

---

## Table of Contents

Generating Registration Information .....	1
<i>getRegistrationInfo</i> .....	6
Additional Topics .....	7

In this article, we describe how we can use the `RGCTranslationUnit` code to generate the registration information for R to access native routines in a DLL. This information provides more structured and robust access to the routines, including some reflection information. Automating the generation of this code simplifies the development and ensures or at least increases the chance that it is correct by minimizing human error. This article describes the computations that are available via the high-level function `getRegistrationInfo`. There is a second implementation also available via the function `generateRegistrationInfo` and these two need to be merged.

## Generating Registration Information

When dynamically loading compiled code into R, we have the opportunity to register the C and FORTRAN routines that can be invoked from R. This helps to identify the intentionally accessible or "exported" routines and can be used to eliminate access to the others in the DLL. The registration also allows us to provide information about the number and type of the parameters each routine expects. This allows R to detect whether a routine is being invoked incorrectly which could lead to memory corruption and a crash or, worse, an erroneous result.

Suppose we have C code with two routines declared as

```
void foo(int *x, int *x_len, double *ans);
```

```
SEXP bar(SEXP a, SEXP b);
```

The first of these can be accessed via the `.C` interface, and the second via the `.Call` interface. We could register these by calling `R_registerRoutines` with information about the set of routines for each interface. The information would be constructed typically as follows

```
static R_CallMethodDef CallEntries[] = {
    {"bar", (DL_FUNC) &bar, 2},
    {NULL, NULL, 0}
};
```

```
static R_CMethodDef CEntries[] = {
    {"foo", (DL_FUNC) &foo, 2, {INTSXP, INTSXP, REALSXP}},
    {NULL, NULL, 0}
};
```

```
R_registerRoutines(dll, CEntries, CallEntries, NULL, NULL, NULL);
```

This is usually done in a routine named `R_init<DLL name>` where we substitute the name of the DLL (without the file extension `.so` or `.dll`). This way, R will automatically call the routine after it has dynamically loaded the DLL.

So far, this is the standard registration tutorial. What we want to talk about in this article is how we can automate the creation of the registration information in

---

`<c:var>CEntries</c:var>`

and

`<c:var>CallEntries</c:var>`

above. If we have a reasonably large number of routines, and/or they are changing not infrequently, creating and maintaining this information can be tedious and hence error prone. So automating its management is desirable.

Now, how can we automate the creation of this information? Essentially, we need to know which routines are being called by the R code and via which interface. So we need to find expressions in the R code that are calls to `.C`, `.Call`, `.External`, `.Call.graphics`, `GraphicsExternal`. Unfortunately, we cannot handle FORTRAN code at present in an automated manner. We can find this information using the function `getNativeRoutineCalls`. This works on a variety of different inputs. One can load a package and then pass the name index of that package in the search path, or alternatively, give the name of the package as in "package:RCurl" or "RCurl". Alternatively, one can give the name of a directory in which to find R files. Or one can give it a list of function objects or a single function object.

### Note

This doesn't yet deal with expressions at the top-level, i.e. outside of functions. This is easy to add, just not done yet.

Generally, the return value from `getNativeRoutineCalls` is a list with an element for each function object which was passed to it, either implicitly or explicitly. Each element is itself a list and has entries for each of the 5 different interface mechanisms, `.C`, `.Call`, `.External`, `.Call.graphics` and `.External.graphics`. These are the names of the entries and the value of each element is either `NULL` or a list of the names or symbols which were referenced in any expression in the function via the interface.

There is now an additional element named "expressions" which contains the actual expressions in which the foreign routine calls are made. This allows us to process them as we have the information about the routines without revisiting all the functions. However, this is relatively inexpensive.

Now that we have the identities of the routines that are called, we can create the registration information. Let's assume that there is no aliasing of the symbols via the `useDynLib()` directive in the `NAMESPACE` file. Since we are automating the registration information, we will typically generate that information also and so there will be no aliasing. If there is to be aliasing, we will create the aliases via a mapping function or prefix/suffix pair when creating the registration information.

So our task is to read the `tu` files and find either the declaration or definition of each routine referenced and ensure that it has the appropriate signature for the interface by which it is being called and also to generate the

We should note that when we find the expression, we can determine the number of arguments which are being passed and also, we can determine the types of literals if there are any and ensure that they are compatible. This allows us to do static or off-line checking rather than run-time/dynamic checking. To do this, we need the expressions themselves and we can squirrel this information away in a single pass or alternatively do a second pass once we have the information about the routines.

Let's work with the XML package as an example.

```
library(XML)
ff = getNativeRoutineCalls("XML")
```

Now, we can see which functions have foreign routine calls.

---

```
names(ff)
```

```
[1] "htmlTreeParse" "libxmlVersion" "newXMLDoc"      "newXMLNode"  
[5] "parseDTD"      "parseURI"        "xmlDOMApply"   "xmlEventParse"  
[9] "xmlTree"       "xmlTreeParse"   "xpathApply"
```

The way this code was written (by yours truly), we expect to see only one foreign routine called in each routine. We can "verify" this with

```
sapply(ff, function(x) sum(sapply(x[1:5], length)))
```

```
htmlTreeParse libxmlVersion      newXMLDoc      newXMLNode      parseDTD  
              1                1                1                2                1  
      parseURI  xmlDOMApply xmlEventParse      xmlTree  xmlTreeParse  
              1                1                1                14                1  
      xpathApply  
              1
```

We only look at the first five elements since the last one is the list of expressions that make the call.

Note that *xmlTree* calls 14 routines and so runs against my expectations.

What are the names of the routines being called in each function, and with which interface (i.e. `.C()`, `.Call()`, ...)? The names of the routines can be found as

```
sapply(ff, function(x) unlist(x[1:5]))
```

```
$htmlTreeParse  
      .Call  
"RS_XML_ParseTree"  
  
$libxmlVersion  
      .Call  
"RS_XML_libxmlVersion"  
  
$newXMLDoc  
      .Call  
"R_newXMLDoc"  
  
$newXMLNode  
      .Call1          .Call2  
"R_newXMLNode" "R_insertXMLNode"  
  
$parseDTD  
      .Call  
"RS_XML_getDTD"  
  
$parseURI  
      .Call  
"R_parseURI"
```

---

```

$xmlDOMApply
    .Call
"RS_XML_RecursiveApply"

$xmlEventParse
    .Call
"RS_XML_Parse"

$xmlTree
    .Call11          .Call12          .Call13          .Call14
  "R_newXMLDtd"    "R_insertXMLNode" "R_newXMLTextNode" "R_xmlNewNs"
    .Call15          .Call16          .Call17          .Call18
  "R_xmlSetNs"    "R_newXMLNode"   "R_insertXMLNode" "R_insertXMLNode"
    .Call19          .Call110         .Call111         .Call112
  "R_xmlNewComment" "R_insertXMLNode" "R_newXMLCDATADataNode" "R_insertXMLNode"
    .Call113         .Call114
  "R_newXMLPINode" "R_insertXMLNode"

$xmlTreeParse
    .Call
"RS_XML_ParseTree"

$xmlpathApply
    .Call
"RS_XML_xpathEval"

```

The names of each character vector give the interface function, but we can compute them more usefully with `sapply(ff, function(x) (names(x)[1:5])[sapply(x[1:5], length) > 0])`

```

htmlTreeParse libxmlVersion    newXMLDoc    newXMLNode    parseDTD
  ".Call"      ".Call"      ".Call"      ".Call"      ".Call"
  parseURI    xmlDOMApply xmlEventParse    xmlTree    xmlTreeParse
  ".Call"      ".Call"      ".Call"      ".Call"      ".Call"
  xpathApply
  ".Call"

```

We need to make certain that each routine is being called via the appropriate interface, that the number of arguments (modulo the `PACKAGE`, `NAOK` and `DUP` arguments for the `.C` calls) are correct, and the the types if determinable are correct.

Now, we need to generate the `tu` files for the package. Until this is incorporated into the R make files, we can do it manually. We add

```

R_INCLUDE_DIR=${R_HOME}/include
R_SHARE_DIR=${R_HOME}/share

%.tu: %.c
$(CC) -fdump-translation-unit $(ALL_CPPFLAGS) $(ALL_CFLAGS) -c -o /dev/null $<

```

---

```
TU_FILES=$(wildcard *.c)
tu: $(TU_FILES:%.c=%.tu)
```

to our package's Makevars.in or Makevars file. And then we can create all the tu files in one command

```
make -f Makevars -f $R_HOME/share/make/shlib.mk tu
```

Note that here we only deal with .c files and we do not use g++ as the compiler as we are only interested in the declarations of the routines and not their bodies. If we want more information, we would use g++.

Now we have the tu files and we can process them.

```
filenames = list.files("/tmp/R/XML/src", ".+\\.tu", full.names = TRUE)
library(RGCCTranslationUnit)
routines = lapply(filenames,
  function(f) {
    p = parseTU(f)
    r = getRoutines(p, gsub("\\.t00\\.tu$", "", basename(f)))
    resolveType(r, p)
  })
names(routines) = basename(filenames)
```

Now we return and find the names of the routines that were actually referenced in R expressions.

```
rRoutines = as.character(unlist(sapply(ff, function(x) unlist(x[1:5]))))
```

All the available routines are given by

```
allRoutines = as.character(unlist(lapply(routines, names)))
```

And we can ensure that there are no missing symbols at this point, i.e. unimplemented routines that are referenced in R code.

```
i = match(rRoutines, allRoutines)
if(any(is.na(i)))
  stop("missing", paste(rRoutines[is.na(i)], collapse = ", "))
```

Now, we can get the registration information.

```
rr = unlist(routines, recursive = FALSE)
names(rr) = gsub(".*\\.\"", "", names(rr))
rr = rr[rRoutines]
```

We need to determine which routines are accessed by which interface so that we can group them in the registration information. We can infer the interface from the usage in the R code, but that does not verify that the routines are compatible with that interface. If the return type is void, then it is a candidate for .C. And if the types are pointers to int, double, char \*, then it is a .C routine.

.Call and .External routines have a return type that is a SEXP and all the parameters are also of type SEXP. In certain circumstances, it is hard to tell the difference between a .External and a .Call. This arises when we have a .Call routine but which has 4 parameters which is the required signature of a .External routine.

In our example, we only have .Call routines. And we have added complexity as the XML source uses a typedefinition to handle the difference between R's native object type (SEXP) and S-Plus version (s\_object). We use USER\_OBJECT\_ as the name of our general object type. And so this appears in the tu and the resolved types, e.g.

```
rr[[1]]$returnType@name
```

---

So, with this package-specific information, we can compare the types to this name.

```
rr[[1]]$returnType@name == "USER_OBJECT_" && all(sapply(rr[[1]]$parameters, function(x) x$type@name)
```

or, more conveniently

```
all(c(rr[[1]]$returnType@name, sapply(rr[[1]]$parameters, function(x) x$type@name))
```

This verifies that all the types are indeed R objects.

If we did not have the typedef, we could replace `USER_OBJECT_` with `SEXP`. And if we are concerned with the potential for a type named `SEXP` that is not actually an R type, we can look at the definition of the typedef

```
class(rr[[1]]$returnType@type)
```

```
[1] "PointerType"  
attr(,"package")  
[1] "RGCCTranslationUnit"
```

```
class(rr[[1]]$returnType@type@type)
```

```
[1] "StructDefinition"  
attr(,"package")  
[1] "RGCCTranslationUnit"
```

```
rr[[1]]$returnType@type@type@name
```

```
[1] "SEXPREC"
```

And if we really want to resolve the `SEXPREC` type, we can do that and ensure that it came from the R header files by returning to the unresolved routines and the tu parser.

We can determine if a routine is compatible with the `.C` interface using the following R functions.

```
# see findFF.R  get_.C_type, is_.C_routine
```

## ***getRegistrationInfo***

All of the computations above are done as part of the high-level function `getRegistrationInfo`. This works with R code and the TU files from all the C/C++ code for a DLL. It uses `getNativeRoutineCalls` to find all the references to the native routines and then it processes the TU files to find all the routines in the compiled code. It then ensures that all the referenced routines are available. After this, it goes about determining the interface by which each routine can be called, i.e. `.C`, `.Call`, `.External`. It uses the R code to resolve the ambiguities between `.Call` and `.External` calls. Having done this, it checks that all the routines are being called correctly from the R code. And finally it returns information about each of the referenced routines giving the the name of the routine, the type of interface by which it can be called and the types of the parameters for a `.C`-callable routine. This information can then be used to generate registration code using `writeCode`. And similarly, the `useDynLib` directive for the `NAMESPACE` file can be generated by calling `writeCode` with "r" as the target.

An test/example of how to use this is given with the following code below. We use the files `foo.R` and `foo.c` in the examples directory of this package. We first generate the `.tu` file with the command

---

```
gcc -fdump-translation-unit -c foo.c -o /dev/null -I`R RHOME`/include
```

Given this, we have all we need and can call the R code to generate the registration information as

```
rfile = system.file("examples", "foo.R", package = "RGCCTranslationUnit")  
regInfo = getRegistrationInfo(rfile, tu.dir = system.file("examples", package = "R
```

Now that we have this information, we can generate the C code to register the routines with R with the command

```
writeCode(regInfo, "native", dll = "duncan", dynamic = FALSE)
```

And we generate the NAMESPACE directive with

```
writeCode(regInfo, "r", dll = "duncan")
```

## Additional Topics

We can also use this tool to identify the routines that should not be exported. Of course, nm could be used for this too to identify all the visible symbols that are routines and then finding the ones that are actually referenced in R. The complement of the latter can all be hidden/not exported.