Key Path Memory Layout

Key path objects are laid out at runtime as a heap object with a variable-sized payload containing a sequence of encoded components describing how the key path traverses a value. When the compiler sees a key path literal, it generates a **key path pattern** that can be efficiently interpreted by the runtime to instantiate a key path object when needed. This document describes the layout of both. The key path pattern layout is designed in such a way that it can be transformed in-place into a key path object with a one-time initialization in the common case where the entire path is fully specialized and crosses no resilience boundaries.

ABI Concerns For Key Paths

For completeness, this document describes the layout of both key path objects and patterns; note however that the instantiated runtime layout of key path objects is an implementation detail of the Swift runtime, and *only key path patterns* are strictly ABI, since they are emitted by the compiler. The runtime has the freedom to change the runtime layout of key path objects, but will have to maintain the ability to instantiate from key path patterns emitted by previous ABI-stable versions of the Swift compiler.

Key Path Objects

Buffer Header

Key path objects begin with the standard Swift heap object header, followed by a key path object header. Relative to the end of the heap object header:

Offset	Description
`0`	Pointer to KVC compatibility C string, or null
`1*sizeof(Int)`	Key path buffer header (32 bits)

If the key path is Cocoa KVC-compatible, the first word will be a pointer to the equivalent KVC string as a null-terminated UTF-8 C string. It will be null otherwise. The **key path buffer header** in the second word contains the following bit fields:

Bits (LSB zero)	Description
0...23	**Buffer size** in bytes
24...29	Reserved. Must be zero in Swift 4...5 runtime
30	1 = Has **reference prefix**, 0 = No reference prefix
31	1 = Is **trivial**, 0 = Has destructor

The *buffer size* indicates the total size in bytes of the components following the key path buffer header. A `ReferenceWritableKeyPath` may have a *reference prefix* of read-only components that can be projected before initiating mutation; bit 30 is set if one is present. A key path may capture values that require cleanup when the key path object is deallocated, but a key path that does not capture any values with cleanups will have the *trivial* bit 31 set to fast-path deallocation.

Components are always pointer-aligned, so the first component always starts at offset `2*sizeof(Int)`. On 64-bit platforms, this leaves four bytes of padding.

Components

After the buffer header, one or more **key path components** appear in memory in sequence. Each component begins with a 32-bit **key path component header** describing the following component.

Bits (LSB zero)	Description
0...23	**Payload** (meaning is dependent on component kind)
24...30	**Component kind**
31	1 = **End of reference prefix**, 0 = Not end of reference prefix

If the key path has a *reference prefix*, then exactly one component must have the *end of reference prefix* bit set in its component header. This indicates that the component after the end of the reference prefix will initiate mutation.

The following *component kinds* are recognized:

Value in bits 24...30	Description
0	Struct/tuple/self stored property
1	Computed
2	Class stored property
3	Optional chaining/forcing/wrapping

- A **struct stored property** component, when given a value of the base type in memory, can project the component value in-place at a fixed offset within the base value. This applies for struct stored properties, tuple fields, and the `.self` identity component (which trivially projects at offset zero). The *payload* contains the offset in bytes of the projected field in the aggregate, or the special value `0xFF_FFFF`, which indicates that the offset is too large to pack into the payload and is stored in the next 32 bits after the header.
- A **class stored property** component, when given a reference to a class instance, can project the component value inside the class instance at a fixed offset. The *payload payload* contains the offset in bytes of the projected field from the address point of the object, or the special value `0xFF_FFFF`, which indicates that the offset is too large to pack into the payload and is stored in the next 32 bits after the header.
- An **optional** component performs an operation involving `Optional` values. The `payload` contains one of the following values:

Value in payload	Description
0	**Optional chaining**
1	**Optional wrapping**
2	**Optional force-unwrapping**

  A *chaining* component behaves like the postfix `?` operator, immediately ending the key path application and returning nil when the base value is nil, or unwrapping the base value and continuing projection on the non-optional payload when non-nil. If an optional chain ends in a non-optional value, an implicit *wrapping*

component is inserted to wrap it up in an optional value. A *force-unwrapping* operator behaves like the postfix `!` operator, trapping if the base value is nil, or unwrapping the value inside the optional if not.

- A **computed** component uses the conservative access pattern of `get` / `set` / `materializeForSet` to project from the base value. This is used as a general fallback component for any key path component without a more specialized representation, including not only computed properties but also subscripts, stored properties that require reabstraction, properties with behaviors or custom key path components (when we get those), and weak or unowned properties. The payload contains additional bitfields describing the component:

Bits (LSB zero)	Description
24	1 = **Has captured arguments**, 0 = no captures
25...26	**Identifier kind**
27	1 = **Settable**, 0 = **Get-Only**
28	1 = **Mutating** (implies settable), 0 = Nonmutating

The component can *capture* context which is stored after the component in the key path object, such as generic arguments from its original context, subscript index arguments, and so on. Bit 24 is set if there are any such captures. Bits 25 and 26 discriminate the *identifier* which is used to determine equality of key paths referring to the same components. If bit 27 is set, then the key path is **settable** and can be written through, and bit 28 indicates whether the set operation **is mutating** to the base value, that is, whether setting through the component changes the base value like a value-semantics property or modifies state indirectly like a class property or `UnsafePointer.pointee`.

After the header, the component contains the following word-aligned fields:

Offset from header	Description
`1*sizeof(Int)`	The **identifier** of the component.
`2*sizeof(Int)`	The **getter function** for the component.
`3*sizeof(Int)`	(if settable) The **setter function** for the component

The combination of the identifier kind bits and the identifier word are compared by the `==` operation on two key paths to determine whether they are equivalent. Neither the kind bits nor the identifier word have any stable semantic meaning other than as unique identifiers. In practice, the compiler picks a stable unique artifact of the underlying declaration, such as the naturally-abstracted getter entry point for a computed property, the offset of a reabstracted stored property, or an Objective-C selector for an imported ObjC property, to identify the component. The identifier kind bits are used to discriminate possibly-overlapping domains.

The getter function is a pointer to a Swift function with the signature `@convention(thin) (@in Base, UnsafeRawPointer) -> @out Value`. When the component is applied, the getter is invoked with a copy of the base value and is passed a pointer to the captured arguments of the component. If the component has no captures, the second argument is undefined.

The setter function is also a pointer to a Swift function. This field is only present if the *settable* bit of the header is set. If the component is nonmutating, then the function has signature `@convention(thin) (@in Base, @in Value, UnsafeRawPointer) -> ()`, or if it is mutating, then the function has

signature `@convention(thin) (@inout Base, @in Value, UnsafeRawPointer) -> ()`. When a mutating application of the key path is completed, the setter is invoked with a copy of the base value (if nonmutating) or a reference to the base value (if mutating), along with a copy of the updated component value, and a pointer to the captured arguments of the component. If the component has no captures, the third argument is undefined.

TODO: Make getter/nonmutating setter take base borrowed, yield borrowed result (materializeForGet); use materializeForSet

If the component has captures, the capture area appears after the other fields, at offset `3*sizeof(Int)` for a get-only component or `4*sizeof(Int)` for a settable component. The area begins with a two-word header:

Offset from start	Description
`0`	Size of captures in bytes
`1*sizeof(Int)`	Pointer to **argument witness table**

followed by the captures themselves. The *argument witness table* contains pointers to functions needed for maintaining the captures:

Offset	Description
`0`	**Destroy**, or null if trivial
`1*sizeof(Int)`	**Copy**
`2*sizeof(Int)`	**Is Equal**
`3*sizeof(Int)`	**Hash**

The *destroy* function, if not null, has signature `@convention(thin) (UnsafeMutableRawPointer) -> ()` and is invoked to destroy the captures when the key path object is deallocated.

The *copy* function has signature `@convention(thin) (_ src: UnsafeRawPointer, _ dest: UnsafeMutableRawPointer) -> ()` and is invoked when the captures need to be copied into a new key path object, for example when two key paths are appended.

The *is equal* function has signature `@convention(thin) (UnsafeRawPointer, UnsafeRawPointer) -> Bool` and is invoked when the component is compared for equality with another computed component with the same identifier.

The *hash* function has signature `@convention(thin) (UnsafeRawPointer, UnsafeRawPointer) -> Int` and is invoked when the key path containing the component is hashed. The implementation understands a return value of zero to mean that the captures should have no effect on the hash value of the key path.

After every component except for the final component, a pointer-aligned pointer to the metadata for the type of the projected component is stored. (The type of the final component can be found from the `Value` generic argument of the `KeyPath<Root, Value>` type.)

Examples

Given:

```
struct A {
  var padding: (128 x UInt8)
  var b: B
}

class B {
  var padding: (240 x UInt8)
  var c: C
}

struct C {
  var padding: (384 x UInt8)
  var d: D
}
```

On a 64-bit platform, a key path object representing `\A.b.c.d` might look like this in memory:

Word	Contents
0	isa pointer to `ReferenceWritableKeyPath<A, D>`
1	reference counts
-	-
2	buffer header 0xC000_0028 - trivial, reference prefix, buffer size 40
-	-
3	component header 0x8000_0080 - struct component, offset 128, end of prefix
4	type metadata pointer for `B`
-	-
5	component header 0x4000_0100 - class component, offset 256
6	type metadata pointer for `C`
-	-
7	component header 0x0000_0180 - struct component, offset 384

If we add:

```
struct D {
  var computed: E { get set }
}

struct E {
  subscript(b: B) -> F { get }
}
```

then `\D.e[B()]` would look like:

Word	Contents
0	isa pointer to `WritableKeyPath<D, E>`
1	reference counts
-	-
2	buffer header 0x0000_0058 - buffer size 88
-	-
3	component header 0x3800_0000 - computed, settable, mutating
4	identifier pointer
5	getter
6	setter
7	type metadata pointer for `F`
-	-
8	component header 0x2100_0000 - computed, has captures
9	identifier pointer
10	getter
11	argument size 8
12	pointer to argument witnesses for releasing/retaining/equating/hashing `B`
13	value of `B()`

Key Path Patterns

(to be written)