libffi: a foreign function interface library

For Version 3.4.2 of libffi

Anthony Green

This manual is for libffi, a portable foreign function interface library.

1 What is libffi?

Compilers for high level languages generate code that follow certain conventions. These conventions are necessary, in part, for separate compilation to work. One such convention is the *calling convention*. The calling convention is a set of assumptions made by the compiler about where function arguments will be found on entry to a function. A calling convention also specifies where the return value for a function is found. The calling convention is also sometimes called the *ABI* or *Application Binary Interface*.

Some programs may not know at the time of compilation what arguments are to be passed to a function. For instance, an interpreter may be told at run-time about the number and types of arguments used to call a given function. '`Libffi`' can be used in such programs to provide a bridge from the interpreter program to compiled code.

The '`libffi`' library provides a portable, high level programming interface to various calling conventions. This allows a programmer to call any function specified by a call interface description at run time.

FFI stands for Foreign Function Interface. A foreign function interface is the popular name for the interface that allows code written in one language to call code written in another language. The '`libffi`' library really only provides the lowest, machine dependent layer of a fully featured foreign function interface. A layer must exist above '`libffi`' that handles type conversions for values passed between the two languages.

2 Using libffi

2.1 The Basics

'`Libffi`' assumes that you have a pointer to the function you wish to call and that you know the number and types of arguments to pass it, as well as the return type of the function.

The first thing you must do is create an `ffi_cif` object that matches the signature of the function you wish to call. This is a separate step because it is common to make multiple calls using a single `ffi_cif`. The *cif* in `ffi_cif` stands for Call InterFace. To prepare a call interface object, use the function `ffi_prep_cif`.

`ffi_status` *ffi_prep_cif* (*ffi_cif **`cif`, *ffi_abi* `abi`, *unsigned int* `nargs`, *ffi_type **`rtype`, *ffi_type ***`argtypes`) [Function]

> This initializes *cif* according to the given parameters.
>
> *abi* is the ABI to use; normally `FFI_DEFAULT_ABI` is what you want. Section 2.4 [Multiple ABIs], page 11 for more information.
>
> *nargs* is the number of arguments that this function accepts.
>
> *rtype* is a pointer to an `ffi_type` structure that describes the return type of the function. See Section 2.3 [Types], page 3.
>
> *argtypes* is a vector of `ffi_type` pointers. *argtypes* must have *nargs* elements. If *nargs* is 0, this argument is ignored.
>
> `ffi_prep_cif` returns a `libffi` status code, of type `ffi_status`. This will be either `FFI_OK` if everything worked properly; `FFI_BAD_TYPEDEF` if one of the `ffi_type` objects is incorrect; or `FFI_BAD_ABI` if the *abi* parameter is invalid.

If the function being called is variadic (varargs) then `ffi_prep_cif_var` must be used instead of `ffi_prep_cif`.

`ffi_status` *ffi_prep_cif_var* (*ffi_cif ****cif***, *ffi_abi* **abi**, *unsigned int* ***nfixedargs***, *unsigned int* ***ntotalargs***, *ffi_type ****rtype***, *ffi_type *****argtypes***) [Function]

This initializes *cif* according to the given parameters for a call to a variadic function. In general its operation is the same as for `ffi_prep_cif` except that:

nfixedargs is the number of fixed arguments, prior to any variadic arguments. It must be greater than zero.

ntotalargs the total number of arguments, including variadic and fixed arguments. *argtypes* must have this many elements.

`ffi_prep_cif_var` will return `FFI_BAD_ARGTYPE` if any of the variable argument types are `ffi_type_float` (promote to `ffi_type_double` first), or any integer type small than an int (promote to an int-sized type first).

Note that, different cif's must be prepped for calls to the same function when different numbers of arguments are passed.

Also note that a call to `ffi_prep_cif_var` with *nfixedargs=nototalargs* is NOT equivalent to a call to `ffi_prep_cif`.

Note that the resulting `ffi_cif` holds pointers to all the `ffi_type` objects that were used during initialization. You must ensure that these type objects have a lifetime at least as long as that of the `ffi_cif`.

To call a function using an initialized `ffi_cif`, use the `ffi_call` function:

`void` *ffi_call* (*ffi_cif ****cif***, *void ****fn***, *void ****rvalue***, *void *****avalues***) [Function]

This calls the function *fn* according to the description given in *cif*. *cif* must have already been prepared using `ffi_prep_cif`.

rvalue is a pointer to a chunk of memory that will hold the result of the function call. This must be large enough to hold the result, no smaller than the system register size (generally 32 or 64 bits), and must be suitably aligned; it is the caller's responsibility to ensure this. If *cif* declares that the function returns `void` (using `ffi_type_void`), then *rvalue* is ignored.

In most situations, '`libffi`' will handle promotion according to the ABI. However, for historical reasons, there is a special case with return values that must be handled by your code. In particular, for integral (not `struct`) types that are narrower than the system register size, the return value will be widened by '`libffi`'. '`libffi`' provides a type, `ffi_arg`, that can be used as the return type. For example, if the CIF was defined with a return type of `char`, '`libffi`' will try to store a full `ffi_arg` into the return value.

avalues is a vector of `void *` pointers that point to the memory locations holding the argument values for a call. If *cif* declares that the function has no arguments (i.e., *nargs* was 0), then *avalues* is ignored. Note that argument values may be modified by the callee (for instance, structs passed by value); the burden of copying pass-by-value arguments is placed on the caller.

Note that while the return value must be register-sized, arguments should exactly match their declared type. For example, if an argument is a `short`, then the entry in *avalues* should point to an object declared as `short`; but if the return type is `short`, then *rvalue* should point to an object declared as a larger type – usually `ffi_arg`.

2.2 Simple Example

Here is a trivial example that calls `puts` a few times.

```
#include <stdio.h>
#include <ffi.h>

int main()
{
  ffi_cif cif;
  ffi_type *args[1];
  void *values[1];
  char *s;
  ffi_arg rc;

  /* Initialize the argument info vectors */
  args[0] = &ffi_type_pointer;
  values[0] = &s;

  /* Initialize the cif */
  if (ffi_prep_cif(&cif, FFI_DEFAULT_ABI, 1,
       &ffi_type_sint, args) == FFI_OK)
    {
      s = "Hello World!";
      ffi_call(&cif, puts, &rc, values);
      /* rc now holds the result of the call to puts */

      /* values holds a pointer to the function's arg, so to
         call puts() again all we need to do is change the
         value of s */
      s = "This is cool!";
      ffi_call(&cif, puts, &rc, values);
    }

  return 0;
}
```

2.3 Types

2.3.1 Primitive Types

`Libffi` provides a number of built-in type descriptors that can be used to describe argument and return types:

`ffi_type_void`
: The type `void`. This cannot be used for argument types, only for return values.

`ffi_type_uint8`
: An unsigned, 8-bit integer type.

`ffi_type_sint8`
: A signed, 8-bit integer type.

`ffi_type_uint16`
: An unsigned, 16-bit integer type.

`ffi_type_sint16`
: A signed, 16-bit integer type.

`ffi_type_uint32`
: An unsigned, 32-bit integer type.

`ffi_type_sint32`
: A signed, 32-bit integer type.

`ffi_type_uint64`
: An unsigned, 64-bit integer type.

`ffi_type_sint64`
: A signed, 64-bit integer type.

`ffi_type_float`
: The C `float` type.

`ffi_type_double`
: The C `double` type.

`ffi_type_uchar`
: The C `unsigned char` type.

`ffi_type_schar`
: The C `signed char` type. (Note that there is not an exact equivalent to the C `char` type in `libffi`; ordinarily you should either use `ffi_type_schar` or `ffi_type_uchar` depending on whether `char` is signed.)

`ffi_type_ushort`
: The C `unsigned short` type.

`ffi_type_sshort`
: The C `short` type.

`ffi_type_uint`
: The C `unsigned int` type.

`ffi_type_sint`
: The C `int` type.

`ffi_type_ulong`
: The C `unsigned long` type.

`ffi_type_slong`
: The C `long` type.

`ffi_type_longdouble`
: On platforms that have a C `long double` type, this is defined. On other platforms, it is not.

`ffi_type_pointer`
: A generic `void *` pointer. You should use this for all pointers, regardless of their real type.

`ffi_type_complex_float`
: The C `_Complex float` type.

`ffi_type_complex_double`
: The C `_Complex double` type.

`ffi_type_complex_longdouble`
: The C `_Complex long double` type. On platforms that have a C `long double` type, this is defined. On other platforms, it is not.

Each of these is of type `ffi_type`, so you must take the address when passing to `ffi_prep_cif`.

2.3.2 Structures

'`libffi`' is perfectly happy passing structures back and forth. You must first describe the structure to '`libffi`' by creating a new `ffi_type` object for it.

`ffi_type` [Data type]

The `ffi_type` has the following members:

`size_t size`
: This is set by `libffi`; you should initialize it to zero.

`unsigned short alignment`
: This is set by `libffi`; you should initialize it to zero.

`unsigned short type`
: For a structure, this should be set to `FFI_TYPE_STRUCT`.

`ffi_type **elements`
: This is a '`NULL`'-terminated array of pointers to `ffi_type` objects. There is one element per field of the struct.

: Note that '`libffi`' has no special support for bit-fields. You must manage these manually.

The `size` and `alignment` fields will be filled in by `ffi_prep_cif` or `ffi_prep_cif_var`, as needed.

2.3.3 Size and Alignment

`libffi` will set the `size` and `alignment` fields of an `ffi_type` object for you. It does so using its knowledge of the ABI.

You might expect that you can simply read these fields for a type that has been laid out by `libffi`. However, there are some caveats.

- The size or alignment of some of the built-in types may vary depending on the chosen ABI.
- The size and alignment of a new structure type will not be set by `libffi` until it has been passed to `ffi_prep_cif` or `ffi_get_struct_offsets`.
- A structure type cannot be shared across ABIs. Instead each ABI needs its own copy of the structure type.

So, before examining these fields, it is safest to pass the `ffi_type` object to `ffi_prep_cif` or `ffi_get_struct_offsets` first. This function will do all the needed setup.

```
ffi_type *desired_type;
ffi_abi desired_abi;
...
ffi_cif cif;
if (ffi_prep_cif (&cif, desired_abi, 0, desired_type, NULL) == FFI_OK)
  {
    size_t size = desired_type->size;
    unsigned short alignment = desired_type->alignment;
  }
```

`libffi` also provides a way to get the offsets of the members of a structure.

`ffi_status` *ffi_get_struct_offsets* (*ffi_abi abi, ffi_type *struct_type, size_t *offsets*) [Function]

Compute the offset of each element of the given structure type. *abi* is the ABI to use; this is needed because in some cases the layout depends on the ABI.

offsets is an out parameter. The caller is responsible for providing enough space for all the results to be written – one element per element type in *struct_type*. If *offsets* is `NULL`, then the type will be laid out but not otherwise modified. This can be useful for accessing the type's size or layout, as mentioned above.

This function returns `FFI_OK` on success; `FFI_BAD_ABI` if *abi* is invalid; or `FFI_BAD_TYPEDEF` if *struct_type* is invalid in some way. Note that only `FFI_STRUCT` types are valid here.

2.3.4 Arrays, Unions, and Enumerations

2.3.4.1 Arrays

'`libffi`' does not have direct support for arrays or unions. However, they can be emulated using structures.

To emulate an array, simply create an `ffi_type` using `FFI_TYPE_STRUCT` with as many members as there are elements in the array.

```
ffi_type array_type;
ffi_type **elements
int i;

elements = malloc ((n + 1) * sizeof (ffi_type *));
for (i = 0; i < n; ++i)
  elements[i] = array_element_type;
```

```
elements[n] = NULL;

array_type.size = array_type.alignment = 0;
array_type.type = FFI_TYPE_STRUCT;
array_type.elements = elements;
```

Note that arrays cannot be passed or returned by value in C – structure types created like this should only be used to refer to members of real `FFI_TYPE_STRUCT` objects.

However, a phony array type like this will not cause any errors from '`libffi`' if you use it as an argument or return type. This may be confusing.

2.3.4.2 Unions

A union can also be emulated using `FFI_TYPE_STRUCT`. In this case, however, you must make sure that the size and alignment match the real requirements of the union.

One simple way to do this is to ensue that each element type is laid out. Then, give the new structure type a single element; the size of the largest element; and the largest alignment seen as well.

This example uses the `ffi_prep_cif` trick to ensure that each element type is laid out.

```
ffi_abi desired_abi;
ffi_type union_type;
ffi_type **union_elements;

int i;
ffi_type element_types[2];

element_types[1] = NULL;

union_type.size = union_type.alignment = 0;
union_type.type = FFI_TYPE_STRUCT;
union_type.elements = element_types;

for (i = 0; union_elements[i]; ++i)
  {
    ffi_cif cif;
    if (ffi_prep_cif (&cif, desired_abi, 0, union_elements[i], NULL) == FFI_OK)
      {
        if (union_elements[i]->size > union_type.size)
          {
            union_type.size = union_elements[i];
            size = union_elements[i]->size;
          }
        if (union_elements[i]->alignment > union_type.alignment)
          union_type.alignment = union_elements[i]->alignment;
      }
  }
```

2.3.4.3 Enumerations

`libffi` does not have any special support for C `enum`s. Although any given `enum` is implemented using a specific underlying integral type, exactly which type will be used cannot be determined by `libffi` – it may depend on the values in the enumeration or on compiler flags such as '`-fshort-enums`'. See Section "Structures unions enumerations and bit-fields implementation" in `gcc`, for more information about how GCC handles enumerations.

2.3.5 Type Example

The following example initializes a `ffi_type` object representing the `tm` struct from Linux's '`time.h`'.

Here is how the struct is defined:

```
struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
    /* Those are for future use. */
    long int __tm_gmtoff__;
    __const char *__tm_zone__;
};
```

Here is the corresponding code to describe this struct to `libffi`:

```
    {
      ffi_type tm_type;
      ffi_type *tm_type_elements[12];
      int i;

      tm_type.size = tm_type.alignment = 0;
      tm_type.type = FFI_TYPE_STRUCT;
      tm_type.elements = &tm_type_elements;

      for (i = 0; i < 9; i++)
          tm_type_elements[i] = &ffi_type_sint;

      tm_type_elements[9] = &ffi_type_slong;
      tm_type_elements[10] = &ffi_type_pointer;
      tm_type_elements[11] = NULL;

      /* tm_type can now be used to represent tm argument types and
 return types for ffi_prep_cif() */
    }
```

2.3.6 Complex Types

'libffi' supports the complex types defined by the C99 standard (_Complex float, _Complex double and _Complex long double with the built-in type descriptors ffi_type_complex_float, ffi_type_complex_double and ffi_type_complex_longdouble.

Custom complex types like _Complex int can also be used. An ffi_type object has to be defined to describe the complex type to 'libffi'.

ffi_type [Data type]

size_t size
: This must be manually set to the size of the complex type.

unsigned short alignment
: This must be manually set to the alignment of the complex type.

unsigned short type
: For a complex type, this must be set to FFI_TYPE_COMPLEX.

ffi_type **elements
: This is a 'NULL'-terminated array of pointers to ffi_type objects. The first element is set to the ffi_type of the complex's base type. The second element must be set to NULL.

The section Section 2.3.7 [Complex Type Example], page 9 shows a way to determine the size and alignment members in a platform independent way.

For platforms that have no complex support in libffi yet, the functions ffi_prep_cif and ffi_prep_args abort the program if they encounter a complex type.

2.3.7 Complex Type Example

This example demonstrates how to use complex types:

```
#include <stdio.h>
#include <ffi.h>
#include <complex.h>

void complex_fn(_Complex float cf,
                _Complex double cd,
                _Complex long double cld)
{
  printf("cf=%f+%fi\ncd=%f+%fi\ncld=%f+%fi\n",
         (float)creal (cf), (float)cimag (cf),
         (float)creal (cd), (float)cimag (cd),
         (float)creal (cld), (float)cimag (cld));
}

int main()
{
  ffi_cif cif;
  ffi_type *args[3];
  void *values[3];
```

```
  _Complex float cf;
  _Complex double cd;
  _Complex long double cld;

  /* Initialize the argument info vectors */
  args[0] = &ffi_type_complex_float;
  args[1] = &ffi_type_complex_double;
  args[2] = &ffi_type_complex_longdouble;
  values[0] = &cf;
  values[1] = &cd;
  values[2] = &cld;

  /* Initialize the cif */
  if (ffi_prep_cif(&cif, FFI_DEFAULT_ABI, 3,
                   &ffi_type_void, args) == FFI_OK)
    {
      cf = 1.0 + 20.0 * I;
      cd = 300.0 + 4000.0 * I;
      cld = 50000.0 + 600000.0 * I;
      /* Call the function */
      ffi_call(&cif, (void (*)(void))complex_fn, 0, values);
    }

  return 0;
}
```

This is an example for defining a custom complex type descriptor for compilers that support them:

```
/*
 * This macro can be used to define new complex type descriptors
 * in a platform independent way.
 *
 * name: Name of the new descriptor is ffi_type_complex_<name>.
 * type: The C base type of the complex type.
 */
#define FFI_COMPLEX_TYPEDEF(name, type, ffitype)             \
  static ffi_type *ffi_elements_complex_##name [2] = {      \
    (ffi_type *)(&ffitype), NULL                             \
  };                                                        \
  struct struct_align_complex_##name {                      \
    char c;                                                  \
    _Complex type x;                                         \
  };                                                        \
  ffi_type ffi_type_complex_##name = {                      \
    sizeof(_Complex type),                                   \
    offsetof(struct struct_align_complex_##name, x),         \
    FFI_TYPE_COMPLEX,                                        \
```

```
    (ffi_type **)ffi_elements_complex_##name                   \
  }

/* Define new complex type descriptors using the macro: */
/* ffi_type_complex_sint */
FFI_COMPLEX_TYPEDEF(sint, int, ffi_type_sint);
/* ffi_type_complex_uchar */
FFI_COMPLEX_TYPEDEF(uchar, unsigned char, ffi_type_uint8);
```

The new type descriptors can then be used like one of the built-in type descriptors in the previous example.

2.4 Multiple ABIs

A given platform may provide multiple different ABIs at once. For instance, the x86 platform has both '`stdcall`' and '`fastcall`' functions.

`libffi` provides some support for this. However, this is necessarily platform-specific.

2.5 The Closure API

`libffi` also provides a way to write a generic function – a function that can accept and decode any combination of arguments. This can be useful when writing an interpreter, or to provide wrappers for arbitrary functions.

This facility is called the *closure API*. Closures are not supported on all platforms; you can check the `FFI_CLOSURES` define to determine whether they are supported on the current platform.

Because closures work by assembling a tiny function at runtime, they require special allocation on platforms that have a non-executable heap. Memory management for closures is handled by a pair of functions:

`void` **ffi_closure_alloc* (*size_t* **`size`**, *void *****`code`**) [Function]

Allocate a chunk of memory holding *size* bytes. This returns a pointer to the writable address, and sets **code* to the corresponding executable address.

size should be sufficient to hold a `ffi_closure` object.

`void` *ffi_closure_free* (*void ****`writable`**) [Function]

Free memory allocated using `ffi_closure_alloc`. The argument is the writable address that was returned.

Once you have allocated the memory for a closure, you must construct a `ffi_cif` describing the function call. Finally you can prepare the closure function:

`ffi_status` *ffi_prep_closure_loc* (*ffi_closure ****`closure`**, *ffi_cif ****`cif`**, *void* (*****`fun`**) (*ffi_cif ****`cif`**, *void ****`ret`**, *void *****`args`**, *void ****`user_data`**), *void ****`user_data`**, *void ****`codeloc`**) [Function]

Prepare a closure function. The arguments to `ffi_prep_closure_loc` are:

closure The address of a `ffi_closure` object; this is the writable address returned by `ffi_closure_alloc`.

cif The `ffi_cif` describing the function parameters. Note that this object, and the types to which it refers, must be kept alive until the closure itself is freed.

user_data An arbitrary datum that is passed, uninterpreted, to your closure function.

codeloc The executable address returned by `ffi_closure_alloc`.

fun The function which will be called when the closure is invoked. It is called with the arguments:

- *cif* The `ffi_cif` passed to `ffi_prep_closure_loc`.
- *ret* A pointer to the memory used for the function's return value.

  If the function is declared as returning `void`, then this value is garbage and should not be used.

  Otherwise, *fun* must fill the object to which this points, following the same special promotion behavior as `ffi_call`. That is, in most cases, *ret* points to an object of exactly the size of the type specified when *cif* was constructed. However, integral types narrower than the system register size are widened. In these cases your program may assume that *ret* points to an `ffi_arg` object.
- *args* A vector of pointers to memory holding the arguments to the function.
- *user_data* The same *user_data* that was passed to `ffi_prep_closure_loc`.

`ffi_prep_closure_loc` will return `FFI_OK` if everything went ok, and one of the other `ffi_status` values on error.

After calling `ffi_prep_closure_loc`, you can cast *codeloc* to the appropriate pointer-to-function type.

You may see old code referring to `ffi_prep_closure`. This function is deprecated, as it cannot handle the need for separate writable and executable addresses.

2.6 Closure Example

A trivial example that creates a new `puts` by binding `fputs` with `stdout`.

```
#include <stdio.h>
#include <ffi.h>

/* Acts like puts with the file given at time of enclosure. */
void puts_binding(ffi_cif *cif, void *ret, void* args[],
                  void *stream)
{
  *(ffi_arg *)ret = fputs(*(char **)args[0], (FILE *)stream);
}
```

```
typedef int (*puts_t)(char *);

int main()
{
  ffi_cif cif;
  ffi_type *args[1];
  ffi_closure *closure;

  void *bound_puts;
  int rc;

  /* Allocate closure and bound_puts */
  closure = ffi_closure_alloc(sizeof(ffi_closure), &bound_puts);

  if (closure)
    {
      /* Initialize the argument info vectors */
      args[0] = &ffi_type_pointer;

      /* Initialize the cif */
      if (ffi_prep_cif(&cif, FFI_DEFAULT_ABI, 1,
                       &ffi_type_sint, args) == FFI_OK)
        {
          /* Initialize the closure, setting stream to stdout */
          if (ffi_prep_closure_loc(closure, &cif, puts_binding,
                                   stdout, bound_puts) == FFI_OK)
            {
              rc = ((puts_t)bound_puts)("Hello World!");
              /* rc now holds the result of the call to fputs */
            }
        }
    }

  /* Deallocate both closure, and bound_puts */
  ffi_closure_free(closure);

  return 0;
}
```

2.7 Thread Safety

`libffi` is not completely thread-safe. However, many parts are, and if you follow some simple rules, you can use it safely in a multi-threaded program.

- `ffi_prep_cif` may modify the `ffi_type` objects passed to it. It is best to ensure that only a single thread prepares a given `ffi_cif` at a time.

- On some platforms, `ffi_prep_cif` may modify the size and alignment of some types, depending on the chosen ABI. On these platforms, if you switch between ABIs, you must ensure that there is only one call to `ffi_prep_cif` at a time.

  Currently the only affected platform is PowerPC and the only affected type is `long double`.

3 Memory Usage

Note that memory allocated by `ffi_closure_alloc` and freed by `ffi_closure_free` does not come from the same general pool of memory that `malloc` and `free` use. To accomodate security settings, '`libffi`' may aquire memory, for example, by mapping temporary files into multiple places in the address space (once to write out the closure, a second to execute it). The search follows this list, using the first that works:

- A anonymous mapping (i.e. not file-backed)
- `memfd_create()`, if the kernel supports it.
- A file created in the directory referenced by the environment variable `LIBFFI_TMPDIR`.
- Likewise for the environment variable `TMPDIR`.
- A file created in `/tmp`.
- A file created in `/var/tmp`.
- A file created in `/dev/shm`.
- A file created in the user's home directory (`$HOME`).
- A file created in any directory listed in `/etc/mtab`.
- A file created in any directory listed in `/proc/mounts`.

If security settings prohibit using any of these for closures, `ffi_closure_alloc` will fail.

4 Missing Features

`libffi` is missing a few features. We welcome patches to add support for these.

- Variadic closures.
- There is no support for bit fields in structures.
- The "raw" API is undocumented.
- The Go API is undocumented.

Index

A

C

F

V