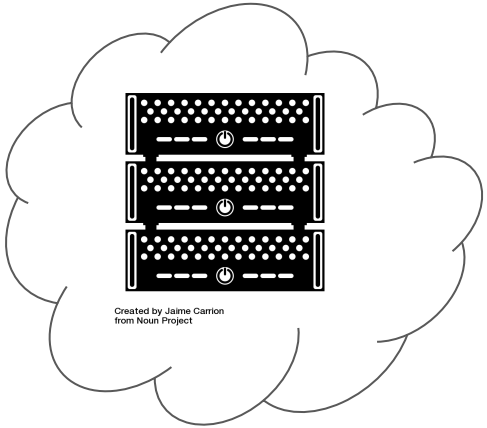
Lessons Learned From An Embedded RTPS in Modern C++

Jackie Kay
9/22/2016

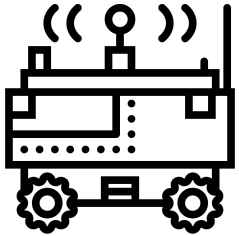
twitter: @jackayline github: jacquelinekay

Getting robots to talk to each other

Created by B. Agustín Amenábar Larraín
from Noun Project



Created by Jaime Carrion
from Noun Project

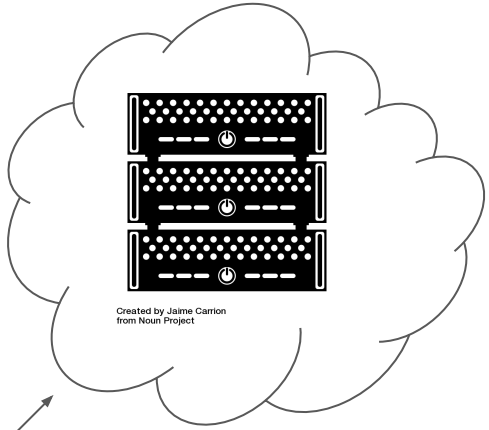


Created by Oliviu Stoian
from Noun Project

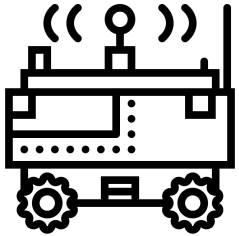
Created by MarkieAnn Packer
from Noun Project

Getting robots to talk to each other

Created by B. Agustín Amenábar Larraín
from Noun Project



Created by Jaime Carrion
from Noun Project

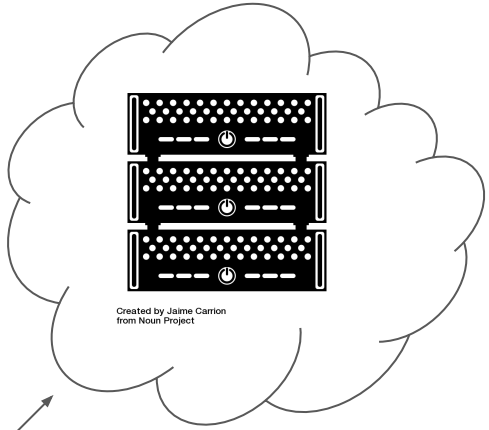


Created by Oliviu Stoian
from Noun Project

Created by MarkieAnn Packer
from Noun Project

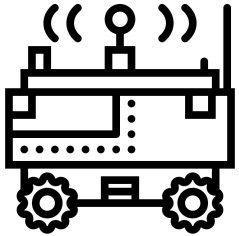
Getting robots to talk to each other

Created by B. Agustín Amenábar Larraín
from Noun Project



Created by Jaime Carrion
from Noun Project

Power managers
etc...



Created by Oliviu Stoian
from Noun Project

Central
CPU

Created by MarkieAnn Packer
from Noun Project

Sensors

Motor Controllers

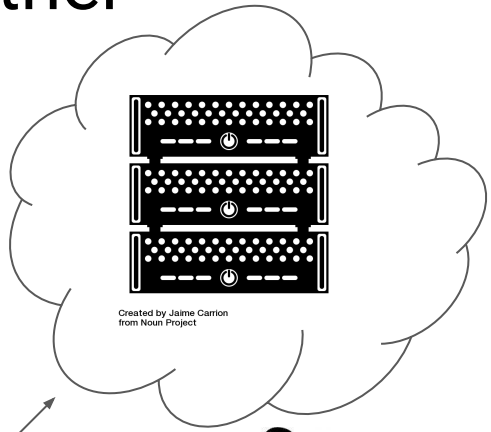
What’s an embedded system?

Category	Example system	Example chip	Memory	CPU speed	Example ISA
Small desktop	Intel NUC	Sandy Bridge	16GB	2.7 GHz	64-bit Intel
Embedded powerhouse	Nvidia TX1	A57	5GB	2 GHz	64-bit ARM
Smartphone	Odroid	Snapdragon	2GB	2 GHz	64-bit ARM
Sensor controller	Pixhawk	Cortex M7	2MB (flash)	216 MHz	32-bit ARM
Low-power, sleeps a lot	MSP430 Launchpad	MSP430	512 KB	25MHz	16-bit AVR, 8-bit AVR

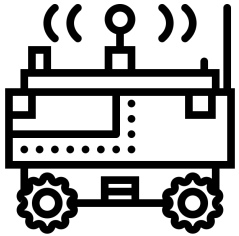
Source: Wikipedia and respective vendor websites

Getting computers to talk to each other

Created by B. Agustín Amenábar Larraín
from Noun Project



Created by Jaime Carrion
from Noun Project



Created by Oliviu Stoian
from Noun Project

Power managers
etc...

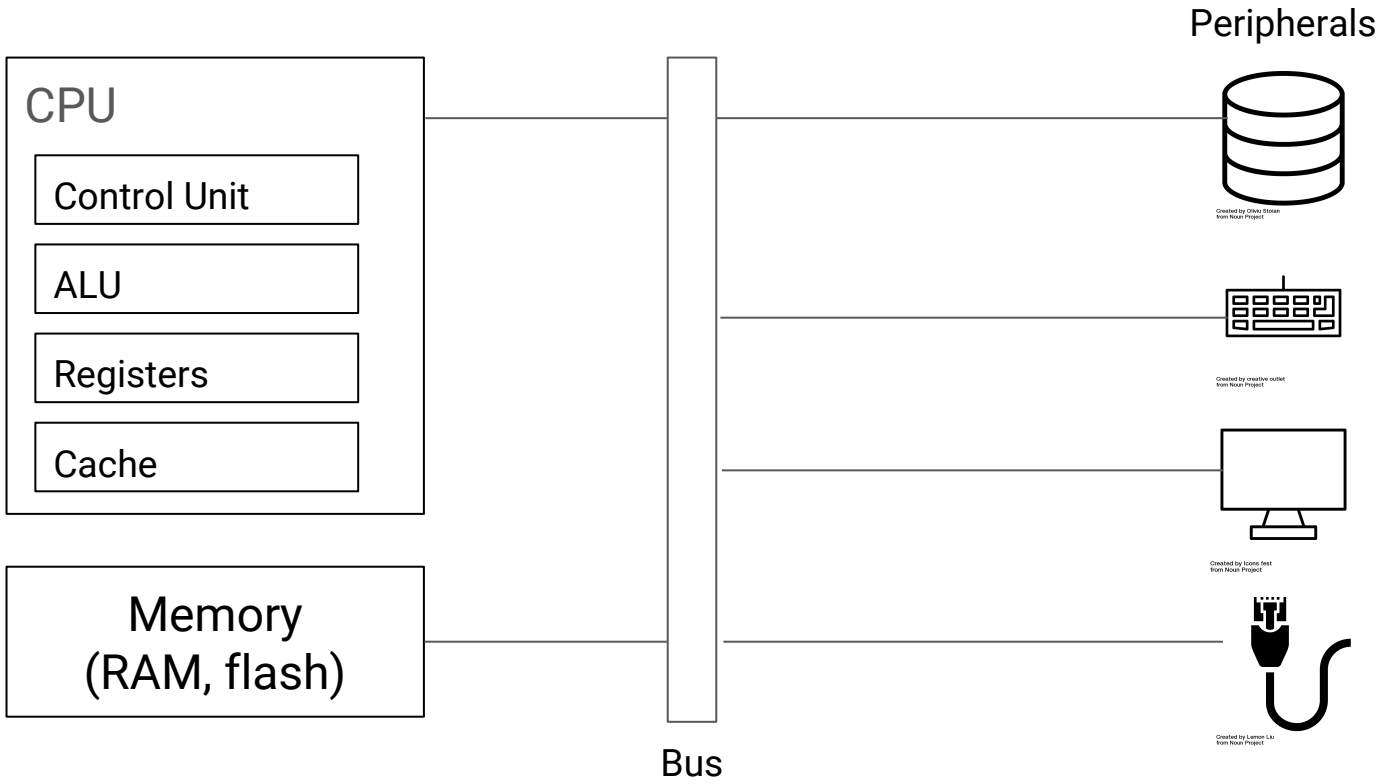
Central
CPU

Created by MarkieAnn Packer
from Noun Project

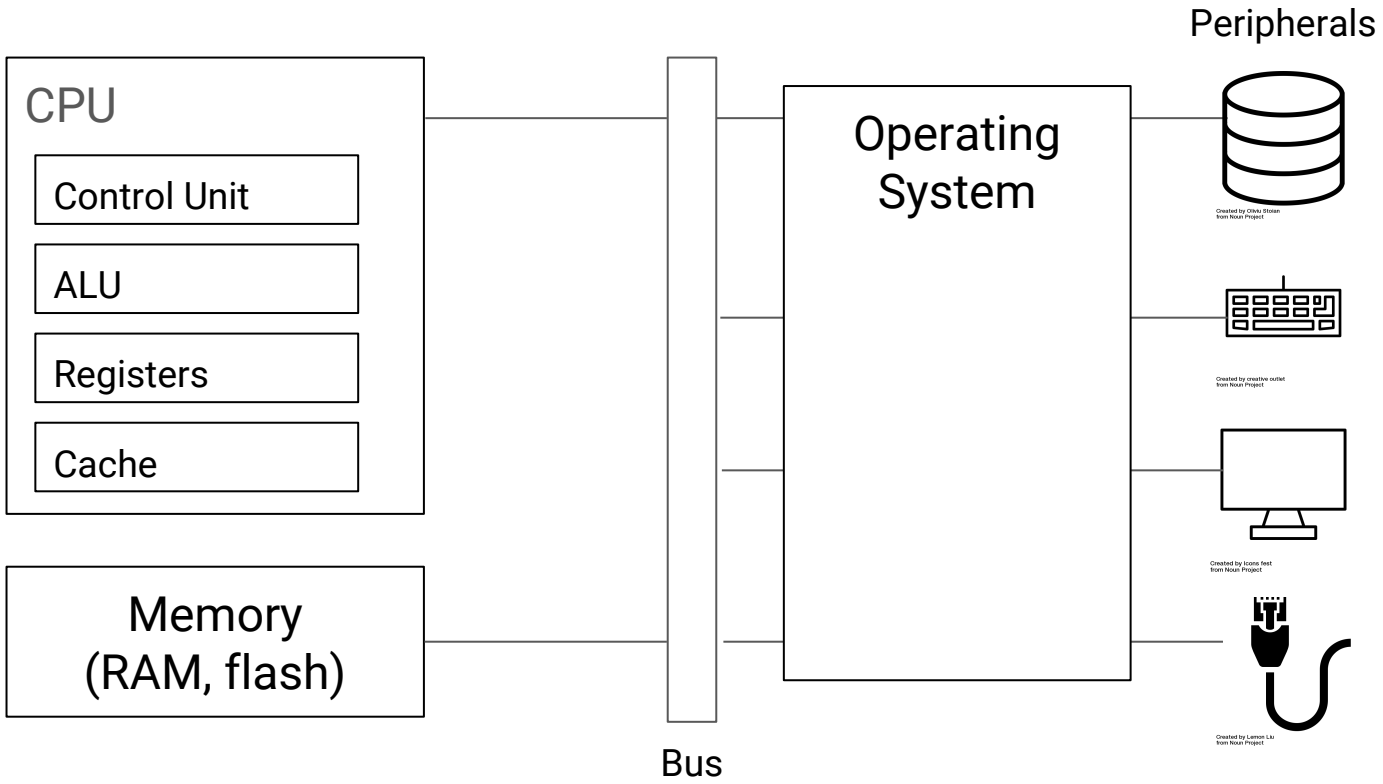
Sensors

Motor Controllers

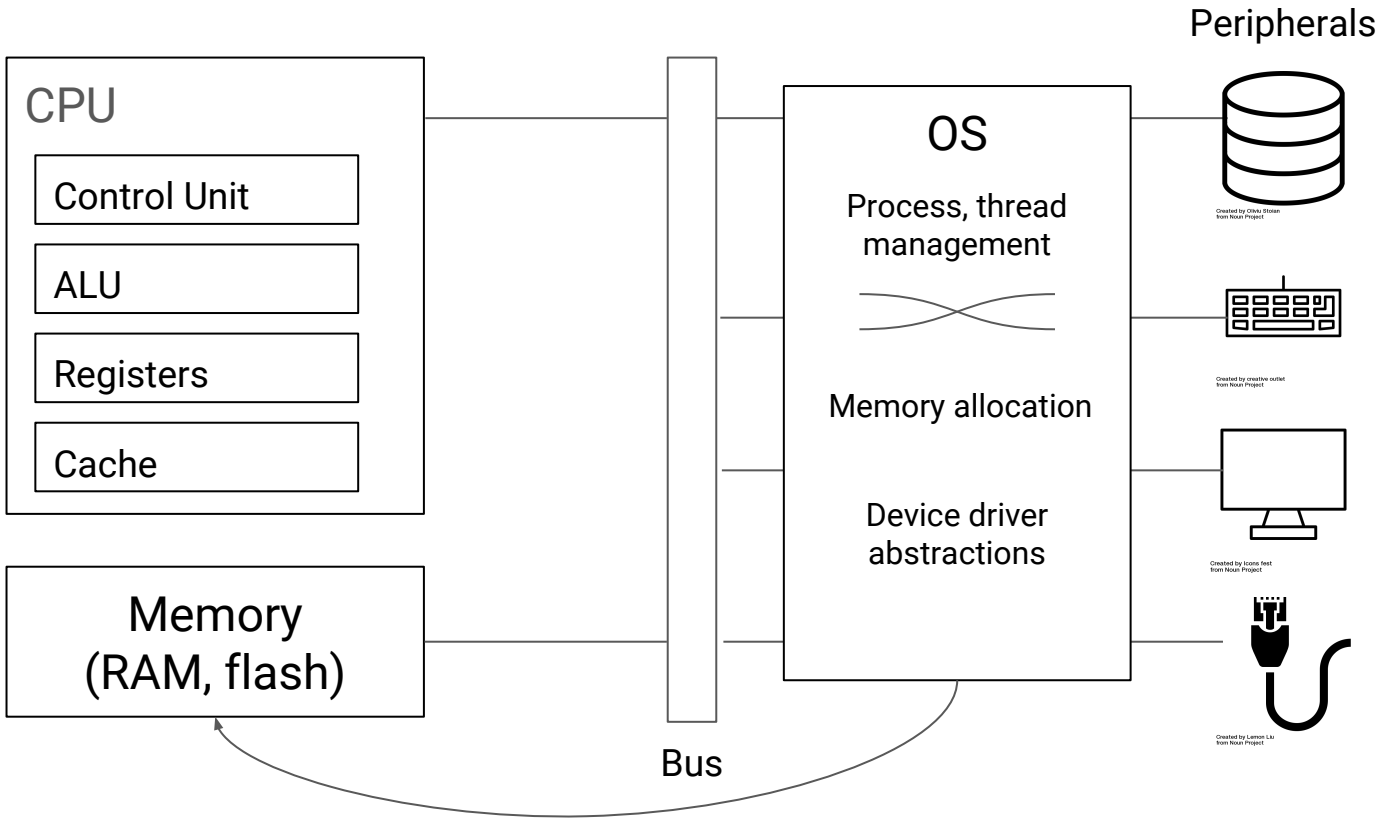
What’s connected to the (micro)processor?



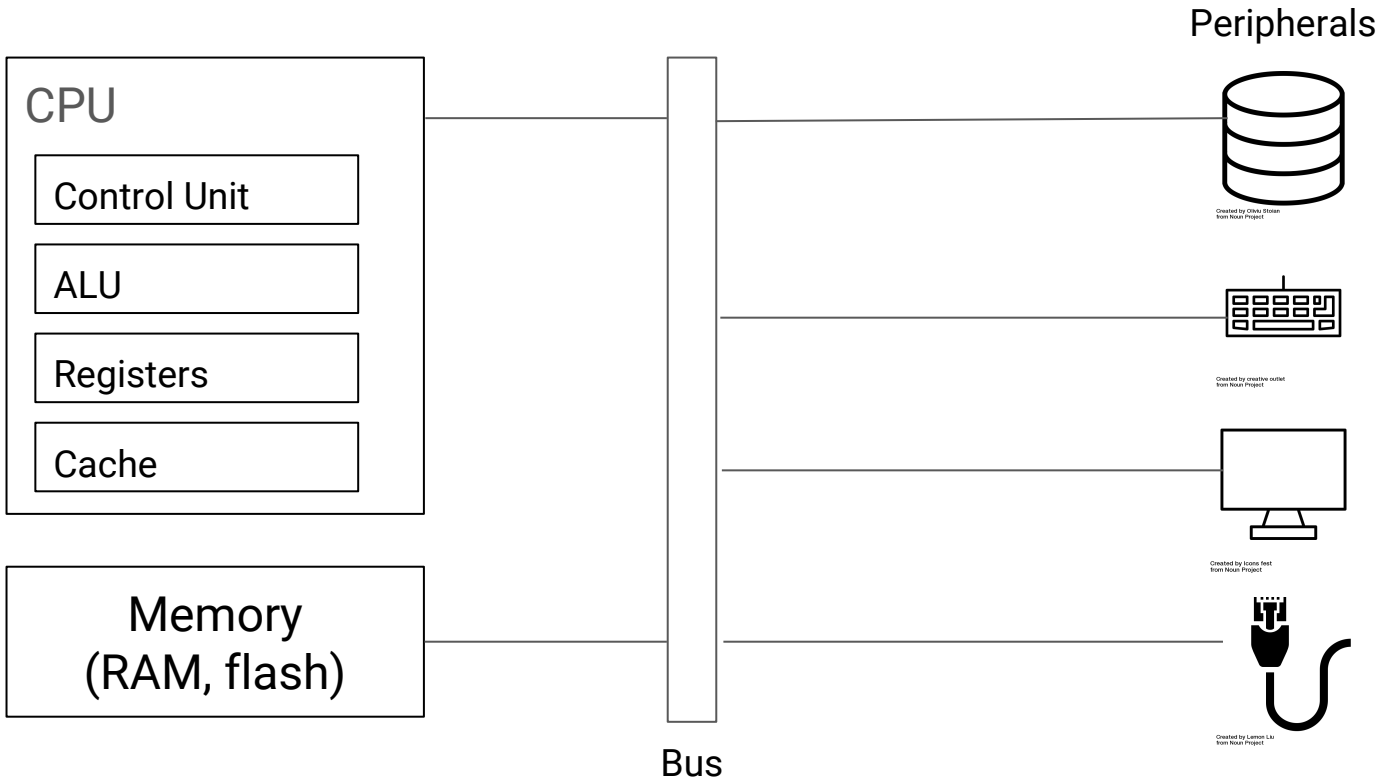
What’s connected to the (micro)processor?



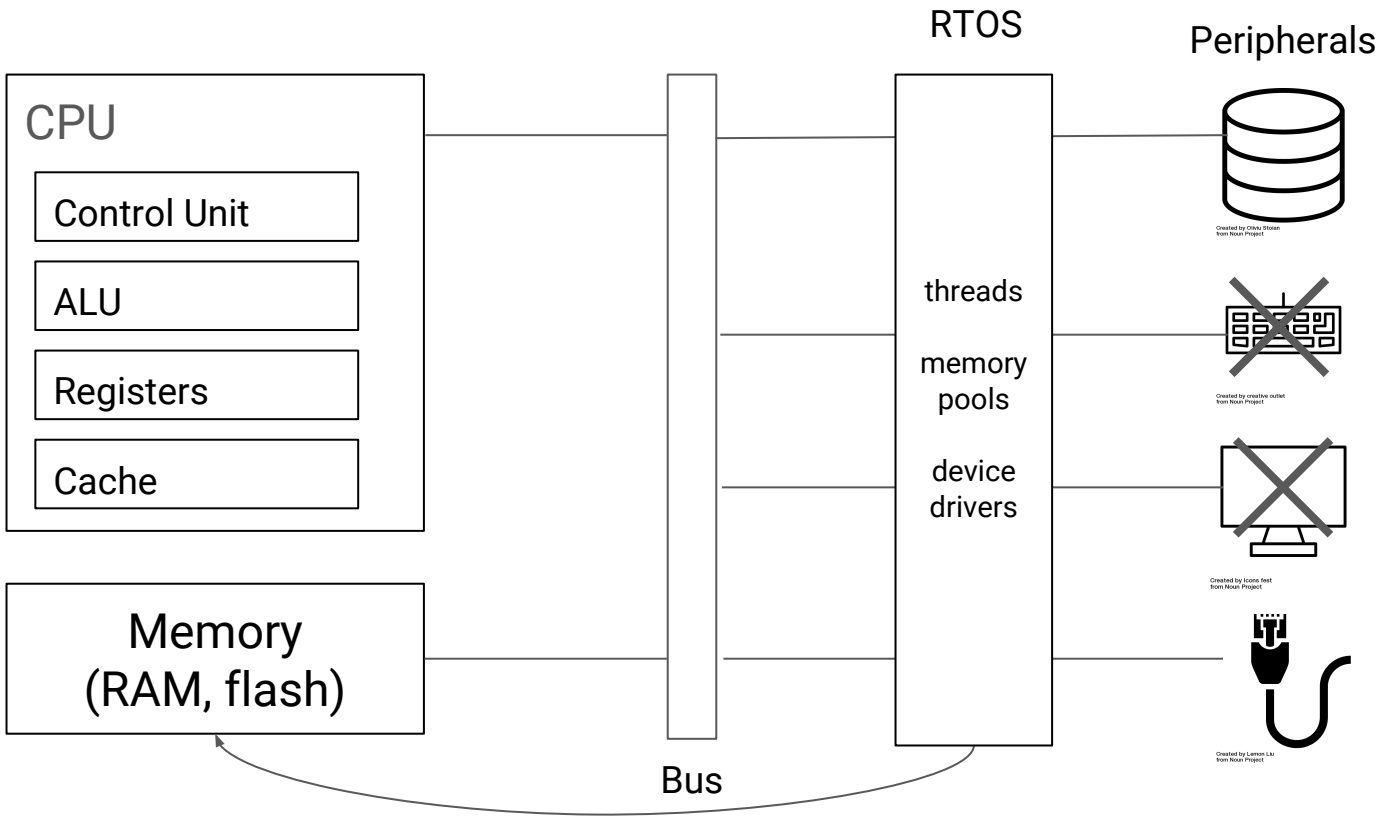
What’s in an OS?



Bare metal


Peripherals
CPU
Control Unit
ALU
Registers
Cache
Memory
(RAM, flash)
Bus

A thin sheath of RTOS?



How thin are we talking here?

`size /vmlinuz:` get the size of the Linux kernel image on my machine

text (bytes)	data (bytes)	bss (bytes)
7044960	0	16166560

How thin are we talking here?

`size /vmlinuz:` get the size of the Linux kernel image on my machine

text (bytes)	data (bytes)	bss (bytes)
7044960	0	16166560

`size cmsis_conformance_test:` get size of cross-compiled binary testing FreeRTOS thread

instructions	Global, static local	Scoped, stack-allocated
text (bytes)	data (bytes)	bss (bytes)
34584	136	7284

Today’s RTOS/Platform:

CMSIS wrapping FreeRTOS on ARM Cortex M7

Today’s Real-Time Operating System/Platform

Cortex Microcontroller System Interface Standard wrapping FreeRTOS

Category	Example system	Example chip	Memory	CPU speed	Example ISA
Small desktop	Intel NUC	Sandy Bridge	16GB	2.7 GHz	64-bit Intel
Embedded powerhouse	Nvidia TX1	A57	5GB	2 GHz	64-bit ARM
Smartphone	Odroid	Snapdragon	2GB	2 GHz	64-bit ARM
Sensor controller	Pixhawk	Arm Cortex M7	2MB (flash)	216 MHz	32-bit ARM
Low-power, sleeps a lot	MSP430 Launchpad	MSP430	512 KB	25MHz	16-bit AVR, 8-bit AVR

Code Economics: C++ features

Objects	Lambdas	Exceptions
Templates	std::function	RTTI
STL Algorithms	Move semantics	STL Containers

Downsizing with compiler options

`-fno-exceptions`

`-fno-rtti`

`-fomit-frame-pointer:` don’t store frame pointer in registers where it’s not needed

`-fshort-enums:` pick smallest possible integer type for underlying type of enums

`-Os`: Optimize for space

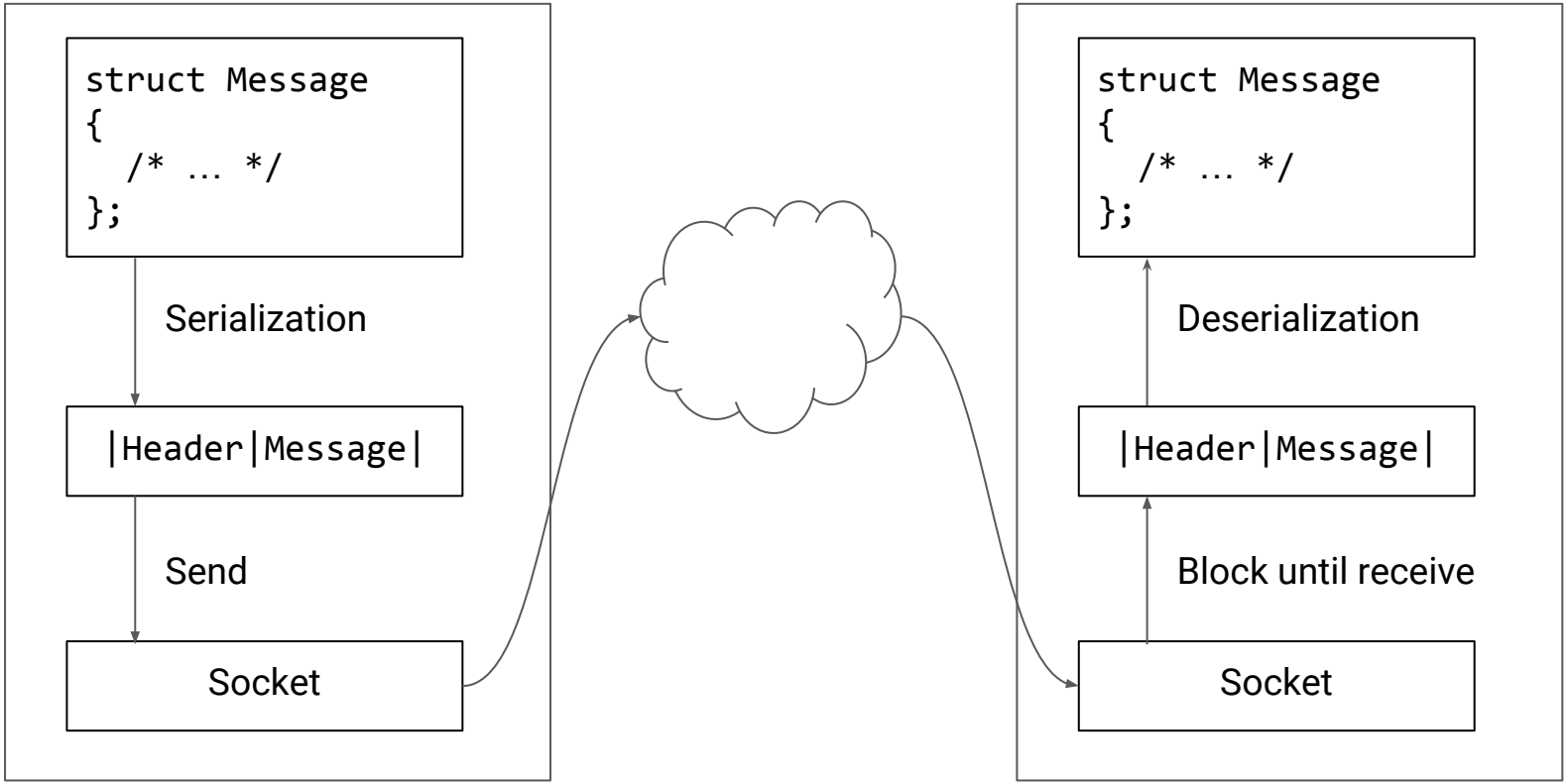
Defining boundaries: linker scripts, interrupt tables

```
._user_heap_stack :
{
  . = ALIGN(8);
  PROVIDE ( end = . );
  PROVIDE ( _end = . );
  . = . + _Min_Heap_Size;
  . = . + _Min_Stack_Size;
  . = ALIGN(8);
} >RAM
```

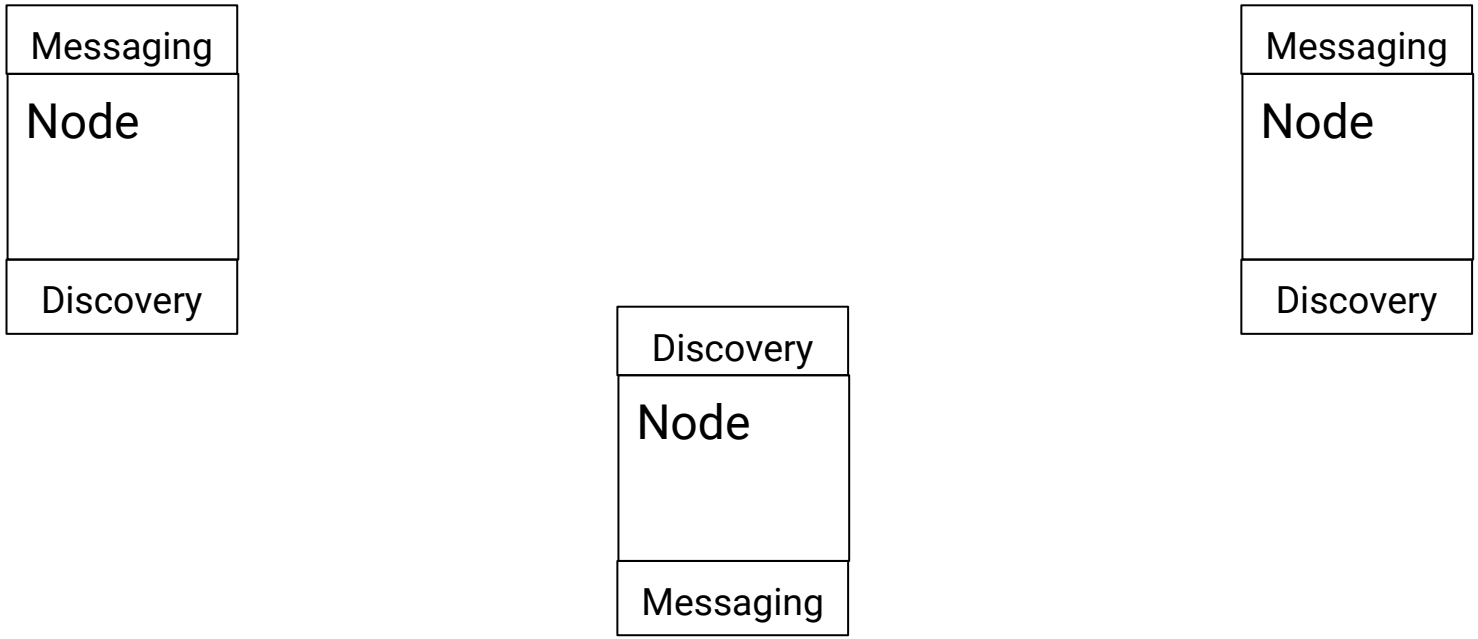
```
__user_initial_stackheap

  LDR     R0, =  Heap_Mem
  LDR     R1, =(Stack_Mem + Stack_Size)
  LDR     R2, = (Heap_Mem +  Heap_Size)
  LDR     R3, = Stack_Mem
  BX      LR

  ALIGN
  ENDIF
  END
```

Writing the code

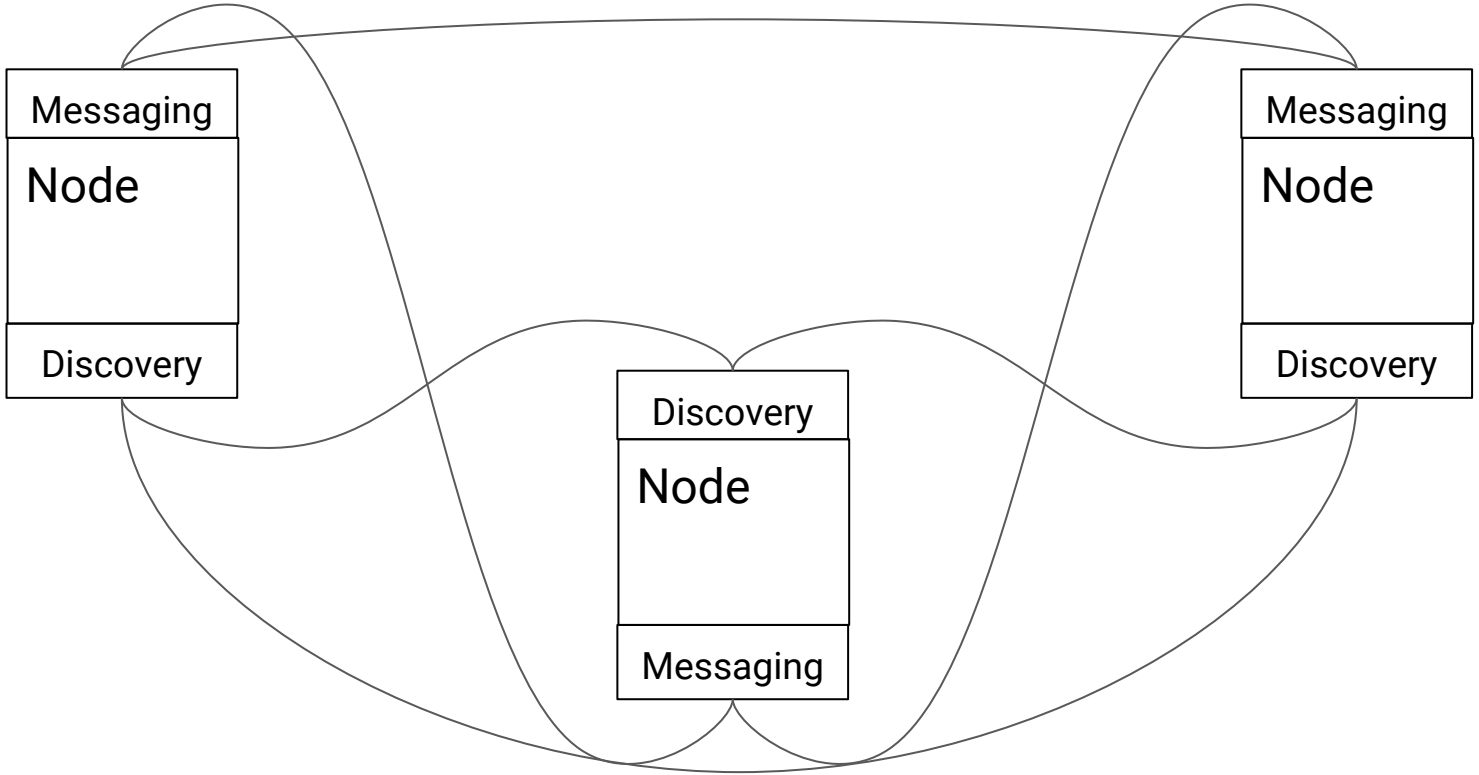
Caught in the middleware



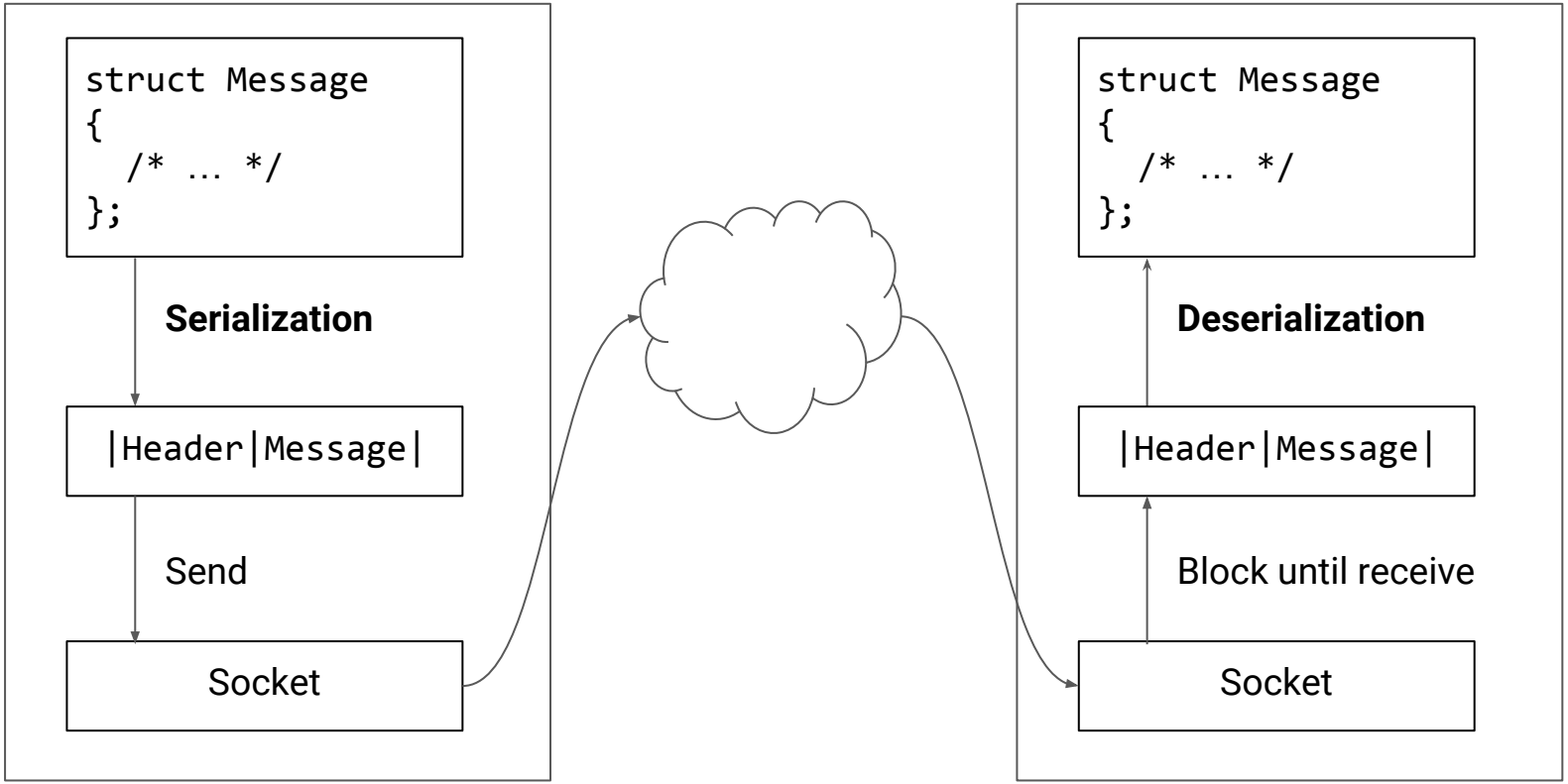
DDS is a bit more interesting



DDS is a bit more interesting



Let’s focus on common patterns



Serialization/deserialization: code generation

```
class Foo : GeneratedMessage {
public:
  void serialize(ByteStream &stream);
  void deserialize(ByteStream &stream);

  void set_field_a(A const& a);
  A get_field_a() const;
  /* etc. */
private:
  A a;
};
```

```
class Bar : GeneratedMessage {
public:
  void serialize(ByteStream &stream);
  void deserialize(ByteStream &stream);

  void set_field_b(B const& b);
  B get_field_b() const;
  /* etc. */
private:
  B b;
};
```

Serialization/deserialization: Using templates

```
template<typename T, typename ArchiveT>
void serialize(
    T const& src, ArchiveT& archive) {
  for (auto &&x : get_fields_of(T)) {
    serialize(T, archive);
  }
}

void(int src, ArchiveT& archive) {
  src >> archive;
}
```

```
template<typename T, typename ArchiveT>
void deserialize(ArchiveT& src, T& dst) {
  for (auto &&x : get_fields_of(T)) {
    deserialize(T, archive);
  }
}

struct Foo {
  A a;
  A2 a2;
  /* etc. */
};
```

Serialization/deserialization: Using templates

```
template<typename T, typename ArchiveT>
void serialize(
    T const& src, ArchiveT& archive) {
  for (auto &&x : get_fields_of(T)) {
    serialize(T, archive);
  }
}

void(int src, ArchiveT& archive) {
  src >> archive;
}
```

```
template<typename T, typename ArchiveT>
void deserialize(ArchiveT& src, T& dst) {
  for (auto &&x : get_fields_of(T)) {
    deserialize(T, archive);
  }
}

struct Foo {
  A a;
  A2 a2;
  /* etc. */
};
```

*This slide is pseudocode

Serialization code example

Serialization benchmarks

	Generic Approach			Code Generation		
	text	data	bss	text	data	bss
Primitive Types	14589	816	296	20861	1224	320
Array of Structs	16409	816	296	39077	1328	320
Struct of Arrays	16405	816	296	28205	1280	360
Nested	23337	816	296	52681	1360	400

Deserialization

```
Packet p = reader_socket.receive();

TypeEnum message_type = p.header;  // runtime-determined

/* ... */

MessageType message;

deserialize(a, message);

execute_callback(message);
```

Deserialization

```
Packet p = reader_socket.receive();

TypeEnum message_type = p.header;  // runtime-determined

/* ... */

MessageType message;

deserialize(a, message);

execute_callback(message);
```

How to translate from `message_type` to `MessageType`?
How to allocate the result of deserialization?

C-style approach

```
switch(message_type) {
  case TypeEnum::ack:
    Ack ack = deserialize(a);
    ack_callback(ack);
    break;
  case TypeEnum::source:
    Source b = deserialize(a);
    source_callback(b);
    break;
  default:
    /* Error-handling */
}
```

C-style approach

```
switch(message_type) {
  case TypeEnum::ack:
    Ack ack = deserialize(a);
    ack_callback(ack);
    break;
  case TypeEnum::source:
    Source b = deserialize(a);
    source_callback(b);
    break;
  default:
    /* Error-handling */
}
```

What if a user of the library wants to register custom type “Foo” with a callback?

C-style approach (with C++ syntax for brevity)

```
switch(message_type) {
  case TypeEnum::ack:
    Ack ack = deserialize(a);
    ack_callback(ack);
    break;
  case TypeEnum::source:
    Source b = deserialize(a);
    source_callback(b);
    break;
  default:
    /* Error-handling */
}
```

What if a user of the library wants to register custom type “Foo” with a callback?

```
case TypeEnum::user_defined:
  callbacks[header.user_type_id](a);

/* user-provided */
void foo_callback(Archive &a) {
  Foo b = deserialize(a);
  /* Process data in b */
}

// User has to add callback to global map
callbacks[“foo”] = foo_callback;
```

Metaprogramming Approach

Register callbacks of different signatures in a compile-time heterogenous map

Reverse-lookup into map at runtime: search through tuple of (id, type value) pairs

“Opt-in” RTTI?

Can’t define callbacks outside of the translation unit of `main()`

… but in embedded we need to compile to one static binary anyway

Type dispatch code example

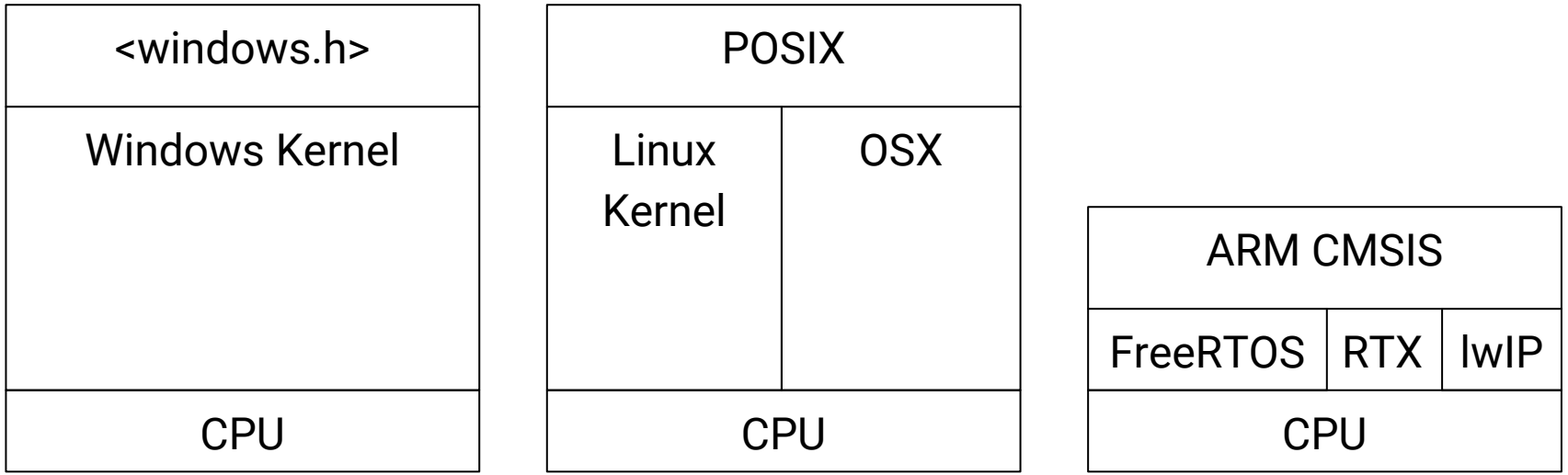
Type dispatch benchmarks

	Generic Approach			Static callback registration		
# callbacks	text	data	bss	text	data	bss
0	9491	784	296	12421	856	296
1	10390	792	296	13111	856	296
2	12114	792	296	14155	856	296
3	14158	792	296	15099	856	296
4	16294	792	296	15447	856	296

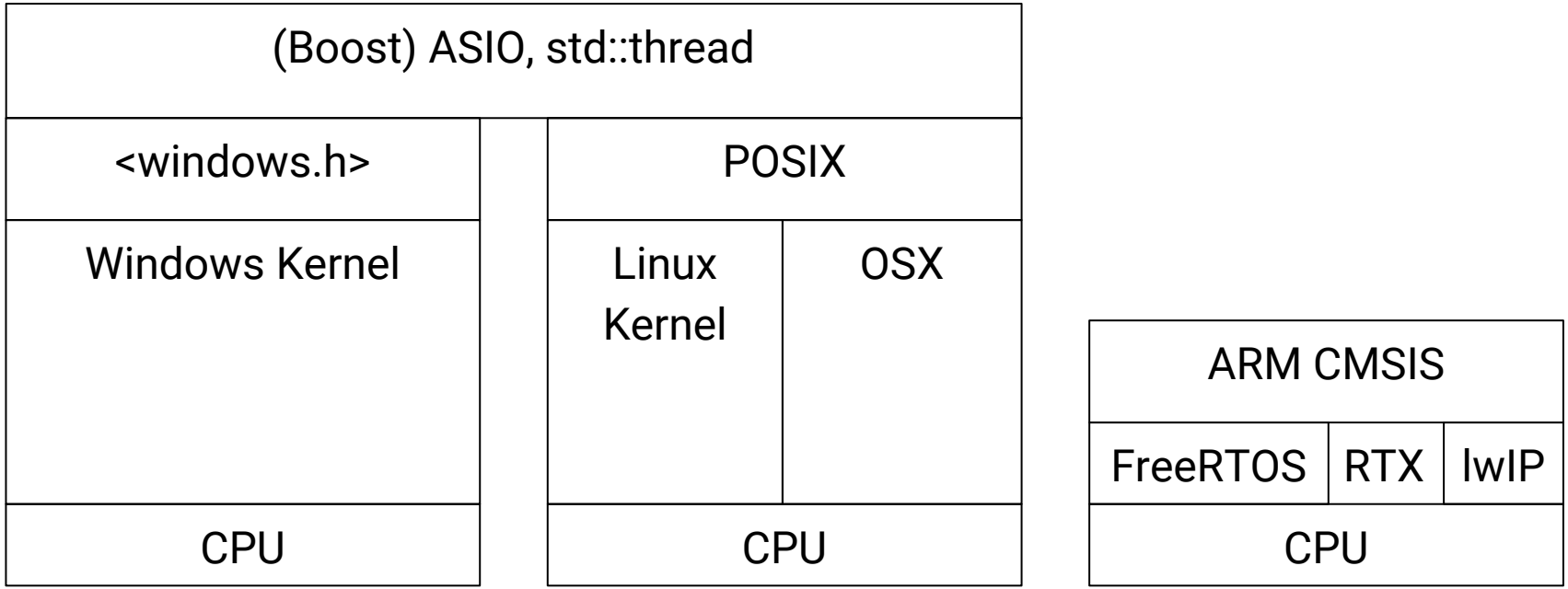
Serialization/Deserialization: Lessons decoded

- Templates can be tricky, always benchmark
- We need a high-quality generic serialization library that doesn’t depend on RTTI and exceptions.

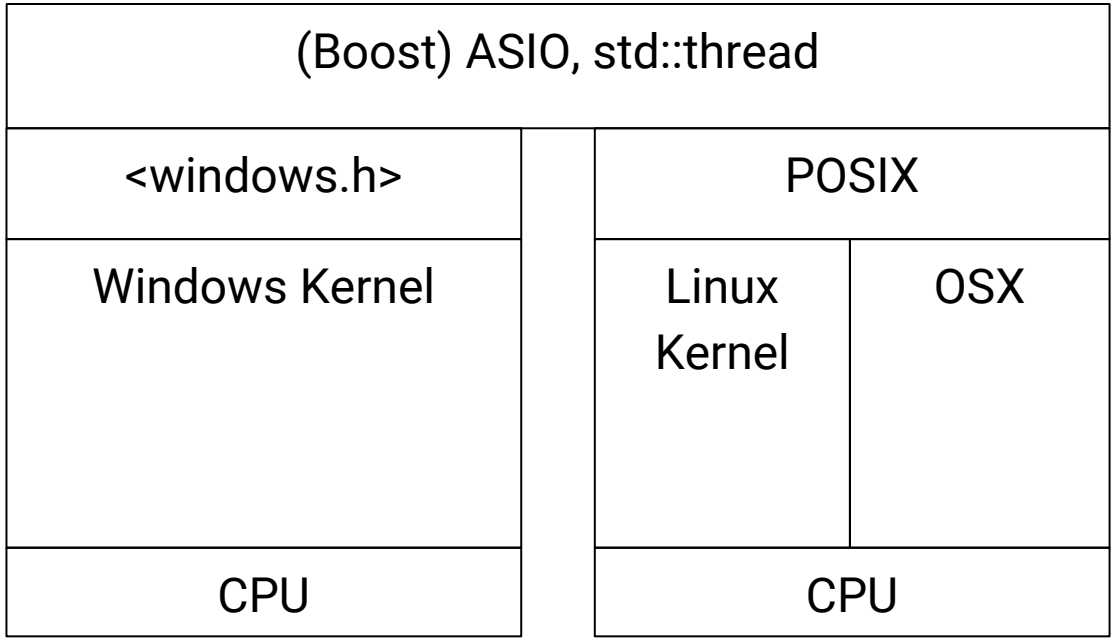
Sockets and threads: standards all the way down



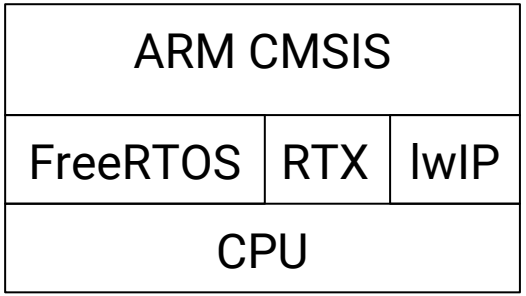
Sockets and threads



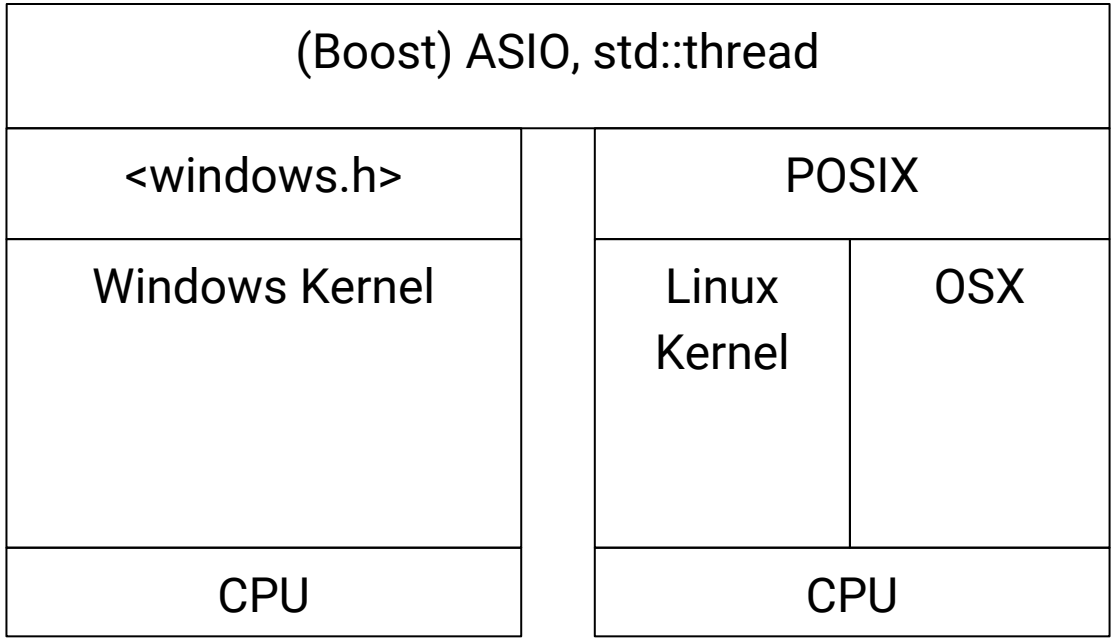
Sockets and threads



“ARM Compiler 6 C++ libraries support the majority of C++11. The major limitation is that Thread support (<thread>) is not available.”
- ARM® Compiler ARM C and C++ Libraries and Floating-Point Support User Guide

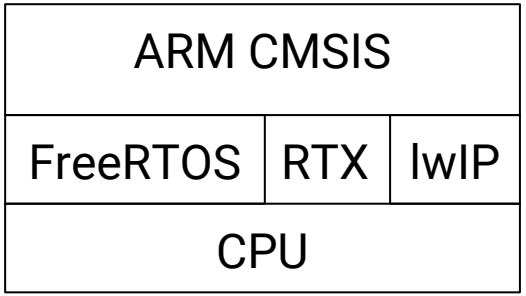


Sockets and threads



“ARM Compiler 6 C++ libraries support the majority of C++11. The major limitation is that Thread support (<thread>) is not available.”
- ARM® Compiler ARM C and C++ Libraries and Floating-Point Support User Guide

:(



Shimming ASIO

Requisites:

Socket descriptors: use lwIP (lightweight IP)

thread::join()

`join` code example

thread::join() quick benchmark

	text	data	bss
Pure C	22176	136	7180
C++ wrapper	28340	148	7280

Besides this overhead, why was <thread> left out?

On desktop, compiler and standard library are tightly coupled, and standard library is increasingly an interface to the OS

On embedded, compiler and interface to the OS are loosely coupled or not coupled at all.

RTOS’s could implement std:: interfaces to what they provide, but usually don’t.

What about DDS/RTPS?

Maybe not a great fit for modern C++: specification is object-oriented, not concept-oriented.

Maybe not a great fit for embedded: high API complexity, many threads/sockets needed for managing discovery across a distributed network of nodes.

Probably the thesis of a talk for an IoT conference!

Conclusions?

Powerful abstractions with low runtime cost and beautiful interfaces can make embedded programming more effective and accessible.

But embedded developers need libraries, cross-platform compatibility, and communication with the rest of the community.

References

- embcxx: https://arobenko.gitbooks.io/bare_metal_cpp
- Cereal: https://uscilab.github.io/cereal
- Hana: http://www.boost.org/doc/libs/1_61_0/libs/hana/doc/html/
- ASIO: http://think-async.com/Asio/asio-1.10.6/doc/
- ARM Info Center: http://infocenter.arm.com/index.html
- CMSIS: http://www.keil.com/pack/doc/CMSIS/General/html/index.html
- Icons are from the Noun Project

Resources and tools:

- OpenOCD: on-chip debugger
- (arm-none-eabi-)binutils: nm, objdump, readelf
- Code bloat analyzers (e.g. https://github.com/evmar/bloat)
- polymcu: CMake-based embedded cross-compilation framework (https://github.com/labapart/polymcu)