

```
my @ftree = map { # add to parent's Child list ( $found );
  s/\s+$/\n/; # strip trailing spaces
  my $depth = if ($s =~ /\s+$/) { $depth + 1 } else { $depth };
  $depth++;
  $parent_stack[ $depth ] = $node;
  # create node structure $parent, $stack[ $depth ] = $node;
  while ( $found -> { P: duff@pobox.com
```

# Programming Parrot

## Adventures in NCI

by Jonathan Scott Duff  
duff@pobox.com



In this article I chronicle my adventures attempting to use Parrot's Native Call Interface (NCI) and hopefully bring you along with me. I'll be using Linux since that's an environment that I'm familiar with, but I understand that the appropriate analogous incantations work on Windows as well.

### ■ What is NCI? -----

Parrot's Native Call Interface allows me to interface directly with functions defined in a C library using only Parrot code to do so. Parrot provides a routine for loading shared libraries and another routine for accessing subroutine symbols defined in those libraries by exposing them as callable Parrot Magic Cookies (PMC) thingies in parrot-land.

Why would I want to link shared libraries directly into my Parrot code? Since Parrot is designed to be the target for a multitude of programming languages, one possible reason might be to facilitate my particular programming language. For instance, if I had an old programming language and environment which acted as a database that was being pushed beyond its meager bounds, I could use Parrot to re-implement the language and environment and use NCI to access modern C libraries for the databasey bits. Another example would be if I were developing a game and needed access to sophisticated graphics routines, I could use NCI for that. Some people will recognize that I'm not just pulling these examples completely out of thin air.

### ■ Diving into the deep end -----

So, where do I get started? I think a good first step might be to write my own C function, generate a shared library, write a Parrot program that loads the library and creates a PMC interface to the function, then execute that function (just to make sure I understand how everything is supposed to work). That's a good first step for me since I've done this once before, though it's been a while. For those of you just starting out with Parrot it would help to first read the documentation available in the Parrot source tree. See the "References" section at the end of the article for more information.

I'm going to gloss over some of the details relating to compiling C source files and focus on the Parrot aspect of things. And, just to warn you, I'm going to give a quick example without much explanation and then back up and explain things in detail.

I'll start by creating a C source file with the following contents:

```
void hello(void)
{
    puts("Hello, World");
}
```

Then I compile the C file and link it into a shared object module like so:

```
$ gcc -Wall -fPIC -c hello.c
$ ld -shared -o hello.so -lc hello.o
```

And finally I write a small Parrot program to test this new shared object library. Never mind what it all means just yet, I'll explain it in a minute:

```
.sub main :main
.local pmc lib, func
# load shared library hello.so
lib = loadlib "hello"
# get a reference to the hello function
func = dlfunc lib, "hello", "v"
func()
.end
```

And in the end, I have the following files: *hello.c*, the C file that contains my subroutine; *hello.o*, the object code; *hello.so*, a dynamic library, and *hello.pir*, my test parrot program written in Parrot Intermediate Representation (PIR).

And now, for the moment of truth! Does it actually work? When I run the command, I get the error output in *Listing 1*.

#### **Listing 1: Error output from my first hello attempt**

```
$ parrot hello.pir
parrot: src/ops/core.ops:1200: Parrot_dlfunc_p_p_sc_sc: Assertion `(interp->ctx.bp_ps.regs_p[-1L-(cur_opcode[2]))->pmc_ext' failed.
Aborted
```