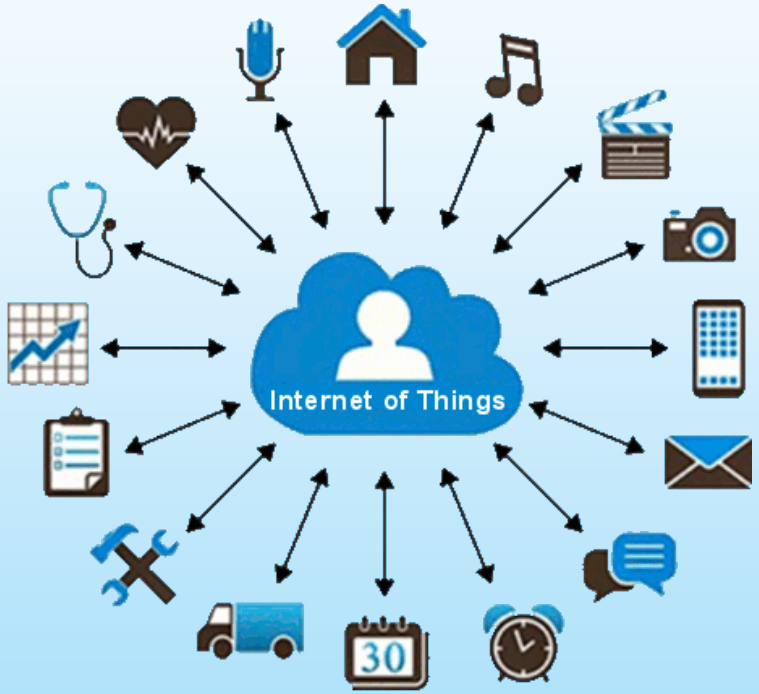
Sensor Networks – Project 1



Coding guidelines

Coding guidelines – C / C++
Know your types!

- Original types (char, int…) have implementation-defined lengths!
  - `char` >= 8 bits, `short int` / `int` >= 16 bits, `long int` >= 32 bits…
- Fixed width integer types since C99 (`stdint.h`) / C++11(`cstdint`)
  - `int`***N***`_t`, `uint`***N***`_t` (***N*** = 8, 16, 32, 64)
- Floating point types have also implementation-defined lengths
  - But <u>usually</u>: `float` → IEEE 754 single precision (binary32)
`double` → IEEE 754 double precision (binary64)
  - Know whether your processor has an FPU or not!
- Special types for specific tasks… with implementation-defined lengths
  - `size_t`: unsigned integer type returned by `sizeof()` (more on that now)
    - `stddef.h` / `cstddef`, >= 16 bits since C99 / C++11

Coding guidelines – C / C++
Know your types! (cont.)

- Operators and keywords that work on types:
  - `sizeof(`*`type`*`), sizeof(`*`expr`*`)`: return the size in bytes of type *`type`* or expression *`expr`* (e.g. a variable)
    - Its return type is `size_t`
  - `typeof(`*`expr`*`)` is equivalent to the type of expression *`expr`*
    - Non-standard extension in GCC and clang for both C and C++
    - Standard in C since C23
  - `decltype(`*`expr`*`)` is similar to (but more complex than) `typeof(`*`expr`*`)`
    - C++, since C++11
- https://cppreference.com is your friend!
  - For both C and C++, despite its name

Coding guidelines – C / C++
So you want to build a (binary) data packet?

```
char tx_buffer[30];

int s1_value = sensor1.read(); // 8 bit
int s2_value = sensor2.read(); // 32 bit
int s3_value = sensor3.read(); // 16 bit

tx_buffer[0] = s1_value & 0xff;
tx_buffer[1] = s2_value & 0xff;
tx_buffer[2] = (s2_value >> 8) & 0xff;
tx_buffer[3] = (s2_value >> 16) & 0xff;
tx_buffer[4] = (s2_value >> 24) & 0xff;
tx_buffer[5] = s3_value & 0xff;
tx_buffer[6] = (s3_value >> 8) & 0xff;

lorawan.send([…], tx_buffer, 30, […]);
```

Coding guidelines – C / C++
Use the right types

```
char tx_buffer[30];

uint8_t   s1_value = sensor1.read();
int32_t   s2_value = sensor2.read();
uint16_t  s3_value = sensor3.read();

tx_buffer[0] = s1_value & 0xff;
tx_buffer[1] = s2_value & 0xff;
tx_buffer[2] = (s2_value >> 8) & 0xff;
tx_buffer[3] = (s2_value >> 16) & 0xff;
tx_buffer[4] = (s2_value >> 24) & 0xff;
tx_buffer[5] = s3_value & 0xff;
tx_buffer[6] = (s3_value >> 8) & 0xff;

lorawan.send([…], tx_buffer, 30, […]);
```

Coding guidelines – C / C++
Define data only in one place, and name it

```
constexpr size_t TX_BUFFER_SIZE = 30;
uint8_t tx_buffer[TX_BUFFER_SIZE];

uint8_t   s1_value = sensor1.read();
int32_t   s2_value = sensor2.read();
uint16_t  s3_value = sensor3.read();

tx_buffer[0] = s1_value & 0xff;
tx_buffer[1] = s2_value & 0xff;
tx_buffer[2] = (s2_value >> 8) & 0xff;
tx_buffer[3] = (s2_value >> 16) & 0xff;
tx_buffer[4] = (s2_value >> 24) & 0xff;
tx_buffer[5] = s3_value & 0xff;
tx_buffer[6] = (s3_value >> 8) & 0xff;

lorawan.send([…], tx_buffer, TX_BUFFER_SIZE, […]);
```

Coding guidelines – C / C++
Define data only in one place, and name it

```
constexpr size_t TX_BUFFER_SIZE = 7;
uint8_t tx_buffer[TX_BUFFER_SIZE];

uint8_t   s1_value = sensor1.read();
int32_t   s2_value = sensor2.read();
uint16_t  s3_value = sensor3.read();

tx_buffer[0] = s1_value & 0xff;
tx_buffer[1] = s2_value & 0xff;
tx_buffer[2] = (s2_value >> 8) & 0xff;
tx_buffer[3] = (s2_value >> 16) & 0xff;
tx_buffer[4] = (s2_value >> 24) & 0xff;
tx_buffer[5] = s3_value & 0xff;
tx_buffer[6] = (s3_value >> 8) & 0xff;

lorawan.send([…], tx_buffer, TX_BUFFER_SIZE, […]);
```

See how much easier it is now to change it!

Coding guidelines – C / C++
Write self-adjusting code

```
constexpr size_t TX_BUFFER_SIZE = 7;
uint8_t tx_buffer[TX_BUFFER_SIZE];

uint8_t   s1_value = sensor1.read();
int32_t   s2_value = sensor2.read();
uint16_t  s3_value = sensor3.read();

size_t pos = 0;
tx_buffer[pos++] = s1_value & 0xff;
tx_buffer[pos++] = s2_value & 0xff;
tx_buffer[pos++] = (s2_value >> 8) & 0xff;
tx_buffer[pos++] = (s2_value >> 16) & 0xff;
tx_buffer[pos++] = (s2_value >> 24) & 0xff;
tx_buffer[pos++] = s3_value & 0xff;
tx_buffer[pos++] = (s3_value >> 8) & 0xff;

lorawan.send([…], tx_buffer, TX_BUFFER_SIZE, […]);
```

Coding guidelines – C / C++
Write self-adjusting code

```
constexpr size_t TX_BUFFER_SIZE = 7;
uint8_t tx_buffer[TX_BUFFER_SIZE];

uint8_t   s1_value = sensor1.read();
int32_t   s2_value = sensor2.read();
uint16_t  s3_value = sensor3.read();

size_t pos = 0;
tx_buffer[pos++] = s1_value & 0xff;
tx_buffer[pos++] = s2_value & 0xff;
tx_buffer[pos++] = (s2_value >> 8) & 0xff;
tx_buffer[pos++] = (s2_value >> 16) & 0xff;
tx_buffer[pos++] = (s2_value >> 24) & 0xff;
tx_buffer[pos++] = s3_value & 0xff;
tx_buffer[pos++] = (s3_value >> 8) & 0xff;

lorawan.send([…], tx_buffer, pos, […]);
```

Now you are sure to always send the right amount of data!

Coding guidelines – C / C++
Write self-adjusting code

```
constexpr size_t TX_BUFFER_SIZE = 30;
uint8_t tx_buffer[TX_BUFFER_SIZE];

uint8_t   s1_value = sensor1.read();
int32_t   s2_value = sensor2.read();
uint16_t  s3_value = sensor3.read();

size_t pos = 0;
tx_buffer[pos++] = s1_value & 0xff;
tx_buffer[pos++] = s2_value & 0xff;
tx_buffer[pos++] = (s2_value >> 8) & 0xff;
tx_buffer[pos++] = (s2_value >> 16) & 0xff;
tx_buffer[pos++] = (s2_value >> 24) & 0xff;
tx_buffer[pos++] = s3_value & 0xff;
tx_buffer[pos++] = (s3_value >> 8) & 0xff;

lorawan.send([…], tx_buffer, pos, […]);
```

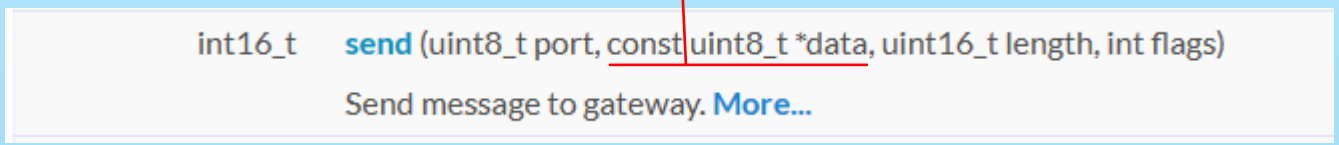
You can even make room for safe testing

Coding guidelines – C / C++
Better check twice than never

```
constexpr size_t TX_BUFFER_SIZE = 30;
uint8_t tx_buffer[TX_BUFFER_SIZE];

uint8_t   s1_value = sensor1.read();
int32_t   s2_value = sensor2.read();
uint16_t  s3_value = sensor3.read();

size_t pos = 0;
tx_buffer[pos++] = s1_value & 0xff;
tx_buffer[pos++] = s2_value & 0xff;
tx_buffer[pos++] = (s2_value >> 8) & 0xff;
tx_buffer[pos++] = (s2_value >> 16) & 0xff;
tx_buffer[pos++] = (s2_value >> 24) & 0xff;
tx_buffer[pos++] = s3_value & 0xff;
tx_buffer[pos++] = (s3_value >> 8) & 0xff;

MBED_ASSERT(pos <= sizeof(tx_buffer));
lorawan.send([…], tx_buffer, pos, […]);
```

Coding guidelines – C / C++
What if I let the compiler do (some of) the work?

```
struct {
    uint8_t   s1_value;
    int32_t   s2_value;
    uint16_t  s3_value;
} frame_data_t;

frame_data_t tx_buffer;

tx_buffer.s1_value = sensor1.read();
tx_buffer.s2_value = sensor2.read();
tx_buffer.s3_value = sensor3.read();

lorawan.send([…], (uint8_t *) &tx_buffer, 7, […]);
```



Coding guidelines – C / C++
What if I let the compiler do all of the work?

```
struct {
    uint8_t s1_value;
    int32_t s2_value;
    uint16_t s3_value;
} frame_data_t;

frame_data_t tx_buffer;

tx_buffer.s1_value = sensor1.read();
tx_buffer.s2_value = sensor2.read();
tx_buffer.s3_value = sensor3.read();

lorawan.send([…], (uint8_t *) &tx_buffer, sizeof(tx_buffer), […]);
```

Beware of padding!!!

!= 7

Coding guidelines – C / C++
What if I let the compiler do all of the work?

```
struct __attribute__((packed)) {
    uint8_t   s1_value;
    int32_t   s2_value;
    uint16_t  s3_value;
} frame_data_t;

frame_data_t tx_buffer;

tx_buffer.s1_value = sensor1.read();
tx_buffer.s2_value = sensor2.read();
tx_buffer.s3_value = sensor3.read();

lorawan.send([…], (uint8_t *) &tx_buffer, sizeof(tx_buffer), […]);
```

It works nicely, but **it's not portable!**

But then again, nothing is *completely* portable in embedded systems

See: - ARM Compiler User's Guide v6.16, section 4.5: Packing data structures
- GCC Manual, https://gcc.gnu.org/onlinedocs/gcc/Common-Type-Attributes.html#index-packed-type-attribute
- Clang documentation, https://clang.llvm.org/docs/AttributeReference.html#packed

Coding guidelines – C / C++
Other *smart* options…

```
constexpr size_t TX_BUFFER_SIZE = 30;
uint8_t tx_buffer[TX_BUFFER_SIZE];

uint8_t   s1_value = sensor1.read();
int32_t   s2_value = sensor2.read();
uint16_t  s3_value = sensor3.read();

size_t pos = 0;
*(typeof(s1_value) *) &tx_buffer[pos] = s1_value;   pos += sizeof(s1_value);
*(typeof(s2_value) *) &tx_buffer[pos] = s2_value;   pos += sizeof(s2_value);
*(typeof(s3_value) *) &tx_buffer[pos] = s3_value;   pos += sizeof(s3_value);

lorawan.send([…], tx_buffer, pos, […]);
```

Beware of illegal unaligned memory accesses!!!

(which are heavily dependent on the processor microarchitecture)

Anyway it's a good idea to order the fields in a way that avoids (legal) unaligned memory accesses, even if you don't use this technique (e.g. int32_t first, uint16_t next, uint8_t last)

Coding guidelines – C / C++
Good enough can be better than the absolute best!

```
constexpr size_t TX_BUFFER_SIZE = 30;
uint8_t tx_buffer[TX_BUFFER_SIZE];

uint8_t   s1_value = sensor1.read();
int32_t   s2_value = sensor2.read();
uint16_t  s3_value = sensor3.read();

size_t pos = 0;
tx_buffer[pos++] = s1_value & 0xff;
tx_buffer[pos++] = s2_value & 0xff;
tx_buffer[pos++] = (s2_value >> 8) & 0xff;
tx_buffer[pos++] = (s2_value >> 16) & 0xff;
tx_buffer[pos++] = (s2_value >> 24) & 0xff;
tx_buffer[pos++] = s3_value & 0xff;
tx_buffer[pos++] = (s3_value >> 8) & 0xff;

MBED_ASSERT(pos <= sizeof(tx_buffer));
lorawan.send([…], tx_buffer, pos, […]);
```

If the code gets too large, chop it in (static) functions!

Coding guidelines – C / C++
What about floats?

```
constexpr size_t TX_BUFFER_SIZE = 30;
uint8_t tx_buffer[TX_BUFFER_SIZE];

float latitude = 40.382278f;

size_t pos = 0;
tx_buffer[pos++] = latitude & 0xff;
tx_buffer[pos++] = (latitude >> 8) & 0xff;
tx_buffer[pos++] = (latitude >> 16) & 0xff;
tx_buffer[pos++] = (latitude >> 24) & 0xff;

lorawan.send([…], tx_buffer, pos, […]);
```

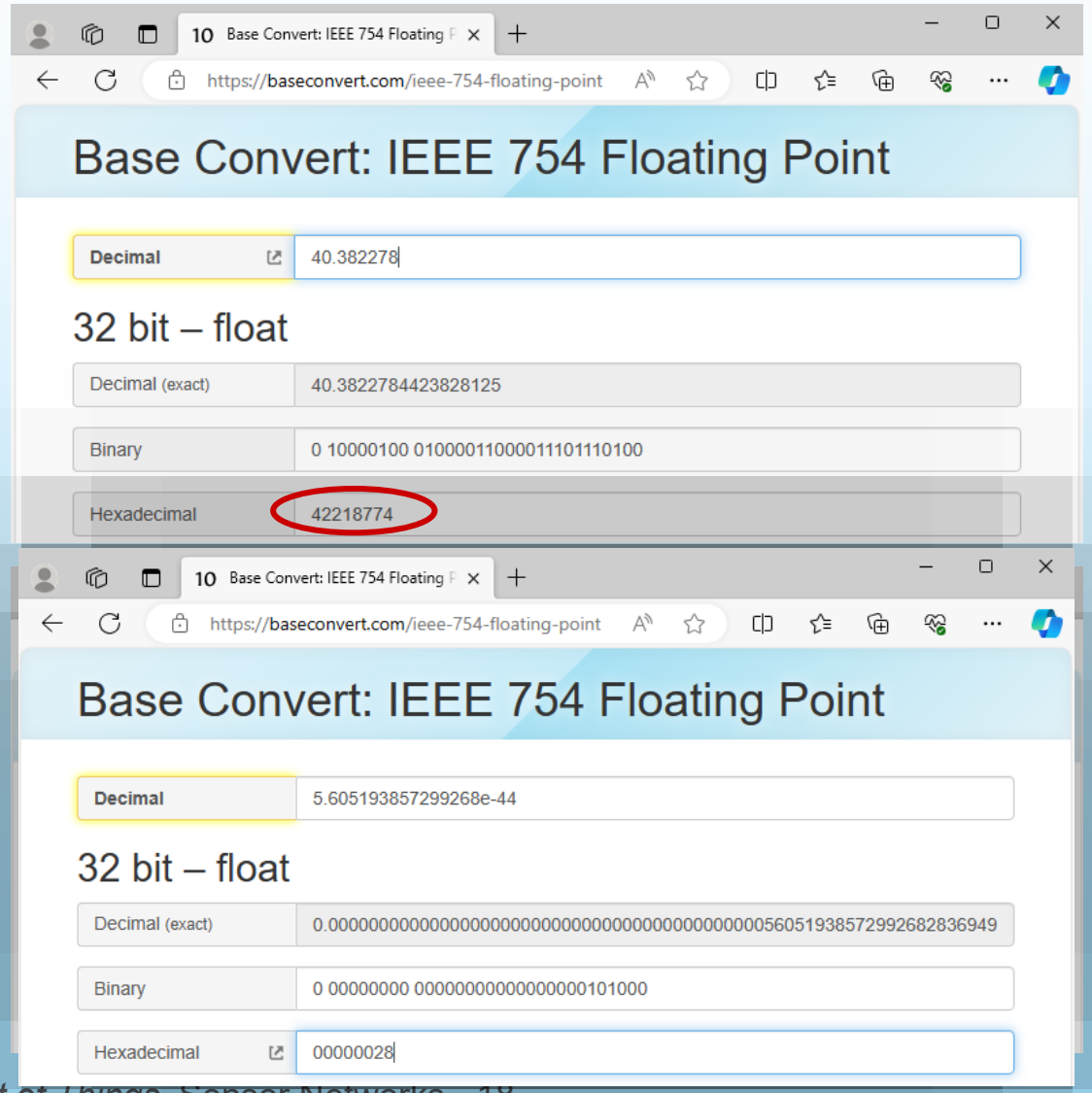
```
356    float latitude = 40.382278f;
357
358    size_t pos = 0;
359    tx_buffer[pos++] = latitude & 0xff;
360    tx_buffer[pos++] = (latitude >> 8) & 0xff;
361    tx_buffer[pos++] = (latitude >> 16) & 0xff;
362    tx_buffer[pos++] = (latitude >> 24) & 0xff;
```

Problems × Output ×

```
Building project sn-lorawan-instr (DISCO_L072CZ_LRWAN1, ARMC6)
Scan: sn-lorawan-instr
Compile [100.0%]: main.cpp
[Error] main.cpp@359,33: invalid operands to binary expression ('float' and 'int')
[Error] main.cpp@360,34: invalid operands to binary expression ('float' and 'int')
[Error] main.cpp@361,34: invalid operands to binary expression ('float' and 'int')
[Error] main.cpp@362,34: invalid operands to binary expression ('float' and 'int')
[ERROR] ./main.cpp:359:33: error: invalid operands to binary expression ('float' and 'int')
tx_buffer[pos++] = latitude & 0xff;
~~~~~~~~ ^ ~~~~
./main.cpp:360:34: error: invalid operands to binary expression ('float' and 'int')
tx_buffer[pos++] = (latitude >> 8) & 0xff;
~~~~~~~~ ^  ~
./main.cpp:361:34: error: invalid operands to binary expression ('float' and 'int')
tx_buffer[pos++] = (latitude >> 16) & 0xff;
~~~~~~~~ ^  ~~
./main.cpp:362:34: error: invalid operands to binary expression ('float' and 'int')
tx_buffer[pos++] = (latitude >> 24) & 0xff;
~~~~~~~~ ^  ~~
4 errors generated.
```

Coding guidelines – C / C++
And if I try to use it as an `int` type?

```
constexpr size_t TX_BUFFER_SIZE = 30;
uint8_t tx_buffer[TX_BUFFER_SIZE];

float    latitude     = 40.382278f;
uint32_t latitude_u32 = latitude;

size_t pos = 0;
tx_buffer[pos++] = latitude_u32 & 0xff;
tx_buffer[pos++] = (latitude_u32 >> 8) & 0xff;
tx_buffer[pos++] = (latitude_u32 >> 16) & 0xff;
tx_buffer[pos++] = (latitude_u32 >> 24) & 0xff;

lorawan.send([…], tx_buffer, pos, […]);
```



The compiler does “the right thing” and **converts** the float value to uint32_t, but 0x42218774 != 0x00000028

Coding guidelines – C / C++
And if I really use it as an `int` type?

```
constexpr size_t TX_BUFFER_SIZE = 30;
uint8_t tx_buffer[TX_BUFFER_SIZE];

float    latitude   = 40.382278f;
uint32_t lat_as_u32 = *(uint32_t *) &latitude;

size_t pos = 0;
tx_buffer[pos++] = lat_as_u32 & 0xff;
tx_buffer[pos++] = (lat_as_u32 >> 8) & 0xff;
tx_buffer[pos++] = (lat_as_u32 >> 16) & 0xff;
tx_buffer[pos++] = (lat_as_u32 >> 24) & 0xff;

lorawan.send([…], tx_buffer, pos, […]);
```

No conversions, just a reinterpretation through the use of pointers

Coding guidelines – C / C++
What if I let the compiler do the work?

```
constexpr size_t TX_BUFFER_SIZE = 30;
uint8_t tx_buffer[TX_BUFFER_SIZE];

struct __attribute__((packed)) {
    float     latitude;
    int32_t   s2_value;
    uint16_t  s3_value;
    uint8_t   s1_value;
} frame_data_t;

frame_data_t *tx_struct_buffer = (frame_data_t *) tx_buffer;

tx_struct_buffer.latitude = 40.382278f;
tx_struct_buffer.s1_value = sensor1.read();
tx_struct_buffer.s2_value = sensor2.read();
tx_struct_buffer.s3_value = sensor3.read();

lorawan.send([…], tx_buffer, sizeof(*tx_struct_buffer), […]);
```

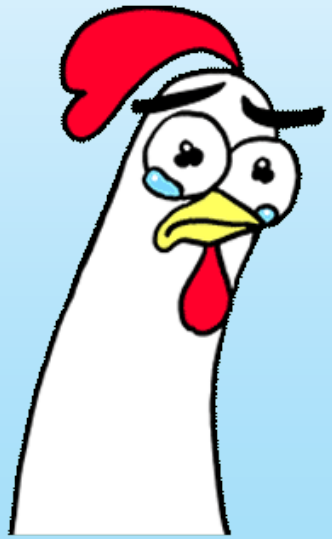
Just remember, packed structs are non-standard and therefore compiler dependent

Coding guidelines – C / C++
Write expressive code

Optimize the algorithm, not the sentences

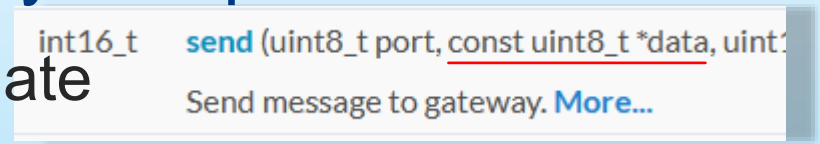
- Prefer, by all means, readable (expressive) code over seemingly clever optimizations
  - Modern compilers are *really* good at optimizing code

```
constexpr size_t TX_BUFFER_SIZE = 30;
uint8_t tx_buffer[TX_BUFFER_SIZE];

float    latitude   = 40.382278f;

size_t pos = 0;
tx_buffer[pos++] = (*(uint32_t *) &latitude) & 0xff;
tx_buffer[pos++] = ((*(uint32_t *) &latitude) >> 8) & 0xff;
tx_buffer[pos++] = ((*(uint32_t *) &latitude) >> 16) & 0xff;
tx_buffer[pos++] = ((*(uint32_t *) &latitude) >> 24) & 0xff;

lorawan.send([…], tx_buffer, pos, […]);
```



Coding guidelines – C / C++
Write expressive code

- Optimize the algorithm, not the sentences
  - Prefer, by all means, readable (expressive) code over seemingly clever optimizations
    - Modern compilers are *really* good at optimizing code
- But some constructs enhance both readability and performance
  - Example: use the `const` qualifier when appropriate



  - Or its relative `constexpr`
    - Since C++11 / C23
    - Guarantees compile-time evaluation
    - Can substitute many preprocessor macros

Coding guidelines – C / C++
Know your system!

- Be aware of processor or system limitations
  - E.g. the Cortex M0+ in B-L072Z-LRWAN1 does *not* have an FPU
    - FP operations are emulated in software!
- Know your goals and restrictions
  - So you can make informed decisions when a trade-off is unavoidable
    - E.g. what's your energy budget *and* how long must it last?
- Measure what you do if at all possible
  - Complex systems show surprising behaviour from time to time
    - It's almost impossible to predict the effect of all interactions

Coding guidelines – Lua
Intro to Lua

- Scripting language
  - Pontifical Catholic University of Rio de Janeiro, MIT License
- ResIOT uses Lua 5.1
- Resources
  - Lua website: https://lua.org/
  - Book: “Programming in Lua”, by Roberto Ierusalimschy
    - 1st edition freely available: https://lua.org/pil/contents.html
  - Language reference: https://lua.org/manual/5.1/
  - ResIOT resources: https://docs.resiot.io
    - ResIOT API reference: Section “Script Lua 5.1 Functions”
    - Examples: Section “Script Lua 5.1 Scenes Examples”
      - Especially example 5, “Payload Parsing”

Coding guidelines – Lua
Intro to Lua (II)

- Dynamically typed
  - Eight basic types: nil, boolean, number, string, userdata, function, thread, and table
  - ResIOT adds a “byte array”
- Many syntax elements taken from C
  - E.g. logical operators (==, !=…), hex constant notation (`0x3f`)…
- Some elements that are different from other languages
  - Sentences *can* end in semicolon, but it is optional and makes no difference
  - Comments
    - Line comments start with two dashes: `--This is a comment`
    - Block comments are enclosed within `--[[` and `]]--`
  - Concatenation operator is two dots: `"a" .. "b"` → `"ab"`

Coding guidelines – Lua
Intro to Lua (III)

Control structures

- `if` *cond* `then` … `elseif` *cond* `then` … `else` … `end`
- `while` *cond* `do` … `end`
- `repeat` … `until` *cond*
- `for` *var* `=` *start*`,` *end*`[,` *incr*`]` `do` … `end`
- `for` *vars* `in` *expr* `do` … `end`

Functions

- `function` *name*`(`*params*`)` … `end`
- Can return multiple results, have variable and named arguments, be used as objects/values for functional programming…
  - E.g. `worked, err = resiot_setnodevalue(…)` returns two values

Coding guidelines – Lua
Intro to Lua (IV)

- Data structures are implemented using tables
  - key / value pairs, like Python dicts or C++ maps
- Arrays
  - ```
    a = {} -- new array
    for i=1, 1000 do
      a[i] = 0
    end
    ```
  - `squares = {1, 4, 9}  -- squares[1] == 1, squares[2] == 4, etc.`
  - The first element has index 1 by convention
- Variables with fields / structures / classes…
  - ```
    a.x = 23
    a.y = "foobar"
    ```

Coding guidelines – Lua
Intro to Lua (V)

- Libraries
  - string, math…
  - `print(string.format("Value of a is %d", a))`
- Many, many other things
  - Check the resources if interested

Coding guidelines – Lua
ResIOT Lua 5.1 library

- Introduces the byte array (BA, ba) type
  - Use it just like a regular array
- Check the examples to get a feel
  - Especially “Payload Parsing”, https://docs.resiot.io/Example5_Payload_Parsing_and_Saving/
  - Binary data (e.g. packet payload) encoded in hex strings, translated to BA or plain strings
- The function names are sometimes awful, but at least they all start with `resiot_`...
  - It’s `resiot_ba2float32LE(…)`, but `resiot_ba2intLE32(…)`
  - "`int`" means *unsigned* integer in function names... except when it means signed
    - `resiot_int16(…)` and `resiot_int32(…)` convert an unsigned integer to a signed one
      - Ain’t it obvious?
  - There’s `resiot_ba2sintBE32(…)` (meaning signed int) but no `resiot_ba2sintLE32(…)`
  - To get a signed LE int from a byte array you must use `resiot_int32(resiot_ba2intLE32(…))`
    - Of course
  - `resiot_float322baLE(…)`? REALLY???