LuaService Reference Manual

alpha-2

Generated by Doxygen 1.5.4

Mon Dec 17 13:33:42 2007

Contents

1 LuaService Overview

This project provides a framework for building Windows Service applications in Lua. A Windows Service is a special application that runs at system boot (or on demand) without an interactive user. One use case is to run Xavante automatically.

This project is hosted at LuaForge (`http://www.luaforge.net/`) as `http://luaforge.net/projects/luaservice/` where full source code, documentation, a bug tracker, and so forth can be found.

- See **Using the LuaService Framework** (p. 79) for a description of how to set up a service with LuaService.
- See **Event Flow Overview** (p. 72) for discussion of the general event flow in a service.
- See **Lua Usage Model** (p. 77) for discussion of how Lua fits in the framework.
- See **Building LuaService** (p. 82) for notes on building LuaService and its documentation from source.
- See **Supporting Tools** (p. 83) for some notes on the software tools used.
- LuaService is open source and licensed under the MIT license as is Lua itself. See **License** (p. 85) for details.

Note that LuaService assumes that lua5.1.dll from the Lua Binaries (`http://luaforge.net/projects/luabinaries/`) distribution is located somewhere that the LocalSystem account can find it. The best way to guarantee this is true is to put a copy of lua5.1.dll in the same folder as LuaService.exe.

LuaService has been tested against both Lua 5.1.1 and Lua 5.1.2. The latter is preferred.

2 LuaService File Index

2.1 LuaService File List

Here is a list of all files with brief descriptions:

3 LuaService Page Index

3.1 LuaService Related Pages

Here is a list of all related documentation pages:

4 LuaService File Documentation

4.1 dox/flowdiagrams.dox File Reference

4.2 dox/luamodel.dox File Reference

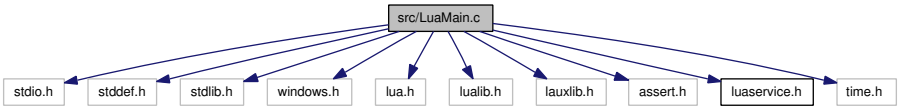
4.3 dox/overview.dox File Reference

4.4 src/LuaMain.c File Reference

Wrap up access to Lua interpretor states.

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <windows.h>
#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>
#include <assert.h>
#include "luaservice.h"
#include <time.h>
```

Include dependency graph for LuaMain.c:



Defines

- #define **fieldint**(f, n)

  Utility macro to add a field to the table at the top of stack.

- #define **fieldstr**(f, s)

  Utility macro to add a field to the table at the top of stack.

- #define **local_getreg**(L, key)

Put a private-keyed registry item on the top of stack.

- #define **local_setreg**(L, key)

  Store the top of stack in a private key in the registry.

Functions

- static int **dbgSleep** (lua_State ∗L)

  Implement the Lua function sleep(ms).

- static int **dbgPrint** (lua_State ∗L)

  Implement the Lua function print(.

- static int **dbgGetCurrentDirectory** (lua_State ∗L)

  Implement the Lua function GetCurrentDirectory().

- static int **dbgStopping** (lua_State ∗L)

  Implement the Lua function stopping().

- static int **dbgTracelevel** (lua_State ∗L)

  Implement the Lua function tracelevel(level).

- static int **dbgGetCurrentConfiguration** (lua_State ∗L)

  Implement the Lua function GetCurrentConfiguration().

- static void **initGlobals** (lua_State ∗L)

  Initialize useful Lua globals.

- static int **pmain** (lua_State ∗L)

  Function called in a protected Lua state.

- static void ∗ **LuaAlloc** (void ∗ud, void ∗ptr, size_t osize, size_t nsize)

  Lua allocator function.

- static int **panic** (lua_State ∗L)

  The panic function for a Lua state.

- **LUAHANDLE LuaWorkerLoad** (**LUAHANDLE** h, const char ∗cmd)

  Create a Lua state with a script loaded.

- **LUAHANDLE LuaWorkerRun** (**LUAHANDLE** h)

Run a pending Lua script.

- void **LuaWorkerCleanup** (**LUAHANDLE** h)
  Clean up after the worker by closing the Lua state.
- char * **LuaResultString** (**LUAHANDLE** h, int item)
  Get a cached worker result item as a string.
- int **LuaResultInt** (**LUAHANDLE** h, int item)
  Get a cached worker result item as an integer.
- char * **LuaResultFieldString** (**LUAHANDLE** h, int item, const char *field)
  Get a field of a cached worker result item as a string.
- int **LuaResultFieldInt** (**LUAHANDLE** h, int item, const char *field)
  Get a field of a cached worker result item as an integer.

Variables

- static const char * **PENDING_WORK** = "Pending Work"
  Private key for a pending compiled but unexecuted Lua chunk.
- static const char * **WORK_RESULTS** = "Work Results"
  Private key for results from a lua chunk.
- static struct luaL_Reg **dbgFunctions** [ ]
  List of Lua callable functions for the service object.

4.4.1 Detailed Description

Wrap up access to Lua interpretor states.

Definition in file **LuaMain.c**.

4.4.2 Define Documentation

4.4.2.1 #define fieldint(f, n)

Value:

```
do{                     \
        lua_pushinteger(L,(n));                 \
        lua_setfield(L,-2,f);                   \
        SvcDebugTrace("  " f ": 0x%x\n", (n));  \
    }while(0)
```

Utility macro to add a field to the table at the top of stack.

Note this assumes that a table is currently at the top of the stack. If that is not true, then this macro probably causes Lua to throw an error.

Parameters:

f String literal field name.

n Integer field value

Definition at line 141 of file LuaMain.c.

Referenced by dbgGetCurrentConfiguration().

4.4.2.2 #define fieldstr(f, s)

Value:

```
do{                     \
        lua_pushstring(L,(s));                  \
        lua_setfield(L,-2,f);                   \
        SvcDebugTraceStr("  " f ": %s\n", (s)); \
    }while(0)
```

Utility macro to add a field to the table at the top of stack.

Note this assumes that a table is currently at the top of the stack. If that is not true, then this macro probably causes Lua to throw an error.

Parameters:

f String literal field name.

s String field value

Definition at line 155 of file LuaMain.c.

Referenced by dbgGetCurrentConfiguration().

4.4.2.3 #define local_getreg(L, key)

Value:

```
do {                    \
                lua_pushlightuserdata(L,(void*)key);    \
                lua_gettable(L,LUA_REGISTRYINDEX);      \
        } while(0)
```

Put a private-keyed registry item on the top of stack.

Retrives the item from the registry keyed by a light user data made from the pointer *key* and pushes it on the stack.

Parameters:

L The Lua state.

key A private key in the form of a pointer to something.

Definition at line 284 of file LuaMain.c.

Referenced by LuaResultFieldInt(), LuaResultFieldString(), LuaResultInt(), LuaResultString(), and pmain().

4.4.2.4 #define local_setreg(L, key)

Value:

```
do {                    \
                lua_pushlightuserdata(L,(void*)key);    \
                lua_insert(L,-2);                       \
                lua_settable(L,LUA_REGISTRYINDEX);      \
    } while(0)
```

Store the top of stack in a private key in the registry.

Stores the top of the stack in the registry keyed by a light user data made from the pointer *key*.

Parameters:

L The Lua state.

key A private key in the form of a pointer to something.

Definition at line 297 of file LuaMain.c.

Referenced by pmain().

4.4.3 Function Documentation

4.4.3.1 static int dbgGetCurrentConfiguration (lua_State * *L*) [static]

Implement the Lua function GetCurrentConfiguration().

Discover some details about the service's configuration as known to the **Service Control Manager** (p. 72) and report them to the debug trace while building a table from them to return.

Parameters:

L Lua state context for the function.

Returns:

The number of values on the Lua stack to be returned to the Lua caller.

Definition at line 171 of file LuaMain.c.

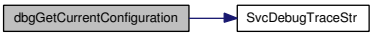
References fieldint, fieldstr, ServiceName, and SvcDebugTraceStr().

```
172 {
173     SC_HANDLE schService;
174     SC_HANDLE schManager;
175     LPQUERY_SERVICE_CONFIG lpqscBuf;
176     LPSERVICE_DESCRIPTION lpqscBuf2;
177     DWORD dwBytesNeeded;
178     const char *name;
179 
180     name = luaL_optstring(L, 1, ServiceName);
181     SvcDebugTraceStr("Get service configuration for %s:\n", name);
182 
183     // Open a handle to the service.
184     schManager = OpenSCManagerA(NULL, NULL, (0
185             |GENERIC_READ
186             |SC_MANAGER_CONNECT
187             |SC_MANAGER_CREATE_SERVICE
188             |SC_MANAGER_ENUMERATE_SERVICE
189     ));
190     if (schManager == NULL)
191         return luaL_error(L, "OpenSCManager failed (%d)", GetLastError());
192     schService = OpenServiceA(schManager, // SCManager database
193             name, // name of service
194             SERVICE_QUERY_CONFIG); // need QUERY access
195     if (schService == NULL) {
196         CloseServiceHandle(schManager);
197         return luaL_error(L, "OpenService failed (%d)", GetLastError());
198     }
199 
200     // Allocate buffers for the configuration information.
201     lpqscBuf = (LPQUERY_SERVICE_CONFIG) LocalAlloc(
202     LPTR, 8192);
203     if (lpqscBuf == NULL) {
204         CloseServiceHandle(schService);
205         CloseServiceHandle(schManager);
206         return luaL_error(L, "Can't allocate lpqscBuf");
207     }
208     lpqscBuf2 = (LPSERVICE_DESCRIPTION) LocalAlloc(
```

```
209     LPTR, 8192);
210     if (lpqscBuf2 == NULL) {
211         LocalFree(lpqscBuf);
212         CloseServiceHandle(schService);
213         CloseServiceHandle(schManager);
214         return luaL_error(L, "Can't allocate lpqscBuf2");
215     }
216
217     // Get the configuration information.
218     if (! QueryServiceConfig(
219             schService,
220             lpqscBuf,
221             8192,
222             &dwBytesNeeded)) {
223         LocalFree(lpqscBuf);
224         LocalFree(lpqscBuf2);
225         CloseServiceHandle(schService);
226         CloseServiceHandle(schManager);
227         return luaL_error(L, "QueryServiceConfig failed (%d)",
228                 GetLastError());
229     }
230     if (! QueryServiceConfig2(
231             schService,
232             SERVICE_CONFIG_DESCRIPTION,
233             (LPBYTE)lpqscBuf2,
234             8192,
235             &dwBytesNeeded)) {
236         LocalFree(lpqscBuf);
237         LocalFree(lpqscBuf2);
238         CloseServiceHandle(schService);
239         CloseServiceHandle(schManager);
240         return luaL_error(L, "QueryServiceConfig2 failed (%d)",
241                 GetLastError());
242     }
243
244     // Build a table of configuration details,
245     // passing them to the trace log along the way
246
247     lua_newtable(L);
248     fieldstr("name", name);
249     fieldint("ServiceType", lpqscBuf->dwServiceType);
250     fieldint("StartType", lpqscBuf->dwStartType);
251     fieldint("ErrorControl", lpqscBuf->dwErrorControl);
252     fieldstr("BinaryPathName", lpqscBuf->lpBinaryPathName);
253     if (lpqscBuf->lpLoadOrderGroup != NULL)
254         fieldstr("LoadOrderGroup", lpqscBuf->lpLoadOrderGroup);
255     if (lpqscBuf->dwTagId != 0)
256         fieldint("TagId", lpqscBuf->dwTagId);
257     if (lpqscBuf->lpDependencies != NULL)
258         fieldstr("Dependencies", lpqscBuf->lpDependencies);
259     if (lpqscBuf->lpServiceStartName != NULL)
260         fieldstr("ServiceStartName", lpqscBuf->lpServiceStartName);
261     if (lpqscBuf2->lpDescription != NULL)
262         fieldstr("Description", lpqscBuf2->lpDescription);
263
264     LocalFree(lpqscBuf);
265     LocalFree(lpqscBuf2);
```

```
266     CloseServiceHandle(schService);
267     CloseServiceHandle(schManager);
268     return 1;
269 }
```

Here is the call graph for this function:



4.4.3.2 static int dbgGetCurrentDirectory (lua_State * *L*) [static]

Implement the Lua function GetCurrentDirectory().

Discover the current directory name and return it to the caller.

Todo

There is a low-probability memory leak here. The buffer used to hold the current directory string came from malloc() and is held across a call to lua_pushlstring() which can potentially throw an error, which will leak the allocated buffer. The other bits of Win32 API wrappers could have similar issues, and should be inspected.

Parameters:

L Lua state context for the function.

Returns:

The number of values on the Lua stack to be returned to the Lua caller.

Definition at line 82 of file LuaMain.c.

```
83 {
84         char *buf = 0;
85         DWORD len = 0;
86         len = GetCurrentDirectoryA(0,NULL);
87         if (!len)
88                 return luaL_error(L, "GetCurrentDirectory failed (%d)", GetLastError());
89         buf = malloc(len+1);
90         if (!buf)
91                 return luaL_error(L,"GetCurrentDirectory can't allocate %ld chars", len);
92         GetCurrentDirectoryA(len+1, buf);
93         lua_pushlstring(L, buf, len);
94         free(buf);
95         return 1;
96 }
```

4.4.3.3 static int dbgPrint (lua_State * *L*) [static]

Implement the Lua function print(.

..).

Construct a message from all the arguments to print(), passing each through the global function tostring() make certain they are strings, and separating them with tab characters. The message is ultimately passed to the Windows service OutputDebugString() for display in a debugger or debug message logger.

Parameters:

L Lua state context for the function.

Returns:

The number of values on the Lua stack to be returned to the Lua caller.

Definition at line 47 of file LuaMain.c.

```
48 {
49         luaL_Buffer b;
50         int n = lua_gettop(L); /* number of arguments */
51         int i;
52         lua_getglobal(L, "tostring");
53         luaL_buffinit(L, &b);
54         for (i=1; i<=n; i++) {
55                 lua_pushvalue(L, n+1); /* b tostring */
56                 lua_pushvalue(L, i);   /* b tostring argi */
57                 lua_call(L, 1, 1);     /* b tostring(argi) */
58                 luaL_addvalue(&b);     /* b */
59                 if (i<n)
60                         luaL_addchar(&b, '\t');
61         }
62         luaL_pushresult(&b);
63         OutputDebugStringA(lua_tostring(L, -1));               //fputs(s, stdout);
64         lua_pop(L,1);
65         return 0;
66 }
```

4.4.3.4 static int dbgSleep (lua_State * *L*) [static]

Implement the Lua function sleep(ms).

Call the Windows Sleep() API to delay thread execution for approximately *ms* ms.

Parameters:

L Lua state context for the function.

Returns:

The number of values on the Lua stack to be returned to the Lua caller.

Definition at line 26 of file LuaMain.c.

```
27 {
28         int t;
29         t = luaL_checkint(L,1);
30         if (t < 0) t = 0;
31         Sleep((DWORD)t);
32         return 0;
33 }
```

4.4.3.5 static int dbgStopping (lua_State * *L*) [static]

Implement the Lua function stopping().

Poll the flag used by the request handler thread to signal that the service should politely halt soon. Returns true if it should shut down, false otherwise.

Parameters:

L Lua state context for the function.

Returns:

The number of values on the Lua stack to be returned to the Lua caller.

Definition at line 108 of file LuaMain.c.

References ServiceStopping.

```
109 {
110         lua_pushboolean(L,ServiceStopping);
111         return 1;
112 }
```

4.4.3.6 static int dbgTracelevel (lua_State * *L*) [static]

Implement the Lua function tracelevel(level).

Control the verbosity of trace output to the debug console.

If level is passed, sets the trace level accordingly. Regardless, it returns the current trace level.

Parameters:

L Lua state context for the function.

Returns:

The number of values on the Lua stack to be returned to the Lua caller.

Definition at line 126 of file LuaMain.c.

References SvcDebugTraceLevel.

```
127 {
128         SvcDebugTraceLevel = luaL_optint(L,-1,SvcDebugTraceLevel);
129         lua_pushinteger(L,SvcDebugTraceLevel);
130         return 1;
131 }
```

4.4.3.7 static void initGlobals (lua_State * *L*) [static]

Initialize useful Lua globals.

The following globals are created in the Lua state:

- service – a table for the service
- service.name – the service name known to the SCM
- service.filename – a string containing the filename of the service program
- service.path – the path of the service folder
- service.sleep(ms) – a function to sleep for *ms* ms
- service.print(...) – like standalone Lua's print(), but with OutputDebugString()
- print – a copy of service.print
- sleep – a copy of service.sleep

The global table package has its path and cpath replaced to reference only the service folder itself.

Parameters:

L Lua state context to get the globals.

Definition at line 336 of file LuaMain.c.

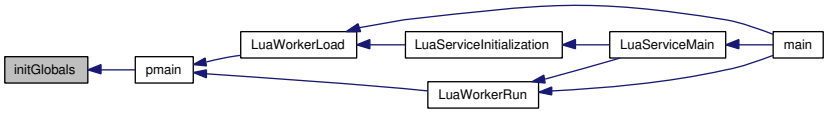
References dbgFunctions, and ServiceName.

Referenced by pmain().

```
337 {
338         char szPath[MAX_PATH + 1];
339         char *cp;
340 
341         lua_newtable(L);
342         GetModuleFileName(GetModuleHandle(NULL), szPath, MAX_PATH);
```

```
343         lua_pushstring(L,szPath);
344         lua_setfield(L,-2,"filename");
345         cp = strrchr(szPath, '\\');
346         if (cp) {
347                 cp[1] = '\0';
348                 lua_pushstring(L,szPath);
349                 lua_setfield(L,-2,"path");
350         }
351         lua_pushstring(L,ServiceName);
352         lua_setfield(L,-2,"name");
353         // define a few useful utility functions
354         luaL_register(L, NULL, dbgFunctions);
355         lua_setglobal(L, "service");
356 #if 0
357         luaL_dostring(L,
358                         "package.path = string.replace([[@?.lua;@?\\init.lua]],'%@',service.path
359                         "package.cpath = string.replace([[@?.dll;@loadall.dll]],'%@',service.pat
360                         );
361 #endif
362         luaL_dostring(L,
363                         "print = service.print\n"
364                         "sleep = service.sleep\n");
365 }
```

Here is the caller graph for this function:



4.4.3.8 static void* LuaAlloc (void * *ud*, void * *ptr*, size_t *osize*, size_t *nsize*) [static]

Lua allocator function.

Borrowed verbatim from the Lua sources, found in lauxlib.c.

Needed to support a direct creation of a Lua state that cannot refer in any way to stdio handles.

Manage a memory block. If *nsize* is non-zero, it will return a pointer to allocated memory that must be passed back here to be freed. If *osize* is non-zero, *ptr* must be non-null and the block it points to will be either freed or reallocated depending on the value of *nsize*.

Note:

This is a good place to introduce memory performance hooks for a Lua state in some future version.

Parameters:

ud Opaque token provided when the Lua state was created.

ptr Pointer to any existing memory for this transaction.

osize Size of the existing memory block.

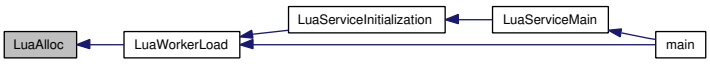
nsize Size of the memory block needed.

Definition at line 508 of file LuaMain.c.

Referenced by LuaWorkerLoad().

```
508                                                                                 {
509         void *retv = NULL;
510         (void)ud;
511         (void)osize;
512         if (nsize == 0) {
513                 free(ptr);
514                 retv = NULL;
515         }
516 #ifdef USE_ONLY_MALLOC
517         else if (osize >= nsize)
518                 retv = ptr;
519         else {
520                 void *p = malloc(nsize);
521                 if (!p)
522                         retv = NULL;
523                 else {
524                         memcpy(p,ptr,osize);
525                         retv = p;
526                 }
527         }
528 #else
529         else
530                 retv = realloc(ptr, nsize);
531 #endif
532 //#define LOG_ALLOCATIONS
533 #ifdef LOG_ALLOCATIONS
534         {
535                 //static clock_t basetime;
536                 FILE *fp = fopen("alloc.log", "at");
537                 assert(fp);
538                 //if (basetime == 0)
539                 //      basetime = clock();
540                 fprintf(fp, "%ld %p %d %d %p\n", clock(), ptr, osize, nsize,
541                                 retv);
542                 fclose(fp);
543         }
544 #endif
545         return retv;
546 }
```

Here is the caller graph for this function:



4.4.3.9 int LuaResultFieldInt (LUAHANDLE *h*, int *item*, const char * *field*)

Get a field of a cached worker result item as an integer.

Parameters:

h An opaque handle returned by a previous call to **LuaWorkerRun()** (p. 21).

item The index of the result item to retrieve. The first result is index 1, consistent with Lua counting.

field The name of the field to retrieve.

Returns:

The integer value or 0 if the field or item doesn't exist or can't be converted to a number.

Definition at line 754 of file LuaMain.c.

References local_getreg, and WORK_RESULTS.

Referenced by main().

```
755 {
756         int ret = 0;
757         lua_State *L = (lua_State*)h;
758         if (!h) return 0;
759         local_getreg(L,WORK_RESULTS);   // table
760         if (lua_type(L,-1) != LUA_TTABLE) {
761                 lua_pop(L,1);
762                 return 0;
763         }
764         lua_rawgeti(L,-1,item);                 // table itemtable
765         if (lua_type(L,-1) != LUA_TTABLE) {
766                 lua_pop(L,2);
767                 return 0;
768         }
769         lua_getfield(L,-1,field);               // table itemtable fieldvalue
770         ret = (int)lua_tointeger(L,-1);
771         lua_pop(L,3);
772         return ret;
773 }
```

Here is the caller graph for this function:

4.4.3.10 char* LuaResultFieldString (LUAHANDLE *h*, int *item*, const char * *field*)

Get a field of a cached worker result item as a string.

Note:

The string returned came from strdup(), and must be freed by the caller.

Parameters:

h An opaque handle returned by a previous call to **LuaWorkerRun()** (p. 21).

item The index of the result item to retrieve. The first result is index 1, consistent with Lua counting.

field The name of the field to retrieve.

Returns:

The string value (from strdup()) or NULL if the field or item doesn't exist or can't be converted to a string.

Definition at line 720 of file LuaMain.c.

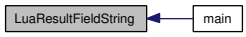
References local_getreg, and WORK_RESULTS.

Referenced by main().

```
721 {
722         char *ret = NULL;
723         lua_State *L = (lua_State*)h;
724         if (!h) return NULL;
725         local_getreg(L,WORK_RESULTS);   // table
726         if (lua_type(L,-1) != LUA_TTABLE) {
727                 lua_pop(L,1);
728                 return NULL;
729         }
730         lua_rawgeti(L,-1,item);                 // table itemtable
731         if (lua_type(L,-1) != LUA_TTABLE) {
732                 lua_pop(L,2);
733                 return NULL;
734         }
735         lua_getfield(L,-1,field);               // table itemtable fieldvalue
736         ret = (char *)lua_tostring(L,-1);
737         if (ret)
```

```
738                 ret = strdup(ret);
739         lua_pop(L,3);
740         return ret;
741 }
```

Here is the caller graph for this function:



4.4.3.11 int LuaResultInt (LUAHANDLE *h*, int *item*)

Get a cached worker result item as an integer.

Parameters:

h An opaque handle returned by a previous call to **LuaWorkerRun()** (p. 21).

item The index of the result item to retrieve. The first result is index 1, consistent with Lua counting.

Returns:

The integer value or 0 if the item doesn't exist or can't be converted to a number.

Definition at line 692 of file LuaMain.c.

References local_getreg, and WORK_RESULTS.

```
693 {
694         int ret = 0;
695         lua_State *L = (lua_State*)h;
696         if (!h) return 0;
697         local_getreg(L,WORK_RESULTS);
698         if (lua_type(L,-1) != LUA_TTABLE) {
699                 lua_pop(L,1);
700                 return 0;
701         }
702         lua_rawgeti(L,-1,item);
703         ret = (int)lua_tointeger(L,-1);
704         lua_pop(L,2);
705         return ret;
706 }
```

4.4.3.12 char* LuaResultString (LUAHANDLE *h*, int *item*)

Get a cached worker result item as a string.

Note:

The string returned came from strdup(), and must be freed by the caller.

Parameters:

h An opaque handle returned by a previous call to **LuaWorkerRun()** (p. 21).

item The index of the result item to retrieve. The first result is index 1, consistent with Lua counting.

Returns:

The string value (from strdup()) or NULL if the item doesn't exist or can't be converted to a string.

Definition at line 664 of file LuaMain.c.

References local_getreg, and WORK_RESULTS.

```
665 {
666         char *ret = NULL;
667         lua_State *L = (lua_State*)h;
668         if (!h) return NULL;
669         local_getreg(L,WORK_RESULTS);
670         if (lua_type(L,-1) != LUA_TTABLE) {
671                 lua_pop(L,1);
672                 return NULL;
673         }
674         lua_rawgeti(L,-1,item);
675         ret = (char *)lua_tostring(L,-1);
676         if (ret)
677                 ret = strdup(ret);
678         lua_pop(L,2);
679         return ret;
680 }
```

4.4.3.13 void LuaWorkerCleanup (LUAHANDLE *h*)

Clean up after the worker by closing the Lua state.

Parameters:

h An opaque handle returned by a previous call to **LuaWorkerRun()** (p. 21).

Definition at line 646 of file LuaMain.c.

Referenced by LuaServiceMain(), and main().

```
647 {
648         lua_State *L=(lua_State*)h;
649         if (h)
650                 lua_close(L);
651 }
```

Here is the caller graph for this function:

4.4.3.14 LUAHANDLE LuaWorkerLoad (LUAHANDLE *h*, const char * *cmd*)

Create a Lua state with a script loaded.

Creates a new Lua state, initializes it with built-in modules and global variables, then loads the specified script or command.

If all is well, the compiled but as yet unexecuted main block of the script is cached in the Lua Registry at index PENDING_WORK (a light user data made from the address of this function).

Parameters:

h Opaque handle to the Lua state to use, or NULL to create a new state.

cmd Script or statement to load

Returns:

An opaque handle identifying the created Lua state.

Definition at line 590 of file LuaMain.c.

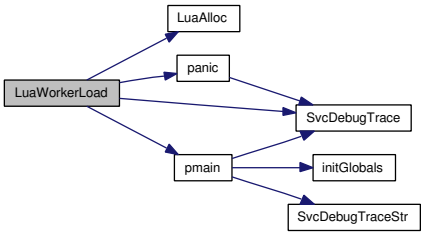
References LuaAlloc(), panic(), pmain(), and SvcDebugTrace().

Referenced by LuaServiceInitialization(), and main().

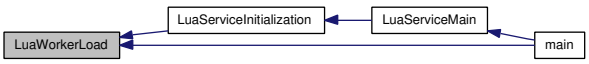
```
591 {
592         int status;
593         lua_State *L=(lua_State*)h;
594 
595         if (!h) {
596 #if 0
597                 L = luaL_newstate();
598 #else
599                 L = lua_newstate(LuaAlloc, NULL);
600 #endif
601                 assert(L);
602                 lua_atpanic(L, &panic);
603         }
604         status = lua_cpcall(L, &pmain, (void*)cmd);
605         if (status) {
606                 SvcDebugTrace("Load script cpcall status %d", status);
607                 SvcDebugTrace((char *)lua_tostring(L,-1),0);
608                 //return NULL;
```

```
609         } else {
610                 SvcDebugTrace("Script loaded ok", 0);
611         }
612         return (void *)L;
613 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



4.4.3.15 LUAHANDLE LuaWorkerRun (LUAHANDLE *h*)

Run a pending Lua script.

The script must have been previously loaded and saved in the Lua registry.

Parameters:

h An opaque handle returned by a previous call to **LuaWorkerRun()** (p. 21).

Returns:

An opaque handle identifying the created Lua state.

Definition at line 622 of file LuaMain.c.

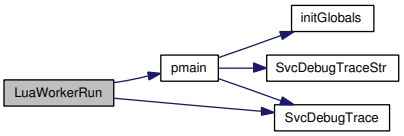
References pmain(), and SvcDebugTrace().

Referenced by LuaServiceMain(), and main().

```
623 {
624         int status;
625         lua_State *L=(lua_State*)h;
```

```
626
627         if (!h) {
628                 SvcDebugTrace("No existing lua state!!!", 0);
629                 return NULL;
630                 //L = lua_open();
631         }
632         status = lua_cpcall(L, &pmain, NULL);
633         if (status) {
634                 SvcDebugTrace("Run script cpcall status %d", status);
635                 SvcDebugTrace((char *)lua_tostring(L,-1),0);
636         } else {
637                 SvcDebugTrace("Script succeeded", 0);
638         }
639         return (void *)L;
640 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

4.4.3.16 static int panic (lua_State * *L*) [static]

The panic function for a Lua state.

This function is called as a last resort if an error is thrown in an unprotected Lua context. It has access to an error message on the top of the Lua stack, but should probably refrain from anything that could throw additional errors, say be attempting to allocate additional memory.

If it returns, Lua is going to call exit(EXIT_FAILURE). Since this is almost certainly running in a thread other than the main thread and, worse, under the supervision of the **Service Control Manager** (p. 72) that could result in the SCM becoming confused about the current state of the service.

To prevent SCM confusion, this function simply calls ExitThread() to kill the current thread without necessarily killing the whole process.

Todo

Should **panic()** (p. 22) also tell the SCM SERVICE_STOPPED?

Definition at line 568 of file LuaMain.c.

References SvcDebugTrace().

Referenced by LuaWorkerLoad().

```
568                                 {
569   (void)L;  /* to avoid warnings */
570   SvcDebugTrace("PANIC: unprotected error in call to Lua API...",0);
571   SvcDebugTrace(lua_tostring(L, -1), 0);
572   ExitThread(EXIT_FAILURE);
573   return 0;
574 }
```

Here is the call graph for this function:

Here is the caller graph for this function:

4.4.3.17 static int pmain (lua_State * *L*) [static]

Function called in a protected Lua state.

Initialize the Lua state if the global service has not been defined, then do something. This function assumes that the caller is living within the constraints of lua_cpcall(), meaning that it is passed exactly one argument on the Lua stack which is a light user-data wrapping an opaque pointer, and it isn't allowed to return anything.

That pointer must be either NULL or a pointer to a C string naming the script file or code fragment to load and execute in the Lua context.

If a NULL is passed, the private registry value PENDING_WORK is pushed and called. Any results are collected in a table and stored in the registry at the private key WORK_-RESULTS, and the value at PENDING_WORK is freed to the the garbage collector.

If a non-null string is passed, it is loaded (but not called) and stored in the registry at the private key PENDING_WORK. Any prevous work results are released to the the garbage collector.

A string is assumed to be a file name, but a future version probably should allow for a litteral script as well.

Parameters:

L Lua state context for the function.

Returns:

The number of values on the Lua stack to be returned to the Lua caller.

Note:

The script file name is always relative to the service folder. This protects against substitution of the script by a third party, at least to some degree.

Definition at line 394 of file LuaMain.c.

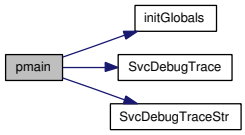
References initGlobals(), local_getreg, local_setreg, PENDING_WORK, SvcDebugTrace(), SvcDebugTraceStr(), and WORK_RESULTS.

Referenced by LuaWorkerLoad(), and LuaWorkerRun().

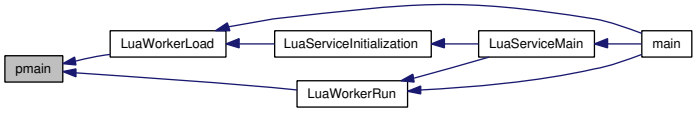
```
395 {
396         char szPath[MAX_PATH+1];
397         char *cp;
398         char *arg;
399         int status;
400 
401         arg = (char *)lua_touserdata(L,-1);
402         lua_getglobal(L, "service");
403         if (!lua_toboolean(L,-1)) {
404                 lua_gc(L, LUA_GCSTOP, 0); /* stop gc during initialization */
405                 luaL_openlibs(L); /* open libraries */
406                 initGlobals(L);
407                 lua_gc(L, LUA_GCRESTART, 0);
408         }
409         lua_pop(L,2); /* don't need the light userdata or service objects on the stack */
410         if (arg) {
411                 // load but don't call the code
412 
413                 // first, release any past results
414                 lua_pushnil(L);
415                 local_setreg(L,WORK_RESULTS);
416                 lua_pushnil(L);
417                 local_setreg(L,PENDING_WORK);
418 
424                 GetModuleFileName(GetModuleHandle(NULL), szPath, MAX_PATH);
425                 cp = strrchr(szPath, '\\');
426                 if (cp) {
427                         cp[1] = '\0';
428                         if ((cp - szPath) + strlen(arg) + 1 > MAX_PATH)
429                                 return luaL_error(L, "Script name '%s%s' too long", szPath, arg)
430                         strcpy(cp+1, arg);
431                 } else {
```

```
432                         return luaL_error(L, "Module name '%s' isn't fully qualified", szPath);
433                 }
434                 SvcDebugTraceStr("Script: %s\n", szPath);
435                 status = luaL_loadfile(L,szPath);
436                 if (status) {
437                         return luaL_error(L,"%s\n",lua_tostring(L,-1));
438                 }
439                 local_setreg(L,PENDING_WORK);
440                 return 0;
441         } else {
442                 int n;
443                 int i;
444                 int results;
445
446                 // call pending code and save any results
447
448                 // first, release any past results
449                 lua_pushnil(L);
450                 local_setreg(L,WORK_RESULTS);
451
452                 lua_createtable(L, 5, 0);
453                 results = lua_gettop(L);
454 //#define NO_DEBUG_TRACEBACK
455 #ifdef NO_DEBUG_TRACEBACK
456                 n = lua_gettop(L);
457                 local_getreg(L,PENDING_WORK);
458                 status = lua_pcall(L,0,LUA_MULTRET,0);
459 #else
460                 lua_getglobal(L,"debug");
461                 // debug
462                 lua_getfield(L, -1, "traceback"); // debug debug.traceback
463                 lua_remove(L, -2); // debug.traceback
464                 n = lua_gettop(L);
465                 local_getreg(L,PENDING_WORK); // debug.traceback function
466                 if (lua_type(L,-1) != LUA_TFUNCTION)
467                         return luaL_error(L,"No pending work function to run");
468                 status = lua_pcall(L, 0, LUA_MULTRET, -2); // debug.traceback ...
469 #endif
470                 if (status) {
471                         return luaL_error(L,"%s\n",lua_tostring(L,-1));
472                 }
473                 SvcDebugTrace("Saved work result count: %d", lua_gettop(L) - n);
474                 for (i=lua_gettop(L); i>n; --i) {
475                         SvcDebugTraceStr("item: %s", lua_typename(L,lua_type(L,-1)));
476                         lua_rawseti(L,results,i-n);
477                 }
478                 assert(lua_gettop(L) == n);
479                 if (lua_gettop(L) != results)
480                         lua_settop(L, results);
481                 assert(lua_type(L,-1) == LUA_TTABLE);
482                 local_setreg(L,WORK_RESULTS);
483                 return 0;
484         }
485 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



4.4.4 Variable Documentation

4.4.4.1 struct luaL_Reg dbgFunctions[ ] [static]

Initial value:

```
 {
                {"sleep", dbgSleep },
                {"print", dbgPrint },
                {"stopping", dbgStopping },
                {"tracelevel", dbgTracelevel },
                {"GetCurrentDirectory", dbgGetCurrentDirectory},
                {"GetCurrentConfiguration", dbgGetCurrentConfiguration},
                {NULL, NULL},
}
```

List of Lua callable functions for the service object.

Each entry creates a single function in the service object.

Definition at line 308 of file LuaMain.c.

Referenced by initGlobals().

4.4.4.2 const char* PENDING_WORK = "Pending Work" [static]

Private key for a pending compiled but unexecuted Lua chunk.

Definition at line 272 of file LuaMain.c.

Referenced by pmain().

4.4.4.3 const char* WORK_RESULTS = "Work Results" [static]

Private key for results from a lua chunk.

Definition at line 274 of file LuaMain.c.

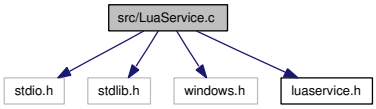
Referenced by LuaResultFieldInt(), LuaResultFieldString(), LuaResultInt(), LuaResultString(), and pmain().

4.5 src/LuaService.c File Reference

Windows Service framework and startup.

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include "luaservice.h"
```

Include dependency graph for LuaService.c:



Functions

- void **SvcDebugTrace** (LPCSTR fmt, DWORD dw)

  Output a debug string.

- void **SvcDebugTraceStr** (LPCSTR fmt, LPCSTR s)

  Output a debug string.

- void WINAPI **LuaServiceCtrlHandler** (DWORD Opcode)

  Service Control Handler.

- DWORD **LuaServiceInitialization** (DWORD argc, LPTSTR *argv, **LUAHANDLE** *ph, DWORD *perror)

  Stup initialization function.

- BOOL **LuaServiceSetStatus** (DWORD dwCurrentState, DWORD dwCheckPoint, DWORD dwWaitHint)
- void WINAPI **LuaServiceMain** (DWORD argc, LPTSTR *argv)

  Service Main function.

- int **main** (int argc, char ∗argv[ ])

  Process entry point.

Variables

- const char ∗ **ServiceName** = "LuaService"

  Service name.

- const char ∗ **ServiceScript** = "service.lua"

  Service launcher script.

- SERVICE_STATUS **LuaServiceStatus**

  Current service status.

- SERVICE_STATUS_HANDLE **LuaServiceStatusHandle**

  Handle to the SCM for the running service to report status.

- HANDLE **ServiceWorkerThread**

  Handle to the thread executing ServiceMain().

- int **SvcDebugTraceLevel** = 0

  Trace level.

- volatile int **ServiceStopping** = 0

  Service Stopping Flag.

4.5.1 Detailed Description

Windows Service framework and startup.

Author:

Ross Berteig
Cheshire Engineering Corp.

Todo

Supporting service PAUSE and CONTINUE control request will require some effort beyond the bare framework guarded by the undefined macro LUASERVICE_CAN_PAUSE_CONTINUE. At minimum, some mechanism must be provided for the Lua side to become aware of the request and actually pause; presumably an Event could be waited on to implement the pause, and signaled to implement continue. However, since we assume that the Lua interpreter itself is not built for threading, we don't have a good means to asynchronously notify the Lua code of the pause request in the first place, which would imply that the Lua code is constantly polling.

Definition in file **LuaService.c**.

4.5.2 Function Documentation

4.5.2.1 void WINAPI LuaServiceCtrlHandler (DWORD *Opcode*)

Service Control Handler.

Called in the main thread when the SCM needs to deliver a status or control request to the service.

Call Context:

Service main thread

Parameters:

Opcode The control operation to handle.

See also:

ssSvc

Definition at line 205 of file LuaService.c.

References LuaServiceStatus, LuaServiceStatusHandle, ServiceStopping, ServiceWorkerThread, and SvcDebugTrace().

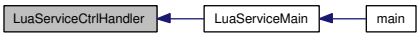
Referenced by LuaServiceMain().

```
206 {
207     DWORD status;
208 
209     SvcDebugTrace("Entered LuaServiceCtrlHandler(%d)\n", Opcode);
210     switch (Opcode) {
211 #ifdef LUASERVICE_CAN_PAUSE_CONTINUE
212     case SERVICE_CONTROL_PAUSE:
213     // Do whatever it takes to pause here.
```

```
214     LuaServiceStatus.dwCurrentState = SERVICE_PAUSED;
215     break;
216
217     case SERVICE_CONTROL_CONTINUE:
218     // Do whatever it takes to continue here.
219     LuaServiceStatus.dwCurrentState = SERVICE_RUNNING;
220     break;
221 #endif
222     case SERVICE_CONTROL_STOP:
223         // Do whatever it takes to stop here.
224         SvcDebugTrace("Telling service to stop\n", 0);
225         ServiceStopping = 1;
226         LuaServiceStatus.dwWin32ExitCode = 0;
227         LuaServiceStatus.dwCurrentState = SERVICE_STOP_PENDING;
228         LuaServiceStatus.dwCheckPoint = 0;
229         LuaServiceStatus.dwWaitHint = 25250;
230
231         if (!SetServiceStatus(LuaServiceStatusHandle, &LuaServiceStatus)) {
232             status = GetLastError();
233             SvcDebugTrace("SetServiceStatus error %ld\n", status);
234         }
235         if (ServiceWorkerThread != NULL) {
236             SvcDebugTrace("Waiting 25 s for worker to stop\n", 0);
237             WaitForSingleObject(ServiceWorkerThread, 25000);
238             CloseHandle(ServiceWorkerThread);
239         }
240         LuaServiceStatus.dwCurrentState = SERVICE_STOPPED;
241         if (!SetServiceStatus(LuaServiceStatusHandle, &LuaServiceStatus)) {
242             status = GetLastError();
243             SvcDebugTrace("SetServiceStatus error %ld\n", status);
244         }
245
246         SvcDebugTrace("Leaving Service\n", 0);
247         return;
248
249     case SERVICE_CONTROL_INTERROGATE:
250         // Fall through to send current status.
251         break;
252
253     default:
254         SvcDebugTrace("Unrecognized opcode %ld\n", Opcode);
255     }
256
257     // Send current status.
258     if (!SetServiceStatus(LuaServiceStatusHandle, &LuaServiceStatus)) {
259         status = GetLastError();
260         SvcDebugTrace("SetServiceStatus error %ld\n", status);
261     }
262     return;
263 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



4.5.2.2 DWORD LuaServiceInitialization (DWORD *argc*, LPTSTR * *argv*, LUAHANDLE * *ph*, DWORD * *perror*)

Stup initialization function.

Initialize Lua state then load and compile our script. The script to run is specified by the `script` field in the table returned by init.lua.

Parameters:

argc Count of arguments passed to the service program by the SCM.

argv Array of argument strings passed to the service program by the SCM.

ph Pointer to a LUAHANDLE that will be written with the handle of an initialized Lua state that has all globals loaded and the service's main script parsed and loaded.

perror Pointer to a DWORD to fill with the Win32 error code that relates to initialization failure, if initialization failed. This value will be passed to the SCM for logging on failure.

Returns:

Zero on success, non-zero exit status on failure. This value will be passed to the SCM for logging on failure.

Definition at line 282 of file LuaService.c.

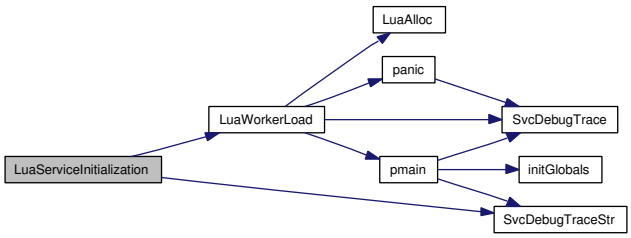
References LuaWorkerLoad(), ServiceScript, ServiceWorkerThread, and SvcDebugTraceStr().

Referenced by LuaServiceMain().

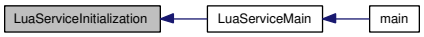
```
284 {
285     if (!DuplicateHandle(GetCurrentProcess(), GetCurrentThread(),
286             GetCurrentProcess(), &ServiceWorkerThread, 0,
287             FALSE,
288             DUPLICATE_SAME_ACCESS)) {
289         *perror = GetLastError();
290         *ph = NULL;
291         return TRUE;
292     }
293     SvcDebugTraceStr("Load LuaService script %s\n", ServiceScript);
294     *ph = LuaWorkerLoad(NULL, ServiceScript);
295     //LuaWorkerSetArgs(argc, argv);
```

```
296     *perror = 0;
297     return NO_ERROR;
298 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



4.5.2.3 void WINAPI LuaServiceMain (DWORD *argc*, LPTSTR * *argv*)

Service Main function.

The entry point of the service's primary worker thread. Since this thread was created by system library code, it apparently has not had the CRT completely initialized.

Todo

Should LuaService push its Lua implementation into a second worker thread that has its CRT properly initialized by using _beginthreadex() to create it instead of CreateThread()?

Call Context:

Service worker thread

Parameters:

argc The count of arguments.

argv The list of arguments.

See also:

Service Application Threads (p. 72)

Definition at line 327 of file LuaService.c.

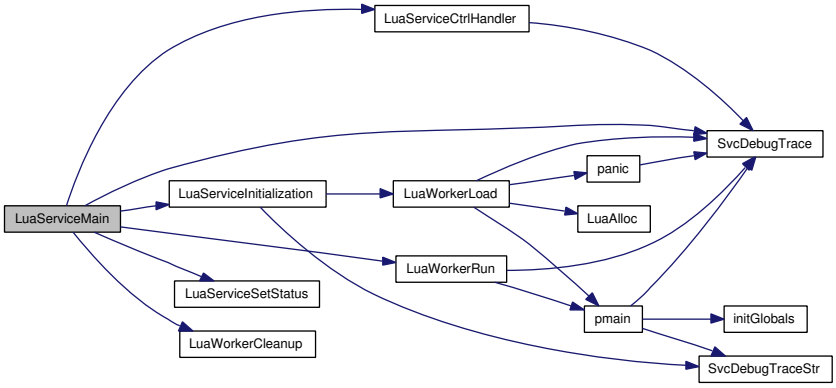
References LuaServiceCtrlHandler(), LuaServiceInitialization(), LuaServiceSetStatus(), LuaServiceStatus, LuaServiceStatusHandle, LuaWorkerCleanup(), LuaWorkerRun(), ServiceName, and SvcDebugTrace().

Referenced by main().

```
328 {
329     DWORD status = 0;
330     DWORD specificError = 0;
331     LUAHANDLE wk = 0;
332 
333     SvcDebugTrace("Entered LuaServiceMain\n", 0);
334 
335     LuaServiceStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS; // SERVICE_WIN32;
336     LuaServiceStatus.dwCurrentState = SERVICE_START_PENDING;
337     LuaServiceStatus.dwControlsAccepted = (0 | SERVICE_ACCEPT_STOP
338                 //| SERVICE_ACCEPT_SHUTDOWN
339 #ifdef LUASERVICE_CAN_PAUSE_CONTINUE
340                 | SERVICE_ACCEPT_PAUSE_CONTINUE
341 #endif
342                 );
343     LuaServiceStatus.dwWin32ExitCode = 0;
344     LuaServiceStatus.dwServiceSpecificExitCode = 0;
345     LuaServiceStatus.dwCheckPoint = 0;
346     LuaServiceStatus.dwWaitHint = 0;
347 
348     LuaServiceStatusHandle = RegisterServiceCtrlHandler(
349             ServiceName,
350             LuaServiceCtrlHandler);
351 
352     if (LuaServiceStatusHandle == (SERVICE_STATUS_HANDLE)0) {
353         SvcDebugTrace("RegisterServiceCtrlHandler failed %d\n",
354                 GetLastError());
355         return;
356     }
357 
358     // Initialization code goes here.
359     LuaServiceSetStatus(SERVICE_START_PENDING, 0, 5000);
360     status = LuaServiceInitialization(argc, (char **)argv, &wk,
361             &specificError);
362     if (status != NO_ERROR) {
363         // Handle error condition
364         LuaServiceStatus.dwCurrentState = SERVICE_STOPPED;
365         LuaServiceStatus.dwCheckPoint = 0;
366         LuaServiceStatus.dwWaitHint = 0;
367         LuaServiceStatus.dwWin32ExitCode = status;
368         LuaServiceStatus.dwServiceSpecificExitCode = specificError;
369 
370         SvcDebugTrace("LuaServiceInitialization exitCode %ld\n", status);
371         SvcDebugTrace("LuaServiceInitialization specificError %ld\n",
372                 specificError);
373 
374         SetServiceStatus(LuaServiceStatusHandle, &LuaServiceStatus);
375         return;
376     }
```

```
377
378     // Initialization complete - report running status.
379     if (!LuaServiceSetStatus(SERVICE_RUNNING, 0, 0)) {
380         status = GetLastError();
381         SvcDebugTrace("SetServiceStatus error %ld\n", status);
382     }
383
384     // do the work of the service by running the loaded script.
385     wk = LuaWorkerRun(wk);
386     LuaWorkerCleanup(wk);
387
388     // we get here only if the script itself returned.
389     SvcDebugTrace("Returning to the Main Thread \n", 0);
390     return;
391 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

4.5.2.4 BOOL LuaServiceSetStatus (DWORD *dwCurrentState*, DWORD *dwCheckPoint*, DWORD *dwWaitHint*)

Definition at line 300 of file LuaService.c.

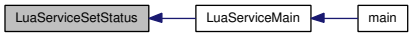
References LuaServiceStatus, and LuaServiceStatusHandle.

Referenced by LuaServiceMain().

```
302 {
```

```
303     LuaServiceStatus.dwCurrentState = dwCurrentState;
304     LuaServiceStatus.dwCheckPoint = dwCheckPoint;
305     LuaServiceStatus.dwWaitHint = dwWaitHint;
306     return SetServiceStatus(LuaServiceStatusHandle, &LuaServiceStatus);
307 }
```

Here is the caller graph for this function:



4.5.2.5 int main (int *argc*, char * *argv*[ ])

Process entry point.

Invoked when the process starts either by a user at a command prompt to setup or control the service, or by the Service Control Manager to start the service.

To Distinguish between the three kinds of service-related programs (the service program, the service control program, and the service configuration program) that we can call StartServiceCtrlDispatcher() early on and use its success or failure to connect to the SCM as an indication of the calling context. If it succeeds, then the process was started by the SCM and is the service program. If it fails with the specific error code ERROR_FAILED_SERVICE_CONTROLLER_CONNECT, then it is not the service program, and it can depend on its command line to distinguish control from configuration. If any other error is returned, then it might have been a service program, but something is so horribly wrong that the service cannot start.

Todo

We could also support running our service thread interactively to support easier debugging. If that is done, then we should consider making **SvcDebugTrace()** (p. 37) and friends, as well as the implementation of service.print() for Lua write to stdout rather than OutputDebugString().

Call Context:

Service, Configuration, Control

Parameters:

argc The count of arguments.

argv The list of arguments.

Returns:

The ANSI C process exit status.

See also:

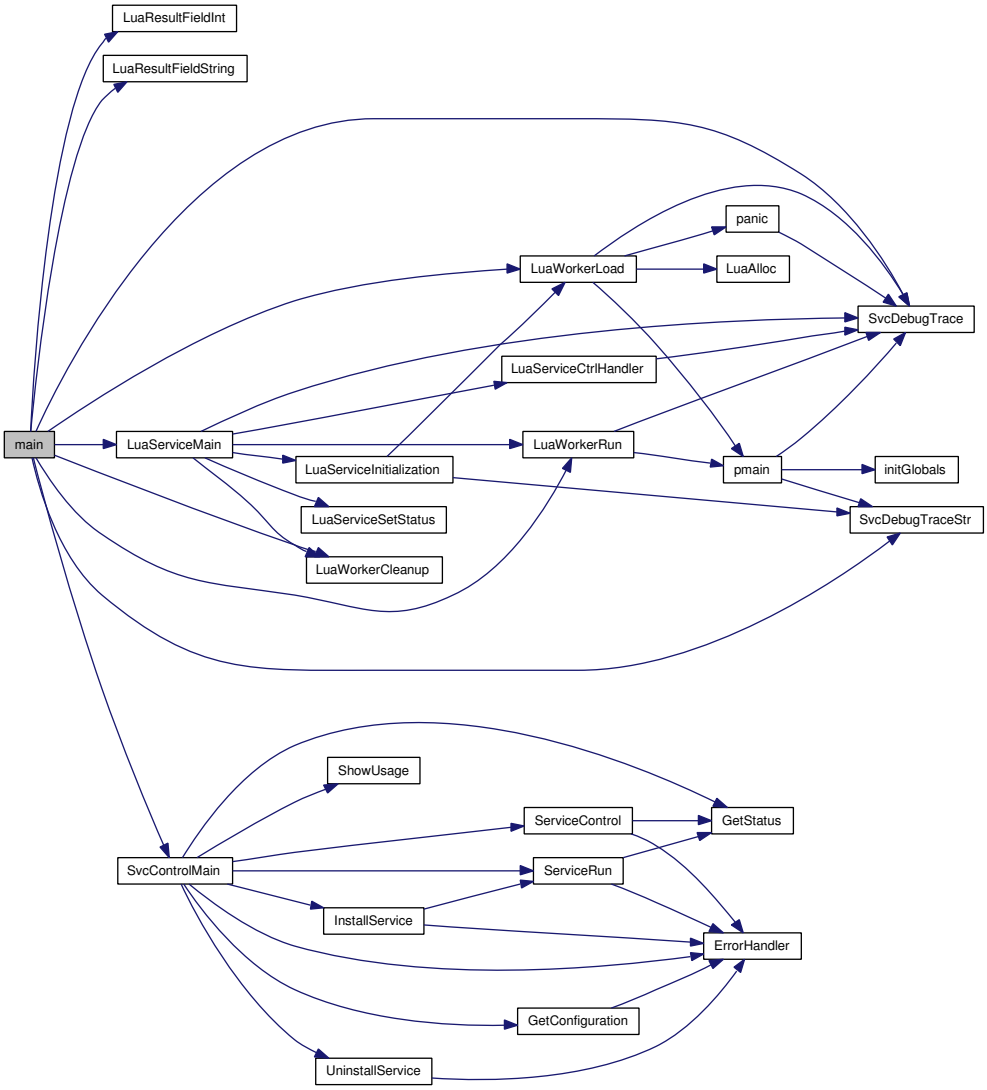
ssSvc

Definition at line 425 of file LuaService.c.

References LuaResultFieldInt(), LuaResultFieldString(), LuaServiceMain(), LuaWorkerCleanup(), LuaWorkerLoad(), LuaWorkerRun(), ServiceName, ServiceScript, SvcControlMain(), SvcDebugTrace(), SvcDebugTraceLevel, and SvcDebugTraceStr().

```
426 {
427     LUAHANDLE lh;
428     SERVICE_TABLE_ENTRY DispatchTable[2]; // note room for terminating record.
429     memset(DispatchTable, 0, sizeof(DispatchTable));
430 
431     SvcDebugTrace("Entered main\n", 0);
432     lh = LuaWorkerLoad(NULL, "init.lua");
433     if (lh) {
434         char *cp;
435         int n;
436         lh = LuaWorkerRun(lh);
437         SvcDebugTrace("... ran init\n", 0);
438         n = LuaResultFieldInt(lh, 1, "tracelevel");
439         SvcDebugTraceLevel = n;
440         cp = LuaResultFieldString(lh, 1, "name");
441         if (cp)
442             ServiceName = cp;
443         SvcDebugTraceStr("... got name %s", cp);
444         cp = LuaResultFieldString(lh, 1, "script");
445         if (cp)
446             ServiceScript = cp;
447         SvcDebugTraceStr("... got script %s", cp);
448         SvcDebugTrace("Finished pre-init\n", 0);
449         LuaWorkerCleanup(lh);
450     }
451     DispatchTable[0].lpServiceName = (LPSTR)ServiceName;
452     DispatchTable[0].lpServiceProc = LuaServiceMain;
453     SvcDebugTraceStr("Service name: %s\n", ServiceName);
454     if (!StartServiceCtrlDispatcher(DispatchTable)) {
455         DWORD err = GetLastError();
456         if (err == ERROR_FAILED_SERVICE_CONTROLLER_CONNECT) {
457             /*
458              * A failure to connect to the SCM implies we are not running
459              * under the SCM's control, so we must not be the actual
460              * service application.
461              *
462              * We try being a controller or configurer instead.
463              */
464             return SvcControlMain(argc, argv);
465         } else {
466             SvcDebugTrace("StartServiceCtrlDispatcher failed %ld\n", err);
467             return EXIT_FAILURE;
468         }
469     }
470     SvcDebugTrace("Leaving main\n", 0);
471     return EXIT_SUCCESS;
472 }
```

Here is the call graph for this function:



4.5.2.6 void SvcDebugTrace (LPCSTR *fmt*, DWORD *dw*)

Output a debug string.

The string is formatted and output only if SvcDebugTraceLevel is greater than zero.

If SvcDebugTraceLevel is 2 or greater, the name of the service will be included in the

output.

If SvcDebugTraceLevel is 3 or greater, the current process and thread ids will be included in the output in addition to the service name.

Call Context:

Service, Configuration, Control

Parameters:

fmt A printf()-like format string with an optional reference to a single DWORD value.

dw A DWORD value to substitute in the message.

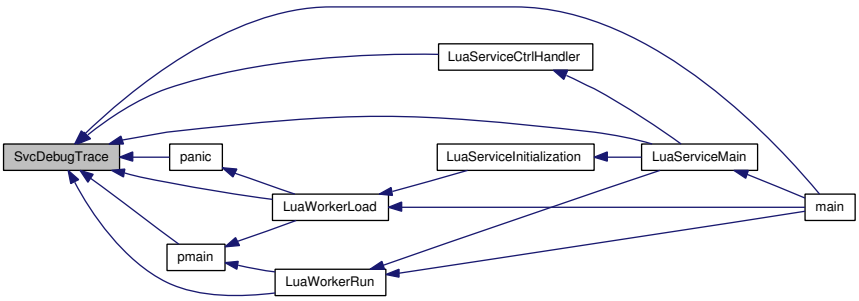
Definition at line 130 of file LuaService.c.

References ServiceName, and SvcDebugTraceLevel.

Referenced by LuaServiceCtrlHandler(), LuaServiceMain(), LuaWorkerLoad(), LuaWorkerRun(), main(), panic(), and pmain().

```
131 {
132     char Buffer[1024];
133     char *cp = Buffer;
134 
135     if (SvcDebugTraceLevel <= 0)
136         return;
137     if (SvcDebugTraceLevel == 2)
138         cp += sprintf(Buffer, "[%s] ", ServiceName);
139     else if (SvcDebugTraceLevel >= 3)
140         cp += sprintf(Buffer, "[%s:%ld/%ld] ", ServiceName,
141                 GetCurrentProcessId(), GetCurrentThreadId());
142     if (fmt == NULL) {
143         strcpy(cp, "-nil-");
144         OutputDebugStringA(Buffer);
145     } else if ((strlen(fmt)+12) < (sizeof(Buffer) - (cp - Buffer))) {
146         sprintf(cp, fmt, dw);
147         OutputDebugStringA(Buffer);
148     } else
149         OutputDebugStringA("--buffer overflow--");
150 }
```

Here is the caller graph for this function:



4.5.2.7 void SvcDebugTraceStr (LPCSTR *fmt*, LPCSTR *s*)

Output a debug string.

The string is formatted and output only if SvcDebugTraceLevel is greater than zero.

If SvcDebugTraceLevel is 2 or greater, the name of the service will be included in the output.

If SvcDebugTraceLevel is 3 or greater, the current process and thread ids will be included in the output in addition to the service name.

Call Context:

Service, Configuration, Control

Parameters:

fmt A printf()-like format string with an optional reference to a single string value.

s A string value to substitute in the message.

Definition at line 170 of file LuaService.c.

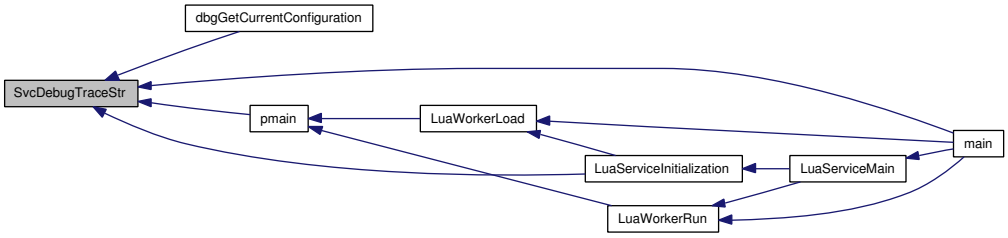
References ServiceName, and SvcDebugTraceLevel.

Referenced by dbgGetCurrentConfiguration(), LuaServiceInitialization(), main(), and pmain().

```
171 {
172     char Buffer[1024];
173     char *cp = Buffer;
174 
175     if (SvcDebugTraceLevel <= 0)
176         return;
```

```
177     if (SvcDebugTraceLevel == 2)
178         cp += sprintf(Buffer, "[%s] ", ServiceName);
179     else if (SvcDebugTraceLevel >= 3)
180         cp += sprintf(Buffer, "[%s:%ld/%ld] ", ServiceName,
181                 GetCurrentProcessId(), GetCurrentThreadId());
182     if (s == NULL)
183         s = "-nil-";
184     if (fmt == NULL)
185         fmt = "-nil-";
186     if ((strlen(fmt)+strlen(s)) < sizeof(Buffer) - (cp - Buffer)) {
187         sprintf(cp, fmt, s);
188         OutputDebugStringA(Buffer);
189     } else
190         OutputDebugStringA("--buffer overflow--");
191 }
```

Here is the caller graph for this function:



4.5.3 Variable Documentation

4.5.3.1 SERVICE_STATUS LuaServiceStatus

Current service status.

Call Context:

Service main and worker threads

Definition at line 62 of file LuaService.c.

Referenced by LuaServiceCtrlHandler(), LuaServiceMain(), and LuaServiceSetStatus().

4.5.3.2 SERVICE_STATUS_HANDLE LuaServiceStatusHandle

Handle to the SCM for the running service to report status.

This is global because it is discovered by the worker thread, and needed by the thread in which the control request handler executes, which is apparently (but not particularly documented) the main thread.

Call Context:

Service main and worker threads

Definition at line 74 of file LuaService.c.

Referenced by LuaServiceCtrlHandler(), LuaServiceMain(), and LuaServiceSetStatus().

4.5.3.3 const char* ServiceName = "LuaService"

Service name.

This string must be unique in the installed system because it is used to identify the service to the SCM. It will appear in the Service control panel, and in other spots where an end-user might see it.

Note:

This value may be configured for a specific installation of this framework by writing a lua script named init.lua that returns a table with a field `name`. The init.lua script must be located in the same folder as LuaService.exe.

Definition at line 42 of file LuaService.c.

Referenced by ChangeConfig(), dbgGetCurrentConfiguration(), GetConfiguration(), initGlobals(), InstallService(), LuaServiceMain(), main(), ServiceControl(), ServiceRun(), SvcControlMain(), SvcDebugTrace(), SvcDebugTraceStr(), and UninstallService().

4.5.3.4 const char* ServiceScript = "service.lua"

Service launcher script.

This string names the Lua script that acts as the main entry point of the service worker thread. This script must be located inside the service's folder or a sub-folder.

Note:

This value may be configured for a specific installation of this framework by writing a lua script named init.lua that returns a table with a field `script`. The init.lua script must be located in the same folder as LuaService.exe.

Definition at line 55 of file LuaService.c.

Referenced by LuaServiceInitialization(), and main().

4.5.3.5 volatile int ServiceStopping = 0

Service Stopping Flag.

Set in the service control request handler to indicate that a STOP request has been received and that the SCM is being informed that the service is now SERVICE_STOP_-PENDING.

About 25 seconds after setting this flag, the service will forcefully die with or without cooperation from the worker thread.

The worker can test this flag from Lua by calling the function service.stopping().

Definition at line 110 of file LuaService.c.

Referenced by dbgStopping(), and LuaServiceCtrlHandler().

4.5.3.6 HANDLE ServiceWorkerThread

Handle to the thread executing ServiceMain().

This will be initialized by ServiceMain() as a duplicate of the thread handle, for use in the **main()** (p. 35) thread so that it has a kernel object on which it can wait for the service thread to have exited when stopping the service.

Since it was created as a duplicate, it must be closed.

Definition at line 85 of file LuaService.c.

Referenced by LuaServiceCtrlHandler(), and LuaServiceInitialization().

4.5.3.7 int SvcDebugTraceLevel = 0

Trace level.

Controls the verbosity of the trace output. The level is tested before any work has been done to format the output, so it is reasonably effective to turn tracing off by setting the level to zero.

Values range from zero (no tracing) and up.

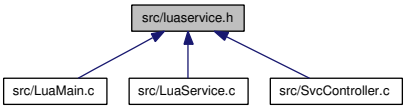
Definition at line 95 of file LuaService.c.

Referenced by dbgTracelevel(), main(), SvcDebugTrace(), and SvcDebugTraceStr().

4.6 src/luaservice.h File Reference

Public declarations.

This graph shows which files directly or indirectly include this file:



Typedefs

- typedef void * **LUAHANDLE**

  An opaque pointer to a Lua state.

Functions

- **LUAHANDLE LuaWorkerLoad** (**LUAHANDLE** h, const char *cmd)

  Create a Lua state with a script loaded.

- **LUAHANDLE LuaWorkerRun** (**LUAHANDLE** h)

  Run a pending Lua script.

- void **LuaWorkerCleanup** (**LUAHANDLE** h)

  Clean up after the worker by closing the Lua state.

- char * **LuaResultString** (**LUAHANDLE** h, int item)

  Get a cached worker result item as a string.

- int **LuaResultInt** (**LUAHANDLE** h, int item)

  Get a cached worker result item as an integer.

- char * **LuaResultFieldString** (**LUAHANDLE** h, int item, const char *field)

  Get a field of a cached worker result item as a string.

- int **LuaResultFieldInt** (**LUAHANDLE** h, int item, const char *field)

  Get a field of a cached worker result item as an integer.

- void **SvcDebugTrace** (LPCSTR fmt, DWORD Status)

  Output a debug string.

- void **SvcDebugTraceStr** (LPCSTR fmt, LPCSTR s)

  Output a debug string.

- int **SvcControlMain** (int argc, char ∗argv[ ])
  Entry point for service control and configuration.

Variables

- int **SvcDebugTraceLevel**
  Trace level.
- const char ∗ **ServiceName**
  Service name.
- const char ∗ **ServiceScript**
  Service launcher script.
- volatile int **ServiceStopping**
  Service Stopping Flag.

4.6.1 Detailed Description

Public declarations.

Definition in file **luaservice.h**.

4.6.2 Typedef Documentation

4.6.2.1 typedef void∗ LUAHANDLE

An opaque pointer to a Lua state.

Definition at line 10 of file luaservice.h.

4.6.3 Function Documentation

4.6.3.1 int LuaResultFieldInt (LUAHANDLE *h*, int *item*, const char ∗ *field*)

Get a field of a cached worker result item as an integer.

Parameters:

h An opaque handle returned by a previous call to **LuaWorkerRun()** (p. 21).

item The index of the result item to retrieve. The first result is index 1, consistent with Lua counting.

field The name of the field to retrieve.

Returns:

The integer value or 0 if the field or item doesn't exist or can't be converted to a number.

Definition at line 754 of file LuaMain.c.

References local_getreg, and WORK_RESULTS.

Referenced by main().

```
755 {
756         int ret = 0;
757         lua_State *L = (lua_State*)h;
758         if (!h) return 0;
759         local_getreg(L,WORK_RESULTS);   // table
760         if (lua_type(L,-1) != LUA_TTABLE) {
761                 lua_pop(L,1);
762                 return 0;
763         }
764         lua_rawgeti(L,-1,item);                 // table itemtable
765         if (lua_type(L,-1) != LUA_TTABLE) {
766                 lua_pop(L,2);
767                 return 0;
768         }
769         lua_getfield(L,-1,field);               // table itemtable fieldvalue
770         ret = (int)lua_tointeger(L,-1);
771         lua_pop(L,3);
772         return ret;
773 }
```

Here is the caller graph for this function:

4.6.3.2 char* LuaResultFieldString (LUAHANDLE *h*, int *item*, const char * *field*)

Get a field of a cached worker result item as a string.

Note:

The string returned came from strdup(), and must be freed by the caller.

Parameters:

h An opaque handle returned by a previous call to **LuaWorkerRun()** (p. 21).

item The index of the result item to retrieve. The first result is index 1, consistent with Lua counting.

field The name of the field to retrieve.

Returns:

The string value (from strdup()) or NULL if the field or item doesn't exist or can't be converted to a string.

Definition at line 720 of file LuaMain.c.

References local_getreg, and WORK_RESULTS.

Referenced by main().

```
721 {
722         char *ret = NULL;
723         lua_State *L = (lua_State*)h;
724         if (!h) return NULL;
725         local_getreg(L,WORK_RESULTS);   // table
726         if (lua_type(L,-1) != LUA_TTABLE) {
727                 lua_pop(L,1);
728                 return NULL;
729         }
730         lua_rawgeti(L,-1,item);                 // table itemtable
731         if (lua_type(L,-1) != LUA_TTABLE) {
732                 lua_pop(L,2);
733                 return NULL;
734         }
735         lua_getfield(L,-1,field);               // table itemtable fieldvalue
736         ret = (char *)lua_tostring(L,-1);
737         if (ret)
738                 ret = strdup(ret);
739         lua_pop(L,3);
740         return ret;
741 }
```

Here is the caller graph for this function:

4.6.3.3 int LuaResultInt (LUAHANDLE *h*, int *item*)

Get a cached worker result item as an integer.

Parameters:

h An opaque handle returned by a previous call to **LuaWorkerRun()** (p. 21).

item The index of the result item to retrieve. The first result is index 1, consistent with Lua counting.

Returns:

The integer value or 0 if the item doesn't exist or can't be converted to a number.

Definition at line 692 of file LuaMain.c.

References local_getreg, and WORK_RESULTS.

```
693 {
694         int ret = 0;
695         lua_State *L = (lua_State*)h;
696         if (!h) return 0;
697         local_getreg(L,WORK_RESULTS);
698         if (lua_type(L,-1) != LUA_TTABLE) {
699                 lua_pop(L,1);
700                 return 0;
701         }
702         lua_rawgeti(L,-1,item);
703         ret = (int)lua_tointeger(L,-1);
704         lua_pop(L,2);
705         return ret;
706 }
```

4.6.3.4 char* LuaResultString (LUAHANDLE *h*, int *item*)

Get a cached worker result item as a string.

Note:

The string returned came from strdup(), and must be freed by the caller.

Parameters:

h An opaque handle returned by a previous call to **LuaWorkerRun()** (p. 21).

item The index of the result item to retrieve. The first result is index 1, consistent with Lua counting.

Returns:

The string value (from strdup()) or NULL if the item doesn't exist or can't be converted to a string.

Definition at line 664 of file LuaMain.c.

References local_getreg, and WORK_RESULTS.

```
665 {
666         char *ret = NULL;
667         lua_State *L = (lua_State*)h;
668         if (!h) return NULL;
669         local_getreg(L,WORK_RESULTS);
670         if (lua_type(L,-1) != LUA_TTABLE) {
671                 lua_pop(L,1);
672                 return NULL;
673         }
674         lua_rawgeti(L,-1,item);
675         ret = (char *)lua_tostring(L,-1);
676         if (ret)
677                 ret = strdup(ret);
678         lua_pop(L,2);
679         return ret;
680 }
```

4.6.3.5 void LuaWorkerCleanup (LUAHANDLE *h*)

Clean up after the worker by closing the Lua state.

Parameters:

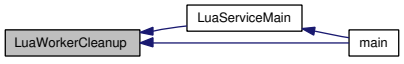
h An opaque handle returned by a previous call to **LuaWorkerRun**() (p. 21).

Definition at line 646 of file LuaMain.c.

Referenced by LuaServiceMain(), and main().

```
647 {
648         lua_State *L=(lua_State*)h;
649         if (h)
650                 lua_close(L);
651 }
```

Here is the caller graph for this function:



4.6.3.6 LUAHANDLE LuaWorkerLoad (LUAHANDLE *h*, const char * *cmd*)

Create a Lua state with a script loaded.

Creates a new Lua state, initializes it with built-in modules and global variables, then loads the specified script or command.

If all is well, the compiled but as yet unexecuted main block of the script is cached in the Lua Registry at index PENDING_WORK (a light user data made from the address of this function).

Parameters:

h Opaque handle to the Lua state to use, or NULL to create a new state.

cmd Script or statement to load

Returns:

An opaque handle identifying the created Lua state.

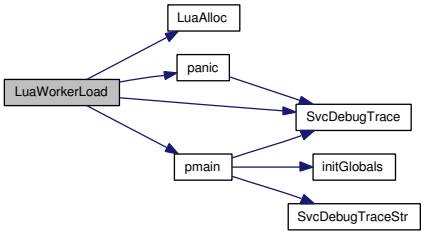
Definition at line 590 of file LuaMain.c.

References LuaAlloc(), panic(), pmain(), and SvcDebugTrace().

Referenced by LuaServiceInitialization(), and main().

```
591 {
592         int status;
593         lua_State *L=(lua_State*)h;
594
595         if (!h) {
596 #if 0
597                 L = luaL_newstate();
598 #else
599                 L = lua_newstate(LuaAlloc, NULL);
600 #endif
601                 assert(L);
602                 lua_atpanic(L, &panic);
603         }
604         status = lua_cpcall(L, &pmain, (void*)cmd);
605         if (status) {
606                 SvcDebugTrace("Load script cpcall status %d", status);
607                 SvcDebugTrace((char *)lua_tostring(L,-1),0);
608                 //return NULL;
609         } else {
610                 SvcDebugTrace("Script loaded ok", 0);
611         }
612         return (void *)L;
613 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

4.6.3.7 LUAHANDLE LuaWorkerRun (LUAHANDLE *h*)

Run a pending Lua script.

The script must have been previously loaded and saved in the Lua registry.

Parameters:

h An opaque handle returned by a previous call to **LuaWorkerRun()** (p. 21).

Returns:

An opaque handle identifying the created Lua state.

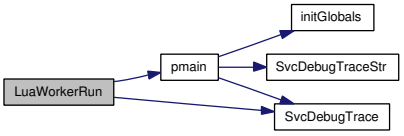
Definition at line 622 of file LuaMain.c.

References pmain(), and SvcDebugTrace().

Referenced by LuaServiceMain(), and main().

```
623 {
624         int status;
625         lua_State *L=(lua_State*)h;
626
627         if (!h) {
628                 SvcDebugTrace("No existing lua state!!!", 0);
629                 return NULL;
630                 //L = lua_open();
631         }
632         status = lua_cpcall(L, &pmain, NULL);
633         if (status) {
634                 SvcDebugTrace("Run script cpcall status %d", status);
635                 SvcDebugTrace((char *)lua_tostring(L,-1),0);
636         } else {
637                 SvcDebugTrace("Script succeeded", 0);
638         }
639         return (void *)L;
640 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

4.6.3.8 int SvcControlMain (int *argc*, char * *argv*[ ])

Entry point for service control and configuration.

Called from **main()** (p. 35) if the program is not running as a service and responsible to the SCM.

Parameters:

argc Count of arguments in *argv*.

argv Array of arguments.

Returns:

Exit status, as from **main()** (p. 35).

Definition at line 39 of file SvcController.c.

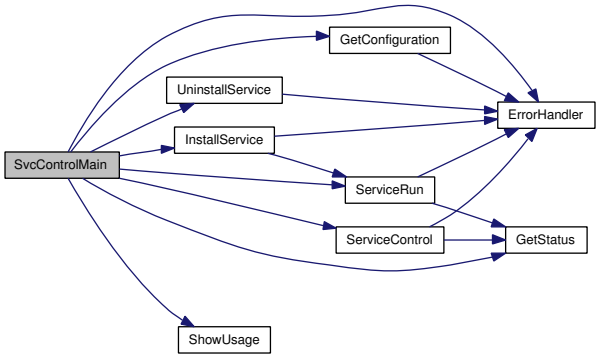
References ErrorHandler(), GetConfiguration(), GetStatus(), InstallService(), ServiceControl(), ServiceName, ServiceRun(), ShowUsage(), and UninstallService().

Referenced by main().

```
39                                              {
40         if (argc == 2) {
41                 if (stricmp("-i", argv[1]) == 0)
42                         InstallService();
43                 else if (stricmp("-u", argv[1]) == 0)
44                         UninstallService();
45                 else if (stricmp("-r", argv[1]) == 0)
46                         ServiceRun();
47                 else if (stricmp("-s", argv[1]) == 0)
```

```
48                      ServiceControl("STOP");
49 #ifdef LUASERVICE_CAN_PAUSE_CONTINUE
50                 else if (stricmp("-p", argv[1]) == 0)
51                 ServiceControl("PAUSE");
52                 else if (stricmp("-c", argv[1]) == 0)
53                 ServiceControl("RESUME");
54 #endif
55                 else if (stricmp("status", argv[1]) == 0) {
56                         SC_HANDLE scm, service;
57                         //Open connection to SCM
58                         scm = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
59                         if (!scm)
60                                 ErrorHandler("OpenSCManager", GetLastError());
61                         //Get service's handle
62                         service = OpenService(scm, ServiceName, SERVICE_ALL_ACCESS);
63                         if (!service)
64                                 ErrorHandler("OpenService", GetLastError());
65                         fputs("STATUS: ", stdout);
66                         GetStatus(service);
67                 } else if (stricmp("config", argv[1]) == 0)
68                         GetConfiguration();
69                 else if (stricmp("help", argv[1]) == 0)
70                         ShowUsage();
71                 //add other custom commands here and
72                 //update ShowUsage function
73                 else {
74                         ShowUsage();
75                         return EXIT_FAILURE;
76                 }
77         } else {
78                 ShowUsage();
79                 return EXIT_FAILURE;
80         }
81         return EXIT_SUCCESS;
82 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

4.6.3.9 void SvcDebugTrace (LPCSTR *fmt*, DWORD *dw*)

Output a debug string.

The string is formatted and output only if SvcDebugTraceLevel is greater than zero.

If SvcDebugTraceLevel is 2 or greater, the name of the service will be included in the output.

If SvcDebugTraceLevel is 3 or greater, the current process and thread ids will be included in the output in addition to the service name.

Call Context:

Service, Configuration, Control

Parameters:

fmt A printf()-like format string with an optional reference to a single DWORD value.

dw A DWORD value to substitute in the message.

Definition at line 130 of file LuaService.c.

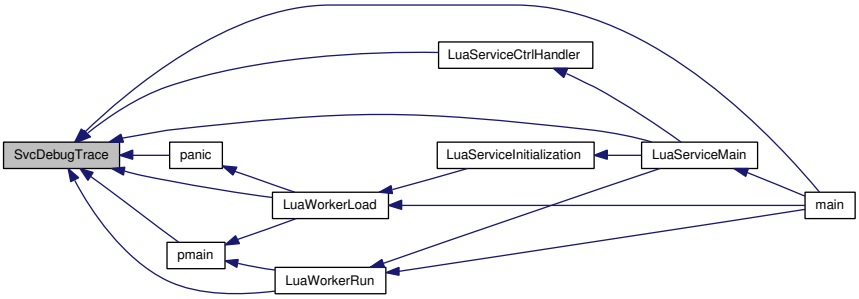
References ServiceName, and SvcDebugTraceLevel.

Referenced by LuaServiceCtrlHandler(), LuaServiceMain(), LuaWorkerLoad(), LuaWorkerRun(), main(), panic(), and pmain().

```
131 {
132     char Buffer[1024];
133     char *cp = Buffer;
134 
135     if (SvcDebugTraceLevel <= 0)
136         return;
137     if (SvcDebugTraceLevel == 2)
138         cp += sprintf(Buffer, "[%s] ", ServiceName);
139     else if (SvcDebugTraceLevel >= 3)
140         cp += sprintf(Buffer, "[%s:%ld/%ld] ", ServiceName,
141                 GetCurrentProcessId(), GetCurrentThreadId());
142     if (fmt == NULL) {
143         strcpy(cp, "-nil-");
144         OutputDebugStringA(Buffer);
145     } else if ((strlen(fmt)+12) < (sizeof(Buffer) - (cp - Buffer))) {
146         sprintf(cp, fmt, dw);
```

```
147         OutputDebugStringA(Buffer);
148     } else
149         OutputDebugStringA("--buffer overflow--");
150 }
```

Here is the caller graph for this function:



4.6.3.10 void SvcDebugTraceStr (LPCSTR *fmt*, LPCSTR *s*)

Output a debug string.

The string is formatted and output only if SvcDebugTraceLevel is greater than zero.

If SvcDebugTraceLevel is 2 or greater, the name of the service will be included in the output.

If SvcDebugTraceLevel is 3 or greater, the current process and thread ids will be included in the output in addition to the service name.

Call Context:

Service, Configuration, Control

Parameters:

fmt A printf()-like format string with an optional reference to a single string value.

s A string value to substitute in the message.

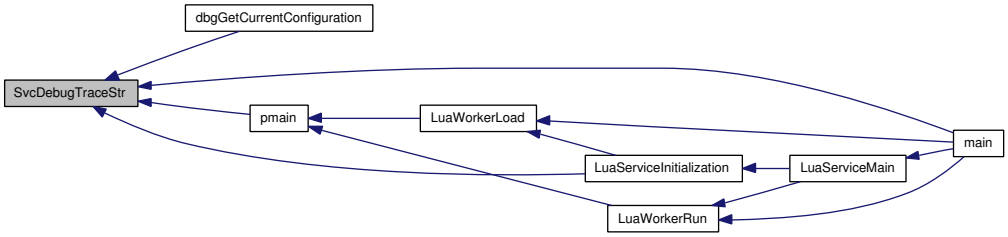
Definition at line 170 of file LuaService.c.

References ServiceName, and SvcDebugTraceLevel.

Referenced by dbgGetCurrentConfiguration(), LuaServiceInitialization(), main(), and pmain().

```
171 {
172     char Buffer[1024];
173     char *cp = Buffer;
174 
175     if (SvcDebugTraceLevel <= 0)
176         return;
177     if (SvcDebugTraceLevel == 2)
178         cp += sprintf(Buffer, "[%s] ", ServiceName);
179     else if (SvcDebugTraceLevel >= 3)
180         cp += sprintf(Buffer, "[%s:%ld/%ld] ", ServiceName,
181                 GetCurrentProcessId(), GetCurrentThreadId());
182     if (s == NULL)
183         s = "-nil-";
184     if (fmt == NULL)
185         fmt = "-nil-";
186     if ((strlen(fmt)+strlen(s)) < sizeof(Buffer) - (cp - Buffer)) {
187         sprintf(cp, fmt, s);
188         OutputDebugStringA(Buffer);
189     } else
190         OutputDebugStringA("--buffer overflow--");
191 }
```

Here is the caller graph for this function:



4.6.4 Variable Documentation

4.6.4.1 const char* ServiceName

Service name.

This string must be unique in the installed system because it is used to identify the service to the SCM. It will appear in the Service control panel, and in other spots where an end-user might see it.

Note:

This value may be configured for a specific installation of this framework by writing a lua script named init.lua that returns a table with a field `name`. The init.lua script must be located in the same folder as LuaService.exe.

Definition at line 42 of file LuaService.c.

Referenced by ChangeConfig(), dbgGetCurrentConfiguration(), GetConfiguration(), initGlobals(), InstallService(), LuaServiceMain(), main(), ServiceControl(), ServiceRun(), SvcControlMain(), SvcDebugTrace(), SvcDebugTraceStr(), and UninstallService().

4.6.4.2 const char* ServiceScript

Service launcher script.

This string names the Lua script that acts as the main entry point of the service worker thread. This script must be located inside the service's folder or a sub-folder.

Note:

> This value may be configured for a specific installation of this framework by writing a lua script named init.lua that returns a table with a field `script`. The init.lua script must be located in the same folder as LuaService.exe.

Definition at line 55 of file LuaService.c.

Referenced by LuaServiceInitialization(), and main().

4.6.4.3 volatile int ServiceStopping

Service Stopping Flag.

Set in the service control request handler to indicate that a STOP request has been received and that the SCM is being informed that the service is now SERVICE_STOP_PENDING.

About 25 seconds after setting this flag, the service will forcefully die with or without cooperation from the worker thread.

The worker can test this flag from Lua by calling the function service.stopping().

Definition at line 110 of file LuaService.c.

Referenced by dbgStopping(), and LuaServiceCtrlHandler().

4.6.4.4 int SvcDebugTraceLevel

Trace level.

Controls the verbosity of the trace output. The level is tested before any work has been done to format the output, so it is reasonably effective to turn tracing off by setting the level to zero.

Values range from zero (no tracing) and up.

Definition at line 95 of file LuaService.c.

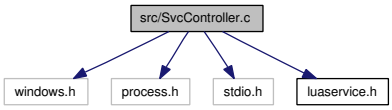
Referenced by dbgTracelevel(), main(), SvcDebugTrace(), and SvcDebugTraceStr().

4.7 src/SvcController.c File Reference

Functions to configure and control a service.

```
#include <windows.h>
#include <process.h>
#include <stdio.h>
#include "luaservice.h"
```

Include dependency graph for SvcController.c:



Functions

- void **ErrorHandler** (char *s, int err)

  Handle a Windows error code.

- void **GetStatus** (SC_HANDLE service)
- void **ShowUsage** ()

  Print a command-line usage message.

- int **InstallService** ()

  Install the service.

- int **UninstallService** ()

  Uninstall the service.

- int **GetConfiguration** ()
- int **ChangeConfig** ()
- int **ServiceRun** ()

  Start the service running.

- int **ServiceControl** (char *CONTROL)

  Send other controls to the service.

- int **SvcControlMain** (int argc, char *argv[ ])

  Entry point for service control and configuration.

4.7.1 Detailed Description

Functions to configure and control a service.

Todo

This mechanism was copied from sample code, and lacks friendly handling of command line arguments. Future versions will likely move some of the less critical details into a Lua script, with the actual service control methods exposed via a built-in module.

Definition in file **SvcController.c**.

4.7.2 Function Documentation

4.7.2.1 int ChangeConfig ()

Definition at line 562 of file SvcController.c.

References ErrorHandler(), and ServiceName.

```
562                     {
563         SC_HANDLE service;
564         SC_HANDLE scm;
565         BOOL success;
566         SC_LOCK lock;
567 
568         //open connection to SCM
569         scm = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS | GENERIC_WRITE);
570         if (!scm)
571                 ErrorHandler("OpenSCManager", GetLastError());
572 
573         //lock the database to guarantee exclusive access
574         lock = LockServiceDatabase(scm);
575         if (lock == 0)
576                 ErrorHandler("LockServiceDatabase", GetLastError());
577 
578         //get service's handle
579         service = OpenService(scm, ServiceName, SERVICE_ALL_ACCESS);
580         if (!service)
581                 ErrorHandler("OpenService", GetLastError());
582 
583         //      serviceType = SERVICE_NO_CHANGE;
584         //      serviceStart = SERVICE_NO_CHANGE;
585         //      serviceError = SERVICE_NO_CHANGE;
586         //      path = 0;
587 
588         //change service config
589         success = ChangeServiceConfig(
590                         service,
591                         SERVICE_NO_CHANGE,
592                         SERVICE_NO_CHANGE,
593                         SERVICE_NO_CHANGE,
```

```
594                         NULL,
595                         NULL,
596                         NULL,
597                         NULL,
598                         NULL,
599                         NULL,
600                         NULL);
601         if (!success) {
602                 UnlockServiceDatabase(lock);
603                 ErrorHandler("ChangeServiceConfig", GetLastError());
604         }
605
606         //unlock database
607         success = UnlockServiceDatabase(lock);
608         if (!success)
609                 ErrorHandler("UnlockServiceDatabase", GetLastError());
610
611         //clean up
612         CloseServiceHandle(service);
613         CloseServiceHandle(scm);
614         return TRUE;
615 }
```

Here is the call graph for this function:

4.7.2.2 void ErrorHandler (char * *s*, int *err*)

Handle a Windows error code.

Looks up the error code *err* in the system table of error messages. If the message exists, it is printed to stderr and to a log file. Otherwise, the printed message and the log entry both indicate that FormatMessage() couldn't identify the error code.

Note:

In the current version, this is only called after a service-related bit of the Windows API fails, so it is highly unlikely that any error code will cause FormatMessage() to fail. However, if a future version allows this function to be called in a wider context or even from Lua code, then it is possible that a provision should be made to look for error messages from additional sources such as any DLLs loaded into the process.

Parameters:

s Name of function or other context that caused the failure.

err Windows error code returned from GetLastError().

Returns:

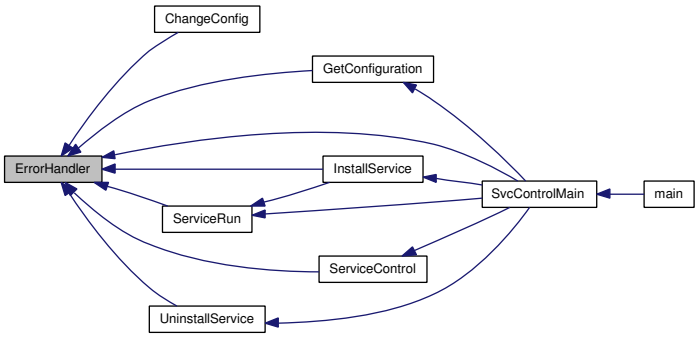
This function calls exit() and does not return to its caller.

Definition at line 160 of file SvcController.c.

Referenced by ChangeConfig(), GetConfiguration(), InstallService(), ServiceControl(), ServiceRun(), SvcControlMain(), and UninstallService().

```
160                                     {
161         LPVOID lpMsgBuf;
162         FILE* pLog;
163         size_t n;
164
165         n = (size_t)FormatMessageA((0 | FORMAT_MESSAGE_ALLOCATE_BUFFER
166                         | FORMAT_MESSAGE_FROM_SYSTEM | 75 ),
167         NULL, err, 0, //MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
168                         (LPTSTR) &lpMsgBuf, 0, NULL);
169         pLog = fopen("LuaService.log", "a");
170         if (n != 0) {
171                 fprintf(stderr,
172                 "%s failed\n"
173                 "Error (%d): %s\n", s, err, (char *)lpMsgBuf);
174                 LocalFree(lpMsgBuf);
175         } else {
176                 fprintf(stderr,
177                 "%s failed\n"
178                 "Error (%d): <<not known to FormatMessage(): %ld>>\n",
179                 s, err, GetLastError());
180         }
181         fprintf(pLog, "%s failed, error code = %d\n", s, err);
182         fclose(pLog);
183         exit(EXIT_FAILURE);
184 }
```

Here is the caller graph for this function:



4.7.2.3 int GetConfiguration ()

Definition at line 524 of file SvcController.c.

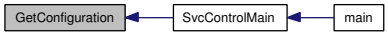
References ErrorHandler(), and ServiceName.

Referenced by SvcControlMain().

```
524                         {
525         SC_HANDLE service;
526         SC_HANDLE scm;
527         BOOL success;
528         LPQUERY_SERVICE_CONFIG buffer;
529         DWORD sizeNeeded;
530 
531         //open connection to SCM
532         scm = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
533         if (!scm)
534                 ErrorHandler("OpenSCManager", GetLastError());
535 
536         //get service's handle
537         service = OpenService(scm, ServiceName, SERVICE_QUERY_CONFIG);
538         if (!service)
539                 ErrorHandler("OpenService", GetLastError());
540 
541         //allocate space for buffer
542         buffer = (LPQUERY_SERVICE_CONFIG)LocalAlloc(LPTR, 4096);
543         // Get the configuration information.
544         success = QueryServiceConfig(service, buffer, 4096, &sizeNeeded);
545         if (!success)
546                 ErrorHandler("QueryServiceConfig", GetLastError());
547 
548         //display the info
549         printf("Service name\t: %s\n", buffer->lpDisplayName);
550         printf("Service type\t: %ld\n", buffer->dwServiceType);
551         printf("Start type\t: %ld\n", buffer->dwStartType);
552         printf("Start name\t: %s\n", buffer->lpServiceStartName);
553         printf("Path\t\t: %s\n", buffer->lpBinaryPathName);
554 
555         LocalFree(buffer);
556 
557         CloseServiceHandle(service);
558         CloseServiceHandle(scm);
559         return TRUE;
560 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



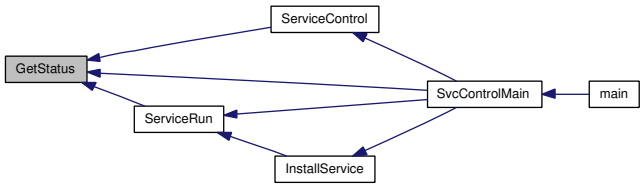
4.7.2.4 void GetStatus (SC_HANDLE *service*)

Definition at line 487 of file SvcController.c.

Referenced by ServiceControl(), ServiceRun(), and SvcControlMain().

```
487                                     {
488         BOOL SUCCESS;
489         SERVICE_STATUS status;
490
491         SUCCESS = QueryServiceStatus(service, &status);
492         switch (status.dwCurrentState) {
493         case SERVICE_RUNNING:
494                 puts("Service RUNNING.");
495                 break;
496         case SERVICE_STOPPED:
497                 puts("Service STOPPED.");
498                 break;
499         case SERVICE_PAUSED:
500                 puts("Service PAUSED.");
501                 break;
502         case SERVICE_CONTINUE_PENDING:
503                 puts("Service is resuming...");
504                 break;
505         case SERVICE_PAUSE_PENDING:
506                 puts("Service is pausing...");
507                 break;
508         case SERVICE_START_PENDING:
509                 puts("Service is starting...");
510                 break;
511         case SERVICE_STOP_PENDING:
512                 puts("Service is stopping...");
513                 break;
514         default:
515                 break;
516         }
517 }
```

Here is the caller graph for this function:



4.7.2.5 int InstallService ()

Install the service.

Asks the **Service Control Manager** (p. 72) to install the service on the local machine.

The service name and display name are both derived from the name field returned by init.lua.

The service is installed as auto-start, using the LocalSystem authority.

Once installed, the SCM is told to start the service.

Returns:

Returns TRUE on success. The current implementation calls **ErrorHandler()** (p. 59) for all significant errors which exits the process and does not return. There appear to be no insignificant errors.

Definition at line 221 of file SvcController.c.

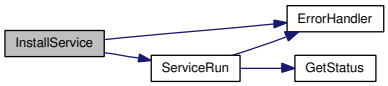
References ErrorHandler(), ServiceName, and ServiceRun().

Referenced by SvcControlMain().

```
221                     {
222         SC_HANDLE newService;
223         SC_HANDLE scm;
224         char szPath[MAX_PATH+3];
225 
226         //get file path
227         szPath[0] = '\"';
228         GetModuleFileName(GetModuleHandle(NULL), szPath+1, MAX_PATH);
229         strcat(szPath, "\"");
230 
231         //open connection to SCM
232         scm = OpenSCManager(NULL, NULL, SC_MANAGER_CREATE_SERVICE);
233         if (!scm)
234                 ErrorHandler("OpenSCManager", GetLastError());
235 
236         //install service
237         newService = CreateService(
238                         scm, //scm database
239                         ServiceName, //service name
240                         ServiceName, //display name
241                         SERVICE_ALL_ACCESS, //access rights to the service
242                         SERVICE_WIN32_OWN_PROCESS, //service type
243                         SERVICE_AUTO_START, //service start type
244                         SERVICE_ERROR_NORMAL, //error control type
245                         szPath, //service path
246                         NULL, //no load ordering group
247                         NULL, //no tag identifier
248                         NULL, //no dependencies
249                         NULL, //LocalSystem account
250                         NULL); //no password
251         if (!newService) {
252                 ErrorHandler("CreateService", GetLastError());
253                 return FALSE;
254         } else {
255                 puts("Service Installed");
```

```
256                 ServiceRun();
257         }
258
259         //clean up
260         CloseServiceHandle(newService);
261         CloseServiceHandle(scm);
262         return TRUE;
263 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

4.7.2.6 int ServiceControl (char * *CONTROL*)

Send other controls to the service.

Asks the **Service Control Manager** (p. 72) to control the service on the local machine based on the parameter.

The service name and display name are both derived from the name field returned by init.lua.

Parameters:

CONTROL The name of the control message to send. The following controls are understood:

- "STOP"

Returns:

Returns TRUE on success. The current implementation calls **ErrorHandler()** (p. 59) for all significant errors which exits the process and does not return. There appear to be no insignificant errors.

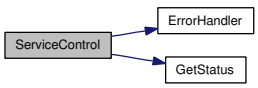
Definition at line 435 of file SvcController.c.

References ErrorHandler(), GetStatus(), and ServiceName.

Referenced by SvcControlMain().

```
435                                 {
436         SC_HANDLE service;
437         SC_HANDLE scm;
438         BOOL SUCCESS= FALSE;
439         SERVICE_STATUS status;
440 
441         //Open connection to SCM
442         scm = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
443         if (!scm)
444                 ErrorHandler("OpenSCManager", GetLastError());
445 
446         //Get service's handle
447         service = OpenService(scm, ServiceName, SERVICE_ALL_ACCESS);
448         if (!service)
449                 ErrorHandler("OpenService", GetLastError());
450 
451         //stop the service
452         if (stricmp(CONTROL, "STOP") == 0) {
453                 puts("Service is stopping...");
454                 SUCCESS = ControlService(service, SERVICE_CONTROL_STOP, &status);
455         }
456 #ifdef LUASERVICE_CAN_PAUSE_CONTINUE
457         //pause the service
458         else if (stricmp(CONTROL, "PAUSE") == 0)
459         {
460                 puts("Service is pausing...");
461                 SUCCESS = ControlService(service, SERVICE_CONTROL_PAUSE, &status);
462         }
463         //continue the service
464         else if (stricmp(CONTROL, "RESUME") == 0)
465         {
466                 puts("Service is resuming...");
467                 SUCCESS = ControlService(service, SERVICE_CONTROL_CONTINUE, &status);
468         }
469 #endif
470         if (!SUCCESS)
471                 ErrorHandler("ControlService", GetLastError());
472         else
473                 GetStatus(service);
474 
475         //Clean up
476         CloseServiceHandle(service);
477         CloseServiceHandle(scm);
478 
479         return TRUE;
480 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

4.7.2.7 int ServiceRun ()

Start the service running.

Asks the **Service Control Manager** (p. 72) to run the service on the local machine.

The service name and display name are both derived from the name field returned by init.lua.

Returns:

Returns TRUE on success. The current implementation calls **ErrorHandler()** (p. 59) for all significant errors which exits the process and does not return. There appear to be no insignificant errors.

Definition at line 338 of file SvcController.c.

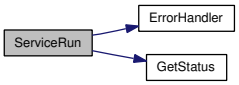
References ErrorHandler(), GetStatus(), and ServiceName.

Referenced by InstallService(), and SvcControlMain().

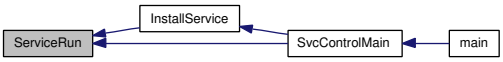
```
338                 {
339         SC_HANDLE scm, Service;
340         SERVICE_STATUS ssStatus;
341         DWORD dwOldCheckPoint;
342         DWORD dwStartTickCount;
343         DWORD dwWaitTime;
344         DWORD dwStatus;
345
346         //open connection to SCM
347         scm = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
348         if (!scm)
349                 ErrorHandler("OpenSCManager", GetLastError());
350
351         //open service
352         Service = OpenService(scm, ServiceName, SERVICE_ALL_ACCESS);
353         if (!Service) {
354                 ErrorHandler("OpenService", GetLastError());
355                 return FALSE;
356         } else {
357                 //start service
358                 StartService(Service, 0, NULL);
359                 GetStatus(Service);
360
361                 // Check the status until the service is no longer start pending.
362                 if (!QueryServiceStatus(Service, &ssStatus) )
363                         ErrorHandler("QueryServiceStatus", GetLastError());
```

```
364                 // Save the tick count and initial checkpoint.
365                 dwStartTickCount = GetTickCount();
366                 dwOldCheckPoint = ssStatus.dwCheckPoint;
367
368                 while (ssStatus.dwCurrentState == SERVICE_START_PENDING) {
369                         // Do not wait longer than the wait hint. A good interval is
370                         // one tenth the wait hint, but no less than 1 second and no
371                         // more than 10 seconds.
372                         dwWaitTime = ssStatus.dwWaitHint / 10;
373
374                         if (dwWaitTime < 1000)
375                                 dwWaitTime = 1000;
376                         else if (dwWaitTime > 10000)
377                                 dwWaitTime = 10000;
378
379                         Sleep(dwWaitTime);
380
381                         // Check the status again.
382                         if (!QueryServiceStatus(Service, &ssStatus) )
383                                 break;
384
385                         if (ssStatus.dwCheckPoint > dwOldCheckPoint) {
386                                 // The service is making progress.
387                                 dwStartTickCount = GetTickCount();
388                                 dwOldCheckPoint = ssStatus.dwCheckPoint;
389                         } else {
390                                 if (GetTickCount()-dwStartTickCount > ssStatus.dwWaitHint) {
391                                         // No progress made within the wait hint
392                                         break;
393                                 }
394                         }
395                 }
396
397                 if (ssStatus.dwCurrentState == SERVICE_RUNNING) {
398                         GetStatus(Service);
399                         dwStatus = NO_ERROR;
400                 } else {
401
402                         puts("\nService not started.");
403                         printf("  Current State: %ld\n", ssStatus.dwCurrentState);
404                         printf("  Exit Code: %ld\n", ssStatus.dwWin32ExitCode);
405                         printf("  Service Specific Exit Code: %ld\n",
406                                         ssStatus.dwServiceSpecificExitCode);
407                         printf("  Check Point: %ld\n", ssStatus.dwCheckPoint);
408                         printf("  Wait Hint: %ld\n", ssStatus.dwWaitHint);
409                         dwStatus = GetLastError();
410                 }
411         }
412
413         CloseServiceHandle(scm);
414         CloseServiceHandle(Service);
415         return TRUE;
416 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



4.7.2.8 void ShowUsage ()

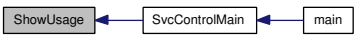
Print a command-line usage message.

Definition at line 188 of file SvcController.c.

Referenced by SvcControlMain().

```
188                 {
189         printf(
190                         "Usage:\n"
191                         "LuaService -i\tInstall service\n"
192                         "LuaService -u\tUninstall service\n"
193                         "LuaService -r\tRun service\n"
194                         "LuaService -s\tStop service\n"
195 #ifdef LUASERVICE_CAN_PAUSE_CONTINUE
196                         "LuaService -p\tPause service\n"
197                         "LuaService -c\tResume service\n"
198 #endif
199                         "LuaService status\tCurrent status\n"
200                         "LuaService help\tDisplay this text\n"
201                         );
202 }
```

Here is the caller graph for this function:



4.7.2.9 int SvcControlMain (int *argc*, char * *argv*[ ])

Entry point for service control and configuration.

Called from **main()** (p. 35) if the program is not running as a service and responsible to the SCM.

Parameters:

argc Count of arguments in *argv*.

argv Array of arguments.

Returns:

Exit status, as from **main()** (p. 35).

Definition at line 39 of file SvcController.c.

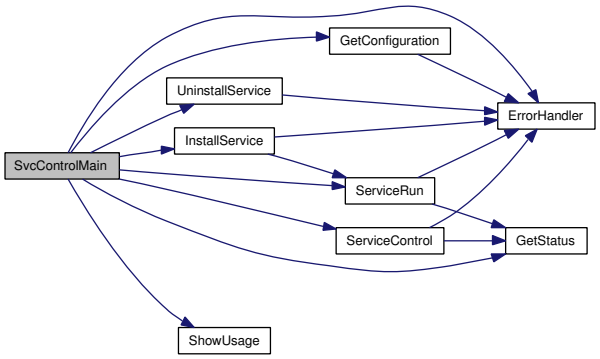
References ErrorHandler(), GetConfiguration(), GetStatus(), InstallService(), ServiceControl(), ServiceName, ServiceRun(), ShowUsage(), and UninstallService().

Referenced by main().

```
39                                          {
40         if (argc == 2) {
41                 if (stricmp("-i", argv[1]) == 0)
42                         InstallService();
43                 else if (stricmp("-u", argv[1]) == 0)
44                         UninstallService();
45                 else if (stricmp("-r", argv[1]) == 0)
46                         ServiceRun();
47                 else if (stricmp("-s", argv[1]) == 0)
48                         ServiceControl("STOP");
49 #ifdef LUASERVICE_CAN_PAUSE_CONTINUE
50                 else if (stricmp("-p", argv[1]) == 0)
51                 ServiceControl("PAUSE");
52                 else if (stricmp("-c", argv[1]) == 0)
53                 ServiceControl("RESUME");
54 #endif
55                 else if (stricmp("status", argv[1]) == 0) {
56                         SC_HANDLE scm, service;
57                         //Open connection to SCM
58                         scm = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
59                         if (!scm)
60                                 ErrorHandler("OpenSCManager", GetLastError());
61                         //Get service's handle
62                         service = OpenService(scm, ServiceName, SERVICE_ALL_ACCESS);
63                         if (!service)
64                                 ErrorHandler("OpenService", GetLastError());
65                         fputs("STATUS: ", stdout);
66                         GetStatus(service);
67                 } else if (stricmp("config", argv[1]) == 0)
68                         GetConfiguration();
69                 else if (stricmp("help", argv[1]) == 0)
70                         ShowUsage();
71                 //add other custom commands here and
72                 //update ShowUsage function
73                 else {
```

```
74                          ShowUsage();
75                          return EXIT_FAILURE;
76                  }
77          } else {
78                  ShowUsage();
79                  return EXIT_FAILURE;
80          }
81          return EXIT_SUCCESS;
82 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

4.7.2.10 int UninstallService ()

Uninstall the service.

Asks the **Service Control Manager** (p. 72) to uninstall the service from the local machine.

The service name and display name are both derived from the name field returned by init.lua.

If the service is currently running, the SCM is told to stop the service and before the uninstall can continue.

Returns:

Returns TRUE on success. The current implementation calls **ErrorHandler()**

(p. 59) for all significant errors which exits the process and does not return. There appear to be no insignificant errors.

Definition at line 281 of file SvcController.c.

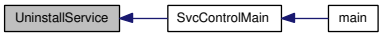
References ErrorHandler(), and ServiceName.

Referenced by SvcControlMain().

```
281                     {
282         SC_HANDLE service;
283         SC_HANDLE scm;
284         int SUCCESS;
285         SERVICE_STATUS status;
286 
287         //Open connection to SCM
288         scm = OpenSCManager(NULL, NULL, SC_MANAGER_CREATE_SERVICE);
289         if (!scm)
290                 ErrorHandler("OpenSCManager", GetLastError());
291 
292         //Get service's handle
293         service = OpenService(scm, ServiceName, SERVICE_ALL_ACCESS | DELETE);
294         if (!service)
295                 ErrorHandler("OpenService", GetLastError());
296 
297         //Get service status
298         SUCCESS = QueryServiceStatus(service, &status);
299         if (!SUCCESS)
300                 ErrorHandler("QueryServiceStatus", GetLastError());
301 
302         //Stop service if necessary
303         if (status.dwCurrentState != SERVICE_STOPPED) {
304                 puts("Stopping service...");
305                 SUCCESS = ControlService(service, SERVICE_CONTROL_STOP, &status);
306                 if (!SUCCESS)
307                         ErrorHandler("ControlService", GetLastError());
308                 Sleep(500);
309         }
310 
311         //Delete service
312         SUCCESS = DeleteService(service);
313         if (SUCCESS)
314                 puts("Service Uninstalled");
315         else
316                 ErrorHandler("DeleteService", GetLastError());
317 
318         //Clean up
319         CloseServiceHandle(service);
320         CloseServiceHandle(scm);
321 
322         return TRUE;
323 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



5 LuaService Page Documentation

5.1 Event Flow Overview

5.1.1 The Key Entities

5.1.1.1 CMD The command prompt or other interactive program such as the computer manager.

5.1.1.2 Controller A generic service control program, which could be a separate utility such as the Services snap-in to Computer Manager, a developer tool such as SC.EXE or the Service Explorer in Visual Studio 2005, or even a special mode of the same executable that acts as the service to be controlled.

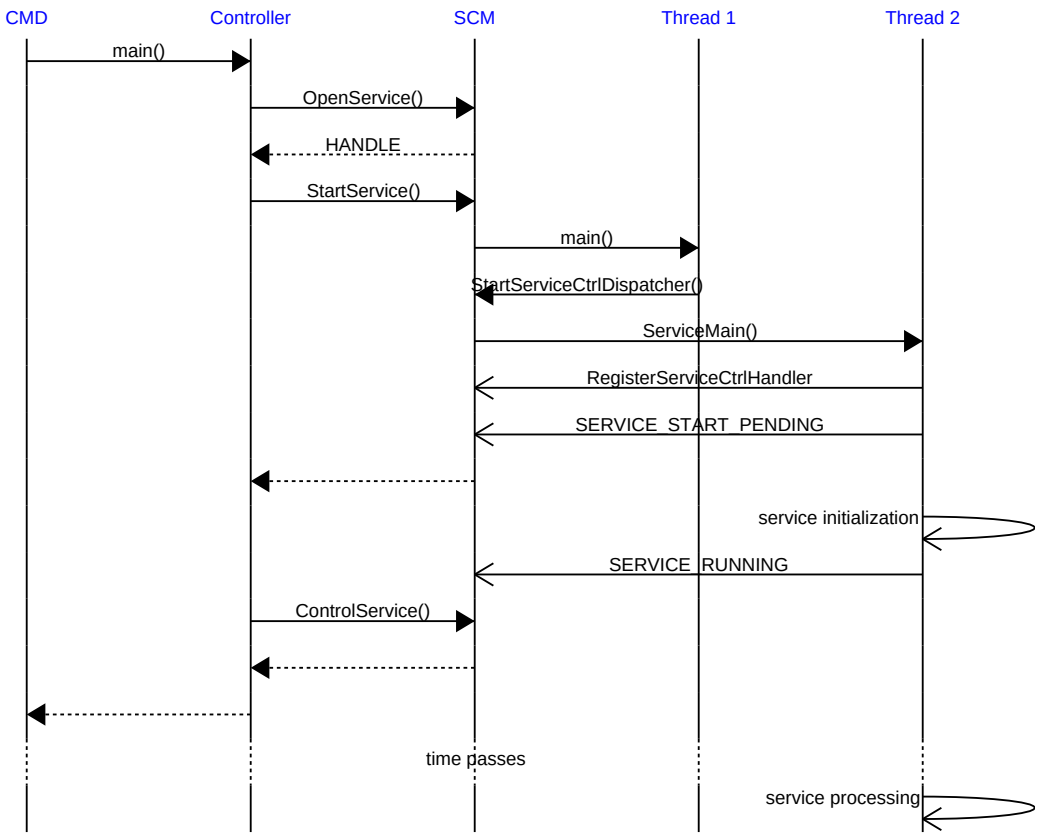
5.1.1.3 Service Control Manager This is a Windows system component that is responsible for starting, managing, and stopping all service applications.

5.1.1.4 Service Application Threads A running service consists of at least two threads. The first is created normally when the system starts the process, and is used to execute the **main()** (p. 35) function, and subsequently to execute the request callback function (i.e. the **LuaServiceCtrlHandler()** (p. 29) function) that handles all service control requests.

The other threads include a worker thread for each service provided by the application (only one in LuaService) that is started by a successful call to StartServiceCtrlDispatcher(). That thread is responsible for per-service initialization, and any ongoing work the service is designed to do. Some kinds of services can allow this thread to exit once initialization is complete. Others may need additional worker threads which should be created during initialization and carefully managed to avoid leakage of allocated resources and misunderstandings about the overall state of the service by the **Service Control Manager** (p. 72).

5.1.2 Starting a Service

The following figure shows the chain of events triggered by a user starting a generic service through a service control program.



1. The user launches some **Controller** (p. 72) program.

2. The controller retrieves a handle to the service by exchanging its name for the handle from the **Service Control Manager** (p. 72).

3. The controller passes the handle to the SCM via a call to StartService().

4. The SCM confirms that the service is not running and needs to start. It discovers from the service database (implemented in the registry) the executable file and command line arguments set when the service was installed.

5. The SCM launches the service application. Its **main()** (p. 35) function starts, and nearly immediately (there is a time limit of 30 seconds before the SCM abandons the service as broken beyond repair) it calls StartServiceCtrlDispatcher() to register the service's name and ServiceMain() entry point. If this application provides multiple services, they are all registered at the same time, and each gets its own thread in which its own ServiceMain() is executed.

6. The SCM starts a thread for each ServiceMain().

7. ServiceMain() calls RegisterServiceCtrlHandler() to identify the callback function used to control that service.

8. ServiceMain() calls SetServiceStatus() to promise that the service is starting (SERVICE_START_PENDING) and to hint about the time required for startup.

9. ServiceMain() does any required service initialization. If the initialization is lengthy, it may be required to make additional calls to SetServiceStatus() to reassure the SCM that it is making progress.

10. If all went well, ServiceMain() calls SetServiceStatus() to announce that the service is started (SERVICE_RUNNING). Otherwise, it will announce SERVICE_STOPPED and provide the failure status codes that the SCM will include in a system event log entry.

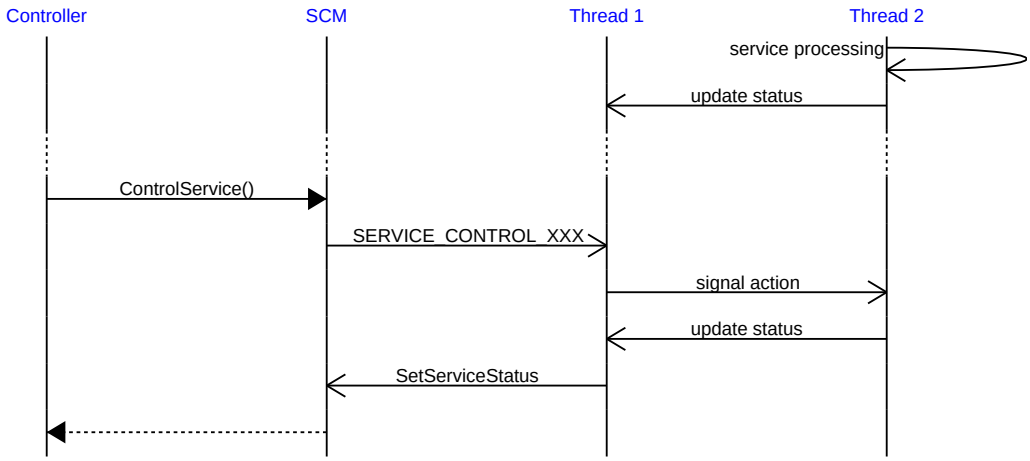
11. The controller's call to StartService() returns at any point after the SCM has tried to launch the service application. The controller can optionally use calls to ControlService() with the request SERVICE_CONTROL_INTERROGATE to monitor the startup progress.

12. The ServiceMain() thread can act as the service's worker thread, interacting with the registered control handler callback function which will be called on **main()** (p. 35)'s thread. This interaction is described in the section **Controlling a Service** (p. 74).

5.1.3 Controlling a Service

The following figure shows the chain of events triggered by a user or the system sending control requests or other notifications to a running generic service. A user sends requests through a service control program. The system acts like a service control

program to send certain notifications such as network reconfiguration events, device removal, and system shutdown. In the following figure, the controller is either a program acting for the user or the system itself, and the distinction is unimportant.

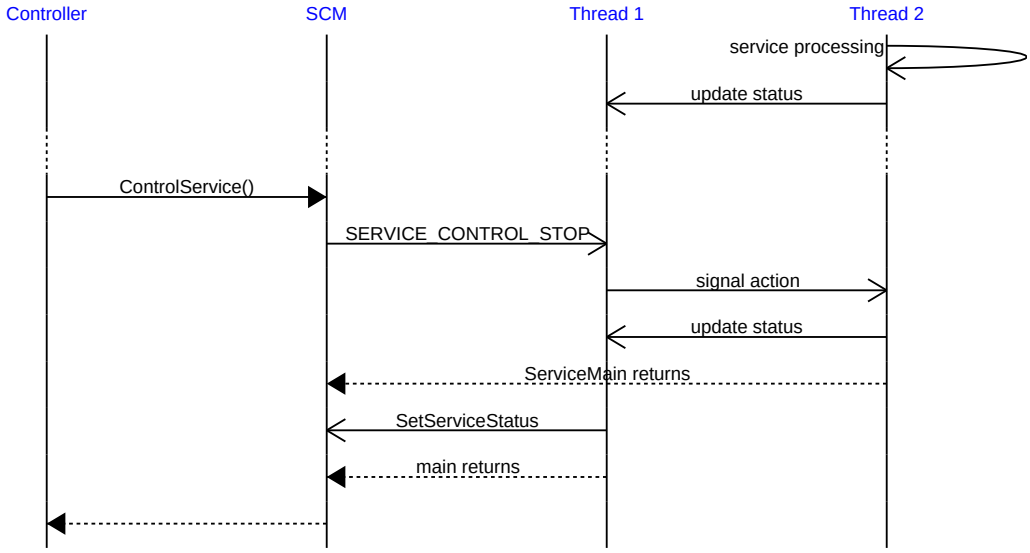


1. The running service probably has a ServiceMain() thread and may have additional threads that are doing whatever it is that this service does.
2. As the service's status changes, it is passed to the **main()** (p. 35) thread for immediate use by an INTERROGATE control request.
3. The **Controller** (p. 72) has a handle for the service that it got from the **Service Control Manager** (p. 72) earlier. It sends a control request to the service via the SCM by calling ControlService().
4. Assuming that the service is thought to be running, the SCM passes the control message by calling the registered control callback function for the service.
5. The callback (which executes in the service's **main()** (p. 35) thread) takes whatever action is required by the control message. Controls that will take a long time to execute should return back an appropriate XXX_PENDING response in a timely fashion and expect that the controller (via the SCM) will monitor its progress with additional INTERROGATE control messages.
6. Acting on the control likely requires that a signal be passed to the ServiceMain() thread and/or any other worker threads involved in the service's implementation.

7. The services current status is returned to the SCM by calling SetServiceStatus().
8. The SCM returns the status to the controller.

5.1.4 Stopping a Service

Stopping a running service is a special case of **Controlling a Service** (p. 74).



1. The running service probably has a ServiceMain() thread and may have additional threads that are doing whatever it is that this service does.
2. As the service's status changes, it is passed to the **main()** (p. 35) thread for immediate use by an INTERROGATE control request.
3. The **Controller** (p. 72) has a handle for the service that it got from the **Service Control Manager** (p. 72) earlier. It sends a STOP control request to the service via the SCM by calling ControlService().
4. Assuming that the service is thought to be running, the SCM passes the control message by calling the registered control callback function for the service.

5. The callback (which executes in the service's **main()** (p. 35) thread) takes whatever action is required by the control message. To STOP the service, it must ask all worker threads belonging to the service to stop any pending work, release any resources, and exit. The ServiceMain() thread must exit by returning from ServiceMain() because there are resources held by that thread that will not be properly released if it ends by any other means.

6. The service cannot stop if other services are dependant on it. Handling this case requires the controller to ask the dependant services to stop, waiting for that to complete, and then asking the target service to stop which should then be possible. That sequence is not shown here.

7. The service can (but should not) ignore the STOP request by returning NO_-ERROR from the handler function without actually stopping. While allowed, this seems like a bad idea.

8. If the service needs extra time to stop, for example because it must wait on several worker threads to safely release their resources, it can return STOP_-PENDING along with a timing hint in a timely fashion and expect that the controller (via the SCM) will monitor its progress with additional INTERROGATE control messages.

9. Once all associated resources and threads are released, it makes its status STOPPED and informs the SCM with a call to SetServiceStatus().

10. Assuming that all the services hosted in this service application are now STOPPED, the **main()** (p. 35) thread exits by returning from **main()** (p. 35) just like any normal process. If it were to return from **main()** (p. 35) before its worker threads ended, they would be killed without a chance to clean up and release any resources they hold; which is not recommended.

11. The SCM returns the status to the controller.

5.2 Lua Usage Model

The framework code provides the basic structure of a service, especially most of the interface between Windows and a service. It supports all of the interaction with the **Service Control Manager** (p. 72) needed to start and stop the service.

5.2.1 LuaService Configuration

An installed LuaService will consist of a folder that holds the framework executable, and at least two Lua source files.

The first Lua file must be named init.lua, and contains a Lua script that returns a single table value. The Lua state it runs in should not be expected to survive into the actual

service thread, so although init.lua can modify its global environment, it can't expect that those changes will have any effect.

The table returned by init.lua must define a service name, which will be used to identify the service to the SCM by both a service controller and the service application. Defining the name at run time allows the same service executable to be used to host any number of distinct services written in Lua on the same Windows machine without recompilation. Note, however, that each service is still hosted by its own service application and that in that case the SCM doesn't always seem to be picky about the service name registered when the service application runs.

The table returned by init.lua should name the service's main script. If not specified, "service.lua" will be assumed. The script must be in the same folder as LuaService.exe and init.lua, or a subfolder of that folder.

```
-- The file init.lua
return {
    name = "UniqueServiceName",
    script = "start.lua",
}
```

5.2.2 LuaService Runtime

The framework assumes that it will be home to only one service. This simplifies implementation details related to handling more than one service hosted in a single process, especially with respect to service lifetimes as opposed to process lifetime. A future version could consider the implications of supporting multiple Lua services hosted in a single application. One natural mechanism would be to host separate Lua contexts in each service thread.

The framework will initialize a single Lua state during service initialization in the worker thread, and will only use that state from the worker thread. The Lua state will contain a global object acting as a proxy for the state of the service, and providing the means for the worker code to indicate its status and progress to the main thread and the SCM.

Functions available from Lua will allow the worker to signal status updates to the SCM. Support will also be provided for Windows API calls handy when debugging a service, such as OutputDebugString(). Support for other Windows API routines should be provided by modules outside the framework proper.

The worker thread will have a means of polling to discover if and when a shutdown request has been delivered. The implementation of that mechanism must take thread safety into account because the service request handler runs in the context of the main thread and therefore does not have access to the worker thread's Lua state.

If a shutdown is requested, the main thread will wait a few seconds for the worker to notice and obey, but unless the worker switches to STOP_PENDING in a timely fashion, it will stop the worker and exit the service program more forcefully.

There should be no real issue with the worker using require() to load modules that provide other functionality. A means will be demonstrated for modules to be statically linked to the framework but not loaded into Lua until requested. The package.path and package.cpath strings will be configured to look for modules only in the LuaService installation folder and in subfolders there. It will explicitly ignore environment variables and the usual default values of the path strings to avoid opening an inadvertant security hole by allowing a module to be loaded from a less trustworthy location.

5.3 Using the LuaService Framework

LuaService is a Windows Service application that hosts a Lua interpreter.

(Future versions may also act as a Windows Service controller, perform configuration tasks, or even run the service's script interactively to aid debugging.)

A LuaService must include the files LuaService.exe and init.lua at a bare minimum. In addition, it should include a Lua script that implements the service (named in init.lua), and may also include any Lua modules (in either .lua or .dll files) that are required to run the service script.

Note that the module search paths specified by package.path and package.cpath are restricted to the service's folder, so any needed modules should be copied into that folder in order for require() to locate them when the service starts.

The LuaService executable can also be used to install, uninstall, start, and stop the service. However, its service installer has limitations (for instance, it can't install the service to run as a specific user) that will require the use of another service installation utility for some configurations. SC.EXE is a generic service configuration and control utility that may be used for these tasks. The Services snapin for the Microsoft Management Console may also be used to start and stop LuaService.

5.3.1 service Table

Both the init and service scripts have a global table named `service` available. That table has the following informational fields:

- `service.name` The name of the service as specified by init.lua, defaulting to "LuaService". This name is known to the SCM, and is found in the registry.
- `service.filename` The fully qualified filename of the service program.
- `service.path` The fully qualified path to the service folder. This is just `service.filename` with LuaService.exe stripped off, and can be assumed to end in a literal backslash character.

It also defines the following utility functions:

- `service.sleep(ms)` Calls Sleep() to make the thread sleep for *ms* ms.

- `service.print(...)` Like standalone Lua.exe's print(), but with its output written to OutputDebugString() instead of stdout, and without any separator characters between print's arguments. Like the stock print(), it passes every argument through the function tostring(). For safety since services don't have access to the user, the global function print is replaced by a copy of this function.

- `service.stopping()` Returns true if the SCM has asked that this service stop soon. The service's main thread has promised the SCM that the STOP request will complete within about 25 seconds, so the script has an obligation to poll this function often enough to be able to stop in time.

- `service.tracelevel(level)` If *level* is not present or is nil, returns the current trace level. If *level* is specified, it is converted to an integer and sets the current trace level. Level 0 (the default) keeps the framework quiet. Levels greater than zero add additional detail to the trace.

5.3.2 Init Script

The init script is executed when LuaService is initially run from its **main()** (p. 35) function. Its purpose is to configure the service executable so that it can find its implementation script.

The Lua state used to load and execute init.lua is abandoned once init.lua returns, so nothing this script does has any effect on the service's execution except its return value.

init.lua should return a table with several named fields.

- `tracelevel` The framework trace level. Defaults to 0.
- `name` The service's name. Defaults to "LuaService".
- `script` The service's implementation script. Defaults to "service.lua".

The following fragment is a sample init.lua for an imaginary Ticker service:

```
-- init.lua for the Ticker service
return {
tracelevel = 0,-- Framework trace level
name = "TickService",-- Service name for SCM
script = "test.lua",-- Script that runs the service
}
```

5.3.3 Service Script

The service implementation script is the whole point of having LuaService.

It is named by the script field of the table returned by `init.lua`, and must be found in the service folder. It may require lua modules, as long as those modules are also found in the service folder.

The following fragment is a sample service script (named test.lua because that is the name used in the **Init Script** (p. 80) `init.lua` shown above) for an imaginary Ticker service:

```
-- Ticker service implementation
--[[----------------------------
This service simply emits a string on the debug console once
every 5 seconds until it is stopped. Not necessarily the most
useful service, but it demonstrates how to construct a service
in the LuaService framework without too many other system
assumptions.

Copy the LuaService executable to this folder.

To use sc.exe to create, start, and stop the service:

sc create TickService binPath= "C:\path\to\this\folder\LuaService.exe"
sc start TickService
sc stop TickService

To use LuaService to create, start, and stop the service:

LuaService -i Create and start the ticker service
LuaService -r  Start the service
LuaService -s Stop the service
LuaService -u Uninstall the service

You will want a debug console that is listening to OutputDebugString() to
see this service do anything at all. DebugView from www.sysinternals.com
is a good choice.
--]]----------------------------

service.print("Ticker service started, named ", service.name)

local i = 0 -- counter
while true do -- loop forever
  service.sleep(5000) -- sleep 5 seconds
  i = i + 1 -- count
  print("tick", i) -- OutputDebugString
  if service.stopping() then -- Test for STOP request
   break --  " and halt service if requested
  end
end
service.print("Ticker service stopped.")
```

Note:

Unless an account is specified when the service was installed, this script is running

in the LocalSystem security context. This is a built-in account in Windows that has a high level of access to the local system (more than the Administrator account), but has very limited access to network resources. As a result, it is a really good idea to keep the service's fold on a disk that is physically attached to the system. LuaService probably cannot run if the folder is on a network drive.

5.4 Building LuaService

To build LuaService from source, you will need many of the tools listed in **Supporting Tools** (p. 83), particularly MinGW and Eclipse including the CDT. In particular, the project does not currently supply either a conventional Makefile or a Visual Studio workspace or project.

5.4.1 Building LuaService.exe

I have found it convenient to use Eclipse to check the project out from the CVS repository at LuaForge, which gets you an Eclipse project in your workspace. With that project loaded, Project|Clean and Project|Build All will get you a complete build of your default configuration.

Package Release\LuaService.exe along with a copy of lua5.1.dll and your custom init.lua and the service's Lua implementation in a single folder. Run `LuaService -i` at a command prompt in your service's folder to install the service in in the SCM's database and start it.

5.4.2 Building the Documentation

To build the documentation, you need to install doxygen, dot, and msggen at a minimum. With these tools installed, the HTML documentation pages can be build by running `doxygen LuaService.doxyfile` at a command prompt in the LuaService workspace folder. To make documentation builds easier, I recommend adding the Eclox plugin to your Eclipse installation. It adds a custom editor for doxyfiles, and adds a toolbar button that runs doxygen in your workspace.

By default, the HTML output is configured to prepare the HTML for collection into a Windows HTML Help file named LuaService.chm. That can only happen if you have the HTML Help Workshop installed, and confirm that the setting for `HHC_LOCATION` in `LuaService.doxyfile` correctly names your installed copy of the HTML Help Compiler HHC.EXE. If that isn't possible, change that setting to have no value and doxygen will stop trying to run the compiler for you. If you have the HTML Help Workshop, the compiled CHM file will be written to `doc\LuaService.chm`. Either way, `doc\html\index.html` will be the root of the HTML documentation set.

To build the PDF document from LaTeX sources under Windows, you will need a LaTeX installation. I have found MiKTeX to be suitable. You also need to post-process

the .tex files written by doxygen to correct some small issues with their LaTeX usage, especially if you let the Eclipse installer locate your workspace folder in the default location on Windows XP (`C:\Documents and Settings\user\workspace\`) because some of the .tex files include references to other files that include the full path name without proper quotation of the spaces. A Perl script found in `...\workspace\LuaScript\fixlatex.plx` makes a temporary copy of LuaService.doxyfile with only the latex output enabled, runs doxygen, edits the generated .tex files, rewrites the latex Makefile to be more consistent with MinGW usage at a command prompt, runs make in the latex folder, and copies the resulting pdf to doc\LuaService.pdf.

5.5 Supporting Tools

Most of these tools aren't mandatory (well, aside from a compiler of some sort that does produce native Win32 binaries without too many wierd DLL dependancies), but they make my life easier during lots of development projects.

- Eclipse platform and IDE, and especially the Eclipse CDT: See `http://www.eclipse.org`, `http://www.eclipse.org/cdt`
- Eclox, a Doxygen frontend plugin for Eclipse. Visit `http://home.gna.org/eclox/`
- Doxygen, a documentation tool for C-like languages. Found at `http://www.doxygen.org/`
- Graphviz Dot, a directed acyclic graph visualization tool used by Doxygen to draw structure and relationship figures. This is a great tool for visualizing arbitrary complex network diagrams since it works from a natural text format describing the nodes and connecting arcs to produce neatly drawn figures. Find it at `http://www.graphviz.org/`
- MSCGen, a Message Sequence Chart generator that integrates well with Doxygen. Find it at `http://www.mcternan.me.uk/mscgen/` and put it in your path so that the @msc and @endmsc directives work.
- MiKTeX, the premier distribution of TeX and LaTeX for Windows. Find it at `http://miktex.org/`. To keep doxygen happy, you probably want to run its package browser and locate and install the optional fancyheader package.
- LuaEclipse, an integrated development environment for the Lua programming language. `http://luaeclipse.luaforge.net/`

- The Lua programming language: `http://www.lua.org/`

- CVS (`http://www.nongnu.org/cvs/`) and CVSNT (`http://www.cvsnt.org/`) The latter is more useful on Windows machines, but beware of its insidiously useful improvements over real CVS, like the cvs ls command and the cvs status -qq command. Once you get used to status having the -qq option as a quick and dirty way to find out what is different in your sandbox, it is really difficult to go back to the real thing. Eclipse's native CVS browsing helps there, however.

- PuTTY, an SSH implementation for Windows: `http://www.chiark.greenend.org.uk/~sgtatham/putty/`

- Diceware at `http://www.diceware.org/` for a good treatment of proper pass phrase generation and management. Look here if you want to be seriously paranoid about private key safety.

- MinGW, minimalist GNU for Windows: `http://www.mingw.org/`

- MSDN is the source of all knowledge of official Windows API and workings.

- DebugView, ProcessMonitor, and ProcessExplorer are invaluable for finding out what is really going on inside a running Windows machine without stepping off the deep end of full kernel debuggin. A serious Windows developer must have these and often other utilities from Sysinternals, now found at Microsoft: `http://www.microsoft.com/technet/sysinternals/`

- SC.EXE, a generic service control utility for developers. This is part of the Windows Platform SDK tools, but may be present on modern windows systems. Try a command like `sc /?` to see if you have it installed already. Some useful incantations include:

  - `sc help` Documentation. Also see the technote in MSDN titled `Using SC.EXE to Develop Windows NT Services` for hints and tips.
  - `sc create LuaService binPath= "c:\full\path\of\LuaService.exe"` Note the space after "binPath=" and the file name. All of sc create's options have that same form, with the option name including the equals character, and using the next command line argument as the value. Don't forget to quote the pathname if it includes any spaces.
  - `sc start LuaService` Tell the SCM to start the service.
  - `sc stop LuaService` Tell the SCM to stop the service.

5.6 License

5.7 Todo List

Global dbgGetCurrentDirectory (p. **10**) There is a low-probability memory leak here. The buffer used to hold the current directory string came from malloc() and is held across a call to lua_pushlstring() which can potentially throw an error, which will leak the allocated buffer. The other bits of Win32 API wrappers could have similar issues, and should be inspected.

Global panic (p. **22**) Should **panic()** (p. 22) also tell the SCM SERVICE_STOPPED?

File LuaService.c (p. **27**) Supporting service PAUSE and CONTINUE control request will require some effort beyond the bare framework guarded by the undefined macro LUASERVICE_CAN_PAUSE_CONTINUE. At minimum, some mechanism must be provided for the Lua side to become aware of the request and actually pause; presumably an Event could be waited on to implement the pause, and signaled to implement continue. However, since we assume that the Lua interpreter itself is not built for threading, we don't have a good means to asynchronously notify the Lua code of the pause request in the first place, which would imply that the Lua code is constantly polling.

Global LuaServiceMain (p. **32**) Should LuaService push its Lua implementation into a second worker thread that has its CRT properly initialized by using _beginthreadex() to create it instead of CreateThread()?

Global main (p. **35**) We could also support running our service thread interactively to support easier debugging. If that is done, then we should consider making **SvcDebugTrace()** (p. 37) and friends, as well as the implementation of service.print() for Lua write to stdout rather than OutputDebugString().

File SvcController.c (p. **57**) This mechanism was copied from sample code, and lacks friendly handling of command line arguments. Future versions will likely move some of the less critical details into a Lua script, with the actual service control methods exposed via a built-in module.

Index