Calling Nautilus functions from Lua interpreter.

Developer's Manual

By:-

Goutham Kannan

Imran Ali Usmani

Under the guidance of :-

Prof. Kyle Hale

Illinois Institute of Technology

This is a developer's manual which talks about how to enable lua to call nautilus functions, i.e. to call nautilus c functions from lua command prompt. These are the steps we followed:

**1. Port Lua to nautilus kernel.**

To run lua over nautilus first we need to port lua code to run in nautilus. Please refer to the document Porting\_Lua\_for\_Nautilus,pdf for detailed information on the steps involved in porting.

**2. Compile nautilus with debug options enabled.**

We compile nautilus code (without lua) with debug options on, i.e. adding *-g* options to *CFALGS* in the *MakeFile* present at the base directory.

**3. Save nautilus.bin information into a text file using readelf command.**

We log the nautilus.bin’s debugger information using the command:

#readelf -wi nautilus.bin >full\_log.txt

**4. Parse the full\_log.txt file to extract all the nautilus functions prototype information.**

Parse the nautilus functions one by from full\_log.txt file. Here is an example:

**Parsing functions information:**

*<0><3201>: Abbrev Number: 1 (DW\_TAG\_compile\_unit)*

*<3202> DW\_AT\_producer : (indirect string, offset: 0xd): GNU C 4.8.4 -mno-red-zone -mcmodel=large -m64 -mtune=generic -march=x86-64 -g -O2 -std=gnu99 -fno-omit-frame-pointer -ffreestanding -fno-stack-protector -fno-strict-aliasing -fno-strict-overflow -fno-common -fgnu89-inline*

*<3206> DW\_AT\_language : 1 (ANSI C)*

*<3207> DW\_AT\_name : (indirect string, offset: 0x1125): src/nautilus/vc.c*

*<320b> DW\_AT\_comp\_dir : (indirect string, offset: 0x228): /home/imran/nautilus*

The <0> tag on the very left tells about a start of a particular c file information till the next <0> tag. We have to look for files inside the '/src/nautilus' directory.

*<1><1d3e>: Abbrev Number: 66 (DW\_TAG\_subprogram)*

*<1d3f> DW\_AT\_external : 1*

*<1d3f> DW\_AT\_name : (indirect string, offset: 0x571): vsnprintf*

*<1d43> DW\_AT\_decl\_file : 1*

*<1d44> DW\_AT\_decl\_line : 1090*

*<1d46> DW\_AT\_prototyped : 1*

*<1d46> DW\_AT\_type : <0x74c>*

*<1d4a> DW\_AT\_low\_pc : 0x3063d0*

*<1d52> DW\_AT\_high\_pc : 0xe58*

*<1d5a> DW\_AT\_frame\_base : 1 byte block: 9c (DW\_OP\_call\_frame\_cfa)*

*<1d5c> DW\_AT\_GNU\_all\_call\_sites: 1*

*<1d5c> DW\_AT\_sibling : <0x23e8>*

*<2><1d60>: Abbrev Number: 33 (DW\_TAG\_formal\_parameter)*

*<1d61> DW\_AT\_name : buf*

*<1d65> DW\_AT\_decl\_file : 1*

*<1d66> DW\_AT\_decl\_line : 1090*

*<1d68> DW\_AT\_type : <0xa2e>*

*<1d6c> DW\_AT\_location : 0x43d4 (location list)*

The <1> tag with 'DW\_TAG\_subprogram' string in same line means start of a function description till next <1> tag.

<2> tag with 'DW\_TAG\_formal\_parameter' string after that means the description about parameters of the function.

'DW\_AT\_name' before <2> tag, means the function name.

'DW\_AT\_type' before <2> tag, has the address where the information about the return type can be found, i.e. at the address <74c> in the above case.

If a function description contains 'DW\_AT\_inline' equals to 1, it is static. We don't need to parse that.

If a function description contains 'DW\_AT\_low\_pc' and 'DW\_AT\_high\_pc', it is also static unless it contains 'DW\_AT\_external' equals to 1.

'DW\_AT\_name' after <2> tag means the parameter name.

'DW\_AT\_type' after <2> tag has the address where its type information can be found, i.e. at the address <a2e>.

**Parsing parameter information:**

*<1><a2e>: Abbrev Number: 17 (DW\_TAG\_pointer\_type)*

*<a2f> DW\_AT\_byte\_size : 8*

*<a30> DW\_AT\_type : <0x811>*

If the type information contains 'DW\_TAG\_pointer\_type' means parameter is a pointer and the pointer type information at the address <811> in the above case.

*<1><278d>: Abbrev Number: 6 (DW\_TAG\_typedef)*

*<278e> DW\_AT\_name : (indirect string, offset: 0x4be): uchar\_t*

*<2792> DW\_AT\_decl\_file : 3*

*<2793> DW\_AT\_decl\_line : 31*

*<2794> DW\_AT\_type : <0x2798>*

If the type information contains 'DW\_TAG\_typedef' means parameter is a typedef and the typedef information at the address <2798> in the above case.'DW\_AT\_name' contains a typedef name.

Parsing of the log files for function names in the file parse\_gdb\_functions.py. Use vi or other editors to view the implementation. The parsed function calls are then resolved to their base types in parse\_gdb.py file.

**5. Create lnautlib.c file in lua directory with nautilus function calls.**

5.1 After we get prototype information about all the nautilus functions, we create lnautlib.c file in 'src/lua\_src/' directory, where we construct lua functions which can be called from the lua command prompt. This lnautlib.c file is similar to the lstrlib.c which provides string operations or lmathlib.c which provides math operations to the lua command prompt. The lua functions inside the lnautlib.c are nothing but the wrappers around the nautilus functions we extracted from the readelf information of the nautilus.bin file.

5.2 Inside the lnautlib.c file we redeclare all the nautilus function prototypes with extern keyword. We also redeclare the structures, unions and typedefs to make them accessible in this file.

5.3 Inside the wrappers the first we add code to extract all the parameters one by one from the command prompt call, i.e. from the lua stack. Then we call the nautilus function and pass all the extracted parameters to the nautilus function. After that we add code to send the return value of the nautilus function back to lua stack, i.e. to the command prompt where it can be collected.

For example, we use the function luaL\_checkint(L,1) to read the second parameter which is an int from the command prompt and then pass it to a nautilus function call. The we collect the return value of the nautilus function and send it back to the lua stack using one of the lua functions such as lua\_pushstring(L,x), if x is a char \*. Here list of lua functions for extracting data from stack and pushing it back:

|  |  |
| --- | --- |
| datatype | Lua function |
| Double or float | luaL\_checknumber() |
| Char \* | luaL\_checkstring() |
| Int, long | luaL\_checkint() |
| Unsigned int or pointer type | luaL\_checkunsigned() or lua\_touserdata() |

Lua functions for extracting data from stack

|  |  |
| --- | --- |
| datatype | Lua function |
| Int or double or float or unsigned int or long | lua\_pushnumber() |
| Char \* | lua\_pushstring() |
| Pointer type | lua\_pushlightuserdata() |

Lua functions for pushing return values back to the stack

5.4 Then we add code to create a mapping of the wrapper function to the actual function name to be called from the command prompt. This mapping is an array of structures and would look like:

static const luaL\_Reg nautlib[] = {

{"naut\_snprintf", naut\_snprintf},

{"naut\_nk\_vc\_getchar", naut\_nk\_vc\_getchar},

… ,

}

5.5 We create the following function which would register the mapping array nautlib inside lua:

LUAMOD\_API int luaopen\_naut (lua\_State \*L) {

luaL\_newlib(L, nautlib);

}

**6. Register lnautlib.c in lua**

We register the luaopen\_naut function inside linit.c, similar to luaopen\_math function of lmathlib.c.

We also add 'lnautlib.o' in 'src/lua\_src/Makefile'.

**7. Compile the kernel again including lua.**

Now our lnautlib.c file is complete. We compile nautilus kernel again including 'src/lua\_src/' directory and our build is ready. For information on calling the nautilus functions from lua prompt see the user manual.

While compiling the lua source inside nautilus we use the conditional compilation using

Kconfig tool. Added the entry for “Lua interpreter support” as a menu option. In this menu, we added two configurations namely “LOAD\_LUA” and “LUA\_TEST”. Both the configs are initialized as bool with default as False.Use any editor to view the file /Konfig for implementation details.

**File specific information:**

1. Step 2 is implemented in **./Makefile** by adding '-g' options in CFALGS.

2. Step 3 is implemented in **./Makefile** as a shell command.

3. Step 4(Parsing functions information) is implemented in **parse\_gdb\_functions.py**.

4. Step 4(Parsing parameter information) to step 6 is implemented in **parse\_gdb.py**.

5. Step 7(Compile the kernel) by adding menu and configurations in **Kconfig**.