a structure with the specified field names and values onto the interpreter's stack.

A simpler mechanism exists provided that one has already defined a C structure with a description of how the structure is laid out. For example, consider a C structure defined by

```
typedef struct
{
   char *s;
   int i;
}
SI_Type;
```

Its layout may be specified via a table of SLang_CStruct_Field_Type entries:

```
SLang_CStruct_Field_Type SI_Type_Layout [] =
{
  MAKE_CSTRUCT_FIELD(SI_Type, s, "string_field", SLANG_STRING_TYPE, 0),
  MAKE_CSTRUCT_FIELD(SI_Type, i, "int_field", SLANG_INT_TYPE, 0),
  SLANG_END_CSTRUCT_TABLE
};
```

Here, MAKE_CSTRUCT_FIELD is a macro taking 5 arguments:

```
MAKE_CSTRUCT_FIELD(C-structure-type,
                   C-field-name,
                   slang-field-name,
                   slang-data-type,
                   is-read-only)
```

The first argument is the structure type, the second is the name of a field of the structure, the third is a string that specifies the name of the corresponding field of the **S-Lang** structure, the fourth argument specifies the field's type, and the last argument specifies whether or not the field should be regarded as read-only.

Once the layout of the structure has been specified, pushing a **S-Lang** version of the structure is trival:

```
int push_struct_example (char *string_value, int int_value)
{
   SI_Type si;

   si.s = string_value;
   si.i = int_value;
   return SLang_push_cstruct ((VOID_STAR)&si, SI_Type_Layout);
}
```

This mechanism of structure creation also permits a **S-Lang** structure to be passed to an intrinsic function through the use of the SLang_pop_cstruct routine, e.g.,

```
void print_si_struct (void)
{
   SI_Type si;
   if (-1 == SLang_pop_cstruct ((VOID_STAR)&si, SI_Type_Layout))
     return;
   printf ("si.i=%d", si.i);
   printf ("si.s=%s", si.s);
   SLang_free_cstruct ((VOID_STAR)&si, SI_Type_Layout);
}
```

Assuming `print_si_struct` exists as an intrinsic function, the **S-Lang** code

```
variable s = struct {string_field, int_field};
s.string_field = "hello";
s.int_field = 20;
print_si_struct (s);
```

would result in the display of

```
 si.i=20;
 si.s=hello
```

Note that the `SLang_free_cstruct` function was called after the contents of `si` were nolonger needed. This was necessary because `SLang_pop_cstruct` allocated memory to set the `char *s` field of `si`. Calling `SLang_free_cstruct` frees up such memory.

Now consider the following:

```
typedef struct
{
   pid_t pid;
   gid_t group;
}
X_t;
```

How should the layout of this structure be defined? One might be tempted to use:

```
SLang_CStruct_Field_Type X_t_Layout [] =
{
  MAKE_CSTRUCT_FIELD(X_t, pid, "pid", SLANG_INT_TYPE, 0),
  MAKE_CSTRUCT_FIELD(X_t, group, "group", SLANG_INT_TYPE, 0),
  SLANG_END_CSTRUCT_TABLE
};
```

However, this assumes `pid_t` and `gid_t` have been typedefed as ints. But what if `gid_t` is a `short`? In such a case, using

```
  MAKE_CSTRUCT_FIELD(X_t, group, "group", SLANG_SHORT_TYPE, 0),
```

would be the appropriate entry for the `group` field. Of course, one has no way of knowing how `gid_t` is declared on other systems. For this reason, it is preferable to use the `MAKE_CSTRUCT_INT_FIELD` macro in cases involving integer valued fields, e.g.,

```
SLang_CStruct_Field_Type X_t_Layout [] =
{
  MAKE_CSTRUCT_INT_FIELD(X_t, pid, "pid", 0),
  MAKE_CSTRUCT_INT_FIELD(X_t, group, "group", 0),
  SLANG_END_CSTRUCT_TABLE
};
```

Before leaving this section, it is important to mention that access to character array fields is not permitted via this interface. That is, a structure such as

```
 typedef struct
 {
    char name[32];
 }
 Name_Type;
```

is not supported since `char name[32]` is not a `SLANG_STRING_TYPE` object. Always keep in mind that a `SLANG_STRING_TYPE` object is a `char *`.

Intrinsic Structures

Here we show how to make intrinsic structures available to the interpreter.

The simplest interface is to structure pointers and not to the actual structures themselves. The latter would require the interpreter to be involved with the creation and destruction of the structures. Dealing with the pointers themselves is far simpler.

As an example, consider an object such as

```
typedef struct _Window_Type
{
   char *title;
   int row;
   int col;
   int width;
   int height;
} Window_Type;
```

which defines a window object with a title, size (`width`, `height`), and location (`row`, `col`).

We can make variables of type `Window_Type` available to the interpreter via a table as follows:

```
static SLang_IStruct_Field_Type Window_Type_Field_Table [] =
{
  MAKE_ISTRUCT_FIELD(Window_Type, title, "title", SLANG_STRING_TYPE, 1),
  MAKE_ISTRUCT_FIELD(Window_Type, row, "row", SLANG_INT_TYPE, 0),
  MAKE_ISTRUCT_FIELD(Window_Type, col, "col", SLANG_INT_TYPE, 0),
  MAKE_ISTRUCT_FIELD(Window_Type, width, "width", SLANG_INT_TYPE, 0),
  MAKE_ISTRUCT_FIELD(Window_Type, height, "height", SLANG_INT_TYPE, 0),
  SLANG_END_ISTRUCT_TABLE
};
```

More precisely, this defines the layout of the `Window_Type` structure. Here, the `title` has been declared as a read-only field. Using

```
  MAKE_ISTRUCT_FIELD(Window_Type, title, "title", SLANG_STRING_TYPE, 0),
```

would allow read-write access.

Now suppose that `My_Window` is a pointer to a `Window_Type` object, i.e.,

```
Window_Type *My_Window;
```

We can make this variable available to the interpreter via the `SLadd_istruct_table` function:

```
if (-1 == SLadd_istruct_table (Window_Type_Field_Table,
                               (VOID_STAR) &My_Window,
                               "My_Win"))
  exit (1);
```

This creates a S-Lang interpreter variable called `My_Win` whose value corresponds to the `My_Win` structure. This would permit one to access the fields of `My_Window` via **S-Lang** statements such as

```
define set_width_and_height (w,h)
{
    My_Win.width = w;
    My_Win.height = h;
}
```

It is extremely important to understand that the interface described in this section does not allow the interpreter to create new instances of `Window_Type` objects. The interface merely defines an association or correspondence between an intrinsic structure pointer and a **S-Lang** variable. For example, if the value of `My_Window` is `NULL`, then `My_Win` would also be `NULL`.

One should be careful in allowing read/write access to character string fields. If read/write access is allowed, then the application should always use the `SLang_create_slstring` and `SLang_free_slstring` functions to set the character string field of the structure.

4.6 Signals

If your program that embeds the interpreter processes signals, then it may be undesirable to allow access to all signals from the interpreter. For example, if your program has a signal handler for `SIGHUP` then it is possible that an interpreter script could specify a different signal handler, which may or may not be desirable. If you do not want to allow the interpreter access to some signal, then that signal can be made off-limits to the interpreter via the `SLsig_forbid_signal` function:

```
/* forbid a signal handler for SIGHUP */
SLsig_forbid_signal (SIGHUP, 1);

/* Allow a signal handler for SIGTERM */
SLsig_forbid_signal (SIGTERM, 0);
```

By default, all signals are allowed access from the interpreter.

4.7 Exceptions

Chapter 5

Keyboard Interface

S-Lang's keyboard interface has been designed to allow an application to read keyboard input from the user in a system-independent manner. The interface consists of a set of low routines for reading single character data as well as a higher level interface (`SLkp`) which utilize **S-Lang**'s keymap facility for reading multi-character sequences.

To initialize the interface, one must first call the function `SLang_init_tty`. Before exiting the program, the function `SLang_reset_tty` must be called to restore the keyboard interface to its original state. Once initialized, the low-level `SLang_getkey` function may be used to read *single* keyboard characters from the terminal. An application using the higher-level `SLkp` interface will read charcters using the `SLkp_getkey` function.

In addition to these basic functions, there are also functions to "unget" keyboard characters, flush the input, detect pending-input with a timeout, etc. These functions are defined below.

5.1 Initializing the Keyboard Interface

The function `SLang_init_tty` must be called to initialize the terminal for single character input. This puts the terminal in a mode usually referred to as "raw" mode.

The prototype for the function is:

```
          int SLang_init_tty (int abort_char, int flow_ctrl, int opost);
```

It takes three parameters that are used to specify how the terminal is to be initialized.

The first parameter, `abort_char`, is used to specify the interrupt character (`SIGINT`). Under MSDOS, this value corresponds to the scan code of the character that will be used to generate the interrupt. For example, under MSDOS, `34` should be used to make `Ctrl-G` generate an interrupt signal since 34 is the scan code for `G`. On other systems, the value of `abort_char` will simply be the ascii value of the control character that will be used to generate the interrupt signal, e.g., `7` for `Ctrl-G`. If `-1` is passed, the interrupt character will not be changed.

Pressing the interrupt character specified by the first argument will generate a signal (`SIGINT`) that may or not be caught by the application. It is up to the application to catch this signal. **S-Lang** provides the function `Slang_set_abort_signal` to make it easy to facilitate this task.

The second parameter is used to specify whether or not flow control should be used. If this parameter is zero, flow control is enabled. If the value is positive, flow control will be disabled. Disabling flow control is necessary to pass certain characters to the application (e.g., `Ctrl-S` and `Ctrl-Q`). Otherwise, the value is negative and the flow control behavior will be inherited from the terminal. The latter interpretation was added to version 2.3.0 of the library; earlier versions disabled flow control for both positive and negative values of this parameter. For some systems such as MSDOS, this parameter is meaningless.

The third parameter, `opost`, is used to turn output processing on or off. If `opost` is zero, output processing is *not* turned on otherwise, output processing is turned on.

The `SLang_init_tty` function returns -1 upon failure. In addition, after it returns, the **S-Lang** global variable `SLang_TT_Baud_Rate` will be set to the baud rate of the terminal if this value can be determined.

Example:

```
      if (-1 == SLang_init_tty (7, 0, 0))  /* For MSDOS, use 34 as scan code */
        {
          fprintf (stderr, "Unable to initialize the terminal.\n");
          exit (1);
        }
      SLang_set_abort_signal (NULL);
```

Here the terminal is initialized such that flow control and output processing are turned off. In addition, the character `Ctrl-G`[1] has been specified to be the interrupt character. The function `SLang_set_abort_signal` is used to install the default **S-Lang** interrupt signal handler.

5.2 Resetting the Keyboard Interface

The function `SLang_reset_tty` must be called to reset the terminal to the state it was in before the call to `SLang_init_tty`. The prototype for this function is:

```
      void SLang_reset_tty (void);
```

Usually this function is only called before the program exits. However, if the program is suspended it should also be called just before suspension.

5.3 Initializing the `SLkp` Routines

Extra initialization of the higher-level `SLkp` functions are required because they are layered on top of the lower level routines. Since the `SLkp_getkey` function is able to process function and arrow keys in a terminal independent manner, it is necessary to call the `SLtt_get_terminfo` function to get information about the escape character sequences that the terminal's function keys send. Once that information is available, the `SLkp_init` function can construct the proper keymaps to process the escape sequences.

[1]For MSDOS systems, use the *scan code* 34 instead of 7 for `Ctrl-G`

This part of the initialization process for an application using this interface will look something like:

```
SLtt_get_terminfo ();
if (-1 == SLkp_init ())
  {
     SLang_doerror ("SLkp_init failed.");
     exit (1);
  }
if (-1 == SLang_init_tty (-1, 0, 1))
  {
     SLang_doerror ("SLang_init_tty failed.");
     exit (1);
  }
```

It is important to check the return status of the `SLkp_init` function which can failed if it cannot allocate enough memory for the keymap.

5.4 Setting the Interrupt Handler

The function `SLang_set_abort_signal` may be used to associate an interrupt handler with the interrupt character that was previously specified by the `SLang_init_tty` function call. The prototype for this function is:

```
void SLang_set_abort_signal (void (*)(int));
```

This function returns nothing and takes a single parameter which is a pointer to a function taking an integer value and returning `void`. If a `NULL` pointer is passed, the default **S-Lang** interrupt handler will be used. The **S-Lang** default interrupt handler under Unix looks like:

```
static void default_sigint (int sig)
{
  SLsignal_intr (SIGINT, default_sigint);
  SLKeyBoard_Quit = 1;
  if (SLang_Ignore_User_Abort == 0)
    SLang_set_error (SL_UserBreak_Error);
}
```

It simply sets the global variable `SLKeyBoard_Quit` to one and if the variable `SLang_Ignore_User_Abort` is non-zero, the error state is set to indicate a user break condition. (The function `SLsignal_intr` is similar to the standard C `signal` function *except that it will interrupt system calls*. Some may not like this behavior and may wish to call this `SLang_set_abort_signal` with a different handler.)

Although the function expressed above is specific to Unix, the analogous routines for other operating systems are equivalent in functionality even though the details of the implementation may vary drastically (e.g., under MSDOS, the hardware keyboard interrupt `int 9h` is hooked).

5.5 Reading Keyboard Input with SLang_getkey

After initializing the keyboard via `SLang_init_tty`, the **S-Lang** function `SLang_getkey` may be used to read characters from the terminal interface. In addition, the function `SLang_input_pending` may be used to determine whether or not keyboard input is available to be read.

These functions have prototypes:

```
unsigned int SLang_getkey (void);
int SLang_input_pending (int tsecs);
```

The `SLang_getkey` function returns a single character from the terminal. Upon failure, it returns `0xFFFF`. If the interrupt character specified by the `SLang_init_tty` function is pressed while this function is called, the function will return the value of the interrupt character and set the **S-Lang** global variable `SLKeyBoard_Quit` to a non-zero value. In addition, if the default **S-Lang** interrupt handler has been specified by a `NULL` argument to the `SLang_set_abort_signal` function, the error state of the library will be set to `SL_UserBreak_Error` *unless* the variable `SLang_Ignore_User_Abort` is non-zero.

The `SLang_getkey` function waits until input is available to be read. The `SLang_input_pending` function may be used to determine whether or not input is ready. It takes a single parameter that indicates the amount of time to wait for input before returning with information regarding the availability of input. This parameter has units of one tenth (1/10) of a second, i.e., to wait one second, the value of the parameter should be `10`. Passing a value of zero causes the function to return right away. `SLang_input_pending` returns a positive integer if input is available or zero if input is not available. It will return -1 if an error occurs.

Here is a simple example that reads keys from the terminal until one presses `Ctrl-G` or until 5 seconds have gone by with no input:

```
#include <stdio.h>
#include <slang.h>
int main ()
{
   int abort_char = 7;  /* For MSDOS, use 34 as scan code */
   unsigned int ch;

   if (-1 == SLang_init_tty (abort_char, 0, 1))
     {
        fprintf (stderr, "Unable to initialize the terminal.\n");
        exit (-1);
     }
   SLang_set_abort_signal (NULL);
   while (1)
     {
        fputs ("\nPress any key.  To quit, press Ctrl-G: ", stdout);
        fflush (stdout);
        if (SLang_input_pending (50) == 0)  /* 50/10 seconds */
          {
             fputs ("Waited too long! Bye\n", stdout);
             break;
```

```
                 }

               ch = SLang_getkey ();
               if (SLang_get_error () == SL_UserBreak_Error)
                 {
                    fputs ("Ctrl-G pressed!  Bye\n", stdout);
                    break;
                 }
               putc ((int) ch, stdout);
            }
          SLang_reset_tty ();
          return 0;
       }
```

5.6 Reading Keyboard Input with SLkp_getkey

Unlike the low-level function `SLang_getkey`, the `SLkp_getkey` function can read a multi-character sequence associated with function keys. The `SLkp_getkey` function uses `SLang_getkey` and **S-Lang**'s keymap facility to process escape sequences. It returns a single integer which describes the key that was pressed:

```
      int SLkp_getkey (void);
```

That is, the `SLkp_getkey` function simple provides a mapping between keys and integers. In this context the integers are called *keysyms*.

For single character input such as generated by the `a` key on the keyboard, the function returns the character that was generated, e.g., `'a'`. For single characters, `SLkp_getkey` will always return an keysym whose value ranges from 0 to 256. For keys that generate multiple character sequences, e.g., a function or arrow key, the function returns an keysym whose value is greater that 256. The actual values of these keysyms are represented as macros defined in the `slang.h` include file. For example, the up arrow key corresponds to the keysym whose value is `SL_KEY_UP`.

Since it is possible for the user to enter a character sequence that does not correspond to any key. If this happens, the special keysym `SL_KEY_ERR` will be returned.

Here is an example of how `SLkp_getkey` may be used by a file viewer:

```
      switch (SLkp_getkey ())
        {
           case ' ':
           case SL_KEY_NPAGE:
              next_page ();
              break;
           case 'b':
           case SL_KEY_PPAGE:
              previous_page ();
              break;
           case '\r':
           case SL_KEY_DOWN:
               next_line ();
               break;
                 .
                 .
            case SL_KEY_ERR:
            default:
               SLtt_beep ();
        }
```

Unlike its lower-level counterpart, SLang_getkey, there do not yet exist any functions in the library that are capable of "ungetting" keysyms. In particular, the SLang_ungetkey function will not work.

5.7 Buffering Input

S-Lang has several functions pushing characters back onto the input stream to be read again later by SLang_getkey. It should be noted that none of the above functions are designed to push back keysyms read by the SLkp_getkey function. These functions are declared as follows:

```
void SLang_ungetkey (unsigned char ch);
void SLang_ungetkey_string (unsigned char *buf, int buflen);
void SLang_buffer_keystring (unsigned char *buf, int buflen);
```

SLang_ungetkey is the most simple of the three functions. It takes a single character a pushes it back on to the input stream. The next call to SLang_getkey will return this character. This function may be used to *peek* at the character to be read by first reading it and then putting it back.

SLang_ungetkey_string has the same function as SLang_ungetkey except that it is able to push more than one character back onto the input stream. Since this function can push back null (ascii 0) characters, the number of characters to push is required as one of the parameters.

The last of these three functions, SLang_buffer_keystring can handle more than one charater but unlike the other two, it places the characters at the *end* of the keyboard buffer instead of at the beginning.

Note that the use of each of these three functions will cause SLang_input_pending to return right away with a non-zero value.

Finally, the **S-Lang** keyboard interface includes the function SLang_flush_input with prototype

```
void SLang_flush_input (void);
```

It may be used to discard *all* input.

Here is a simple example that looks to see what the next key to be read is if one is available:

```
int peek_key ()
{
   int ch;
   if (SLang_input_pending (0) == 0) return -1;
   ch = SLang_getkey ();
   SLang_ungetkey (ch);
```

```
         return ch;
      }
```

5.8 Global Variables

Although the following **S-Lang** global variables have already been mentioned earlier, they are gathered together here for completeness.

`int SLang_Ignore_User_Abort;` If non-zero, pressing the interrupt character will not result in the libraries error state set to `SL_UserBreak_Error`.

`volatile int SLKeyBoard_Quit;` This variable is set to a non-zero value when the interrupt character is pressed. If the interrupt character is pressed when `SLang_getkey` is called, the interrupt character will be returned from `SLang_getkey`.

`int SLang_TT_Baud_Rate;` On systems which support it, this variable is set to the value of the terminal's baud rate after the call to `SLang_init_tty`.

Chapter 6

Readline Interface

The **S-Lang** library includes simple but capable readline functionality in its `SLrline` layer. The `SLrline` routines provide a simple mechanism for an application to get prompted input from a user with command line editing, completions, and history recall.

The use of the `SLrline` routines will be illustrated with a few simple examples. All of the examples given in this section may be found in the file `demo/rline.c` in the **S-Lang** source code distribution. For clarity, the code shown below omits most error checking.

6.1 Introduction

The first example simply reads input from the user until the user enters `quit`:

```
SLrline_Type *rl;
SLang_init_tty (-1, 0, 1);
rl = SLrline_open (80, SL_RLINE_BLINK_MATCH);
while (1)
  {
    char *line;
    unsigned int len;

    line = SLrline_read_line (rl, "prompt>", &len);
    if (line == NULL) break;
    if (0 == strcmp (line, "quit"))
      {
        SLfree (line);
        break;
      }
    (void) fprintf (stdout, "\nRead %d bytes: %s\n", strlen(line), line);
    SLfree (line);
  }
SLrline_close (rl);
SLang_reset_tty ();
```

In this example, the `SLtt` interface functions `SLang_init_tty` and `SLang_reset_tty` functions have been used to open and close the terminal for reading input. By default, the `SLrline` functions use the `SLang_getkey` function to read characters and assume that the terminal has been properly initialized before use.

The `SLrline_open` function was used to create an instance of an `SLrline_Type` object. The function takes two arguments: and edit window display width (80 above), and a set of flags. In this case, the `SL_RLINE_BLINK_MATCH` flags has been used to turn on parenthesis blinking. Once finished, the `SLrline_Type` object must be freed using the `SLrline_close` function.

The actual reading of the line occurs in the `SLrline_read_line` function, which takes an `SLrline_Type` instance and a string representing the prompt to be used. The line itself is returned as a malloced `char *` and must be freed using the `SLfree` function after used. The length (in bytes) of the line is returned via the parameter list.

If an end-of-file character (`^D` on Unix) was entered at the beginning of a line, the `SLrline_read_line` function will return `NULL`. However, it also return `NULL` if an error of some sort was encountered. The only way to tell the difference between these two conditions is to call `SLang_get_error`.

The above code fragment did not provide for any sort of `SIGINT` handling. Without such a provision, pressing `^C` at the prompt could be enough to kill the application. This is especially undesirable if one wants to press `^C` to abort the call to `SLrline_read_line`. The function `example_2` in `demo/rline.c` shows code to handle this situation as well as distinguish between EOF and other errors.

6.2 Interpreter Interface

`SLrline` features such as command-line completion, vi-emulation, and so on are implemented through callbacks or hooks from the `SLrline` functions to the **S-Lang** interpreter. Hence, this functionality is only available to applications that make use of the interpreter.

TBD...

Chapter 7

Screen Management

The **S-Lang** library provides two interfaces to terminal independent routines for manipulating the display on a terminal. The highest level interface, known as the SLsmg interface is discussed in this section. It provides high level screen management functions for manipulating the display in an optimal manner and is similar in spirit to the curses library. The lowest level interface, or the SLtt interface, is used by the SLsmg routines to actually perform the task of writing to the display. This interface is discussed in another section. Like the keyboard routines, the SLsmg routines are *platform independent* and work the same on MSDOS, OS/2, Unix, and VMS.

The screen management, or SLsmg, routines are initialized by function SLsmg_init_smg. Once initialized, the application uses various SLsmg functions to write to a *virtual* display. This does not cause the *physical* terminal display to be updated immediately. The physical display is updated to look like the virtual display only after a call to the function SLsmg_refresh. Before exiting, the application using these routines is required to call SLsmg_reset_smg to reset the display system.

The following subsections explore **S-Lang**'s screen management system in greater detail.

7.1 Initialization

The function SLsmg_init_smg must be called before any other SLsmg function can be used. It has the simple prototype:

```
        int SLsmg_init_smg (void);
```

It returns zero if successful or -1 if it cannot allocate space for the virtual display.

For this routine to properly initialize the virtual display, the capabilities of the terminal must be known as well as the size of the *physical* display. For these reasons, the lower level SLtt routines come into play. In particular, before the first call to SLsmg_init_smg, the application is required to call the function SLtt_get_terminfo before calling SLsmg_init_smg.

The SLtt_get_terminfo function sets the global variables SLtt_Screen_Rows and SLtt_Screen_Cols to the values appropriate for the terminal. It does this by calling the SLtt_get_screen_size function to query the terminal driver for the appropriate values for these variables. From this point on, it is up to the application to maintain the correct values for these variables by calling the SLtt_get_screen_size function whenever the display size changes, e.g., in response to a SIGWINCH signal. Finally, if the application is going to read characters from the keyboard, it is also a good idea to initialize the keyboard routines at this point as well.

7.2 Resetting SLsmg

Before the program exits or suspends, the function SLsmg_reset_smg should be called to shutdown the display system. This function has the prototype

```
void SLsmg_reset_smg (void);
```

This will deallocate any memory allocated for the virtual screen and reset the terminal's display.

Basically, a program that uses the SLsmg screen management functions and **S-Lang**'s keyboard interface will look something like:

```
#include <slang.h>
int main ()
{
   SLtt_get_terminfo ();
   SLang_init_tty (-1, 0, 0);
   SLsmg_init_smg ();

   /* do stuff .... */

   SLsmg_reset_smg ();
   SLang_reset_tty ();
   return 0;
}
```

If this program is compiled and run, all it will do is clear the screen and position the cursor at the bottom of the display. In the following sections, other SLsmg functions will be introduced which may be used to make this simple program do much more.

7.3 Handling Screen Resize Events

The function SLsmg_reinit_smg is designed to be used in conjunction with resize events.

Under Unix-like operating systems, when the size of the display changes, the application will be sent a SIGWINCH signal. To properly handle this signal, the SLsmg routines must be reinitialized to use the new display size. This may be accomplished by calling SLtt_get_screen_size to get the new size, followed by SLsmg_reinit_smg to reinitialize the SLsmg interface to use the new size. Keep in mind that these routines should not be called from within the signal handler. The following code illustrates the main ideas involved in handling such events:

```
static volatile int Screen_Size_Changed;
static sigwinch_handler (int sig)
{
     Screen_Size_Changed = 1;
     SLsignal (SIGWINCH, sigwinch_handler);
  }

  int main (int argc, char **argv)
  {
     SLsignal (SIGWINCH, sigwinch_handler);
     SLsmg_init_smg ();
      .
      .
     /* Now enter main loop */
     while (not_done)
       {
          if (Screen_Size_Changed)
            {
               SLtt_get_screen_size ();
               SLsmg_reinit_smg ();
               redraw_display ();
            }
          .
          .
       }
    return 0;
  }
```

7.4 SLsmg Functions

In the previous sections, functions for initializing and shutting down the SLsmg routines were discussed. In this section, the rest of the SLsmg functions are presented. These functions act only on the *virtual* display. The *physical* display is updated when the SLsmg_refresh function is called and *not until that time*. This function has the simple prototype:

```
void SLsmg_refresh (void);
```

7.4.1 Positioning the cursor

The SLsmg_gotorc function is used to position the cursor at a given row and column. The prototype for this function is:

```
 void SLsmg_gotorc (int row, int col);
```

The origin of the screen is at the top left corner and is given the coordinate (0, 0), i.e., the top row of the screen corresponds to row = 0 and the first column corresponds to col = 0. The last row of the screen is given by row = SLtt_Screen_Rows - 1.

It is possible to change the origin of the coordinate system by using the function SLsmg_set_screen_start with prototype:

```
void SLsmg_set_screen_start (int *r, int *c);
```

This function takes pointers to the new values of the first row and first column. It returns the previous values by modifying the values of the integers at the addresses specified by the parameter list. A `NULL` pointer may be passed to indicate that the origin is to be set to its initial value of 0. For example,

```
        int r = 10;
        SLsmg_set_screen_start (&r, NULL);
```

sets the origin to (10, 0) and after the function returns, the variable `r` will have the value of the previous row origin.

7.4.2 Writing to the Display

`SLsmg` has several routines for outputting text to the virtual display. The following points should be understood:

- The text is output at the position of the cursor of the virtual display and the cursor is advanced to the position that corresponds to the end of the text.
- Text does *not* wrap at the boundary of the display— it is trucated. This behavior seems to be more useful in practice since most programs that would use screen management tend to be line oriented.
- Control characters are displayed in a two character sequence representation with `^` as the first character. That is, `Ctrl-X` is output as `^X`.
- The behavior of the newline character depends upon the value of the `SLsmg_Newline_Behavior` variable. It may be set to any one of the following values:

  `SLSMG_NEWLINE_IGNORED` : If a newline character is encountered when writing a string to the virtual display, the characters in the string following the newline character will not be written. In other words, the newline character will act like a string termination character. This is the default setting for the `SLsmg_Newline_Behavior`.

  `SLSMG_NEWLINE_MOVES` : If a newline character is when writing to the virtual display, the following characters will be written to the beginning of the next row.

  `SLSMG_NEWLINE_SCROLLS` : When set to this value and a newline character is output at the bottom of the virtual display, the display will scroll up. Otherwise the behavior will be the same as that of `SLSMG_NEWLINE_MOVES`.

  `SLSMG_NEWLINE_PRINTABLE` : When set to this value, a newline character will be printed as the two characters sequence `^J`.

Although the some of the above items might appear to be too restrictive, in practice this is not seem to be the case. In fact, the design of the output routines was influenced by their actual use and modified to simplify the code of the application utilizing them.

`void SLsmg_write_char (char ch);`

Write a single character to the virtual display.

`void SLsmg_write_nchars (char *str, int len);`

Write `len` characters pointed to by `str` to the virtual display.

`void SLsmg_write_string (char *str);`

Write the null terminated string given by pointer `str` to the virtual display. This function is a wrapper around `SLsmg_write_nchars`.

`void SLsmg_write_nstring (char *str, int n);`

The purpose of this function is to write a null terminated string to a field that is at most `n` cells wide. Each double-wide character in the string will use two cells. If the string is not big enough to fill the `n` cells, the rest of the cells will be filled with space characters. This function is a wrapper around `SLsmg_write_wrapped_string`.

`void SLsmg_write_wrapped_string(SLuchar_Type *str, int r, int c, unsigned int dr, unsigned int dc, int`

The purpose of this function is to write a string `str` to a box defined by rows and columns satisfying `r<=row<r+dc` and `c<=column<c+dc`. The string will be wrapped at the column boundaries of the box and truncated if its size exceeds to size of the box. If the total size of the string is less than that of the box, and the `fill` parameter is non-zero, then the rest of the cells in the box will be filled with space characters. Currently the wrapping algorithm is very simple and knows nothing about word boundaries.

`void SLsmg_printf (char *fmt, ...);`

This function is similar to `printf` except that it writes to the `SLsmg` virtual display.

`void SLsmg_vprintf (char *, va_list);`

Like `SLsmg_printf` but uses a variable argument list.

7.4.3 Erasing the Display

The following functions may be used to fill portions of the display with blank characters. The attributes of blank character are the current attributes. (See below for a discussion of character attributes)

`void SLsmg_erase_eol (void);`

Erase line from current position to the end of the line.

`void SLsmg_erase_eos (void);`

Erase from the current position to the end of the screen.

`void SLsmg_cls (void);`

Clear the entire virtual display.

7.4.4 Setting Character Attributes

Character attributes define the visual characteristics the character possesses when it is displayed. Visual characteristics include the foreground and background colors as well as other attributes such as blinking, bold, and so on. Since `SLsmg` takes a different approach to this problem than other screen management libraries an explanation of this approach is given here. This approach has been motivated by experience with programs that require some sort of screen management.

Most programs that use `SLsmg` are composed of specific textual objects or objects made up of line drawing characters. For example, consider an application with a menu bar with drop down menus. The menus might be enclosed by some sort of frame or perhaps a shadow. The basic idea is to associate an integer to each of the objects (e.g., menu bar, shadow, current menu item, etc.) and create a mapping from the integer to the set of attributes. In the terminology of `SLsmg`, the integer is simply called an *object*.

For example, the menu bar might be associated with the object `1`, the drop down menu could be object `2`, the shadow could be object `3`, and so on.

The range of values for the object integer is restricted from 0 up to and including 255 on all systems except MSDOS where the maximum allowed integer is 15[1]. The object numbered zero should not be regarding as an object at all. Rather it should be regarded as all *other* objects that have not explicitly been given an object number. `SLsmg`, or more precisely `SLtt`, refers to the attributes of this special object as the *default* or *normal* attributes.

The `SLsmg` routines know nothing about the mapping of the color to the attributes associated with the color. The actual mapping takes place at a lower level in the `SLtt` routines. Hence, to map an object to the actual set of attributes requires a call to any of the following `SLtt` routines:

```
void SLtt_set_color (int obj, char *name, char *fg, char *bg);
void SLtt_set_color_object (int obj, SLtt_Char_Type attr);
void SLtt_set_mono (int obj, char *, SLtt_Char_Type attr);
```

Only the first of these routines will be discussed briefly here. The latter two functions allow more fine control over the object to attribute mapping (such as assigning a "blink" attribute to the object). For a more full explanation on all of these routines see the section about the `SLtt` interface.

The `SLtt_set_color` function takes four parameters. The first parameter, `obj`, is simply the integer of the object for which attributes are to be assigned. The second parameter is currently unused by these routines. The third and forth parameters, `fg` and `bg`, are the names of the foreground and background color to be used associated with the object. The strings that one can use for the third and fourth parameters can be any one of the 16 colors:

```
"black"                "gray"
"red"                  "brightred"
"green"                "brightgreen"
"brown"                "yellow"
"blue"                 "brightblue"
"magenta"              "brightmagenta"
"cyan"                 "brightcyan"
```

[1]This difference is due to memory constraints imposed by MSDOS. This restriction might be removed in a future version of the library.

```
        "lightgray"            "white"
```

The value of the foreground parameter `fg` can be anyone of these sixteen colors. However, on most terminals, the background color will can only be one of the colors listed in the first column[2].

Of course not all terminals are color terminals. If the **S-Lang** global variable `SLtt_Use_Ansi_Colors` is non-zero, the terminal is assumed to be a color terminal. The `SLtt_get_terminfo` will try to determine whether or not the terminal supports colors and set this variable accordingly. It does this by looking for the capability in the terminfo/termcap database. Unfortunately many Unix databases lack this information and so the `SLtt_get_terminfo` routine will check whether or not the environment variable `COLORTERM` exists. If it exists, the terminal will be assumed to support ANSI colors and `SLtt_Use_Ansi_Colors` will be set to one. Nevertheless, the application should provide some other mechanism to set this variable, e.g., via a command line parameter.

When the `SLtt_Use_Ansi_Colors` variable is zero, all objects with numbers greater than one will be displayed in inverse video[3].

With this background, the `SLsmg` functions for setting the character attributes can now be defined. These functions simply set the object attributes that are to be assigned to *subsequent* characters written to the virtual display. For this reason, the new attribute is called the *current* attribute.

`void SLsmg_set_color (int obj);`

Set the current attribute to those of object `obj`.

`void SLsmg_normal_video (void);`

This function is equivalent to `SLsmg_set_color (0)`.

`void SLsmg_reverse_video (void);`

This function is equivalent to `SLsmg_set_color (1)`. On monochrome terminals, it is equivalent to setting the subsequent character attributes to inverse video.

Unfortunately there does not seem to be a standard way for the application or, in particular, the library to determine which color will be used by the terminal for the default background. Such information would be useful in initializing the foreground and background colors associated with the default color object (0). For this reason, it is up to the application to provide some means for the user to indicate what these colors are for the particular terminal setup. To facilitate this, the `SLtt_get_terminfo` function checks for the existence of the `COLORFGBG` environment variable. If this variable exists, its value will be used to initialize the colors associated with the default color object. Specifically, the value is assumed to consist of a foreground color name and a background color name separated by a semicolon. For example, if the value of `COLORFGBG` is `lightgray;blue`, the default color object will be initialized to represent a `lightgray` foreground upon a `blue` background.

7.4.5 Lines and Alternate Character Sets

The **S-Lang** screen management library also includes routines for turning on and turning off alternate character sets. This is especially useful for drawing horizontal and vertical lines.

[2]This is also true on the Linux console. However, it need not be the case and hopefully the designers of Linux will someday remove this restriction.

[3]This behavior can be modified by using the `SLtt_set_mono` function call.

`void SLsmg_set_char_set (int flag);`

If `flag` is non-zero, subsequent write functions will use characters from the alternate character set. If `flag` is zero, the default, or, ordinary character set will be used.

`void SLsmg_draw_hline (int len);`

Draw a horizontal line from the current position to the column that is `len` characters to the right.

`void SLsmg_draw_vline (int len);`

Draw a horizontal line from the current position to the row that is `len` rows below.

`void SLsmg_draw_box (int r, int c, int dr, int dc);`

Draw a box whose upper right corner is at row `r` and column `c`. The box spans `dr` rows and `dc` columns. The current position will be left at row `r` and column `c`.

7.4.6 Miscellaneous Functions

`void SLsmg_touch_lines (int r, int n);`

Mark screen rows numbered `r`, `r + 1`, ... `r + (n - 1)` as modified. When `SLsmg_refresh` is called, these rows will be completely redrawn.

`int SLsmg_char_at(SLsmg_Char_Type *ch);`

Returns the character and its attributes at the current position. The SLsmg_Char_Type object is a structure representing the character cell:

```
#define SLSMG_MAX_CHARS_PER_CELL 5
typedef struct
 {
    unsigned int nchars;
    SLwchar_Type wchars[SLSMG_MAX_CHARS_PER_CELL];
    SLsmg_Color_Type color;
 }
 SLsmg_Char_Type;
```

Normally the value of the `nchars` field will be 1 to indicate that the character contains precisely one character whose value is given by the zeroth element of the wchars array of the structure. The value of `nchars` will be greater than one if the character cell also contains so-called Unicode combining characters. The combining characters are given by the elements 1 through `nchars-1` of the `wchars` array. If `nchars` is 0, then the character cell represents the second half of a double-wide character.

The `color` field repesents both the color of the character cell and the alternate character set setting of the cell. This value may be bitwise-anded with `SLSMG_COLOR_MASK` to obtain the cell's color, and bitwise-anded with `SLSMG_ACS_MASK` to determine whether or not the alternate-character set setting is in effect for the cell (zero or non-zero).

7.5 Variables

The following **S-Lang** global variables are used by the `SLsmg` interface. Some of these have been previously discussed.

`int SLtt_Screen_Rows; int SLtt_Screen_Cols;` The number of rows and columns of the *physical* display. If either of these numbers changes, the functions `SLsmg_reset_smg` and `SLsmg_init_smg` should be called again so that the `SLsmg` routines can re-adjust to the new size.

`int SLsmg_Tab_Width;` Set this variable to the tab width that will be used when expanding tab characters. The default is 8.

`int SLsmg_Display_Eight_Bit;` This variable determines how characters with the high bit set are to be output. Specifically, a character with the high bit set with a value greater than or equal to this value is output as is; otherwise, it will be output in a 7-bit representation. The default value for this variable is `128` for MSDOS and `160` for other systems (ISO-Latin). In UTF-8 mode, the value of this variable is 0.

`int SLtt_Use_Ansi_Colors;` If this value is non-zero, the terminal is assumed to support ANSI colors otherwise it is assumed to be monochrome. The default is 0.

`int SLtt_Term_Cannot_Scroll;` If this value is zero, the `SLsmg` will attempt to scroll the physical display to optimize the update. If it is non-zero, the screen management routines will not perform this optimization. For some applications, this variable should be set to zero. The default value is set by the `SLtt_get_terminfo` function.

7.6 Hints for using SLsmg

This section discusses some general design issues that one must face when writing an application that requires some sort of screen management.

Chapter 8

Signal Functions

Almost all non-trivial programs must worry about signals. This is especially true for programs that use the **S-Lang** terminal input/output and screen management routines. Unfortunately, there is no fixed way to handle signals; otherwise, the Unix kernel would take care of all issues regarding signals and the application programmer would never have to worry about them. For this reason, none of the routines in the **S-Lang** library catch signals; however, some of the routines block the delivery of signals during crucial moments. It is up to the application programmer to install handlers for the various signals of interest.

If the application makes use of the interpreter, then a signal handler for `SIGINT` should be installed to allow the user to break out of the interpreter via, e.g., `Ctrl-C`. In order for this to work, the signal handler should call `SLang_set_error` to generate a `SL_UserBreak_Error` exception, i.e.,

```
void sigint_handler (int sig)
{
   if (SLang_Ignore_User_Abort == 0)
     SLang_set_error (SL_UserBreak_Error);
}
```

Applications that use the `tty getkey` routines or the screen management routines must worry about signals such as:

`SIGINT`	`interrupt`
`SIGTSTP`	`stop`
`SIGQUIT`	`quit`
`SIGTTOU`	`background write`
`SIGTTIN`	`background read`
`SIGWINCH`	`window resize`

It is important that handlers be established for these signals while the either the `SLsmg` routines or the `getkey` routines are initialized. The `SLang_init_tty`, `SLang_reset_tty`, `SLsmg_init_smg`, and `SLsmg_reset_smg` functions block these signals from occurring while they are being called.

Since a signal can be delivered at any time, it is important for the signal handler to call only functions that can be called from a signal handler. This usually means that such function must be re-entrant. In particular, the `SLsmg` routines are *not* re-entrant; hence, they should not be called when a signal is being processed unless the application can ensure that the signal was not delivered while an SLsmg function was called. This statement applies to many other functions such as malloc, or, more generally, any function that calls malloc. The upshot is that the signal handler should not attempt to do too much except set a global variable for the application to look at while not in a signal handler.

The **S-Lang** library provides two functions for blocking and unblocking the above signals:

```
int SLsig_block_signals (void);
int SLsig_unblock_signals (void);
```

It should be noted that for every call to SLsig_block_signals, a corresponding call should be made to SLsig_unblock_signals, e.g.,

```
void update_screen ()
{
   SLsig_block_signals ();

   /* Call SLsmg functions */
       .
       .
   SLsig_unblock_signals ();
}
```

See demo/pager.c for examples.

Chapter 9

Searching Functions

The S-Lang library incorporates two types of searches: Regular expression pattern matching and ordinary searching.

9.1 Simple Searches

S-Lang's **SLsearch** interface functions a convenient interface to the famous Boyer-Moore fast searching algrothim. The searches can go in either a forward or backward direction and and may be performed with or without regard to case. Moreover, UTF-8 encoded strings are fully supported by the interface.

9.2 Regular Expressions

!!! No documentation available yet !!!

Appendix A

S-Lang 2 API NEWS and UPGRADE information

The **S-Lang** API underwent a number of changes for version 2. In particular, the following interfaces have been affected:

```
SLsmg
SLregexp
SLsearch
SLrline
SLprep
slang interpreter modules
```

Detailed information about these changes is given below. Other changes include:

- UTF-8 encoded strings are now supported at both the C library level and the interpreter.
- Error handling by the interpreter has been rewritten to support try/catch style exception. Applications may also define application-specific error codes.
- The library may be compiled with large-file-support.

See the relevant chapters in this manual for more information.

A.1 SLang_Error

The `SLang_Error` variable is nolonger part of the API. Change code such as

```
SLang_Error = foo;
if (SLang_Error == bar) ...
```

to

```
SLang_set_error (foo);
if (bar == SLang_get_error ()) ...
```

A.2 SLsmg/SLtt Functions

The changes to these functions were dictated by the new UTF-8 support. For the most part, the changes should be transparent but some functions and variables have been changed.

- SLtt_Use_Blink_For_ACS is nolonger supported, nor necessary I think only DOSEMU uses this.
- Prototypes for `SLsmg_draw_object` and `SLsmg_write_char` were changed to use wide characters (SLwchar_Type).
- `SLsmg_Char_Type` was changed from an `unsigned short` to a structure. This change was necessary in order to support combining characters and double width unicode characters. This change may affect the use of the following functions:

  ```
  SLsmg_char_at
  SLsmg_read_raw
  SLsmg_write_raw
  SLsmg_write_color_chars
  ```

- The `SLSMG_BUILD_CHAR` macro has been removed. The `SLSMG_EXTRACT_CHAR` macro will continue to work as long as combining characters are not present.
- The prototype for `SLsmg_char_at` was changed to

  ```
  int SLsmg_char_at (SLsmg_Char_Type *);
  ```

A.3 SLsearch Functions

`SLsearch_Type` is now an opaque type. Code such as

```
SLsearch_Type st;
SLsearch_init (string, 1, 0, &st);
   .
   .
s = SLsearch (buf, bufmax, &st);
```

which searches forward in `buf` for `string` must be changed to

```
SLsearch_Type *st = SLsearch_open (string, SLSEARCH_CASELESS);
if (st == NULL)
  return;
   .
   .
s = SLsearch_forward (st, buf, bufmax);
   .
   .
SLsearch_close (st);
```

A.4 Regular Expression Functions

The slang v1 regular expression API has been redesigned in order to facilitate the incorporation of third party regular expression engines.

New functions include:

```
SLregexp_compile
SLregexp_free
SLregexp_match
SLregexp_nth_match
SLregexp_get_hints
```

The plan is to migrate to the use of the PCRE regular expressions for version 2.2. As such, you may find it convenient to adopt the PCRE library now instead of updating to the changed **S-Lang** API.

A.5 Readline Functions

The readline interface has changed in order to make it easier to use. Using it now is as simple as:

```
SLrline_Type *rli;
rli = SLrline_open (SLtt_Screen_Cols, flags);
buf = SLrline_read_line (rli, prompt, &len);
/* Use buf */
   .
   .
SLfree (buf);
SLrline_close (rli);
```

See how it is used in `slsh/readline.c`.

A.6 Preprocessor Interface

SLPreprocess_Type was renamed to SLprep_Type and made opaque. New functions include:

```
SLprep_new
SLprep_delete
SLprep_set_flags
SLprep_set_comment
SLprep_set_prefix
SLprep_set_exists_hook
SLprep_set_eval_hook
```

If you currently use:

```
SLPreprocess_Type pt;
SLprep_open_prep (&pt);
   .
```

```
          .
        SLprep_close_prep (&pt);
```

Then change it to:

```
        SLprep_Type *pt;
        pt = SLprep_new ();
          .
          .
        SLprep_delete (pt);
```

A.7 Functions dealing with the interpreter C API

- `SLang_pop_double` has been changed to be more like the other `SLang_pop_*` functions. Now, it may be used as:

  ```
        double x;
        if (-1 == SLang_pop_double (&x))
          .
          .
  ```

- All the functions that previously took an "unsigned char" to specify a slang data type have changed to require an `SLtype`. Previously, `SLtype` was typedefed to be an `unsigned char`, but now it is an `int`.
- The `SLang_Class_Type` object is now an opaque type. If you were previously accessing its fields directly, then you will have to change the code to use one of the accessor functions.

A.8 Modules

- In order to support the loading of a module into multiple namespaces, a module's init function may be called more than once. See modules/README for more information.
- The `init_<module>_module` function is no longer supported because it did not support namespaces. Use the `init_<module>_module_ns` function instead.

Appendix B

Copyright

The **S-Lang** library is distributed under the terms of the GNU General Public License.

B.1 The GNU Public License

```
                    GNU GENERAL PUBLIC LICENSE
                       Version 2, June 1991

 Copyright (C) 1989, 1991 Free Software Foundation, Inc.
                       59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble
```

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

```
                    GNU GENERAL PUBLIC LICENSE
   TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION
```

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

```
    a) You must cause the modified files to carry prominent notices
    stating that you changed the files and the date of any change.

    b) You must cause any work that you distribute or publish, that in
    whole or in part contains or is derived from the Program or any
    part thereof, to be licensed as a whole at no charge to all third
    parties under the terms of this License.
```

```
c) If the modified program normally reads commands interactively
when run, you must cause it, when started running for such
interactive use in the most ordinary way, to print or display an
announcement including an appropriate copyright notice and a
notice that there is no warranty (or else, saying that you provide
a warranty) and that users may redistribute the program under
these conditions, and telling the user how to view a copy of this
License.  (Exception: if the Program itself is interactive but
does not normally print such an announcement, your work based on
the Program is not required to print an announcement.)
```

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

```
a) Accompany it with the complete corresponding machine-readable
source code, which must be distributed under the terms of Sections
1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three
years, to give any third party, for a charge no more than your
cost of physically performing source distribution, a complete
machine-readable copy of the corresponding source code, to be
distributed under the terms of Sections 1 and 2 above on a medium
customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer
to distribute corresponding source code.  (This alternative is
allowed only for noncommercial distribution and only if you
received the program in object code or executable form with such
an offer, in accord with Subsection b above.)
```

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus

any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

```
                            NO WARRANTY

  11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY
FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW.  EXCEPT WHEN
OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES
PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED
OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.  THE ENTIRE RISK AS
TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU.  SHOULD THE
PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING,
REPAIR OR CORRECTION.

  12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR
REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES,
INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING
OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED
TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY
YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER
PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE
POSSIBILITY OF SUCH DAMAGES.

                     END OF TERMS AND CONDITIONS
```

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) 19yy  <name of author>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

B.2 The Unicode Inc. Copyright

This software makes use of the Unicode tables published by Unicode, Inc under the following terms:

```
COPYRIGHT AND PERMISSION NOTICE

Copyright (c) 1991-2009 Unicode, Inc. All rights reserved. Distributed
under the Terms of Use in http://www.unicode.org/copyright.html.

Permission is hereby granted, free of charge, to any person
obtaining a copy of the Unicode data files and any associated
documentation (the "Data Files") or Unicode software and any
associated documentation (the "Software") to deal in the Data Files
or Software without restriction, including without limitation the
rights to use, copy, modify, merge, publish, distribute, and/or sell
copies of the Data Files or Software, and to permit persons to whom
the Data Files or Software are furnished to do so, provided that (a)
the above copyright notice(s) and this permission notice appear with
all copies of the Data Files or Software, (b) both the above
copyright notice(s) and this permission notice appear in associated
documentation, and (c) there is clear notice in each modified Data
File or in the Software as well as in the documentation associated
with the Data File(s) or Software that the data or software has been
modified.

THE DATA FILES AND SOFTWARE ARE PROVIDED "AS IS", WITHOUT WARRANTY
OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE
COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR
ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THE DATA FILES OR SOFTWARE.

Except as contained in this notice, the name of a copyright holder
shall not be used in advertising or otherwise to promote the sale,
use or other dealings in these Data Files or Software without prior
written authorization of the copyright holder.
```