GNUstep Objective-C ABI version 10

David Chisnall

July 26, 2024

Contents

Chapter 1

Introduction

The GNUstep Objective-C runtime has a complicated history. It began as the Étoilé Objective-C runtime, a research prototype that adopted a lot of ideas from the VPRI Combined Object-Lambda Architecture (COLA) model and was intended to support languages like Self (prototype-based, multiple inheritance) as well as Objective-C. This code was repurposed as libobjc2 to provide an Objective-C runtime that clang could use. At the time, the GCC Objective-C runtime had a GPLv2 exemption that applied only to code compiled with GCC and so any code compiled with clang and linked against the GCC runtime had to be licensed as GPLv2 or later.

GCC's Objective-C support at this time was lacking a number of features of more modern Objective-C (for example, declared properties) and showed no signs of improving.

Eventually libobjc2 was adopted by the GNUstep project and became the GNUstep Objective-C runtime. It was intended as a drop-in replacement for the GCC runtime and so adopted the GCC Objective-C ABI and extended it in a variety of backwards-compatible ways.

The GCC ABI was, itself, inherited from the original NeXT Objective-C runtime. The Free Software Foundation used the GPL and the threat of legal action to force NeXT to release their GCC changes to support Objective-C. They were left with some shockingly bad code, which was completely useless without an Objective-C runtime. The GCC team committed the shockingly bad code and wrote a runtime that was almost, but not quite, compatible with the NeXT one. In particular, it did not implement `objc_msgSend`, which requires hand-written assembly, and instead modified the compiler to call a function to look up the method and then to call the result, giving a portable pure-C design.

As such, the ABI supported by the GNUstep Objective-C runtime dates back to 1988 and is starting to show its age. It includes a number of hacks and misfeatures that are in dire need of replacing. This document describes the new ABI used by version 2.0 of the runtime.

1.1 Mistakes

Supporting a non-fragile ABI was one of the early design goals of the GNUstep Objective-C runtime. When Apple switched to a new runtime, they were able

to require that everyone recompiled all of their code to support the non-fragile ABI and, in particular, were able to support only the new ABI on ARM and on 64-bit platforms.

At the time, it was not possible to persuade everyone to recompile all of their code for a new GNUstep runtime, and so I made a number of questionable design decisions to allow classes compiled with the non-fragile ABI to subclass ones compiled with the fragile ABI. These decisions have led to some issues where code using the non-fragile ABI ends up being fragile.

The new ABI makes no attempt to support mixing old and new ABI code. The runtime will work with either, but not with both at the same time. It will upgrade the old structures to the new on load (consuming more memory and providing an incentive to recompile) and will then use only the new structures internally.

The GNUstep runtime incorporates the safe method caching mechanism originally from the Étoilé runtime. Unfortunately, this added some significant memory overhead (each selector, class pair needed a separate version). In the new design, we move to a single 64-bit version counter (which, if incremented once per cycle on a 2GHz CPU, will still not overflow after a few millennia). This saves memory (around 5% total memory usage in a microbenchmark that simply sends a `+class` message to every class in the Foundation framework) at the cost of increasing the rate of cache invalidations. Because method replacement is relatively rare in Objective-C, this extra overhead is relatively rare.

1.2 Changed circumstances

When the original GCC runtime was released, linkers were designed primarily to work with C. C guarantees that each symbol is defined in precisely one compilation unit. In contrast, C++ (10 years away from standardisation at the time the GCC runtime was released) has a number of language features that rely on symbols appearing in multiple compilation units. The original 4Front C++ compiler worked by compiling without emitting any of these, parsing the linker errors, and then recompiling adding missing ones.

More modern implementations of C++ emit these symbols in every compilation unit that references them and rely on the linker to discard duplicates. Modern linkers support *COMDATs* for this purpose.

The NeXT runtime was able to work slightly differently. The Mach-O binary format (used by NeXT and Apple) provides a mechanism for registering code that will handle loading for certain sections, thus delegating some linker functionality to the runtime.

In addition to COMDATs, modern linkers support generating symbols that correspond to the start and end of sections. This makes it possible for the new ABI to emit all declarations of a particular kind in a section and for the runtime to then receive an array of all of the objects in that section.

Chapter 2

Entry point

The legacy GCC ABI provided a `__objc_exec_class` function that registered all of the Objective-C data for a single compilation unit. This has two downsides:

- It means that the `+load` methods will be called one at a time, as classes are loaded, because the runtime has no way of knowing when an entire library has been loaded.
- It prevents any deduplication between compilation units, and so a selector used in 100 .m files and linked into a single binary will occur 100 times and be passed to the runtime for merging 100 times.

2.1 The new entry point

The new runtime provides an `__objc_load` function for loading an entire library at a time. This function takes a pointer to the structure shown in Listing **??**.

For the current ABI, the `version` field must always be zero. This field exists to allow future versions of the ABI to add new fields to the end, which can be ignored by older runtime implementations.

The remaining fields all contain pointers to the start and end of a section. The sections are listed in table **??**.

The `__objc_selectors` section contains all of the selectors referenced by this library. As described in chapter **??**, these are deduplicated by the linker, so each library should contain only one copy of any given selector.

Similarly, the `__objc_classes`, `__objc_cats`, and `__objc_protocols` sections contain classes, categories, and protocols: the three top-level structural components in an Objective-C program. These are all described in later chapters.

The `__objc_class_refs` section contains variables that are used for accessing classes. These are described in Section **??** and provide loose coupling between the representation of the class and accesses to it.

The `__objc_protocol_refs` section contains variables that point to protocols in the same way. This indirection layer makes it possible for future versions of the ABI to make incompatible changes to the protocol structure and for the runtime to upgrade old libraries on load.

Listing 2.1: The Objective-C library description structure. [From loader.c]

```
struct objc_init
{
  uint64_t version;
  SEL sel_begin;
  SEL sel_end;
  Class *cls_begin;
  Class *cls_end;
  Class *cls_ref_begin;
  Class *cls_ref_end;
  struct objc_category *cat_begin;
  struct objc_category *cat_end;
  struct objc_protocol *proto_begin;
  struct objc_protocol *proto_end;
  struct objc_protocol **proto_ref_begin;
  struct objc_protocol **proto_ref_end;
  struct objc_alias *alias_begin;
  struct objc_alias *alias_end;
  struct nsstr *strings_begin;
  struct nsstr *strings_end;
};
```

Prefix	Section
sel_	__objc_selectors
cls_	__objc_classes
cls_ref_	__objc_class_refs
cat_	__objc_cats
proto_	__objc_protocols
proto_ref_	__objc_protocol_refs
alias	__objc_class_aliases

Table 2.1: Section names for Objective-C components.

2.2 Compiler responsibilities

For each compilation unit, the compiler must emit a copy of both the `objc_init` structure and a function that passes it to the runtime, in such a way that the linker will preserve a single copy. On ELF platforms, these are hidden weak symbols with a comdat matching their name. The load function is called `.objcv2_load_function` and the initializer structure is called `.objc_init` (the dot prefix preventing conflicts with any C symbols). The compiler also emits a `.objc_ctor` variable in the `.ctors` section, with a `.objc_ctor` comdat.

The end result after linking is a single copy of the `.objc_ctor` variable in the `.ctors` section, which causes a single copy of the `.objcv2_load_function` to be called, passing a single copy of the `.objc_init` structure to the runtime on binary load.

The `.objc_init` structure is initialised by the `__start_{section name}` and `__stop_{section name}` symbols, which the linker will replace with relocations describing the start and end of each section.

The linker does not automatically initialise these variables if the sections do not exist, so compilation units that do not include any entries for one or more of them must emit a zero-filled section. The runtime will then ignore the zero entry.

Chapter 3

Selectors

Typed selectors are one of the largest differences between the GNU family of runtimes (GCC, GNUstep, ObjFW) and the NeXT (NeXT, macOS, iOS) family. In the NeXT design, selectors are just (interned) strings representing the selector name. This can cause stack corruption when different leafs in the class hierarchy implement methods with the same name but different types and some code sends a message to one of them using a variable of type `id`. In the GNU family, they are a pair of the method name and the type encoding.

The GNUstep ABI represents selectors using the structure described in Listing **??**. The first field is a union of the value emitted by the compiler and the value used by the runtime. The compiler initialises the `name` field with the string representation of the selector name, but when the runtime registers the selector it will replace this with an integer value that uniquely identifies the selector (it will also store the name in a table at this index so selectors can be mapped back to names easily).

3.1 Symbol naming

In this ABI, unlike the GCC ABI, we try to ensure that the linker removes as much duplicate data as possible. As such, each selector, selector name, and selector type encoding is emitted as a weak symbol with a well-known name

Listing 3.1: The selector structure. [From selector.h]

```
struct objc_selector
{
  union
  {

    const char *name;

    uintptr_t index;
  };

  const char * types;
};
```

name, with hidden visibility. When linking, the linker will discard all except for one (though different shared libraries will have different copies).

The selector names are emitted as `.objc_sel_name_{selector name}`, the type encodings as `.objc_sel_name_{mangled type encoding}` and the selectors themselves as `.objc_sel_name_{selector name}_{mangled type encoding}`. The *mangled* type encoding replaces the @ character with a `'\1'` byte. This mangling prevents conflicts with symbol versioning (which uses the @ character to separate the symbol name from its version).

This deduplication is not required for correctness: the runtime ensures that selectors have unique indexes, but should reduce the binary size.

Chapter 4

Classes

The class structure is shown in Listing **??**. Each class is emitted as an instance of this structure as a symbol called `_OBJC_CLASS_{class name}`. The `isa` pointer for the class is initialised to point to another class structure describing the metaclass, in which all class methods and properties are defined.

The `super_class` field is initialised to the class structure for the superclass. If the runtime is upgrading the class structure to a newer version of the ABI then this pointer will be updated to the upgraded version of the class on load.

The `name` field points to a null-terminated string containing the class name. The `version` field should be initialised to zero.

The `info` field is used internally as a bitfield. The compiler is responsible for setting this to 0 for classes and 1 for metaclasses. The remaining low 8 bits are reserved for use by future versions of the ABI. All higher bits are reserved for the runtime to use for dynamic properties of classes.

The `abi_version` field is used to differentiate different versions of the class ABI structure and is currently always zero.

The `ivars`, `methods`, `properties` and `protocols` fields describe the class and are explained in the next sections.

4.1 Class metadata

Most of the class metadata uses the pattern similar to the following:

```
struct metadata_element;

struct metadata_list
{
  int count;
  int size;
  struct metadata_element elements[];
}
```

In this example, `metadata_list` describes an ordered collection of `metadata_element` elements. The `count` field indicates how many elements there are in the `elements` array. The array is appended to the structure, but *is not guaranteed to be a C array of the element type.* To allow for future expansion, the `size` field in the list structure defines the size of one element in the array. Future versions of the ABI are able to increase the size of `metadata_element`, without breaking existing

Listing 4.1: The class structure. [From class.h]

```
struct objc_class
{

  Class                       isa;

  Class                       super_class;

  const char                 *name;

  long                        version;

  unsigned long               info;

  long                        instance_size;

  struct objc_ivar_list      *ivars;

  struct objc_method_list    *methods;

  void                       *dtable;

  Class                       subclass_list;

  IMP                         cxx_construct;

  IMP                         cxx_destruct;

  Class                       sibling_class;


  struct objc_protocol_list *protocols;

  struct reference_list      *extra_data;

  long                        abi_version;

  struct objc_property_list *properties;
};
```

Listing 4.2: The instance variable list structure. [From ivar.h]

```
struct objc_ivar_list
{

  int               count;

  size_t            size;

  struct objc_ivar ivar_list[];
};
```

Listing 4.3: The instance variable structure. [From ivar.h]

```
struct objc_ivar
{

  const char *name;

  const char *type;

  int        *offset;

  uint32_t    size;

  uint32_t    flags;
};
```

versions of the runtime. Existing versions of the runtime will simply ignore any missing fields.

This pattern is used for instance variables, methods, and properties.

4.1.1 Instance variables

Instance variables are defined in list shown in Listing **??**, which follows the structure outlined in Section **??**. The entries in the list are elements of the `struct objc_ivar` structure, described in Listing **??**. Future versions of the ABI may add additional fields, in which case they should increase the value of `size` in the list structure.

The `name` field points to a null-terminated string containing the name of the instance variable. The `type` field contains the extended type encoding of the instance variable.

The `offset` field contains a pointer to the instance variable offset variable. This is of the form `__objc_ivar_offset_{class name}.{ivar name}.{type encoding}`. This variable is a 32-bit integer, which restricts objects to 4GB plus the size of the last field. The type encoding is in the traditional format, with the mangling defined in Section **??** applied. This means that, for example, changing the type of an instance variable from `NSString*` to `NSConstantString*` will not cause a linker failure, but changing its type from `int` to `float` will.

The `flags` field is a bitmask. The first two bits indicate the ownership, as shown in Listing **??**. This is always `ownership_invalid` (0) for instance variables that are not objects.

The next bit (bit 2) indicates that the `type` field contains an extended type

Listing 4.4: Instance variable ownership. [From ivar.h]

```
typedef enum {

  ownership_invalid = 0,

  ownership_strong  = 1,

  ownership_weak    = 2,

  ownership_unsafe  = 3
} objc_ivar_ownership;
```

Listing 4.5: The method structure. [From method.h]

```
struct objc_method_list
{

  struct objc_method_list  *next;

  int                       count;

  size_t                    size;

  struct objc_method        methods[];
};
```

encoding and should always be set in generated code. This bit exists so that instance variable structures generated from older ABIs can be automatically upgraded. The next six bits (bits 3–8) contain the base-2 logarithm of the alignment. This is enough to describe any power of two alignment from 0 to 2^{63}. There is no point supporting 2^{64}-byte alignment because we currently don't support any platforms with greater than 2^{64}-byte address spaces[1] and on such a platform there can be at most one object requiring 2^{64}-byte alignment, and it therefore fit inside an object. Instance variable offsets must currently be within a 4GB (2^{32}-byte) range and so even 6 bits is somewhat excessive.

4.1.2 Methods

Methods are defined in list shown in Listing **??**, which follows the structure outlined in Section ??. The entries in the list are elements of the `struct objc_method` structure, described in Listing **??**. Future versions of the ABI may add additional fields, in which case they should increase the value of `size` in the list structure.

The method list structure contains a `next` pointer so that method lists in categories and classes can be combined. This should always be initialised to null in the compiler.

The method structure is comparatively simple. The first field (`imp`) is a pointer to the method. The second field is a pointer to the selector, which should be generated as described in Chapter ??. The final field is the extended type encoding. Note that the selector is typed and incorporates the traditional type

[1]Most 64-bit platforms currently support only 2^{48}- or 2^{56}-byte address spaces.

Listing 4.6: The method structure. [From method.h]

```
struct objc_method
{

  IMP          imp;

  SEL          selector;

  const char *types;
};
```

Listing 4.7: The protocol structure. [From protocol.h]

```
struct objc_protocol_list
{

  struct objc_protocol_list *next;

  size_t                     count;

  struct objc_protocol      *list[];
};
```

encoding. This allows the runtime to return either the traditional or extended type encoding, as required.

4.1.3 Protocols

Protocols adopted by a class are stored in the `objc_protocol_list` structure, described in Listing **??**. These do not have a field indicating the size, because protocols are referenced by pointer.

Protocols are emitted as described in Chapter **??** and referenced directly in this structure. Explicit `@protocol` references are handled via an indirection layer, but it is safe to reference the global variable describing a protocol directly in this structure. If a future version of the runtime wishes to update the protocol structure then it is able to do so and update the pointers in the protocol list to point to the upgraded structures.

4.1.4 Declared properties

Declared properties are defined in list shown in Listing **??**, which follows the structure outlined in Section ??. The entries in the list are elements of the `struct objc_property` structure, described in Listing **??**. Future versions of the ABI may add additional fields, in which case they should increase the value of `size` in the list structure.

4.2 Class references

Each entry in the `__objc_class_refs` section is a symbol (in a COMDAT of the same name) called `_OBJC_CLASS_REF_{class name}`, which is initialised to point to a variable called `_OBJC_CLASS_{class name}`, which is the symbol for the class. This

Listing 4.8: The property structure. [From properties.h]

```
struct objc_property_list
{

  int count;

  int size;
  /*
   * The next property in a linked list.
   */
  struct objc_property_list *next;

  struct objc_property properties[];
};
```

Listing 4.9: The property structure. [From properties.h]

```
struct objc_property
{

  const char *name;

  const char *attributes;

  const char *type;

  SEL getter;

  SEL setter;
};
```

and the class structure are the *only* place where the `_OBJC_CLASS_{class name}` symbols may be referenced.

All other accesses to the class (i.e. from message sends to classes or to `super`) must be via a load of the `_OBJC_CLASS_REF_{class name}` variable.

The current version of the runtime ignores this section, but if a future runtime changes the class structure then it can update these pointers to heap-allocated versions of the new structure.

Chapter 5

Categories

Chapter 6

Protocols

Chapter 7

Encoding strings

Objective-C defines three kinds of type encoding string:

Traditional type encodings come from the NeXT Objective-C implementation[1] and are widely used in reflection. The `@encode` directive generates a type encoding in this form.

Extended type encodings were introduced by Apple. They extend the traditional type encodings and provide types for classes and parameter types for blocks.

Property attribute encodings define the properties of an attribute string.

The type encoding always uses the underlying type, ignoring `typedef`s. This means that type encodings are not stable across platforms. For example, `int64_t` may be treated as `long` or `long long`, depending on the target.

7.1 Traditional type encodings

Traditional type encodings are intended to be able to encode any C or Objective-C 1.0 types, providing only information that cannot be obtained via other introspection interfaces.

7.1.1 Primitive types

All primitive C types are represented in traditional type encodings using a single character, listed in Table **??**. For each C type that has `signed` and `unsigned` variants, the encoding format uses the same letter with the uppercase letter for the `unsigned` form and the lowercase letter for the `signed` form.

Note that, in C, all types except for `char` are implicitly signed if the `signed` keyword is omitted. In contrast, whether `char` is equivalent to `signed char` or `unsigned char` is implementation dependent. The type encoding for `char` will always match the underlying type.

BOOL

[1] Or possibly the earlier StepStone version?

Character	Type
`c`	`signed char`
`C`	`unsigned char`
`s`	`signed short`
`S`	`unsigned short`
`i`	`signed int`
`I`	`unsigned int`
`l`	`signed long`
`L`	`unsigned long`
`q`	`signed long long`
`Q`	`unsigned long long`
`f`	`float`
`d`	`double`
`B`	`_Bool` (`bool` in C++)
`v`	`void`
`?`	Unknown type

Table 7.1: Type encodings of primitive types.

7.1.2 Composite types

C contains two composite types; arrays and structures. C also provides pointers, for describing indirection. C++ classes are, for the purpose of type encodings, treated as structures.

Array are encoded in square brackets, with the number of elements followed by the element type. For example:

```
int array[42];
```

The type encoding of `array` will be `[42i]`.

Structure encodings are in braces, with the name of the structure, followed by an equals sign, followed by the encodings of all elements. For example:

```
struct Z
{
  int x;
  float y;
};
```

The encoding of `struct Z` will be `Z=if`. These encodings are combined, for example an array of 10 elements of `struct Z` would be encoded as `[10Z=if]`.

Pointers are described by prefixing the type with a caret (`^`). For example, a pointer to `struct Z` would be encoded as `^Z=if`. Pointers to incomplete structures—either forward definitions or structures currently in the process of being defined—or any pointers to structures from within other structures include the name but not the fields in the structure definition. This allows recursive structures to be represented, for example:

```
struct Recursive
{
  int x;
  struct Recursive *r;
};
```

Character	Type
`@`	`id`
`#`	`Class`
`:`	`SEL`

Table 7.2: Type encodings of Objective-C types.

This structure will yield a type encoding of `Recursive=i^Recursive`. The compiler is not required to detect recursion and may simply refer to any structure referenced by pointer omitting its encoding.

C++ references—including r-value references—are encoded as pointers.

7.1.3 C strings

A single asterisk is used as shorthand when encoding `char*`. This was intended to be an encoding for null-terminated C strings, so that the Distributed Objects system was able to copy the string.

Unfortunately, recent versions of clang will generate `*` as the encoding for both `signed char*` and even for `BOOL*` (because `BOOL` is a `typedef` for `char`. As such, an encoding of `*` gives strictly less information than `^C` or `^c` and so its use should be discouraged.

7.1.4 Objective-C types

Objective-C introduces `id`, `Class`, `SEL` and `BOOL` types. Of these, `BOOL` is a `typedef` for an underlying C type (`signed char` on Apple platforms, `unsigned char` on most GCC Objective-C platforms, `int` on VxWorks) and so does not get a new type encoding. The other types are encoded as described in Table **??**.

7.1.5 Method encodings

Method type encodings describe the argument frame. They include numbers that describe where in a classic all-arguments-on-the-stack calling convention the arguments would reside. This is not tremendously useful in most contexts, though the size can be helpful when allocating space to store an invocation.

The format for method encodings begins with the encoding of the return type, followed by the total size of the arguments in bytes. Each argument is then listed, followed by its offset in the argument frame (the offset it would be in a `struct` containing all of the arguments).

For example, the method:

```
- (int)foo: (float)a;
```

May encode as `i20@0:8f16`, assuming that pointers are 8 bytes and `int` and `float` are each 4 bytes. Note that the `self` and `_cmd` parameters are explicit here, in the `@0` (object argument at offset 0) and `:8` (`SEL` argument at offset 8) parts of the encoding.

Methods declared in protocols may include some of the qualifiers described in Table **??**. Each of these precedes the encoding for the type that it is qualifying.

Character	Type
n	in
N	inout
O	bycopy
o	out
R	byref
r	const
V	oneway

Table 7.3: Type encodings of Objective-C method argument qualifier types.

7.1.6 Blocks and function pointers

In the traditional type encoding format, functions are treated as unknown and so function pointers are encoded as `^?` (pointer to unknown type). Blocks did not exist when the format was defined and so are encoded as `@?`, signifying an unknown type that is also an object.

7.2 Extended type encodings

The extended type encoding format is a superset of the traditional format, providing extended information about blocks and objects. Objective-C object pointer types include the class or protocol types for which they are valid, blocks include their full argument signature in the same format as a method encoding.

The extended type encoding for an object is encoded in double quotes after the `@` symbol. If a class is specified, then this includes the name, followed by any specified protocols in angle brackets. For example:

```
@class NSObject;
@protocol Proto1;
@protocol Proto2;

NSObject *obj;
id<Proto> proto;
NSObject<Proto1> *qualObj;
NSObject<Proto1,Proto2> *qualObj2;
```

The extended encoding for `obj` is `@"NSObject"`. There can be at most one class type in an extended encoding, but multiple protocol types. The `proto` variable does not include a type, indicating an object that may be of any class but must confirm to the protocol `Proto`. This is encoded as `@"<Proto>"`. If more than one protocol is specified, then they are listed in turn. The encoding of `qualObj` is therefore `@"NSObject<Proto1>"` and the encoding of `qualObj2` is `@"NSObject<Proto1><Proto2>"`. Note that, in the encoding of `qualObj2`, each protocol is listed separately in angle brackets, unlike the Objective-C syntax where they are listed as a comma-separated list inside a single pair of angle brackets.

7.3 Property attribute encodings

Chapter 8

Message sending