

# Не для слабонервных: Погружение в мир без правил с пакетом unsafe



Нина Пакшина



# Ограничения обычных указателей в Go



Указатель с типом T1 не может быть  
преобразован в произвольный тип

$T1 \neq T2$

$*T1 \not\rightarrow *T2$

Указатель типа T1 нельзя сравнивать с  
указателем типа T2

$T1 \neq T2$

$T1 \vee T2 \neq \text{nil}$

$*T1 \nless *T2$

Переменную, хранящей указатель типа T1,  
нельзя переписать указателем T2

T1 != T2

T1 v T2 != nil

\*T1 ~~=~~ \*T2

# Не поддерживают адресную арифметику

`p++`, `p-2`, `p--`, etc

Как обойти эти  
ограничения?



# Пакет unsafe



Позволяет обойти  
безопасность типов в Go



# ОСНОВНЫЕ КОМПОНЕНТЫ unsafe

# unsafe.Pointer

Указатель на произвольный тип (ArbitraryType)

```
x := uint32(4)
```

```
p := unsafe.Pointer(&x)
```

Может быть получен из обычного  
указателя и преобразован в  
указатель любого типа

```
x := uint32(4)
rp := &x
p := unsafe.Pointer(rp)
newPtr := (*int32)(p)
```

Может быть преобразован в uintptr и  
обратно\*

```
x := uint32(4)
```

```
u := uintptr(unsafe.Pointer(&x))
```

```
p := unsafe.Pointer(u)
```

Может быть преобразован в любой  
тип\*

```
x := uint32(4294967295)
z := *(*int32)(unsafe.Pointer(&x)) //тип int32
```

# unsafe.Sizeof

```
s1 := [4]byte{0x00, 0x00, 0x00, 0x00}  
unsafe.Sizeof(s1) → ?
```

→ 4

```
s2 := []byte{0x00, 0x00, 0x00, 0x00}  
unsafe.Sizeof(s) → ?
```

→ 24

ptr → 8

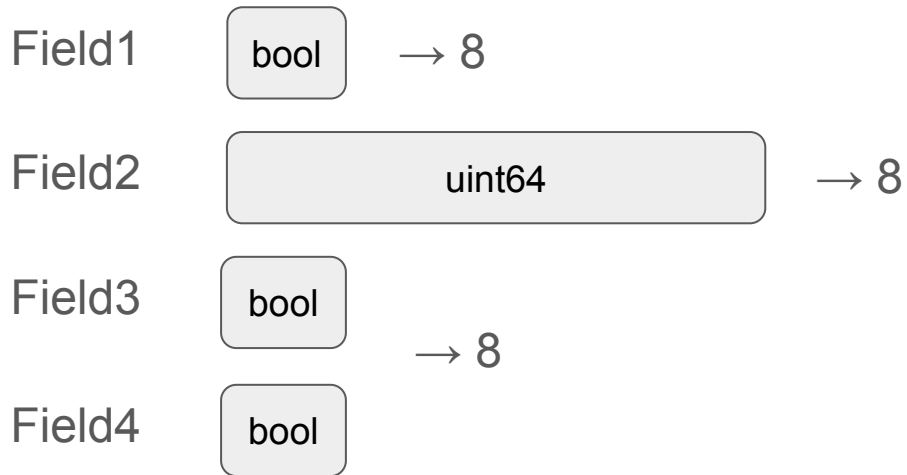
len → 8

cap → 8

# unsafe.Sizeof

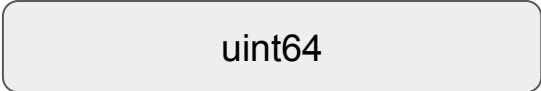
```
s := myStruct{}
```

`unsafe.Sizeof(s)` → 24



# unsafe.Sizeof

```
s := myStruct{}
```

Field2            → 8

Field1            unsafe.Sizeof(s)      → 16

Field3            → 8      unsafe.Sizeof(s.Field4) → 1

Field4      



# unsafe.Alignof

```
s := myStruct{}
```

Field2

uint64

Field1

bool

Field3

bool

Field4

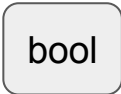
bool

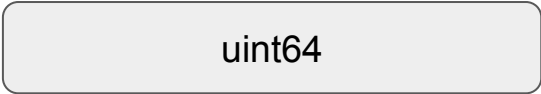
`unsafe.Alignof(s) → 8`

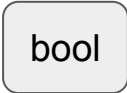
# unsafe.Offsetof

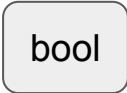
```
s := myStruct{}
```

```
unsafe.Offsetof(s.Field3) → 16
```

Field1       bool      → 0

Field2       uint64      → 8

Field3       bool      → 16

Field4       bool      → 17

# Адресная арифметика

Если знаем:

- начальный адрес
- смещение
- длину данных

Можем получить/преобразовать данные

# Массив

```
s1 := [4]byte{0x01, 0x02, 0x03, 0x04}
```

Указатель на  
начало массива

Размер элемента

```
p1 := unsafe.Pointer(uintptr(unsafe.Pointer(&s1)) + unsafe.Sizeof(s1[0]))
```

Числовое представление адреса

```
x1 := *(*byte)(unsafe.Pointer(&p1))
```

```
x1 → 0x02
```

# Слайс

```
s2 := []byte{0x01, 0x02, 0x03, 0x04}
```

```
x2 := *(*byte)(unsafe.Pointer(uintptr(unsafe.Pointer(&s2)) +  
+ unsafe.Sizeof(s2[0])))
```

$x2 \rightarrow$  Ерунда

```
x2 := *(*byte)(unsafe.Pointer(uintptr(unsafe.Pointer(&s2[0])) + 1))
```

# Преобразование данных

```
x := uint32(4294967295)
```

```
unsafe.Sizeof(x) → 4 byte
```



$x \rightarrow [4]\text{byte}$

Начальный адрес  
`unsafe.Pointer(&x)`

4 байта

0x0...01

0xFF

0x0...02

0xFF

0x0...03

0xFF

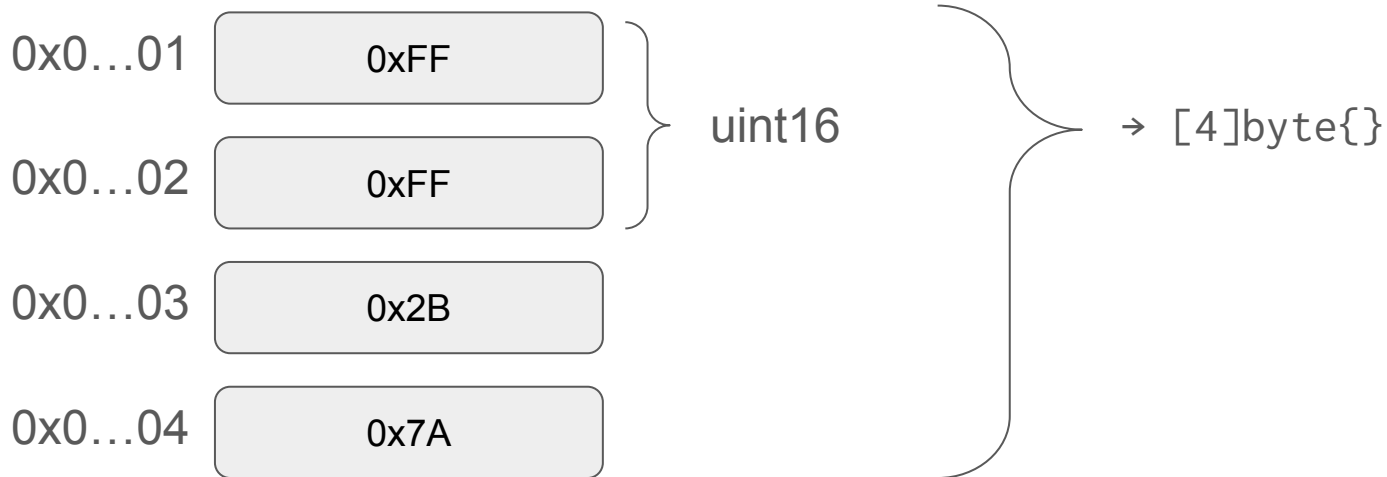
0x0...04

0xFF

```
s := *(*[4]byte)(unsafe.Pointer(&x))
```

```
s → [4]byte{0xFF, 0xFF, 0xFF, 0xFF}
```

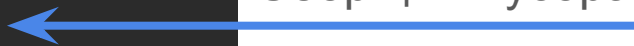
Преобразование указателя с \*T1 на \*T2  
возможно, если T2 не больше T1 по размеру и  
они имеют эквивалентную структуру памяти.



# Преобразование из `unsafe.Pointer` в `uintptr`, но не обратно

```
x := uint16(4)
u := uintptr(unsafe.Pointer(&x))
p := unsafe.Pointer(u)
```

Сборщик мусора





# Преобразование из `unsafe.Pointer` в `uintptr`, но не обратно

```
x := uint16(4)
u := uintptr(unsafe.Pointer(&x))
p := unsafe.Pointer(u)
```

Сборщик мусора

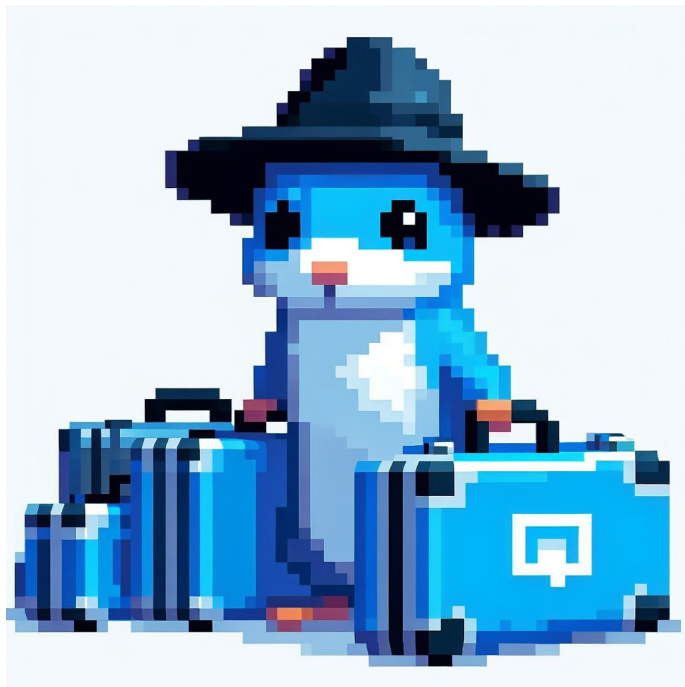
```
// Registers
offset := unsafe.Offsetof(header.ByteCount) + unsafe.Sizeof(header.ByteCount)
regAddr := unsafe.Pointer(uintptr(unsafe.Pointer(&bytes[0])) + offset)
```

# Проверка паттернов

go vet ./...



# Кейсы



# syscall.Syscall



```
func Syscall(trap, a1, a2, a3 uintptr) (r1, r2 uintptr, err syscall.ErrorString)
func Syscall6(trap, a1, a2, a3, a4, a5, a6 uintptr) (r1, r2 uintptr, err syscall.ErrorString)
func RawSyscall(trap, a1, a2, a3 uintptr) (r1, r2, err uintptr)
func RawSyscall6(trap, a1, a2, a3, a4, a5, a6 uintptr) (r1, r2, err uintptr)
```

```
_, _, errno := syscall.Syscall(syscall.SYS_MSYNC, uintptr(p), uintptr(len(b)), uintptr(flags))
if errno != 0 {
    return errno
}
return err: nil
```

```
//go:uintptrkeepalive
//go:nosplit
//go:linkname Syscall
func Syscall(trap, a1, a2, a3 uintptr) (r1, r2 uintptr, err Errno) {
    runtime_entersyscall()
```

```
// If marked as "go:uintptrkeepalive", don't inline, since the
// keep alive information is lost during inlining.
//
// TODO(prattmic): This is handled on calls during escape analysis,
// which is after inlining. Move prior to inlining so the keep-alive is
// maintained after inlining.
if fn.Pragma&ir.UintptrKeepAlive != 0 {
    reason = "marked as having a keep-alive uintptr argument"
    return
}
```

`go:uintptrkeepalive`

-----

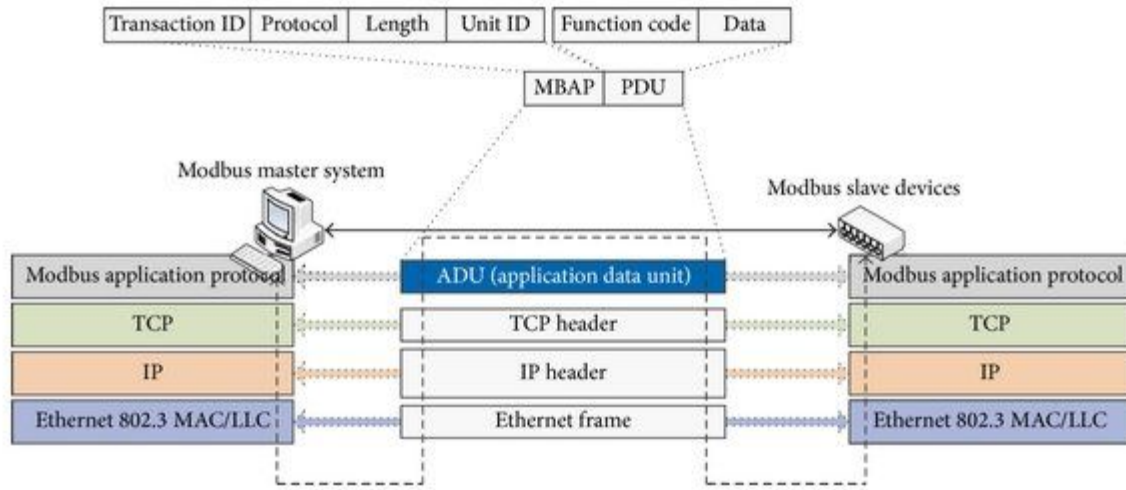
The `//go:uintptrkeepalive` directive must be followed by a function declaration.

It specifies that the function's `uintptr` arguments may be pointer values that have been converted to `uintptr` and must be kept alive for the duration of the call, even though from the types alone it would appear that the object is no longer needed during the call.

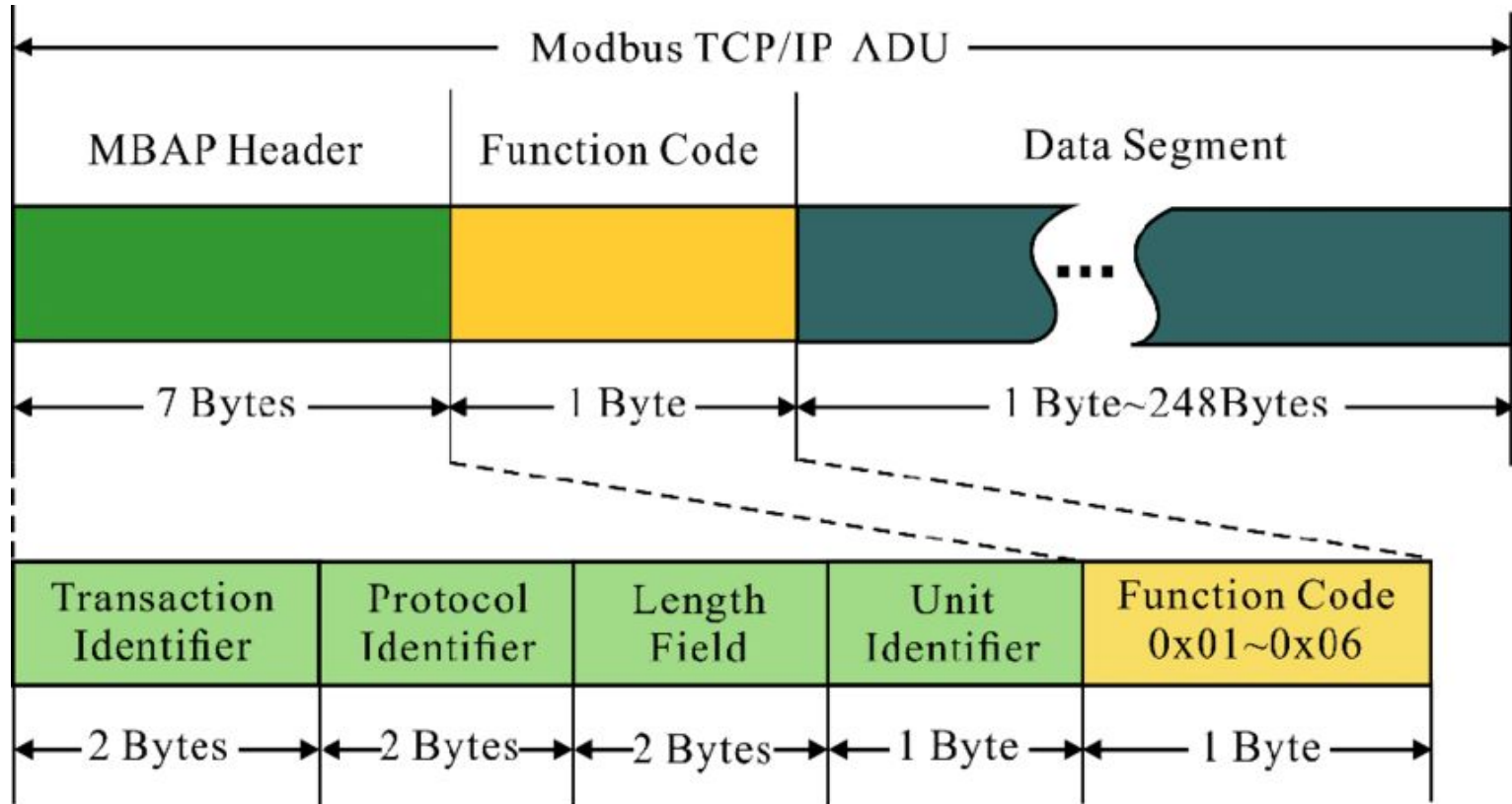
This directive is similar to `//go:uintptrescapes`, but it does not force arguments to escape. Since stack growth does not understand these arguments, this directive must be used with `//go:nosplit` (in the marked function and all transitive calls) to prevent stack growth.

The conversion from pointer to `uintptr` must appear in the argument list of any call to this function. This directive is used for some low-level system call implementations.

# Сырые сокеты и протоколы



# Modbus TCP/IP





# ADU Modbus

<b>0001</b>	<b>Transaction identifier</b>	Transaction Identifier
<b>0000</b>	<b>Protocol identifier</b>	Protocol Identifier
<b>0009</b>	<b>The length</b> (9 bytes are followed)	Message Length
<b>11</b>	<b>The device address</b> (17 = 11 hex)	Unit Identifier
<b>03</b>	<b>Function code</b> (read Analog Output Holding Registers)	Function Code
<b>06</b>	<b>The number of bytes later</b> (6 bytes are followed)	Byte Count
<b>02</b>	<b>Value of the high register bit</b> (02 hex)	Register value Hi (AO0)
<b>2B</b>	<b>Early discharge value register</b> (2B hex)	Register value Lo (AO0)
<b>00</b>	<b>Value of the high register bit</b> (00 hex)	Register value Hi (AO1)
<b>64</b>	<b>Value of the low register bit</b> (64 hex)	Register value Lo (AO1)
<b>00</b>	<b>Value of the high register bit</b> (00 hex)	Register value Hi (AO2)
<b>7F</b>	<b>Early discharge value register</b> (7F hex)	Register value Lo (AO2)

# Модбас ADU

```
[ ]byte{
```

```
    0x00, 0x00,
```

Transaction ID

```
    0x01, 0x00,
```

Protocol ID

```
    0x09, 0x00,
```

Length

```
    0x00,
```

Device address

```
    0x03,
```

Function code

```
    0x06,
```

Bytes count

```
    0x2B, 0x02, 0x64, 0x00, 0x7F, 0x00,
```

Data registers

```
}
```

# Модбас PDU

```
type ModbusHeader struct {  
    TransactionID    uint16  
    ProtocolID       uint16  
    Length           uint16  
    DeviceAddress     uint8  
    FunctionCode      uint8  
    BytesCount        uint8  
}
```

```
Registers := []uint16{}
```

# Из байт в структуру

```
[]byte{0x00, 0x00, 0x01, 0x00, 0x09, 0x00, 0x00, 0x03, 0x06, 0x2B,  
0x02, 0x64, 0x00, 0x7F, 0x00}
```

```
func getModbusPDU(bytes []byte) (ModbusHeader, []uint16) {  
    // Header  
    header := *(*ModbusHeader)(unsafe.Pointer(&bytes[0]))  
  
    // Registers  
    offset := unsafe.Offsetof(header.ByteCount) + unsafe.Sizeof(header.ByteCount)  
    regAddr := unsafe.Pointer(uintptr(unsafe.Pointer(&bytes[0])) + offset)  
    bytesAddr := (*uint16)(regAddr)  
    registers := unsafe.Slice(bytesAddr, int(header.ByteCount / 2))  
    return header, registers  
}
```

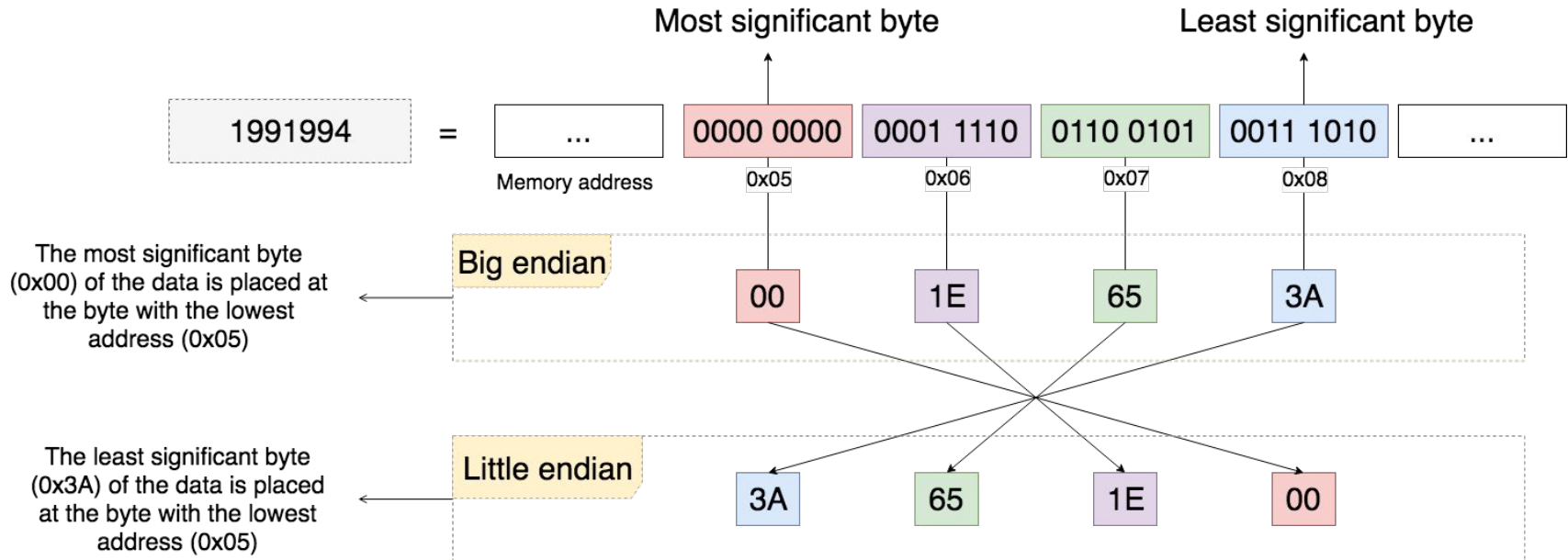
# Порядок байтов



Big endian и Little endian

123 - Big endian

321 - Little endian



# Как это влияет на unsafe

Перевод `[]byte{...}` → коварен

Little endian (младший бит вначале)

`[]byte{0x00, 0x01} → uint16 = 0x0100 = 256`

Big endian (старший бит вначале)

`[]byte{0x00, 0x01} → uint16 = 0x0001 = 1`

# Порядок байтов

```
//  
//go:build 386 || amd64 || arm || arm64 || loong64 || ppc64le || mips64le || mipsle || riscv64 || was  
  
package os  
  
const isBigEndian = false
```

```
//  
//go:build ppc64 || s390x || mips || mips64  
  
package os  
  
const isBigEndian = true
```



```
var (  
    modbusFrame = ModbusHeader{  
        TransactionID: 0,  
        ProtocolID:    0,  
        Length:        5,  
        Address:        5,  
        FunctionCode:  3,  
        ByteCount:      3,  
    }  
    registers = []uint16{255, 255, 0}  
)
```

Из структуры в байты

```
func getBytes(pdu ModbusHeader, registers []uint16) []byte {  
    if len(registers) < 1 {  
        return nil  
    }  
    headerSize := unsafe.Offsetof(pdu.ByteCount) + unsafe.Sizeof(pdu.ByteCount)  
    addrH := (*byte)(unsafe.Pointer(&pdu))  
    bytesHeader := unsafe.Slice(addrH, headerSize)  
  
    addrR := (*byte)(unsafe.Pointer(&registers[0]))  
    bytesRegisters := unsafe.Slice(addrR, pdu.ByteCount*uint8(unsafe.Sizeof(registers[0])))  
    return append(bytesHeader, bytesRegisters...)  
}
```

Понадобится ли мне сейчас в  
работе `unsafe`?

Скорее всего нет, но...

1. Особые кейсы
2. Устройство памяти
3. Адресная арифметика
4. Читать исходники go

