A Portable C Function Call Interface

M. Anton Ertl[*]
TU Wien

Abstract

Many Forth systems provide means to call C functions, but these interfaces are not designed to be portable between platforms: A call to a C library function that works on one platform may fail on the next platform, because the parameter and return value types of the C function may be different. In this paper, we present an interface that avoids this problem: In particular, the actual calls can be made platform-independent; a part of the declarations is platform-dependent, but can be generated automatically from C `.h`-files.

1 Introduction

Many operating system and library calls have their interfaces specified as C prototypes and are called using C calling conventions. As a result, C has become a kind of lingua franca when interfacing with other languages; other languages generally interface to C, and "foreign function call" libraries like ffcall and libffi are actually only designed for interfacing with C.

This paper discusses the design of a C interface for Forth. The main goals of this interface are:

Portability of Forth code It should be possible to write Forth code with calls to C such that it works unchanged across different platforms. The portability of the C function declarations would also be nice, but may only be partially achievable, as we will see.

Programmer convenience It should be easy to call the C functions using the existing documentation for them. The need for declaring C functions should be eliminated if possible.

Avoid losing bits During conversions between Forth and C types, bits should only be cut off in places where the programmer has some control over what these bits are.

Full domain Allow using all possible values as arguments to functions. This goal conflicts with the no-bit-loss goal.

Many Forth systems already have a C call interface. However, they all fail the portability goal. Indeed, many of the interfaces contain artifacts like the reversal of the arguments that are specific to the platform and the Forth system involved.

This paper does not deal with access to C `structs`, `unions` or memory accesses to C types. In addition to some of the problems discussed here, these issues also pose additional problems, and require additional effort to solve them.

2 Problems and choices

This section discusses the problems that we encounter when we design a C call interface, and outlines some of the design decisions. Our actual interface is presented in Section 3. If the present section appears to be complicated and lengthy, this is due to the complex subject matter. Feel free to skip to Section 3, and only read this section to learn about the reasons for this design.

2.1 Parameter order

For user convenience, the parameter order is the same as in the C code and (more importantly) the documentation of the C function. I.e., the rightmost parameter in C is on top of its stack in Forth, and the leftmost parameter deepest.

Some existing implementations use the reverse order (leftmost parameter on top of stack), because that is easier to implement for their systems on the IA-32 architecture (where C passes parameters on the native stack, with the leftmost parameter on top).

However, the reverse order is inconvenient for the users, and error-prone. Typically, neither the normal nor the reverse order are what a Forth programmer would have designed for best use in Forth, so some stack juggling is often necessary; performing this stack juggling while mentally reversing the order of parameters given in the documentation is hard and frequently leads to errors.

Also, all recent calling conventions pass the first few parameters in registers, including the calling conventions used for Unix and Windows on the AMD64 architecture, which will gradually replace

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

the IA-32 architecture and its stack-based calling convention in the next years.

Finally, the implementation benefits of the reverse order are not just restricted to an obsolescent architecture, but also to a specific design of the Forth system: It requires that stack items are kept in memory, with the data stack pointer being `esp`, and that floating-point values are kept on the data stack. Sophisticated native-code compilers keep stack items in registers, and less sophisticated systems like Gforth do not use `esp` for the data stack pointer. And nearly all Forth systems use a separate floating-point stack.

2.2 Types

The main problem with the calling C functions is: Which Forth types should we pass for various parameters, and what type should we expect as return value?

A simple approach would be to let all C integer and pointer types correspond to Forth cells and all C floating-point types to Forth floats, for both parameters and return values. This would satisfy the portability and convenience goals.

Unfortunately, some C integer types are larger than a Forth cell on some platforms; e.g., `off_t` may be 64 bits wide even on 32-bit platforms. Consider a call to this C function:

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

If we pass a cell for the offset parameter, we are not able to pass all the possible offsets that `lseek` can take, so we miss the full-domain goal. What's worse, the result of the function is truncated to fit into a cell, so we lose bits, contrary to our goal.

So we might actually prefer to call the C function `lseek` with the following stack effect:

```
( n-fd d-offset n-whence -- d )
```

Bit loss vs. full domain

When we call `lseek`, the d-offset argument may be too large (e.g., on a 64-bit system, where d is 128 bits and `off_t` 64 bits; or on a 32-bit system with a 32-bit `off_t`), and may be truncated on passing it to `lseek`, losing bits. This is the conflict between the full-domain goal and the loss-avoiding goal. However, in this case the problem is not that bad, because the programmer has some control over the situation; e.g., he will typically pass an offset that comes from an earlier call to `lseek`, or use a small (constant) offset that is known not to be damaged by truncation on any platform.[1]

So, in general, for functions we call, we usually want to have a Forth type for the arguments that is at least as big as the C type (the full-domain trumps bit-loss here); for the return value, we want a Forth type that it at least as big as the C type, to avoid bit-loss.

For callbacks (Forth words that we pass to C as C function pointers and that the C code then calls), we want to have the Forth types for the arguments at least as big as the C type to avoid bit-loss. For the return value, we again want to provide a type at least as big as the C type to be able to return all values out of the codomain of the function (and avoiding the bit-loss is again the responsibility of the programmer).

So, in all cases we want a Forth type that is at least as big as the C type. A way to ensure that this is as often the case as possible would be to use double-cells for integer types in all places. However, that approach conflicts with the convenience goal. Actually, most C types fit into a single cell on all 32-bit and larger platforms[2], and there are only few, such as `off_t`, that are larger on some platform. So actually single cells should be the usual case, and double cells the exception.

You may wonder where the asymmetry between Forth and C types comes from. It comes from the situation for which we are designing: We have a bunch of independently developed C functions that are called from a Forth program that is designed to call these C functions; and for callbacks, the words that are called back are designed to be called back from these independently developed C functions. If we designed an interface for calling independently developed Forth code from (dependent) C code, we would use C types that are at least as big as Forth types.

2.3 Determining the Forth type

Can we determine the Forth type of a parameter from the C type?

We cannot determine it from the basic C type, because the basic type of the parameter might be different on different platforms. E.g., `off_t` is not a basic C type; it is usually mapped to `long` or `long long`. If we use a single cell for `long` and a double cell for `long long` then we would get different stack effects for `lseek` on different platforms, breaking portability. This approach is implemented in Gforth's current C interface, and it is broken; fortunately parameters that may be `long long` are rare, so this problem is rare.

Can we determine it from the derived C type, e.g., `off_t`? In principle this is a good idea. It

[1] It might still be a good idea to have an (optional) runtime check that the truncation really loses only redundant bits.

[2] We can restrict our view to such big platforms in many cases, because the library we want to call (e.g. OpenGL) does not exist on smaller platforms

certainly can be used as a guideline when deciding which Forth types should be used when the programmer declares the Forth type manually, as in our interface below.

One might also consider to generate the Forth type automatically from the C prototype information (from the `.h`-files) and a table of C-to-Forth type mappings. However, while this strategy would work in most cases, it would not be entirely portable, because the prototypes in the `.h`-files are not necessarily the same on all platforms. E.g., on some old Unix versions the `.h`-files probably contain `long` in place of `off_t`, and that would typically be mapped to a single cell (whereas `off_t` would typically be mapped to a double cell).

The reason why such differences in `.h` files are not a problem for C is that C performs automatic conversion between different integer types. The reason that they would be problems for Forth is that Forth requires explicit conversion between some integer types (in particular, between single-cell and double-cell types).

Floating point

For floating-point parameters, the situation is much simpler: We only have one Forth on-stack floating-point type, so we have to convert every C type to that, and have to convert that to every C floating-point type. There may be some bit loss involved, so the programmer should know what he is doing. The bit loss will usually occur in the form of rounding, which will be acceptable in many situations, but may lead to hard-to-find errors in other cases.

C performs automatic type conversion between integer and floating-point types, so in theory a given parameter might be an integer type on one platform and a floating-point type on another platform. However, this does not happen in practice.

Addresses/Pointers

In this paper we assume that C pointers are represented as simple flat addresses. There may be some platforms around where this is not the case, but we feel that such platforms are not worth catering for, because:

- These platforms are relatively exotic, and it is not clear that ANS Forth systems exist for them at all, much less that they would want to use a portable Forth-to-C interface.
- Catering for them would probably complicate the interface significantly.
- Many programmers would probably make mistakes in using such a more complicated interface without noticing (because the result would run in a flat-address system), resulting in programs that don't port to non-flat machines despite the interface complications.

Moreover, we could not cater for such platforms, because we do not have enough experience with a wide-enough range of such platforms to design a general way of dealing with them.

Pointers necessarily always fit into a cell (since addresses fit into a cell), so the type problem is trivial for passing and returning pointers: just use a cell for every pointer.

However, there is a problem in what can be done with pointers. We cannot easily fetch the data they are pointing to or store data there, because we don't know how to access it. We leave this memory access problem to a future paper.

Still, we can do something useful with such pointers: we can pass them to other C functions; E.g., that is the only use that even C programmers make of some pointer types, such as `FILE *`.

Structs/Unions

In C you can pass structs and unions as parameters to a function, and the function can return a struct or union. We do not attack this problem in this paper.

Fortunately, the library functions I have come across usually do not make use of this feature of the C language, but prefer to pass pointers to structs rather than pass structs by value. However, this is not necessarily the case for all libraries.

Varargs

Some C functions (e.g., `printf`) can be called at different places with different numbers and types of parameters (varargs functions). The Forth system does not know how many of the values on the stacks are intended to be arguments to the C function, which of the values on the stacks correspond to which C type, etc. Therefore, the Forth programmer has to make the Forth and C types used in the concrete call explicit.

This can be done by putting that information near the call (probably right before it).

Another option would be to declare several Forth words (with different names) for the C function, each with a different parameter pattern, and then use the right name for the desired parameter pattern in the call.

2.4 Case sensitivity

Another potential problem is that C names are matched case sensitively, whereas in Forth names that may only differ in case may be treated as being the same; and most Forth systems are actually implemented case-insensitively.

Fortunately, C programmers usually do not use case sensitivity to distinguish functions[3].

Moreover, a C function may have the same name as an existing Forth word (e.g., abs), so one would shadow the other.

One solution for both problems would be to define the C functions in a separate, case-sensitive wordlist. However, while Gforth has such case-sensitive wordlists (tables), most Forth systems do not have them. Moreover, dealing with collisions through wordlists is cumbersome.

Another solution is to provide a different Forth name for the problematic C name, and use this Forth name to refer to the C function in Forth code.

3 The C function call interface

The C interface consists of three parts, used in this order:

Declare Forth types and name This part is platform independent.

Declare C types and name This part is platform dependent, but can be generated automatically from .h-files.

Call the C function This part is platform independent.

3.1 Declaration, Forth part

In the Forth part of the declaration, you declare the Forth name, which C function it corresponds to, and what the Forth types of the parameters are. For our lseek example, the Forth declaration might look like this:

```
c-function dlseek lseek n d n -- d
```

This declares a Forth word dlseek for the C function lseek with the Forth stack effect n d n -- d.

C-function parses the whole sequence up to the --, plus the following return value. The allowable types for the parameters and the return value are:

n w A single cell.

d A double cell.

r A float.

void Used as return type if the function does not return a value.

func Used to pass a C function pointer.

The Forth part of the declaration is optional. If it is not present, the word gets a default name and default parameter and return types, as follows:

- The default Forth name is the same name as the C function name.
- The default type for an integer or pointer type in C is a single cell.
- The default type for a floating-point type in C is a float.

In most cases, these defaults are the desired names and types, so only few explicit Forth-part declarations are necessary.

If you do not use the default types, it is probably also a good idea to use a non-default name (like dlseek in our example), to make the programmer and reader more aware of the non-default types.

3.2 Declaration, C part

The C part of the declaration specifies the basic C types for the parameter and return values on the specific platform, like this:

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

Of course, on a different platform one might need a different declaration, e.g.,

```
c-types lseek int long int -- long
```

Again, c-types parses everything up to --, plus the return type. The possible types are: schar short int long longlong uchar ushort uint ulong ulonglong ptr float double longdouble void func.

Note that this declaration can be created automatically out of the prototype for lseek and the type declaration of off_t:

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

So, while these declarations are platform-specific, it is possible to write a parser that processes the .h-files of the platform at hand, takes the the C functions that are declared there, and performs C part declarations for the Forth system.

[3] There may be case-insensitive collisions between constants or types and functions, though.

3.3 Calling the C function

Once a C function is declared, calling it works just like with any other Forth word. E.g., for our dlseek a call might look like this[4]:

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

3.4 Varargs

Functions with variable numbers or types of arguments can be handled by declaring each argument pattern separately:

```
c-function sn-printf printf w n -- n
c-types printf ptr long -- int

c-function sr-printf printf w r -- n
c-types printf ptr double -- n

s\" %ld\0" drop 20   sn-printf .
s\" %f\0"  drop 2.5e sr-printf .
```

3.5 Callbacks

Consider the ANSI C function qsort:

```
void qsort(void *base, size_t nmemb,
           size_t size,
           int(*compar)(const void *,
                        const void *));
```

When you call it, you have to pass a C function pointer for the last argument. You may want to let qsort call a Forth word through that function pointer (a callback); then you have to provide a C function pointer for the Forth word. An example of such a word (useful with qsort) would be:

```
: n-compare ( addr1 addr2 -- n )
 @ swap @ swap - ;
```

Ideally we would like to call qsort like this:

```
: sort-cells ( addr u -- )
  1 cells ['] n-compare qsort ;
```

However, a Forth execution token is not a C function pointer, and qsort would not know how to execute it, so we have to get a little more involved. First we define a word compar for the kind of function pointers that qsort wants, as usual in two parts:

```
c-function-ptr       compar w w -- n
c-function-ptr-types compar ptr ptr -- int
```

The resulting compar is a defining word for creating specific function pointers[5], like this:

```
['] n-compare compar fptr-n-compare
```

And now you can use that for calling qsort:

```
: sort-cells ( addr u -- )
  1 cells fptr-n-compare qsort ;
```

4 Status

This C interface is currently just a paper design, but its implementation is planned for the near future.

5 Conclusion

Designing a C interface that allows platform-independent calls to C functions, is convenient to program, and has some other nice properties poses a number of subproblems, in particular the mismatch between the type systems of Forth and the C. In this paper we discuss these problems and present a solution: The declartation of parameter types is divided into: a platform-independent Forth-type part, with defaults that make most such declarations unnecessary; and a platform-dependent C-type part that can be generated out of C's .h-files. The main part of the Forth code, that part that contains the calls to C, is platform-independent.

Acknowledgments

I thank Sergey N. Baranov for his helpful comments on a draft version of the paper.

[4] Of course, there is still the question of where the SEEK_SET is coming from; this is a constant with a platform-specific value, and would ideally also be created by our .h-file processor.

[5] An alternative would have been to make compar just a conversion word that would typically be used with constant, but that might encourage the users to call it several times with the same execution token, and that might cost memory every time.