# The Go low-level calling convention on x86-64 (updated)

What's new in 2020 and in Go 1.15

Raphael 'kena' Poss

November 2020

**modified**: 2020-12-01
**subtitle**: What's new in 2020 and in Go 1.15
**slug**: go-calling-convention-x86-64-2020
**category**: Programming
**tags**: golang, compilers, analysis, programming languages, c++
**series**: Go low-level code analysis

# Contents

> **Note**
> The latest version of this document can be found online at https://dr-knz.net/go-calling-convention-x86-64-2020.html. Alternate formats: Source, PDF.

# Introduction

Two years ago, this article reviewed the low-level code generation of the Go compiler, as of version 1.10. A few things have changed since, and so an update is in order.

As previously, All tests below are performed using the `freebsd/amd64` target; this time using go 1.15.5. The assembly listing are produced with `go tool objdump` and use the Go assembler syntax.

# Calling convention

## Arguments and return value

The mechanisms for passing arguments and return values remain largely unchanged since go 1.10: they are passed via memory, on the stack.

## Call sequence: how a function gets called

As in Go 1.10, in 1.15 a function places arguments for its callee into its activation record, and makes space for the return value there as well. The callee writes the return value the caller's activation record.

As before, a side effect of this design is that when a function returns the same value as one of its callees, it needs to read the return value from the callee from its own activation record, then place it back onto the stack at a return value in its caller's activation record. Tail call optimizations (TCO) thus remain impossible.

Additionally, function prologues remain largely unchanged:

- a function that uses local variables needs to set up an activation record by adjusting the SP register, and does this always in its prologue.
- as before, every function also sets up a frame pointer in the BP register to facilitate exception unwinds.
- as before, a function that uses more than a few words of stack, or that performs a function call, also needs to check the remaining size of the stack upfront and allocate more stack if needed. This is because Go allocates tiny stacks to goroutines by default.

Naturally, the epilogue un-does these operations.

Here is an example function prologue and epilogue, taken from one of the Go runtime's internal functions:

```
internal/cpu.Initialize:
        ; Check remaining stack size:
        MOVQ FS:0xfffffff8, CX
        CMPQ 0x10(CX), SP ; at least 24 bytes on the stack?
        JBE 0x401047      ; no: go to block at end of function below

        ; Allocate activation record:
```

```
        SUBQ $0x18, SP    ; 24 bytes in activation record

        ; Set up the frame pointer
        MOVQ BP, 0x10(SP) ; BP is callee-save: store it
        LEAQ 0x10(SP), BP ; set up new frame pointer
        ...

        MOVQ 0x10(SP), BP ; restore the caller's frame pointer
        ADDQ $0x18, SP    ; deallocate the activation record
        RET               ; return

0x401047:
        CALL runtime.morestack_noctxt(SB) ; alloc more stack
        JMP internal/cpu.Initialize(SB) ; restart
```

### Callee-save registers—or not

Go 1.15 did not evolve from 1.10: there are still no callee-save registers. All temporaries are spilled to the stack upon a function call.

### The cost of pointers and interfaces

The layout of pointers and interfaces remains unchanged:

- a pointer takes one word.
- an interface takes two words: one for the vtable, one for the reference to the object.
- strings have two words: the length and a pointer to the string's bytes.
- slices have three words: the length, the capacity and a pointer to the data.

As before, interface references to empty structs use a zero pointer as object reference; the entirety of the implementation is decided by the vtable pointer.

When an object implements an interface by value, *in the general case* converting the object to an interface reference moves the object to the heap.

To see how this happens, we can use the following code:

```
// Define a struct type implementing the interface by value.
type bar struct{ x int }
func (bar) foo() {}
// Define a global variable so we don't use the heap allocator.
var y bar

// Make an interface value.
func MakeInterface2() Foo { return y }
```

This gives us:

```
MakeInterface2:
   ; <function prologue>

   ; write y to 0(SP), as an argument to runtime.convT64
   0x45c55d                488b057cc70900          MOVQ main.y(SB), AX
   0x45c564                48890424                MOVQ AX, 0(SP)
   ; call runtime.convT64, this converts the object to a heap reference
   0x45c568                e833c5faff              CALL runtime.convT64(SB)
```

```
; extract the return value
0 x45c56d               488b442408              MOVQ 0x8(SP), AX
; take the vtable pointer
0 x45c572               488d0d07c00200          LEAQ go.itab.main.bar,main.Foo(SB), CX
; write both to the return value slot for MakeInterface2
0 x45c579               48894c2420              MOVQ CX, 0x20(SP)
0 x45c57e               4889442428              MOVQ AX, 0x28(SP)

; <function epilogue>
0 x45c58c               c3                      RET
```

This code generation in v1.15 is slightly different from what it was in v1.10; back then we would see instead:

```
MakeInterface2:
   ; <function prologue>

   ; take the vtable pointer
   0 x4805dd             488d053c020400          LEAQ go.itab.src.bar,src.Foo(SB), AX
   ; pass it as argument to convT2I64
   0 x4805e4             48890424                MOVQ AX, 0(SP)
   ; take the address of y
   0 x4805e8             488d05e9f10b00          LEAQ main.y(SB), AX
   ; pass it as argument to convT2I64
   0 x4805ef             4889442408              MOVQ AX, 0x8(SP)
   ; convert to interface reference
   0 x4805f4             e8e7b2f8ff              CALL runtime.convT2I64(SB)
   ; copy the return value from runtime.convT2I64 to the return slot of MakeInterface2
   0 x4805f9             488b442410              MOVQ 0x10(SP), AX
   0 x4805fe             488b4c2418              MOVQ 0x18(SP), CX
   0 x480603             4889442430              MOVQ AX, 0x30(SP)
   0 x480608             48894c2438              MOVQ CX, 0x38(SP)

   ; <function epilogue>
   0 x480616             c3                      RET
```

This is a new optimization: the conversion of an object to an interface reference now costs 7 instructions instead of 9 previously. The main change is that previously, `runtime.convT2I64` was responsible both for moving the object to the heap and attaching the vtable pointer; whereas in v1.15 `runtime.convT64` just moves the object to the heap and returns a naked pointer, and the caller is responsible for attaching the vtable pointer.

Additionally, another optimization is performed inside the `convT64` function: for certain specific values, no heap allocation is performed. In v1.10, this optimization was restricted to the case of a single-word value or struct that was initialized to its default (all zero bytes). In v1.15, the optimization was extended to include all integer values smaller than 256 (i.e. 0x00-0xFF).

This optimization is available for all word-sized types or smaller. For example, it works with an integer type implementing the interface directly, as well as for a struct with a single integer field.

### Vararg calls

Go supports variable numbers of arguments, via the `...` construct. In a nutshell, the caller prepares a slice object on the stack and makes it point at the positional arguments (also on the stack), then passes that slice as fixed-position argument to the callee.

In addition to this, if the vararg list was declared with an interface type (which is a common case, for example `fmt.Printf` has `...interface{}`), a conversion from each argument value to an interface reference

must also take place. This conversion moves each argument to the heap in the general case, with a "small numbers optimization" as described in the previous section.

Let us see how this looks like. First we can look at the case of a vararg list that is *not* an interface type:

```
func f(...int) {}

var x,y,z,w int
func caller() {
   f(x,y,z,w)
}
```

This gives us:

```
caller:
    ; <function prologue>

    ; fill the slice:
    XORPS X0, X0          ; set 2 words (128 bit) to zero in X0
    MOVUPS X0, 0x18(SP)   ; initialize the 4-element slice to zero
    MOVUPS X0, 0x28(SP)   ; initialize the 4-element slice to zero
    MOVQ main.x(SB), AX
    MOVQ AX, 0x18(SP)     ; store x into 1st position
    MOVQ main.y(SB), AX
    MOVQ AX, 0x20(SP)     ; store y into 2nd position
    MOVQ main.z(SB), AX
    MOVQ AX, 0x28(SP)     ; store z into 3rd position
    MOVQ main.w(SB), AX
    MOVQ AX, 0x30(SP)     ; store w into 4th position

    ; prepare the slice as outgoing argument:
    LEAQ 0x18(SP), AX     ; store the base address
    MOVQ AX, 0(SP)
    MOVQ $0x4, 0x8(SP)    ; store the length
    MOVQ $0x4, 0x10(SP)   ; store the capacity

    CALL main.g(SB)       ; call the function

    ; <function epilogue>
    RET
```

So far, no surprises. It may be interesting to note that Go always wastes instructions to pre-initialize the vararg slice to zero, even though it immediately populates it afterwards with the argument values. A C++ compiler would not do that for vararg calls and simply writes the argument directly to their final spots.

We can then compare what happens when the function takes its arguments using an interface type:

```
// note: now we have an interface type.
func f(...interface{}) {}

var x,y,z,w int
func caller() {
   f(x,y,z,w)
}
```

This gives us:

```
caller:
    ; <function prologue>

    ; fill the slice:
```

```
XORPS X0, X0         ; zero out the slice
MOVUPS X0, 0x38(SP)
MOVUPS X0, 0x48(SP)
MOVUPS X0, 0x58(SP)
MOVUPS X0, 0x68(SP)

MOVQ main.x(SB), AX
MOVQ AX, 0x30(SP)    ; copy x on the stack, out of the slice
LEAQ 0x7995(IP), AX
MOVQ AX, 0x38(SP)    ; place x's interface{} vtable ptr in the slice
LEAQ 0x30(SP), CX
MOVQ CX, 0x40(SP)    ; place the address of x's copy in the slice

MOVQ main.y(SB), CX
MOVQ CX, 0x28(SP)    ; copy y on the stack, out of the slice
MOVQ AX, 0x48(SP)    ; place the same vtable ptr as x in the slice
LEAQ 0x28(SP), CX
MOVQ CX, 0x50(SP)    ; place the address of y's copy in the slice

MOVQ main.z(SB), CX
MOVQ CX, 0x20(SP)    ; copy z on the stack, out of the slice
MOVQ AX, 0x58(SP)    ; place the same vtable ptr as x in the slice
LEAQ 0x20(SP), CX
MOVQ CX, 0x60(SP)    ; place the address of z's copy in the slice

MOVQ main.w(SB), CX  ; copy w on the stack, out of the slice
MOVQ CX, 0x18(SP)
MOVQ AX, 0x68(SP)    ; place the same vtable ptr as x in the slice
LEAQ 0x18(SP), AX
MOVQ AX, 0x70(SP)    ; place the address of w's copy in the slice

LEAQ 0x38(SP), AX
MOVQ AX, 0(SP)       ; set the slice base address as argument
MOVQ $0x4, 0x8(SP)   ; the slice's size
MOVQ $0x4, 0x10(SP)  ; the slice's capacity

CALL main.g(SB)       ; call the function

; <function epilogue>
RET
```

Here are the main differences:

- Each position in the vararg slice now has two words instead of just one.
- Each value passed must be passed by reference. Simple object types (such as integer heres) are merely copied into the caller's activation record, and the address of their stack copy is added to the slice.

  (The reason why the object is first copied to the stack, instead of placing the address to the global variable directly in the slice, is that Go must preserve the sequential semantics that the value is sampled at the time the call is performed, and will not change in the slice even if the global variable is modified in the callee or another goroutine.)
- If the interface was non-trivial, we would also see a call to *runtime.convT* for each argument.

## Exception handling

### Implementation of `defer`

`defer` is the keyword by which a programmer can specify one or more callback functions to call on every return path, including exception unwinds. This helps implement RAII patterns in Go.

In Go 1.10, each use of `defer` was translated to a *call* to `runtime.deferproc` which would register the deferred call onto the current goroutine's unwind stack. Because this was done via a call, every function containing the `defer` keyword also had to check its stack size and prepare an activation record, *even for functions that did not perform function calls otherwise.* In other words, the cost of using `defer` in "leaf" functions was rather high.

Additionally, in Go 1.10, the compiler would place a call to `runtime.deferreturn` on every return path, and that runtime function was responsible for performing the defer calls.

An example of these previous mechanisms is given in the previous analysis.

In contrast, Go 1.15 contains two optimizations that make the implementation of `defer` rather different in the case when `defer` is used unconditionally—i.e. it is always reached from the function's entry point. In that case:

- when there are 8 uses of `defer` or fewer, the Go compiler optimizes them by writing the callbacks to the function's activation record directly. There is no need for a `runtime.deferproc` any more in that case. During exception unwinding, the unwinding code knows where to look for defers in each activation record.
- separately, the compiler also emits the full call sequences to the deferred functions in every return path, so that the natural return control flow performs these calls, and there is no call to `runtime.deferreturn`.

This way, a function containing 8 or less unconditional `defers` to functions that themselves can be inlined does not pay the overhead of setting up a caller context if it does not otherwise perform function calls.

(Naturally, these optimizations do not work if a `defer` is conditional, or occurs inside a loop.)

Here is an example:

```
func Defer1() int { defer f(); return 123 }
```

This compiles to:

```
Defer1:
      ; <function prologue>

      0x45c4bd        MOVQ $0x0, AX
      0x45c4c4        MOVQ AX, 0x8(SP)   ; set up a word full with zeroes
      0x45c4c9        MOVB $0x0, 0x7(SP) ; set the first byte to zero (redundant)

      ; write zero to the return value slot
      0x45c4ce        MOVQ $0x0, 0x20(SP)

      ; defer the call to f()
      0x45c4d7        LEAQ 0x1b672(IP), AX
      0x45c4de        MOVQ AX, 0x8(SP)      ; write the address of f
      0x45c4e3        MOVB $0x1, 0x7(SP)    ; let the runtime know there is 1 defer

      ; write the return value 123
      0x45c4e8        MOVQ $0x7b, 0x20(SP)
```

```
; un-defer
0x45c4f1        MOVB $0x0, 0x7(SP)    ; let the runtime know there is no more defer
; final call to f() on the return path
0x45c4f6        CALL main.f(SB)

; <function epilogue>
0x45c504        RET

; the following code is called during unwinds after a recover,
; not on the common case:
0x45c505        CALL runtime.deferreturn(SB)
0x45c50a        MOVQ 0x10(SP), BP
0x45c50f        ADDQ $0x18, SP
0x45c513        RET
```

In this example, the `f()` function is non-inlinable so the call to `f()` remains explicit in the generated code. If `f()` had been inlinable, then the return paths would be simplified and the `Defer1()` function would not need an activation record.

### Implementation of `panic`

Throwing an already-built value as an exception within a function body works in Go 1.15 very much like in Go 1.10: as a call to the function `runtime.gopanic`. This function takes an argument of type `interface{}`; therefore, whatever value is passed must be promoted to an interface reference as explained above.

Here is an example:

```
func Panic() { panic(123) }
```

This compiles to:

```
Panic:
   ; <function prologue>

   ; load the vtable for interface{}:
   LEAQ 0x78dc(IP), AX
   MOVQ AX, 0(SP)

   ; load the address of a static copy of the
   ; integer value 123:
   LEAQ 0x2afa9(IP), AX
   MOVQ AX, 0x8(SP)

   ; call gopanic:
   CALL runtime.gopanic(SB)
   NOPL
   ; note: function epilogue omitted in this case
```

There is no surprise here—the compiler knows that the function never returns and thus the return path (and, in this case, the entirety of the function's epilogue) is omitted.

There is a small optimization in v1.15 compared to v1.10: the padding instruction after the call used to be an undefined 2-byte opcode 0x0F0B (disassembled as UD2); this is now generated as a regular NOP, which is smaller (just 1 byte, 0x90).

### Catching exceptions: `defer + recover`

The mechanism for catching exception has not changed: any use of `recover()` in the source is compiled as a regular function call to `runtime.gorecover`. As in previous versions, this halts the exception propagation and outputs the panic object as return value.

## Summary and conclusions

In this chapter, we revisited the findings from two years ago.

To a large extent, the low-level calling convention in Go v1.15 is not much different from what it was in v1.10; the previous observations thus largely remain unchanged:

- arguments and return values are still passed via memory.
- activation records are still registered dynamically, instead of using static unwinding tables as is done in e.g. C++.
- pointers occupy one word; `string` values and interface references, two; and slices occupy three words.
- the promotion of objects to interface references, when they implement the interface by value, requires a move to the heap via a call to a runtime function in the general case.

The notable changes are as follows:

- converting a value to an interface reference has become simpler, as the caller does not pass the vtable pointer to the runtime `conv` functions any more. This saves instructions on the way in and out of the conversion.
- the "small value" optimization, which aims to avoid a heap allocation when promoting a value to an interface reference, has been extended to all 1-word values from zero up to and including 255.
- when a function contains 8 or less non-conditional uses of `defer`, an optimization kicks in that prevents calls to `runtime.deferproc` and `runtime.deferreturn` entirely. In that case, the deferred callback information is stored in the function's activation record. The exception unwinding code is now equipped to find deferred callbacks there, in addition to the goroutine struct. This greatly reduces the runtime overhead in the common case when a function only uses `defer` once or twice, in the main control path.

Additionally, this time we visited a more detailed example of calling a vararg function, with the step-by-step construction of the argument slice.

Because of the lack of major changes, the open question from last time just as valid with Go 1.15 as it was in 1.10:

> What is cheaper: handling exceptions via `panic` / `recover`, or passing and testing error results with `if err := ...; err != nil { return err }`?

This question is non-trivial because the cost of a `panic` call and the top-level error recovery with `defer` and `recover` can be amortized across a workload. Where would the inflection point lie?

We will revisit this question in the next part.

Also in the series:

- The Go low-level calling convention on x86-64
- Measuring argument passing in Go and C++
- Measuring multiple return values in Go and C++
- Measuring errors vs. exceptions in Go and C++

---

## Copyright and licensing