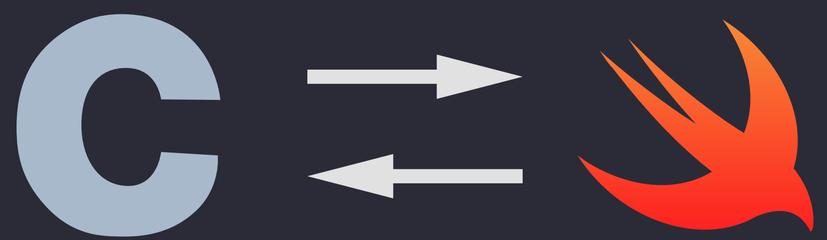
C Interoperability with Swift



Why not just use Swift?

“Don't trim your toe nails with a lawn mower”

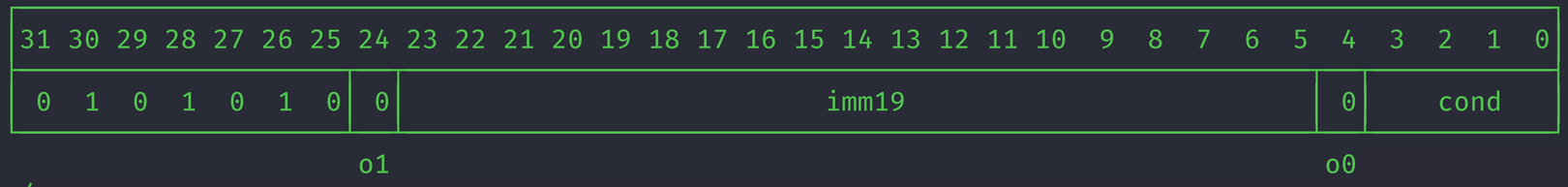
–Some guy on the internet

C has Macros

This is a good thing

```
/*
 ┌──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┐
 │31│30│29│28│27│26│25│24│23│22│21│20│19│18│17│16│15│14│13│12│11│10│ 9│ 8│ 7│ 6│ 5│ 4│ 3│ 2│ 1│ 0│
 ├──┼──┼──┼──┼──┼──┼──┼──┼──┼──┼──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┤
 │sf│op│ S│ 1│ 0│ 0│ 0│ 1│shift│                          imm12                          │       Rn      │       Rd      │
 └──┴──┴──┴──┴──┴──┴──┴──┴─────┴─────────────────────────────────────────────────────────────┴───────────────┴───────────────┘
 */
#define encode(NAME, op, S) \
EXPORT u32 encode ## NAME ## i(bool use64Bits, u32 shift, u32 imm12, ARM64Reg Rn, ARM64Reg Rd) { \
}

encode(ADD, 0, 0);
encode(ADDS, 0, 1);
encode(SUB, 1, 0);
encode(SUBS, 1, 1);
#undef encode
```

```
/*
```




```
 */
#define encode(NAME, cond) \
EXPORT u32 encode ## NAME(u32 imm19) { \
}

encode(BEQ, 0b0000);
encode(BNE, 0b0001);
encode(BMI, 0b0100);
encode(BPL, 0b0101);
encode(BVS, 0b0110);
encode(BVC, 0b0111);
encode(BHI, 0b1000);
encode(BLS, 0b1001);
encode(BGE, 0b1010);
encode(BLT, 0b1011);
encode(BGT, 0b1100);
encode(BLE, 0b1101);
encode(BAL, 0b1110);
#undef encode
```

```
#define encode /* ... */

encode(STXRB,   0b00, 0, 0, 0, 0);
encode(STLXRB,  0b00, 0, 0, 0, 1);
encode(LDXRB,   0b00, 0, 1, 0, 0);
encode(LDAXRB,  0b00, 0, 1, 0, 1);
encode(STLRB,   0b00, 1, 0, 0, 1);
encode(LDARB,   0b00, 1, 1, 0, 1);
encode(STXRH,   0b01, 0, 0, 0, 0);
encode(STLXRH,  0b01, 0, 0, 0, 1);
encode(LDXRH,   0b01, 0, 1, 0, 0);
encode(LDAXRH,  0b01, 0, 1, 0, 1);
encode(STLRH,   0b01, 1, 0, 0, 1);
encode(LDARH,   0b01, 1, 1, 0, 1);
encode(STXR32,  0b10, 0, 0, 0, 0);
encode(STLXR32, 0b10, 0, 0, 0, 1);
encode(STXP32,  0b10, 0, 0, 1, 0);
encode(STLXP32, 0b10, 0, 0, 1, 1);
encode(LDXR32,  0b10, 0, 1, 0, 0);
encode(LDAXR32, 0b10, 0, 1, 0, 1);
encode(LDXP32,  0b10, 0, 1, 1, 0);
encode(LDAXP32, 0b10, 0, 1, 1, 1);
encode(STLR32,  0b10, 1, 0, 0, 1);
encode(LDAR32,  0b10, 1, 1, 0, 1);
encode(STXR64,  0b11, 0, 0, 0, 0);
encode(STLXR64, 0b11, 0, 0, 0, 1);
encode(STXP64,  0b11, 0, 0, 1, 0);
encode(STLXP64, 0b11, 0, 0, 1, 1);
encode(LDXR64,  0b11, 0, 1, 0, 0);
encode(LDAXR64, 0b11, 0, 1, 0, 1);
encode(LDXP64,  0b11, 0, 1, 1, 0);
encode(LDAXP64, 0b11, 0, 1, 1, 1);
encode(STLR64,  0b11, 1, 0, 0, 1);
encode(LDAR64,  0b11, 1, 1, 0, 1);
#undef encode
```

C has a faster type checker

```
(sf<<31)|(op<<30)|(S<<29)|(0b10001<<24)|(shift<<22)|(imm12<<10)|(Rn<<5)|Rd
```

Remember me?

Expression was too complex to be solved in reasonable time; consider breaking up the expression into distinct sub-expressions

Distinct sub expression?

```
var val: UInt32 = 0
val |= sf      << 31
val |= op      << 30
val |= S       << 29
val |= 0b10001 << 24
val |= shift   << 22
val |= imm12   << 10
val |= Rn      << 5
val |= Rd
```

x 300

The problem with using C

```
encodeMOVZi(true, 0, 2, ARM64RegR0)
```

What is true?

Is this the immediate?

Is this the immediate?

We would much rather write `.R0`

This is what we want

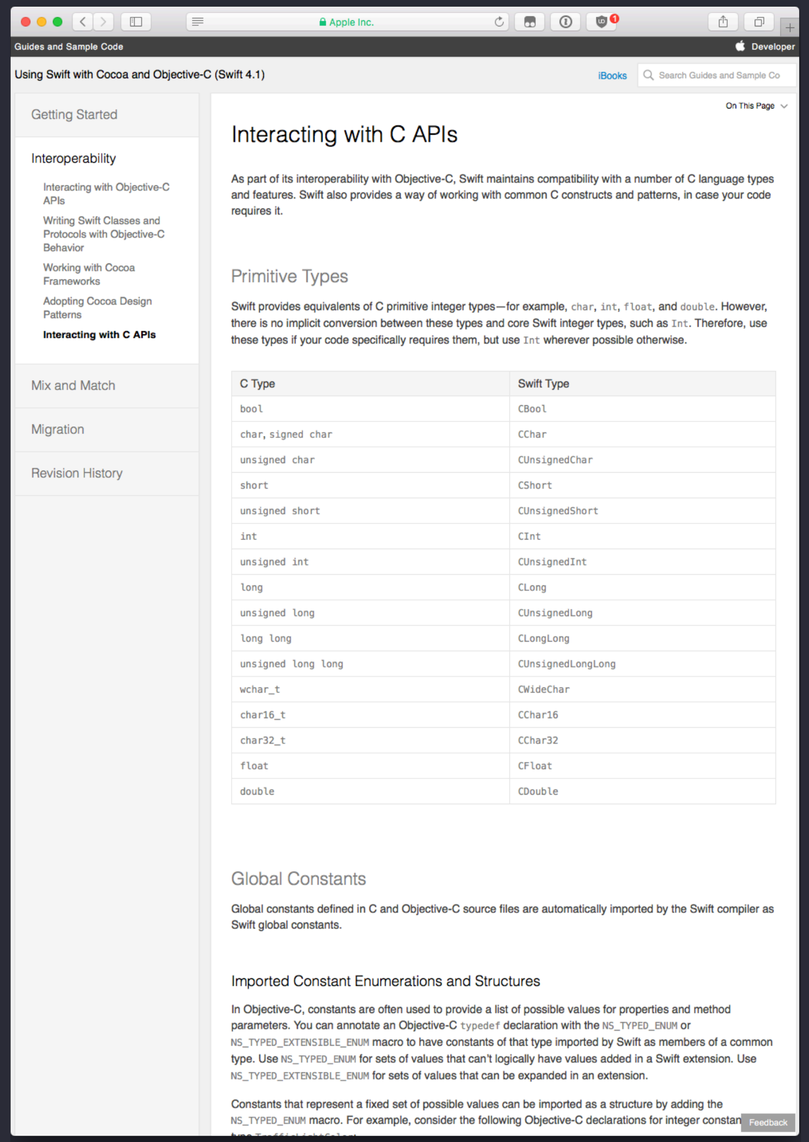
```
var mov = encodeMOVZi(use64Bits: true, hw: 0, imm16: 2, .R0)
var add = encodeADDi(use64Bits: true, shift: 0, imm12: 2, .R0, .R0)
var sub = encodeSUBi(use64Bits: true, shift: 0, imm12: 1, .R0, .R0)
```

- Argument Labels
- Dot syntax for enums

Where to start?

Apple Developer Documentation

- Enumerations
- Imported Constant Enumerations and Structures
- Importing Functions as Type Members



https://developer.apple.com/library/content/documentation/Swift/Conceptual/BuildingCocoaApps/InteractingWithCAPIs.html

Constant Enumerations and Structures

Annotated C

```
typedef NS_ENUM(NSInteger, UITableViewCellStyle) {
    UITableViewCellStyleDefault,
    UITableViewCellStyleValue1,
    UITableViewCellStyleValue2,
    UITableViewCellStyleSubtitle
};
```

Generated Swift Interface

```
enum UITableViewCellStyle: Int {
    case `default`
    case value1
    case value2
    case subtitle
}
```

Constant Enumerations and Structures

Annotated C

```
typedef long TrafficLightColor NS_TYPED_ENUM;

TrafficLightColor const TrafficLightColorRed;
TrafficLightColor const TrafficLightColorYellow;
TrafficLightColor const TrafficLightColorGreen;
```

Generated Swift Interface

```
struct TrafficLightColor: RawRepresentable, Equatable, Hashable {
    typealias RawValue = Int

    init(rawValue: RawValue)
    var rawValue: RawValue { get }

    static var red: TrafficLightColor { get }
    static var yellow: TrafficLightColor { get }
    static var green: TrafficLightColor { get }
}
```

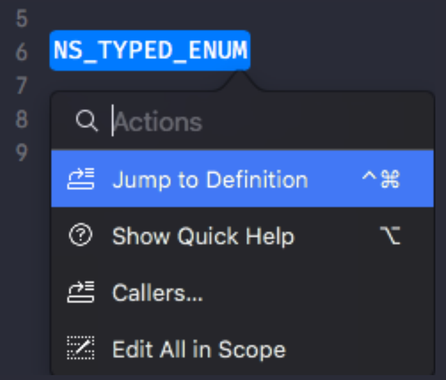
Importing Functions as Type Members

Annotated C

```
Color ColorCreateWithCMYK(float c, float m, float y, float k)
  CF_SWIFT_NAME(Color.init(c:m:y:k:));

float ColorGetHue(Color color)
  CF_SWIFT_NAME(getter:Color.hue(self:));

void ColorSetHue(Color color, float hue)
  CF_SWIFT_NAME(setter:Color.hue(self:newValue:));
```

Generated Swift Interface

```
extension Color {
    init(c: Float, m: Float, y: Float, k: Float)

    var hue: Float { get set }
}
```


NS_TYPED_ENUM
Actions
Jump to Definition ^⌘
Show Quick Help ⌥
Callers...
Edit All in Scope

```
// Foundation/NSObjCRuntime.h
#define NS_TYPED_ENUM _NS_TYPED_ENUM
#define _NS_TYPED_ENUM _CF_TYPED_ENUM
// CoreFoundation/CFAvailability.h
#if __has_attribute(swift_wrapper)
#define _CF_TYPED_ENUM
__attribute__((swift_wrapper(enum)))
#else
#define _CF_TYPED_ENUM
#endif

// Foundation/NSObjCRuntime.h
#define NS_SWIFT_NAME(_name) CF_SWIFT_NAME(_name)
// CoreFoundation/CFBase.h
#define CF_SWIFT_NAME(_name)
__attribute__((swift_name(#_name)))
```

Defining our own

```
#if __has_attribute(swift_wrapper)
#define BRIDGE_ENUM_TO_SWIFT
    __attribute__((swift_wrapper(enum)))
#else
#define BRIDGE_ENUM_TO_SWIFT
#endif

#if __has_attribute(swift_name)
# define BRIDGE_TO_SWIFT_WITH_NAME(_NAME)
    __attribute__((swift_name(#_NAME)))
#else
# define BRIDGE_TO_SWIFT_WITH_NAME(_NAME)
#endif
```

Annotated C "enums"

```
typedef u32 ARM64Reg BRIDGE_ENUM_TO_SWIFT;
ARM64Reg const ARM64RegR0  =  0;
ARM64Reg const ARM64RegR1  =  1;
ARM64Reg const ARM64RegR2  =  2;
ARM64Reg const ARM64RegR3  =  3;
ARM64Reg const ARM64RegR4  =  4;
//  ...
```

Annotated C functions

```
// MARK: Add/subtract (immediate)
/*
```

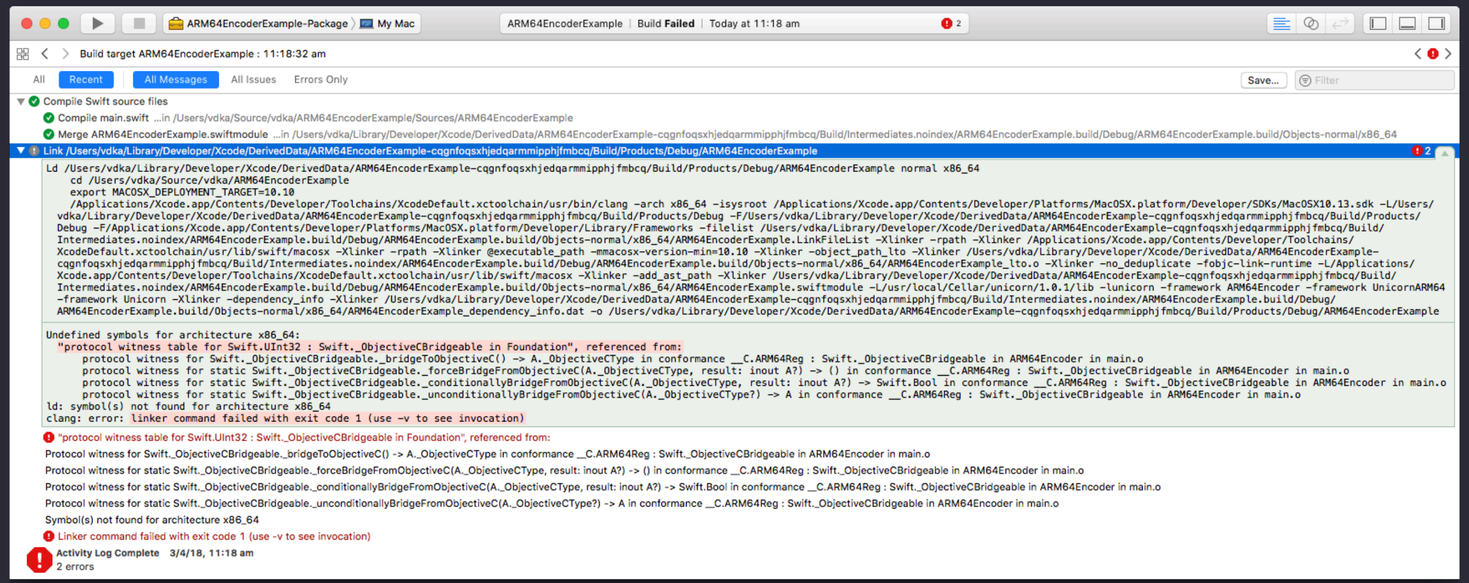
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	0	0	0	1	shift																							
																						imm12									
																						Rn									
																						Rd									

```
 */
#define encode(NAME, op, S) \
BRIDGE_TO_SWIFT_WITH_NAME(encode ## NAME ## i(use64Bits:shift:imm12:_:_:)) \
EXPORT u32 encode ## NAME ## i(bool use64Bits, u32 shift, u32 imm12, ARM64Reg Rn, ARM64Reg Rd) { \
    assert(canPack(shift, 2)); \
    assert(canPack(imm12, 12)); \
    u32 sf = use64Bits ? 1 : 0; \
    Rn = encodeARM64Reg(Rn, ARM64RegSP); \
    Rd = encodeARM64Reg(Rd, ARM64RegSP); \
    return (sf << 31) | (op << 30) | (S << 29) | (0b10001 << 24) | (shift << 22) | (imm12 << 10) | (Rn << 5) | Rd; \
}

encode(ADD, 0, 0);
encode(ADDS, 0, 1);
encode(SUB, 1, 0);
encode(SUBS, 1, 1);
#undef encode
```

Swift Package Manager

Swift Package Manager

```
▷ tree
.
├── Package.swift
└── Sources
    └── ARM64Encoder
        ├── encoder.c
        └── include
            └── encoder.h

3 directories, 4 files

▷ cat Package.swift
// swift-tools-version:4.0

import PackageDescription

let package = Package(
    name: "ARM64Encoder",
    products: [
        .library(name: "ARM64Encoder", targets: ["ARM64Encoder"]),
    ],
    targets: [
        .target(name: "ARM64Encoder", dependencies: []),
    ]
)
```



Big gotcha

Remember to `import Foundation`

```
import Unicorn
import UnicornARM64
import ARM64Encoder
import Foundation

var mov = encodeMOVZi(use64Bits: true, hw: 0, imm16: 2, .R0)
var add = encodeADDi(use64Bits: true, shift: 0, imm12: 2, .R0, .R0)
var sub = encodeSUBi(use64Bits: true, shift: 0, imm12: 1, .R0, .R0)

var instructions: [UInt8] = []
instructions += withUnsafeBytes(of: &mov, Array.init)
instructions += withUnsafeBytes(of: &add, Array.init)
instructions += withUnsafeBytes(of: &sub, Array.init)

let address: UInt64 = 0x400000
var uc: OpaquePointer?
var r0: UInt64 = 0x0 // R0 register (also called X0 for 64 bits)

// Initialize emulator in ARM64 mode
try open(arch: .arm64, mode: .arm, &uc)

// map 2MB memory for this emulation & write machine code to be emulated
try memMap(uc, address: 0, size: numericCast(ADDRESS + 2 * 1024), perms: .all)
try memWrite(uc, address: address, bytes: instructions)

try emuStart(uc, begin: address, until: address + numericCast(instructions.count))

// now print out some registers
print("Emulation done. Below is the CPU context")

try regRead(uc, regid: ARM64Reg.x0, value: &r0)
print(">>> r0 = 0x\(String(r0, radix: 16))")
```

GitHub

vdka/ARM64Encoder

Questions?