Gforth's libcc C Function Call Interface

M. Anton Ertl*
TU Wien

Abstract

A major problem in our earlier proposal for a C interface was that a part of the interface was not portable between platforms. The libcc interface solves this problem by using a C compiler and its `.h`-files. The `.h`-files contain knowledge about the specific platform, and the C compiler automatically inserts the necessary conversions between Forth and C types. In this paper we describe the libcc implementation and interface. We also discuss how a Forth-C interface might be standardized.

1 Introduction

The programming interfaces of many useful libraries are defined as collections of C functions and C data structures, so being able to call C functions is a very useful capability for a Forth system.

In an earlier paper [Ert06], we discussed general design issues for a function call interface and designed a C interface based on these ideas that was intended to be implemented using foreign function call libraries like the ffcall libraries[1] or libffi.

In the meantime, we have explored a new implementation approach for foreign function calls that makes it possible to eliminate the non-portable C part of the declaration in most cases. In this paper we describe this implementation approach (Section 3) and the resulting C interface (Section 4). We also mention some of the issues in standardizing a C interface (Section 5).

2 Portability

Portability is a central problem addressed in our earlier work, and it has led to our new implementation approach, so we revisit it here.

The primary form of portability that we are interested in is in being able to use the same Forth program on several platforms (e.g., Linux-i386, Linux-AMD64, MacOS X, and Cygwin), even if the Forth program calls C functions. Portability between Forth systems is another issue that we discuss in Section 5.

The platform portability problem is that equivalent C functions (same name, same functionality) have different argument and return types on different platforms. A typical example is the POSIX function

```
off_t lseek(int fd, off_t offset,
            int whence);
```

On some ancient platforms, this function is instead defined as follows:

```
long lseek(int fd, long offset,
           int whence);
```

However, even if we use the official POSIX definition, `off_t` can be different things on different platforms: either `long long` or `long`; what's worse, it can even be different things on the same platform, depending on the way that `lseek()` etc. are compiled[2].

In our earlier work we propose to deal with this problem by having a platform-dependent C part in the function declaration, e.g.:

```
c-types lseek int longlong int -- longlong
```

We suggested that these C parts could be generated automatically out of the `.h`-files, reducing the amount of per-platform work needed.

How does C avoid this problem? Primarily by having `.h`-files with standard names, that contain (among other things) platform-specific definitions, e.g., of types like `off_t` (that's why automatioc generation of C parts out of `.h`-files would help.

C has another benefit: the C compiler knows more about data types of parameters than a Forth compiler (at least if the function prototype is `#included`); this allows it to insert the necessary type conversions transparently.

3 Implementation approach

Instead of implementing a C parser that translates `.h`-files into Forth, libcc uses the existing C compiler's knowledge of C and its ability to read and

*Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; `anton@mips.complang.tuwien.ac.at`

[1] `http://www.haible.de/bruno/packages-ffcall.html`

[2] Whether `_FILE_OFFSET_BITS` is set to 64 or not.

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: EuroForth 2007 Conference Proceedings, p. 7-11

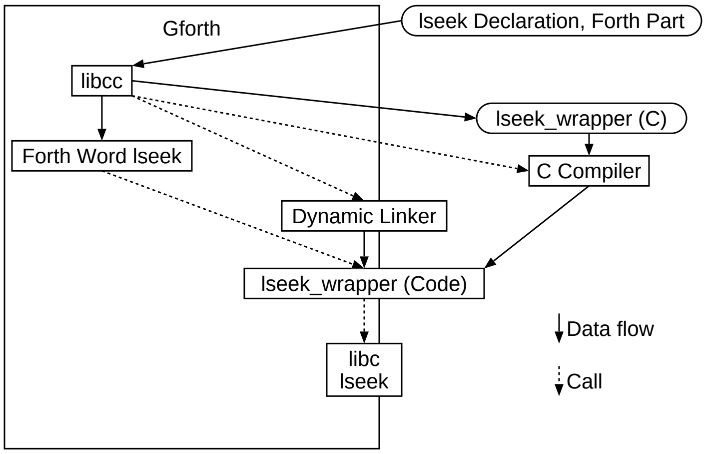

Figure 1: A foreign function call for `lseek` with libcc

understand `.h`-files, and to insert the necessary type conversions.

For each Forth part of a foreign function declaration, libcc generates a wrapper function in C that accesses the cells and floats on the Forth stacks and passes them as parameters to the target C function (e.g., `lseek()`), then calls the target function, and finally takes the return value and pushes it on the appropriate Forth stack. The wrapper function is later compiled by a C compiler, which automatically inserts the necessary conversions between the Forth types and the C types, and also knows about the C calling convention on the particular platform.

Finally, the wrapper function and the target function are linked dynamically, and when the Forth program calls the word corresponding to the target function, it calls the wrapper function, which in turn calls the target function. All the wrapper functions can be called through a simple indirect call, without needing a library like libffi.

This approach is depicted in Fig. 1.

A major advantage of this approach is that any tricks that the `.h`-files play with C macros (e.g., renaming C functions, as is usually done for `lseek()`) take full effect in the wrapper function, so it will call exactly the function that we were interested in.

Disadvantages of this approach are:

- It requires a C compiler at run-time.
- The actual call of the C compiler is quite target-specific (in particular because it is necessary to generate a dynamically linkable binary). Fortunately that's just one small piece of code per platform.
- The compilation of a wrapper function is much more expensive than similar operations necessary with ffcall and libffi. We are exploring batching and caching to reduce this cost, and will report on that in more detail in future papers.

4 The libcc interface

This section describes the libcc interface, in particular the differences from our earlier interface proposal [Ert06]. This section gives an overview, mainly through the `lseek` example. For a complete specification, read the current version of the Gforth manual.[3]

4.1 Declarations

As in our earlier work [Ert06], the declarations for a target C function consist of a C part and a Forth part. The Forth part is exactly as described in our earlier work, but the C part is completely different: it consists of lines of C code prefixed with `\C`; the programmer has to ensure that the C code declares the target C function (and its parameter and return types). A complete set of declarations for two `lseek`-calling words can look as follows:

```
\c #define _FILE_OFFSET_BITS 64
\c #include <sys/types.h>
\c #include <unistd.h>
c-function lseek lseek n n n -- n
c-function dlseek lseek n d n -- d
```

In our implementation the `\C` declarations are copied to the generated C code in front of the wrapper function, making the target function known to the C compiler, which can then generate the right type conversions for the parameters and return value.

[3] `http://www.complang.tuwien.ac.at/forth/gforth/cvs-public/`

In this example we have two Forth parts, resulting in two words, both calling the C function `lseek()`, but with different stack effects: the Forth word `lseek` has the stack effect `( n n n -- n )`, whereas `dlseek` has the stack effect `( n d n -- d )`.

4.2 Calls

Calls work exactly as described in our earlier work: The programmer has to push the arguments left-to-right on the stack; the argument and return value types are determined by the Forth part of the declaration and are independent of the concrete C types used by the function. Example:

```
fd @ 0. SEEK_SET dlseek -1. d= if
  ... \ error handling
then
```

4.3 Variadic Functions

To call C functions that can take a variable number and variable types of arguments, like `printf()`, the programmer can declare several words, each with a desired parameter set, and then call these words by name. E.g.:

```
\c #include <stdio.h>
c-function printf-nr printf a n r -- n
c-function printf-rn printf a r n -- n
s\" n=%d r=%f\n\0" drop -5 -0.5e printf-nr
s\" r=%f n=%d\n\0" drop -0.5e -5 printf-rn
```

One problem here is that the C compiler does not know the desired type of the varargs and does not insert conversions to this type; this can lead to portability problems. One solution to this problem is to perform the conversion explictly at the C level. E.g.:

```
\c #define printfll(s,ll) \
\c             printf(s,(long long)ll)
c-function printfll printfll a n -- n
```

Here, we define a macro `printfll` that converts the second parameter explicitly to the `long long` type, and then calls `printf()`. The Forth word then calls `printf` through this macro.

4.4 Variables, Constants

C variables and constants can be converted to Forth words with this interface, by treating them like functions:

```
\c #include <unistd.h>
\c #define seek_set_macro() (SEEK_SET)
c-function SEEK_SET seek_set_macro -- n
```

However, this method is quite heavyweight, both in the amount of typing and in memory consumption, so we may introduce a lighter-weight way to import C constants and variables in the future.

4.5 Calling C function pointers

Programmers can also call C functions for which they have C function pointers, not the name:

```
\c typedef int (* fun1)(int);
\c #define call_fun1(par1,fptr) \
\c             ((fun1)fptr)(par1)
c-function call-fun1 call_fun1 n func -- n
```

The Forth word `call-fun1` now works similar to `execute`, but instead of execution tokens it takes C function pointers of this particular type. The programmer has to define one such call word for each function pointer type he wants to call.

4.6 Callbacks

Conversely, the programmer sometimes wants to generate C function pointers for Forth words, because he has to pass them to some C function directly or through a data structure; this is usually known as callback (the library calls back into the application program through this function pointer).

We have not implemented callback support in libcc yet, and have not fixed the interface yet.

4.7 Future work

The main other missing piece of the C interface is dealing with structs and unions: accessing their fields and passing them to or returning them from functions. Field access is particularly hairy, for the following reasons:

- Each structure constitutes a separate name space, and we have to find a practical way to map C field names to Forth.
- On fetching from or storing into fields we have to convert between the C type and the Forth type. In general the C type is platform-dependent, so if we want to be portable, we cannot use some general fetching/storing words, but have to use one that is specific to the field.

5 Standards

In the long run, it would be nice to be able to port programs that call C libraries between Forth systems. In order to do that, we need to standardize the C interface. However, the interface outlined above is implementation-specific in the C part of the declarations, so standardizing on it completely is not going to happen. In the following, we explore various levels of standardization that might be more successful. Note that this discussion is not tied to the interface described in the rest of the paper, but applies to any C interface.

5.1 Calls only

The most important portability issue is the calls of the foreign functions themselves, because they tend to be distributed across a large amount of code, and changing the calls would be a lot of work and error-prone. So the word name, parameter order, parameter Forth types and return value Forth type of the word for calling a particular C function should be the same across Forth systems.

However, while the word name and parameter order can be standardized for all calls, the parameter and return value Forth types are specific to each particular function-calling word, so standardizing that means standardizing either the words themselves, or having a standard way to specify these types, i.e., the Forth part of the declaration.

5.2 Specific libraries

For widely-used libraries and APIs, one could standardize the Forth interfaces of the calls. The amount of work required for that could be reduced by just specifying what C type corresponds to what Forth type. E.g., for the POSIX.1-2001 API one could specify that all integer types would be single cells in Forth, with the exception of a few (e.g., `off_t`) that would be double cells.

In order to be able to use these words portably, it would also be helpful to specify a way in which the interface would become available; e.g., what file needs to be `required`, and maybe in which wordlist these words reside afterwards.

One potential problem in that respect is that some APIs have a lot of optional extensions (similar to ANS Forth). One would have to find ways to specify these extensions.

5.3 Forth part of the declaration

While such an approach is probably satisfactory to programmers using these libraries, each Forth system would need its own files with the declarations of all these C-function words. By standardizing the Forth part of the declaration, these parts could be shared between the Forth systems, which should reduce the effort for all involved parties, and also reduce the number of bugs encountered by an individual user.

5.4 C part of the declaration

As mentioned above, standardizing the C part is hard, because it depends on the way that the Forth system implements the C interface (e.g., compare the way this is done in this paper with that in our earlier work [Ert06]), and because it can depend on the platform (e.g., consider the two different C parts for `lseek()` in our earlier work).

What might be possible to address the Forth system issue is to specify the several syntaxes appropriate for different implementation approaches. A particular Forth system would interpret one of them and ignore all the others. The C part of an interface definition might then contain several ways to specify the same thing, and all Forth systems that implement at least one of these ways could work with that.

That solution has its problems, but it is probably hard to find consensus on anything stronger.

6 Related Work

SWIG (Simplified Wrapper and Interface Generator)[4] is a tool for interfacing scripting languages to C. It works in a way similar to libcc. However, after a superficial investigation of SWIG we decided against using SWIG and for implementing libcc directly in Forth; a still-remembered reason was that we heard that adapting SWIG for a new language would require more than 1000 lines of code; currently libcc has 484 lines.

7 Conclusion

The libcc interface uses actual C code for the C part of a C function declaration; this makes it possible to make the C part platform-independent in most cases, because it can just `#include` a standard `.h`-file that contains the platform-specific C code; so the whole interface to most C functions can be completely portable between platforms.

The libcc interface offers a number of other advantages over the interface proposed earlier [Ert06], such as more capabilities when dealing with variadic functions, and the ability to call C function pointers without needing special support.

The implementation of the libcc interface works as follows: For every target function it generates a wrapper function in C, compiles it and dynamically links it into Gforth; the wrapper function is trivial to call, and performs all the stack accesses and (implicitly through the C compiler) argument conversions.

The standardisation of a C interface between Forth systems can be divided into several parts,

[4] `http://www.swig.org/`

with the actual calls probably the easiest to find consensus on, and the declarations being progressively harder.

References

[Ert06] M. Anton Ertl. A portable C function call interface. In *22nd EuroForth Conference*, pages 47–51, 2006.