Swift ABI Stability Manifesto

- Authors: Michael Ilseman (compiled through conversations with many others)

Introduction

The Big Picture

One of the top priorities for Swift right now is compatibility across future Swift versions. Compatibility aims at accomplishing two goals:

1. **Source compatibility** means that newer compilers can compile code written in an older version of Swift. This aims to reduce the migration pain that Swift developers face when migrating to a newer Swift version. Without source compatibility, projects face version-lock where all source code in a project and its packages must be written in the same version of Swift. With source compatibility, package authors will be able to maintain a single code base across multiple Swift versions while allowing their users to use a newer version of Swift.
2. **Binary framework & runtime compatibility** enables the distribution of frameworks in a binary form that works across multiple Swift versions. Binary frameworks include both a *Swift module file*, which communicates source-level information of the framework's API, and a *shared library*, which provides the compiled implementation that is loaded at runtime. Thus, there are two necessary goals for binary framework compatibility:
    - **Module format stability** stabilizes the module file, which is the compiler's representation of the public interfaces of a framework. This includes API declarations and inlineable code. The module file is used by the compiler for necessary tasks such as type checking and code generation when compiling client code using a framework.
    - **ABI stability** enables binary compatibility between applications and libraries compiled with different Swift versions. It is the focus of the rest of this document.

This document is an exploration and explanation of Swift's ABI alongside the goals and investigations needed before declaring Swift's ABI stable. It is meant to be a resource to the community as well as a declaration of the direction of Swift's ABI.

Throughout this document there will be references to issues in Swift's issue tracking system denoted by "SR-xxxx". These references track open engineering and design tasks for Swift's ABI.

What Is ABI?

At runtime, Swift program binaries interact with other libraries and components through an ABI. ABI is Application Binary Interface, or the specification to which independently compiled binary entities must conform to be linked together and executed. These binary entities must agree on many low level details: how to call functions, how their data is represented in memory, and even where their metadata is and how to access it.

ABI is per-platform, as it is a low level concern influenced by both the architecture and the OS. Most platform vendors define a "standard ABI" which is used for C code and built on by C-family languages. Swift, however, is a very different language from C and has its own per-platform ABI. While most of this document is platform-agnostic, platform-specific concerns have influenced details of the design and implementation of Swift's ABI. For details on each platform's standard ABI, refer to the Appendix.

What Is ABI Stability?

ABI stability means locking down the ABI to the point that future compiler versions can produce binaries conforming to the stable ABI. Once an ABI is stable, it tends to persist for the rest of the platform's lifetime due to ever-increasing mutual dependencies.

ABI stability only affects invariants of externally visible public interfaces and symbols. Internal symbols, conventions, and layout can continue to change without breaking the ABI. For example, future compilers are free to change the calling conventions for internal function calls so long as the public interfaces are preserved.

Decisions about the ABI will have long-term ramifications and may limit the ways in which the language can grow and evolve in the future. Future Swift versions can add new, orthogonal aspects to the ABI, but any inefficiencies or inflexibilities present when stability is declared will (effectively) persist forever for that platform.

ABI changes that are new and orthogonal are called *ABI-additive* changes. ABI-additive changes may be taken advantage of when the minimum targeted Swift version supports them. This allows us to extend or progressively lock down more of the ABI. These may be ABI additions to support new features or that allow for more efficient data access. Examples appear throughout this document.

What Does ABI Stability Enable?

ABI stability enables OS vendors to embed a Swift standard library and runtime that is compatible with applications built with older or newer versions of Swift. This would remove the need for apps to distribute their own copy of these libraries on those platforms. It also allows for better decoupling of tools and better integration into the OS.

As noted earlier, ABI stability is necessary, though not sufficient, for binary frameworks. Module format stability is also required and is beyond the scope of this document.

Library Evolution

Expressive and performance-focused languages which have binary interfaces tend to exhibit the fragile binary interface problem, which makes it difficult for any library or component to change over time without requiring every user to recompile with new versions of that library. A major push in Swift currently is the plan for Library Evolution, which aims to grant flexibility for library authors to maintain backwards and forwards binary compatibility. Many implementation concerns therein could have an impact on ABI.

One of the goals of rolling out ABI stability is to remain flexible enough to accommodate library evolution changes without limiting the design space. Library evolution concerns will be addressed in each individual section, though a common refrain will be that the details are still undecided.

Components of the Swift ABI

In practice, ABI concerns can be tightly coupled. But, as a conceptual model, I'd like to break them out into 6 separate classifications:

1. Types, such as structs and classes, must have a defined in-memory layout for instances of that type. For binary entities to interoperate, they must share the same layout conventions. This layout is discussed in the Data Layout section.

2. Type metadata is used extensively by Swift programs, the Swift runtime, reflection, and tools such as debuggers and visualizers. This metadata must either have a defined memory layout, or have a set of defined APIs for querying the metadata of a type. Type metadata is discussed in the Type Metadata section.

3. Every exported or external symbol in a library needs a unique name upon which binary entities can agree. Swift provides function overloading and contextual name spaces (such as modules and types), which means that any name in source code might not be globally unique. A unique name is produced through a technique called *name mangling*. Swift's name mangling scheme is discussed in the Mangling section.

4. Functions must know how to call each other, which entails such things as the layout of the call stack, what registers are preserved, and ownership conventions. Calling conventions are discussed in the Calling Convention section.

5. Swift ships with a runtime library which handles such things as dynamic casting, reference counting, reflection, etc. Compiled Swift programs make

external calls out to this runtime. Thus, Swift runtime API is Swift ABI. Runtime API stability is discussed in the Runtime section.

6. Swift ships with a standard library that defines many common types, structures, and operations on these. For a shipped standard library to work with applications written in different versions of Swift, it must expose a stable API. Thus, Swift Standard Library API is Swift ABI, as well as the layout of many of the types it defines. Swift standard library ABI stability concerns are discussed in the Standard Library section.

Data Layout

Background

First, let's define some common terminology.

- An *object* is a stored entity of some type, meaning it has a location in memory or in registers. Objects can be values of struct/enum type, class instances, references to class instances, values of protocol type, or even closures. This is in contrast to the class-based-OO definition of object as being an instance of a class.
- A *data member* of an object is any value that requires layout within the object itself. Data members include an object's stored properties and associated values.
- A *spare bit* is a bit that is unused by objects of a given type. These often arise due to things such as alignment, padding, and address spaces, further described below.
- An *extra inhabitant* is a bit pattern that does not represent a valid value for objects of a given type. For example, a simple C-like enum with 3 cases can fit in 2 bits, where it will have one extra inhabitant: the fourth unused bit pattern.

Data layout, also known as type layout, specifies the in-memory layout of an object's data. This includes the size of an object in memory, the alignment of an object (defined later), and how to find each data member within an object.

An object has a statically known layout if the compiler is able to determine its layout at compilation time. Objects whose layout is not determinable until runtime have *opaque layout*. Such objects are further discussed in the opaque layout section

Layout and Properties of Types For every type `T` in Swift with statically known layout, the ABI specifies a means to determine:

- The *alignment* for that type: for `x : T`, the address of `x` modulo alignment is always zero.
- The *size* for that type: the byte size (possibly 0) without padding at the end.

- The *offset* for each data member (if applicable): the address at which every member resides, relative to the object's base address.

Derived from alignment and size is the *stride* of the type, which is the size of objects of that type rounded up to alignment (minimum 1). The stride is mostly useful for objects laid out contiguously in memory, such as in arrays.

Some types have interesting properties:

- A type is *trivial*, also known as POD ("plain ol' data"), if it merely stores data and has no extra copy, move, or destruction semantics. Trivial objects can be copied by replicating their bits, and are destroyed through deallocation. A type is trivial only if all data members are also trivial.
- A type is *bitwise movable* if there are no side table references dependent on its address. A move operation can occur when an object is copied from one location into another and the original location is no longer used. Bitwise movable objects are moved by performing a bitwise copy and then invalidating the original location. A type is bitwise movable only if all its data members are also bitwise movable. All trivial types are bitwise movable.

An example of a trivial type is a Point struct that contains two Double fields: an x coordinate and a y coordinate. This struct is trivial, as it can be copied merely by copying its bits and its destruction performs no extra operations.

An example of a bitwise movable, but non-trivial, type is a struct that contains a reference to a class instance. Objects of that type cannot be copied merely by copying their bits, because a retain operation must be performed on the reference. Upon destruction, such objects must perform a release. However, the object can be moved from one address to another by copying its bits provided the original location is invalidated, keeping the overall retain count unchanged.

An example of a type that is neither trivial nor bitwise movable is a struct containing a weak reference. Weak references are tracked in a side table so that they can be nil-ed out when the referenced object is destroyed. When moving an object of such type from one address to another, the side table must be updated to refer to the weak reference's new address.

Opaque Layout Opaque layout occurs whenever the layout is not known until runtime. This can come up for unspecialized generics, which do not have a known layout at compilation time. It can also come up for resilient types, which are described in the next section.

The size and alignment of an object of opaque layout, as well as whether it is trivial or bitwise movable, is determined by querying its value witness table, which is described further in the value witness table section. The offsets for data members are determined by querying the type's metadata, which is described further in the value metadata section. Objects of opaque layout must typically be passed indirectly, described further in the function signature lowering section.

The Swift runtime interacts with objects of opaque layout through pointers, and thus they must be addressable, described further in the abstraction levels section.

In practice, layout might be partially-known at compilation time. An example is a generic struct over type `T` that stores an integer as well as an object of type `T`. In this case, the layout of the integer itself is known and its location within the generic struct might be as well, depending on the specifics of the layout algorithm. However, the generic stored property has opaque layout, and thus the struct overall has an unknown size and alignment. We are investigating how to most efficiently lay out partially-opaque aggregates [SR-3722]. This will likely entail placing the opaque members at the end in order to guarantee known offsets of non-opaque data members.

Library Evolution Library evolution introduces *resilient* layouts of public types by default and provides new annotations that freeze the layout for performance. A resilient layout avoids many of the pitfalls of the fragile binary problem by making the layout opaque. Resilient types have far more freedom to change and evolve without breaking binary compatibility: public data members can be rearranged, added, and even removed (by providing a computed getter/setter instead). The new annotations provide the ability to relinquish these freedoms by making stricter guarantees about their layout in order to be more efficiently compiled and accessed.

In order to allow for cross-module optimizations for modules that are distributed together, there is the concept of a *resilience domain*. A resilience domain is a grouping of modules which are version-locked with each other and thus do not have binary compatibility across multiple version requirements with each other. See Resilience Domains for more details.

Resilient types are required to have opaque layout when exposed outside their resilience domain. Inside a resilience domain, this requirement is lifted and their layout may be statically known or opaque as determined by their type (see previous section).

Annotations may be applied to a library's types in future versions of that library, in which case the annotations are versioned, yet the library remains binary compatible. How this will impact the ABI is still under investigation [SR-3911].

Abstraction Levels All types in Swift conceptually exist at multiple levels of abstraction. For example, an `Int` value is of a concrete type and can be passed to functions in registers. But, that same value might be passed to a function expecting a generic type `T`, which has opaque layout. Since the function is expecting its argument to be passed indirectly, the integer value must be promoted to the stack. When that value has type `T`, it is said to be at a higher abstraction level than when it was an integer. Moving between abstraction levels is done through a process called *reabstraction*.

For many types in Swift, reabstraction involves directly copying the value to memory so that it is addressable. Reabstraction may be more complicated for tuples and higher-order functions, explained later in the tuples layout section and the function signature lowering section.

A Tour of Types

What follows is a breakdown of the different kinds of types in Swift and what needs to be specified.

Structs The layout algorithm for structs should result in an efficient use of space, possibly by laying out fields in a different order than declared [SR-3723]. We may want a fully declaration-order-agnostic algorithm to allow data members to be reordered in source without breaking binary compatibility [SR-3724]. We also need to consider whether, by default, we want to ensure struct data members are addressable (i.e. byte-aligned) or if we'd rather do bit-packing to save space [SR-3725].

Zero sized structs do not take up any space as data members and struct members may be laid out in the padding of sub-structs. We may want to explore whether there are implementation benefits to capping alignment at some number, e.g. 16 on many platforms [SR-3912].

Tuples Tuples are similar to anonymous structs, but they differ in that they exhibit structural subtyping: a tuple of type e.g. `(Bool, Bool)` can be passed anywhere expecting generic types `(T, U)`. But, the type `(T, U)` exists at a higher abstraction level than `(Bool, Bool)`. Due to this, tuples may face more expensive reabstraction costs if their layout is aggressively packed. Reabstracting such a tuple would involve splitting and promoting each element into their own addresses.

This may be an argument for a simple, declaration-order, non bit-packed layout algorithm for tuples. Tuples are often used for small local values and rarely persisted across ABI boundaries in a way that aggressive packing is performance-critical. This would also be more consistent with how fixed-size C arrays are presented in Swift, which are imported as tuples.

We should investigate whether to aggressively bit-pack tuple elements similarly to structs, paying the reabstraction costs, or if the benefits are not worth the costs [SR-3726].

Tuples should be binary compatible between labeled and unlabeled tuples of the same type and structure.

Enums A value of enum type exists as one of many variants or cases. Determining which is the job of the *discriminator*, also known as a tag, which is an integer value denoting which case is presently stored. To save space, discriminators can be put in spare bits or be represented by extra inhabitants.

`@closed` enums, that is enums that can't have cases added to them later, can be classified into the following:

- Degenerate - zero cased, or single cased without an associated value
- Trivial - no associated values
- Single payload - an enum where only one case has associated values
- Multi-payload - an enum that has multiple cases with associated values

Degenerate enums take zero space. Trivial enums are just their discriminator.

Single payload enums try to fit their discriminator in the payload's extra inhabitants for the non-payload cases, otherwise they will store the discriminator after the payload. When the discriminator is stored after the payload, the bits are not set for the payload case. The payload is guaranteed to be layout compatible with the enum as the payload case does not use any extra inhabitants. Storing the discriminator after the payload may also result in more efficient layout of aggregates containing the enum, due to alignment.

The layout algorithm for multi-payload enums is more complicated and still needs to be developed [SR-3727]. The algorithm should try to rearrange payloads so as to coalesce cases and save space. This rearrangement can also improve performance and code size. For example, if ARC-ed payload components reside in the same location, operations like copy can be done directly on the values without extensive switching.

Enum raw values are not ABI, as they are implemented as code present in the computed property getter and setter. `@objc` enums are C-compatible, which means they must be trivial.

Library evolution adds the notion of `@open` enums (which will also be resilient), which allow library owners to add new cases and reorder existing cases without breaking binary compatibility. How this is accomplished is still to be determined.

Classes There are two constructs present when discussing about class layout: *class instances*, which reside on the heap, and *references* to class instances, which are reference-counted pointers.

Class Instances The layout of class instances is mostly opaque. This is to avoid the vexing problem of fragile binary interfaces, also known as the "fragile base class problem", in which seemingly minor changes to a base class break binary compatibility with subclasses.

The run-time type of a non-final class instance or a class existential is not known statically. To facilitate dynamic casts, the object must store a pointer to its type, called the *isa* pointer. The *isa* pointer is always stored at offset 0 within the object. How that type is represented and what information it provides is part of the class's metadata and is covered in the class metadata section. Similarly, the function for a non-final method call is also not known statically and is dispatched

based on the run-time type. Method dispatch is covered in the method dispatch section.

Class instances will, as part of ABI-stability, guarantee a word-sized field of opaque data following the isa field that may be used for reference counting by the runtime [SR-4353]. But, the format and conventions of this opaque data will not be ABI at first in order to have more flexibility for language or implementation changes. Instead, runtime functions provide the means to interact with reference counts. This opaque data and its conventions may be locked down for more efficient access in the future, which will be an ABI-additive change.

References Classes are reference types. This means that Swift code dealing with class instances does so through references, which are pointers at the binary level. These references participate in automatic reference counting (ARC).

References to Objective-C-compatible class instances (i.e. those that inherit from an Objective-C class or are imported from Objective-C) must provide the same bit-level guarantees to the Objective-C runtime as Objective-C references. Thus, such references are opaque: they have no guarantees other than that nil is 0 and provide no extra inhabitants.

References to native, non-Objective-C-compatible Swift class instances do not have this constraint. The alignment of native Swift class instances is part of ABI, providing spare bits in the lower bits of references. Platforms may also provide spare bits (typically upper bits) and extra inhabitants (typically lower addresses) for references due to limited address spaces.

We may want to explore using spare bits in references to store local reference counts in order to perform some ARC operations more efficiently [SR-3728]. These would need to be flushed to the object whenever a reference may escape or the local reference count reaches zero. If these local reference counts can cross ABI boundaries, then such a change will have to be implemented in an ABI-additive way with deployment target checking.

Existential Containers Any discussion of existentials quickly becomes bogged down in obscure terminology, so let's first establish some background surrounding the terms *existential values*, *existential containers*, and *witness tables*.

In type theory, an existential type describes an interface of an abstract type. Values of an existential type are *existential values*. These arise in Swift when an object's type is a protocol: storing or passing an object of protocol type means that the actual run-time type is opaque (not known at compile time, and thus neither is its layout). But, that opaque type has known interfaces because that type conforms to the protocol.

A type's conformance to a protocol consists of functions (whether methods or getters and setters), but the specific addresses of those functions are not known at compilation time for existential values as their actual type is not known

until run time. This is a similar situation as with references to non-final class instances, and is solved using a similar technique. *Witness tables* are tables of function pointers implementing a protocol conformance and are further discussed in the witness table section.

Existential containers store values of protocol or protocol composition type alongside corresponding witness tables for each protocol conformance. For existentials that are not class-constrained (may be value types or classes), the container needs to store:

- the value itself: either in an inline buffer or as a pointer to out-of-line storage
- a pointer to the type metadata
- a witness table pointer for every conformance.

Class-constrained existentials omit the metadata pointer (as the object itself contains a pointer to its type), as well as any excess inline buffer space. `Any`, which is an existential value without any conformances, has no witness table pointer.

We are re-evaluating the inline buffer size for existential containers prior to ABI stability [SR-3340]. We are also considering making the out-of-line allocation be copy-on-write (COW) [SR-4330]. We should also explore "exploding" existential parameters, i.e. converting an existential parameter into a protocol-constrained generic parameter [SR-4331].

Declaring Stability

ABI stability means nailing down type layout and making decisions about how to handle the concerns of Library Evolution. The end result will be a technical specification of the layout algorithms that future compilers must adhere to in order to ensure binary compatibility [SR-3730].

For all of the areas discussed above, more aggressive layout improvements may be invented in the post-ABI stability future. For example, we may want to explore rearranging and packing nested type data members with outer type data members. Such improvements would have to be done in an ABI-additive fashion through deployment target and/or min-version checking. This may mean that the module file will need to track per-type ABI versioning information.

A potentially out of date description of Swift's current type layout can be found in the Type Layout docs.

Type Metadata

While data layout specifies the layout of objects of a given type, *type metadata* holds information about the types themselves. The information available and how to access this information is part of Swift ABI.

Swift keeps metadata records for every *concrete type*. Concrete types include all non-generic types as well as generic types with concrete type parameters. These records are created by the compiler as well as lazily created at run time (e.g. for generic type instantiations). This metadata stores information about its type, discussed in each section below.

A potential approach to stability mechanism is to provide metadata read/write functions alongside the runtime to interact with metadata, giving some freedom to the underlying structures to grow and change. This effectively makes large portions of metadata opaque. But, certain fields require access to be as efficient as possible (e.g. dynamic casts, calling into witness tables) and the performance hit from going through an intermediary function would be unacceptable. Thus, we will probably freeze the performance-critical parts and use accessor functions for the rest [SR-3923].

Metadata has many historical artifacts in its representation that we want to clean up [SR-3924]. We also want to make small tweaks to present more semantic information in the metadata, to enable better future tooling and features such as reflection [SR-3925]. Some of these need to be done before declaring ABI stability and some may be additive.

Declaring Stability Stabilizing the ABI means producing a precise technical specification for the fixed part of the metadata layout of all language constructs so that future compilers and tools can continue to read and write them. A prose description is not necessarily needed, though explanations are useful. We will also want to carve out extra space for areas where it is likely to be needed for future functionality [SR-3731].

For more, but potentially out of date, details see the Type Metadata docs.

Generic Parameters

Swift has a powerful generics system, which shows up both at compilation time (through specialization optimizations) and at run time when the type is unknown. Swift types may be parameterized over generic types, and thus every type's metadata describes whether generic type parameters are present and if so provides information about them.

At run time, objects only have concrete types. If the type in source code is generic, the concrete type is an instantiation of that generic type. Generic instantiation metadata provide type metadata for each generic type parameter. If the generic type is constrained, corresponding witness tables for each protocol conformance are also provided in the metadata.

Value Metadata

Named value types store the type name (currently mangled but we are investigating un-mangled [SR-3926]) and a pointer to the type's parent for nested

types.

Value type metadata also has kind-specific entries. Struct metadata stores information about its fields, field offsets, field names, and field metadata. Enum metadata stores information about its cases, payload sizes, and payload metadata. Tuple metadata stores information about its elements and labels.

Value Witness Tables Every concrete type has a *value witness table* that provides information about how to lay out and manipulate values of that type. When a value type has opaque layout, the actual layout and properties of that value type are not known at compilation time, so the value witness table is consulted.

The value witness table stores whether a type is trivial and/or bitwise movable, whether there are extra inhabitants and if so how to store and retrieve them, etc. For enums, the value witness table will also provide functionality for interacting with the discriminator. There may be more efficient ways of representing enums that simplify this functionality (or provide a fast path), and that's under investigation [SR-4332].

These value witness tables may be constructed statically for known values or dynamically for some generic values. While every unique type in Swift has a unique metadata pointer, value witness tables can be shared by types so long as the information provided is identical (i.e. same layout). Value witness tables always represent a type at its highest abstraction level. The value witness table entries and structure need to be locked down for ABI stability [SR-3927].

Class Metadata

Swift class metadata is layout-compatible with Objective-C class objects on Apple's platforms, which places requirements on the contents of the first section of class metadata. In this first section, entries such as super class pointers, instance size, instance alignment, flags, and opaque data for the Objective-C runtime are stored.

Following that are superclass members, parent type metadata, generic parameter metadata, class members, and *vtables*, described below. Library evolution may present many changes to what exactly is present and will likely make many of the contents opaque to accommodate changes [SR-4343].

Method Dispatch Invoking a non-final instance method involves calling a function that is not known at compile time: it must be resolved at run time. This is solved through the use of a *vtable*, or virtual method table (so called because overridable methods are also known as "virtual" methods). A *vtable* is a table of function pointers to a class or subclass's implementation of overridable methods. If the vtable is determined to be part of ABI, it needs a layout algorithm that also provides flexibility for library evolution.

Alternatively, we may decide to perform inter-module calls through opaque *thunks*, or compiler-created intermediary functions, which then perform either direct or vtable dispatch as needed [SR-3928]. This enables greater library evolution without breaking binary compatibility by allowing internal class hierarchies to change. This would also unify non-final method dispatch between open and non-open classes while still allowing for aggressive compiler optimizations like de-virtualization for non-open classes. This approach would make vtables not be ABI, as that part of the type metadata would effectively be opaque to another module.

Protocol and Existential Metadata

Protocol Witness Tables The protocol witness table is a function table of a type's conformance to the protocol's interfaces. If the protocol also has an associated type requirement, then the witness table will store the metadata for the associated type. Protocol witness tables are used with existential containers where the run time type is not known.

Protocol witness tables may be created dynamically by the runtime or statically by the compiler. The layout of a protocol witness table is ABI and we need to determine a layout algorithm that also accommodates library evolution concerns, where additional protocol requirements may be added with default fall-backs [SR-3732].

Existential Metadata Existential type metadata contains the number of witness tables present, whether the type is class-constrained, and a *protocol descriptor* for each protocol constraint. A protocol descriptor describes an individual protocol constraint, such as whether it is class-constrained, the size of conforming witness tables, and protocol descriptors for any protocols it refines. Protocol descriptors are layout compatible with the Objective-C runtime's protocol records on Apple platforms. The format of the existential type metadata needs to be reviewed as part of the ABI definition [SR-4341].

Function Metadata

In addition to common metadata entries, function type metadata stores information about the function signature: parameter and result type metadata, calling convention, per-parameter ownership conventions, and whether the function throws. Function type metadata always represents the function at its highest abstraction level, which is explained later in the function signature lowering section. Function parameters are currently modeled with a tuple-based design, but this should be updated to match modern Swift [SR-4333]. As more ownership semantics are modeled, more information may be stored about each parameter.

Mangling

Mangling is used to produce unique symbols. It applies to both external (public) symbols as well as internal or hidden symbols. Only the mangling scheme for external symbols is part of ABI.

ABI stability means a stable mangling scheme, fully specified so that future compilers and tools can honor it. For a potentially out-of-date specification of what the mangling currently looks like, see the Name Mangling docs.

There are some corner cases currently in the mangling scheme that should be fixed before declaring ABI stability. We need to come up with a canonicalization of generic and protocol requirements to allow for order-agnostic mangling [SR-3733]. We also may decide to more carefully mangle variadicity of function parameters, etc [SR-3734]. Most often, though, mangling improvements focus on reducing symbol size.

Mangling design centers around coming up with short and efficient manglings that still retain important properties such as uniqueness and integration with existing tools and formats. Given the prevalence of public symbols in libraries and frameworks, and debugging symbols in applications, the symbol names themselves can make up a significant portion of binary size. Reducing this impact is a major focus of stabilizing the mangling. Post-ABI-stability, any new manglings or techniques must be additive and must support the existing manglings.

There are many ways to improve the existing mangling without major impact on existing tools. Throughout these endeavors, we will be empirically measuring and tracking symbol size and its impact on binary size. ABI work on mangling focuses on producing *compact manglings* and using *suffix differentiation*.

Compact Manglings

Minor tweaks to shorten the mangling can have a beneficial impact on all Swift program binary sizes. These tweaks should compact existing manglings while preserving a simple unique mapping. One example is not distinguishing between struct/enum in mangling structures, which would also provide more library evolution freedom [SR-3930]. We are considering dropping some internal witness table symbols when they don't provide any meaningful information conducive to debugging [SR-3931]. We recently overhauled word substitutions in mangling, with the goal of reducing as much redundancy in names as possible [SR-4344].

There are other aggressive directions to investigate as well, such as mangling based on a known overload set for non-resilient functions. This does have the downside of making manglings unstable when new overloads are added, so its benefits would have to be carefully weighed [SR-3933].

Any more ambitious reimagining of how to store symbols such as aggressive whole-library symbol name compression would have to be done in tight coupling

with existing low level tools. Unfortunately, this might make some of the more ambitious options infeasible in time for ABI stability. They could be rolled out as ABI-additive using deployment target checking in the future.

Suffix Differentiation

There are many existing low level tools and formats that store and consume the symbol information, and some of them use efficient storage techniques such as tries. Suffix differentiation is about adjusting the mangling in ways that take advantage of them: by distinguishing manglings through suffixes, i.e. having common shared prefixes. This is currently underway and is resulting in binary size reductions for platforms that use these techniques [SR-3932].

Calling Convention

For the purposes of this document, "standard calling convention" refers to the C calling convention for a given platform (see appendix), and "Swift calling convention" refers to the calling convention used by Swift code when calling other Swift code. One of the first steps toward ABI stability is for Swift to adopt the Swift calling convention [SR-4346]. The Swift runtime uses the standard calling convention, though it may make alterations (see section Runtime calling convention).

Calling convention stability pertains to public interfaces. The Swift compiler is free to choose any convention for internal (intra-module) functions and calls.

For rationale and potentially-out-of-date details, see the Swift Calling Convention Whitepaper. As part of nailing down the calling conventions, that document will either be updated with the final specifications of the calling conventions or else moved to a rationale document and a more succinct and rigorous specification put in its place.

Register convention

This section will be using the terms *callee-saved* and *scratch* to classify registers as part of a register convention.

- A *callee-saved register* must be preserved over the duration of a function call. If a called function (the *callee*) wishes to change the value stored in the register, it must restore it before returning.
- A *scratch* register, also known as caller-saved or callee-clobbered, is not preserved over the duration of a function call. If the register's value must be preserved, code surrounding a function call must save and restore the value.

Swift uses roughly the same categorization of registers as the standard calling convention. But, for some platforms, the Swift calling convention adds additional

situational uses of some callee-saved registers: the *call context* register and the *error* register.

Call Context Register The value held in the *call context* register depends on the kind of function called:

- Instance methods on class types: pointer to self
- Class methods: pointer to type metadata (which may be subclass metadata)
- Mutating method on value types: pointer to the value (i.e. value is passed indirectly)
- Non-mutating methods on value types: self may fit in one or more registers, else passed indirectly
- *Thick closures*, i.e. closures requiring a context: the closure context

Having the call context register be callee-saved is advantageous. It keeps the register stable across calls, where the context is very likely to be used and reused in subsequent or nested calls. Additionally, this makes partial application free as well as converting thin closures to thick.

Error Register Throwing functions communicate error values to their callers through the *error* register on some platforms. The error register holds a pointer to the error value if an error occurred, otherwise 0. The caller of a throwing function is expected to quickly check for 0 before continuing on with non-error code, otherwise branching to code to handle or propagate the error. Using a callee-saved register for the error register enables free conversion from non-throwing to throwing functions, which is required to honor the subtyping relationship.

The specific registers used in these roles are documented in the calling convention summary document.

Function Signature Lowering

Function signature lowering is the mapping of a function's source-language type, which includes formal parameters and results, all the way down to a physical convention, which dictates what values are stored in what registers and what values to pass on the stack.

ABI stability requires nailing down and fully specifying this algorithm so that future Swift versions can lower Swift types to the same physical call signature as prior Swift versions [SR-4349]. More in-depth descriptions and rationale of function signature lowering can be found in the function signature lowering docs.

Lowering the result value is usually done first, with a certain number of registers designated to hold the result value if it fits, otherwise the result value is passed on the stack. A good heuristic is needed for the limit and is architecture specific (e.g. 4 registers on modern 64-bit architectures) [SR-3946].

Next comes lowering parameters, which proceeds greedily by trying to fit values into registers from left-to-right, though some parameters may be re-ordered.

For example, closures are best placed at the end to take advantage of ABI compatibility between thick closures and thin ones without a context.

Some values must be passed and returned indirectly as they are *address only*. Address only values include non-bitwise-copyable values, values with opaque layout, and non-class-constrained existential values. Even if the runtime type would normally be passed in a register, or even if the type is statically known at the call-site, if the callee receives or returns values with opaque layout, they must be passed or returned indirectly.

We should investigate whether it makes sense to split values with partially opaque layout by passing the non-opaque parts in registers [SR-3947].

Parameter ownership is not reflected in the physical calling convention, though it will be noted in the mangling of the function name. Default argument expressions will not be ABI, as they will be emitted into the caller. This means that a library can add, modify, or remove default argument expressions without breaking binary compatibility (though modifying/removing may break source compatibility).

Lowering Higher-Order Functions Passing or returning higher-order functions may involve undergoing reabstraction, which requires that the compiler creates a thunk mapping between the actual calling convention and the expected calling convention.

For example, let's say there are two functions:

```
func add1(_ i: Int) -> Int { return i+1 }
func apply<T,U>(_ f: (T) -> U, _ x: T) -> U { return f(x) }
```

`apply`'s function parameter `f` must take and return its values indirectly, as `T` and `U` have opaque layout. If `add1` is passed to `apply`, the compiler will create a thunk for `apply` to call that takes a parameter indirectly and calls `add1` by passing it in register. The thunk will then receive the result in register and return it indirectly back to `apply`.

Stack Invariants

Calling conventions include invariants about the call stack, such as stack alignment. Unless there is a very compelling reason to deviate, Swift should just honor the stack invariants of the standard calling convention. This is because Swift functions may share their call stack with non-Swift code. For example a Swift function that calls an Objective-C function, which in turn calls another Swift function, would want to maintain the proper stack alignment (and any other stack invariants) for all calls. This is far simpler if they both honor the same invariants.

Runtime Calling Convention

The Swift runtime uses the standard calling convention, though it may evolve to preserve more invariants. It's likely beneficial to introduce one or a number of tweaks to the scratch register sets of some runtime functions. Swift code that makes a call into the runtime assumes some registers are scratch, i.e. clobbered by the runtime function. But, some runtime functions may not need as many scratch registers and can guarantee more registers as callee-saved. Every formerly-scratch register that is denoted callee-saved (i.e. runtime function saved) relieves the register pressure of the surrounding Swift code making the runtime call.

Such changes to runtime functions can be rolled out incrementally in the future, and they are backwards compatible so long as no version of that function ever clobbers the now-saved registers. But, such a change is ratcheting, that is every register that is changed to be runtime-saved can no longer go back to being scratch without breaking binary compatibility. If the reduced scratch register set causes the runtime function to spill, then the whole exercise was pointless and actively harmful. Great care should be taken and testing applied for any change to ensure that the runtime function never spills in the future.

Runtime

Swift exposes a runtime that provides APIs for compiled code. Calls into the Swift runtime are produced by the compiler for concerns such as memory management and run-time type information. Additionally, the runtime exposes low-level reflection APIs that are useful to the standard library and some users.

Every existing runtime function will need to be audited for its desirability and behavior [SR-3735]. For every function, we need to evaluate whether we want the API as is:

- If yes, then we need to precisely specify the semantics and guarantees of the API.
- If not, we need to either change, remove, or replace the API, and precisely specify the new semantics.

The runtime is also responsible for lazily creating new type metadata entries at run time, either for generic type instantiations or for resilient constructs. Library evolution in general introduces a whole new category of needs from the runtime by making data and metadata more opaque, requiring interaction to be done through runtime APIs. Additionally, ownership semantics may require new runtime APIs or modifications to existing APIs. These new runtime needs are still under investigation [SR-4352].

There are many potential future directions to open up the ABI and operate on less-opaque data directly, as well as techniques such as call-site caching. These are ABI-additive, and will be interesting to explore in the future.

For a potentially-out-of-date listing of runtime symbols and some details, see

the Runtime docs.

Standard Library

Any standard library API shipped post-ABI-stability must be supported into the future to ensure binary compatibility. The standard library will also be utilizing resilience annotations and *inlineable* code. Inlineable code is code that is bundled with the client's code, and is available for inlining to the optimizer if it decides to do so. The standard library faces the following (non-exhaustive) list of challenges for ensuring binary compatibility:

- Published public functions and types cannot be removed or changed in ways that break binary compatibility.
- Choosing what code to make inlineable will affect performance and flexibility.
- Internal functions called by inlineable code become ABI, and are subject to the same binary compatibility concerns as public functions.
- Non-resilient types cannot change their layout.
- Protocols cannot add new requirements.

Inlineability

Inlineable code that calls internal functions makes those internal functions ABI, as the client code will be making external calls to them. Thus, many internal interfaces in the standard library will need to be locked down if called from inlineable code. Whether to mark code inlineable will have to carefully weigh performance requirements against keeping flexibility for future changes.

This tradeoff between performance and flexibility also affects the ability to deploy bug fixes and performance improvements to users. Users that have inlined code from the standard library will not be able to get bug fixes and performance improvements in an OS update without performing a recompilation with the new library. For more information on this topic, see Inlineable Functions.

Upcoming Changes

While the standard library is already ensuring source stability, it will be changing many of its fundamental underlying representations this year. When ABI stability lands, the standard library will be severely limited in the kinds of changes it can make to existing APIs and non-resilient types. Getting the standard library in the right place is of critical importance.

The programming model for String is still being redesigned [SR-4354], and many types such as Int are undergoing implementation changes [SR-3196]. At the same time, the standard library is simultaneously switching to new compiler features such as conditional conformances to clean up and deliver the best APIs [SR-3458].

Another goal of Swift is to improve the applicability of Swift to systems programming. Ownership semantics may make a large impact, including things such as improved `inout` semantics that allow for efficient and safe array slicing. Providing the right abstractions for efficient use of contiguous memory is still under investigation [SR-4355].

Next Steps

All progress and issue tracking will be done through JIRA on bugs.swift.org, using the "AffectsABI" label. We will make an ABI stability dashboard to more easily monitor specifics and progress. The next step is to start making issues for everything that needs fixing and issues for the directions we want to explore.

This document will be a living document until ABI stability is reached, updated with new findings and JIRA issues as they come up. After ABI stability is achieved, this document should be succeeded by technical specifications of Swift's ABI.

Issue tracking alone doesn't effectively communicate the overall progress and when ABI stability can be expected to land. Some issues take longer than others and there isn't a good indication of how long the known tasks will take, nor of how many unknown issues are yet to be filed. For that, a higher level view of the overall status will be provided, possibly on swift.org.

Appendix

Standard ABIs

Apple ARM64 iOS platform ABI is a vendor-specific variant of ARM's AAPCS64.

Apple ARM32 iOS platform ABI is similarly a variant of ARM's AAPCS.

Apple x86-64 MacOS platform ABI is based off of the generic System V ABI also used by BSD and Linux.

Apple i386 MacOS platform ABI is similarly based off of the generic i386 System V ABI.