Australian National University

And in to one end he plugged the whole of reality, as extrapolated from
a fairy cake, and in the other end he plugged

Embedded Systems Architecture

so that when he turned it on she saw in one instant the whole infinity of creation

ENGN8537

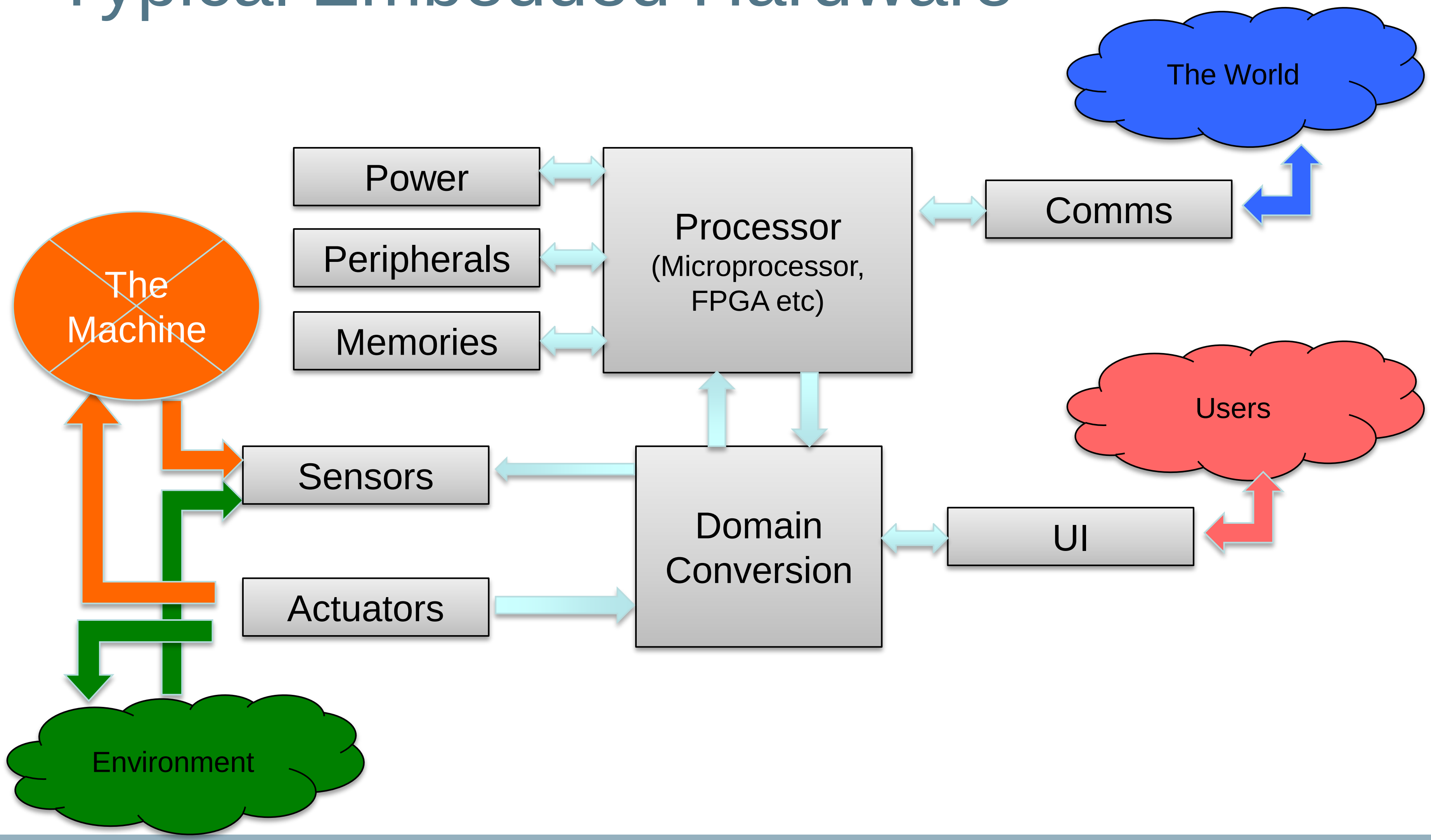
Embedded Systems

Embedded System Architecture

The architecture of an Embedded System is often thought to contain hardware and hardware interfaces only. Certainly this is the portion of the Embedded System that is typically given to Electronic Engineers to design.

Embedded Systems must interact closely with their environment, so the correct definition and specification of interfaces is certainly of great importance.

Typical Embedded Hardware

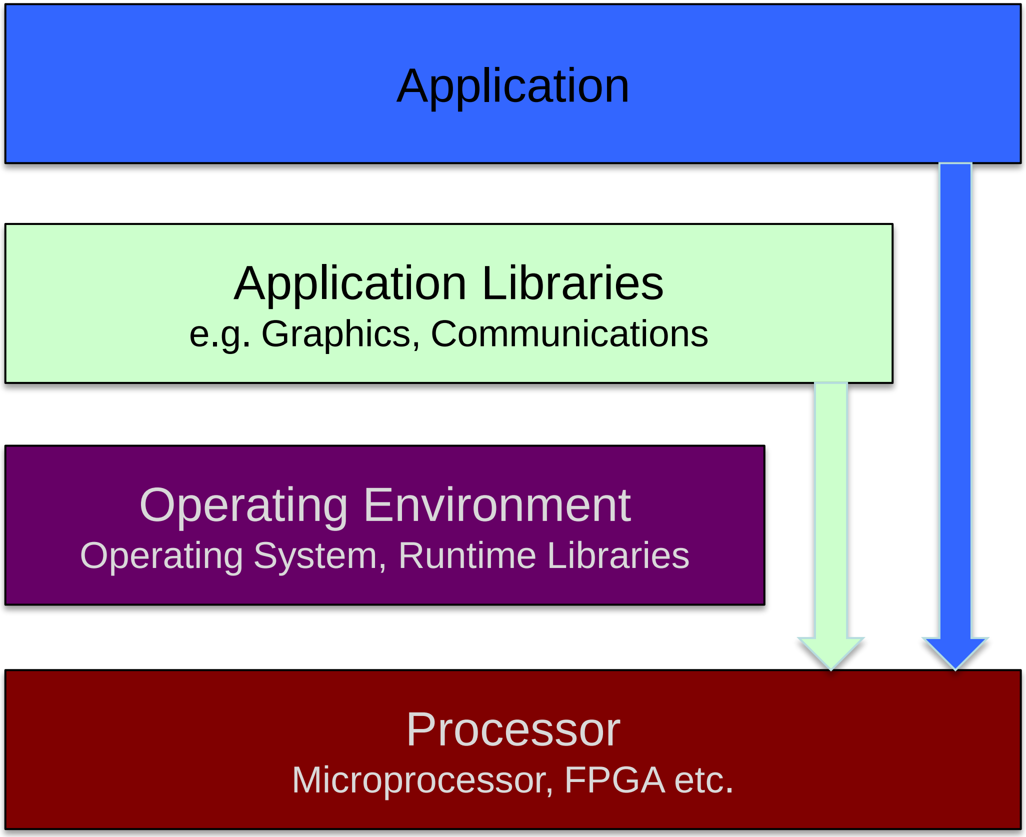


Embedded System Architecture

Embedded Systems typically have stricter constraints on power consumption, space, performance any many other attributes that can’t be described by the hardware alone.

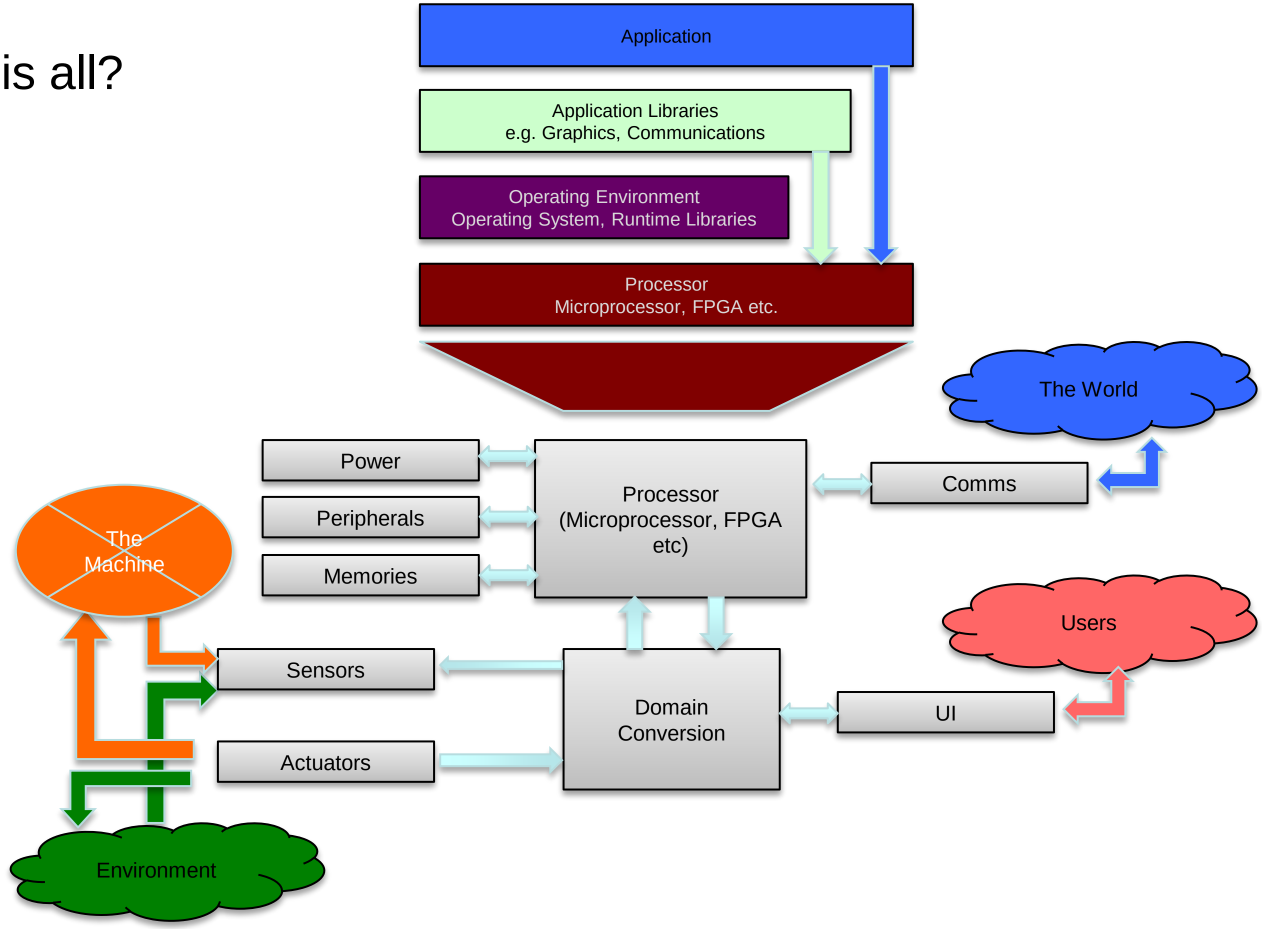
The software stack forms an integral part of the overall systems architecture and must certainly be considered when designing the system.

Typical Embedded Software


Application
Application Libraries
e.g. Graphics, Communications
Operating Environment
Operating System, Runtime Libraries
Processor
Microprocessor, FPGA etc.

Typical Embedded System?

So is this all?



Embedded System Architecture

The tight constraints surrounding Embedded Systems don’t stop at physical parameters.

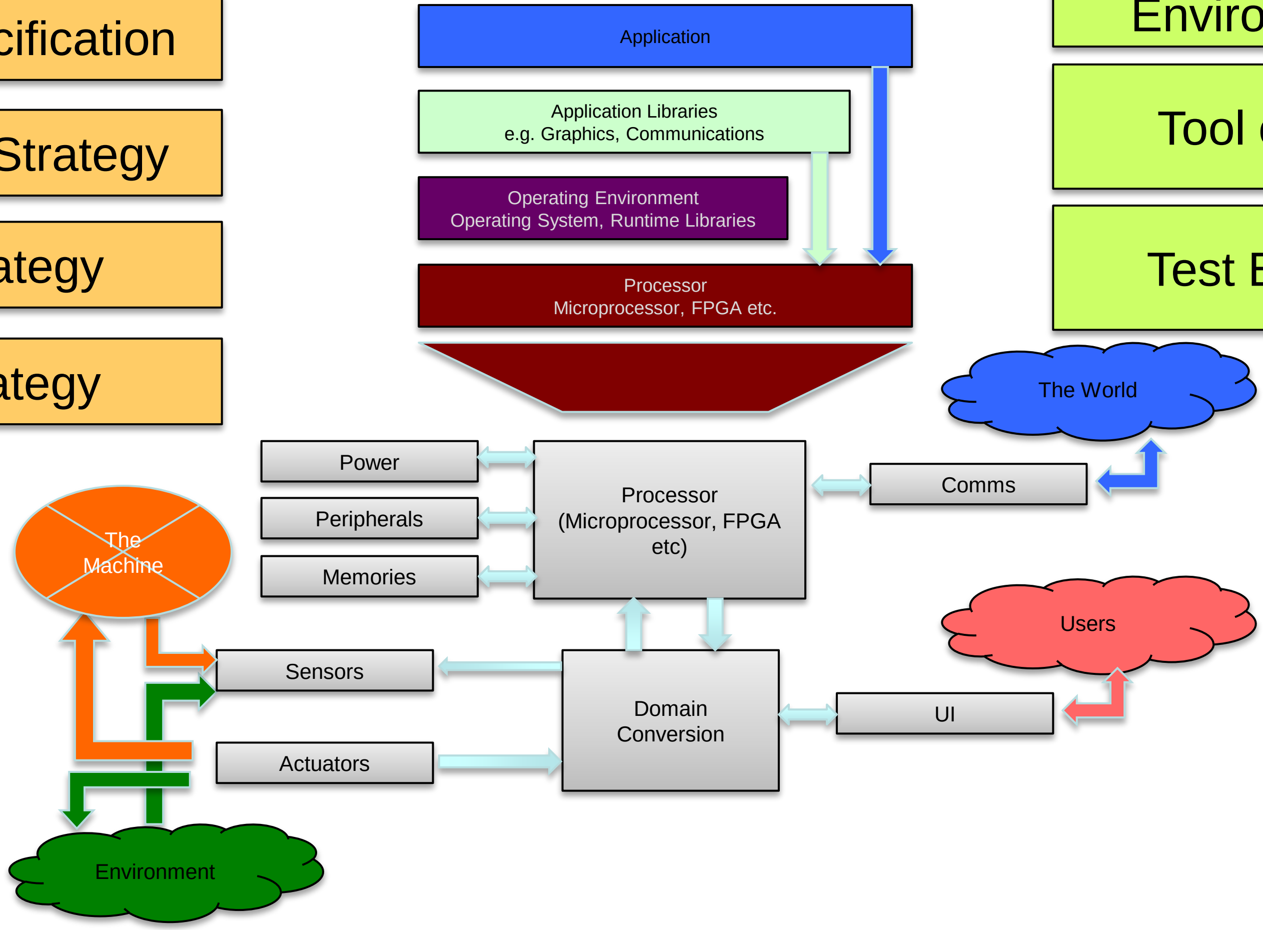
The successful design of an Embedded System has to note the fact that such systems are commodities, but also that they are rarely sold as a final product.

An Embedded System architecture has to recognize the interfaces between the system and the *business processes* around it, not just the physical processes.

These business processes include budgets and lead times and therefore come back and affect the choice of development system as well.

Typical Embedded System

- Reliability Specification
- Development Strategy
- Testing Strategy
- Market Strategy



Application

Application Libraries
e.g. Graphics, Communications

Operating Environment
Operating System, Runtime Libraries

Processor
Microprocessor, FPGA etc.

The World

Power

Peripherals

Memories

Processor
(Microprocessor, FPGA etc)

Comms

The Machine

Users

Sensors

UI

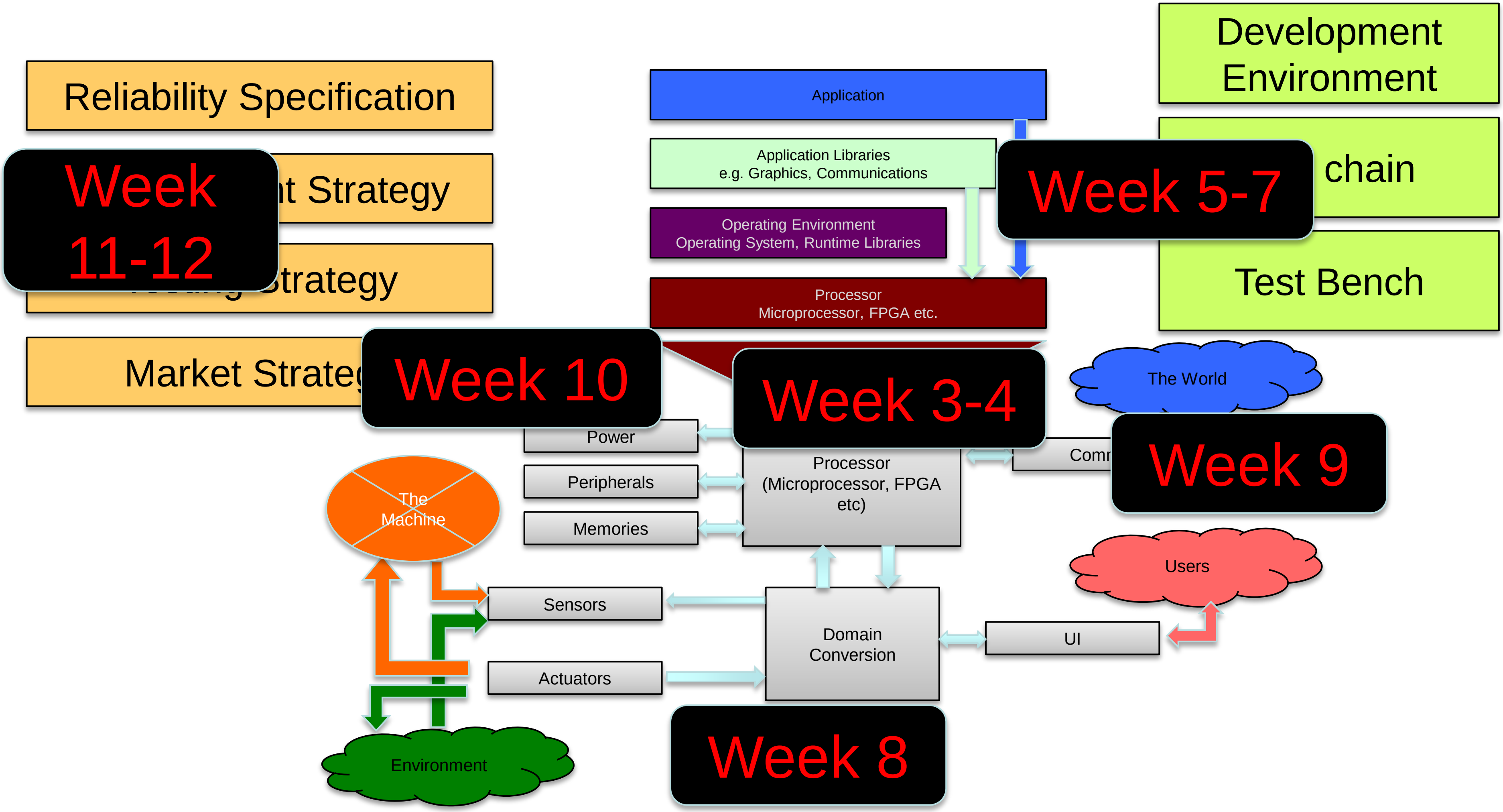
Domain Conversion

Actuators

Environment

- Development Environment
- Tool chain
- Test Bench

Typical Embedded System



Motivation 1

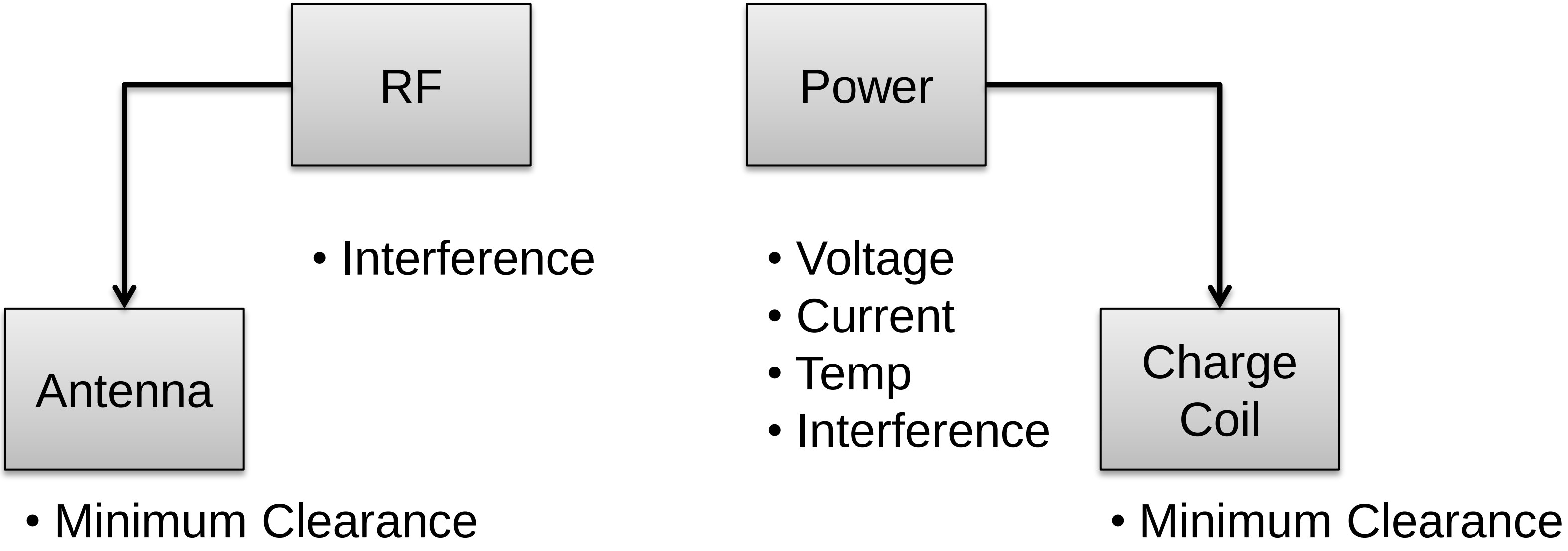
Here we have a (very small) subset of a high level architecture diagram for an embedded system along with the specifications relevant to each other

- Interference

- Voltage
- Current
- Temp
- Interference

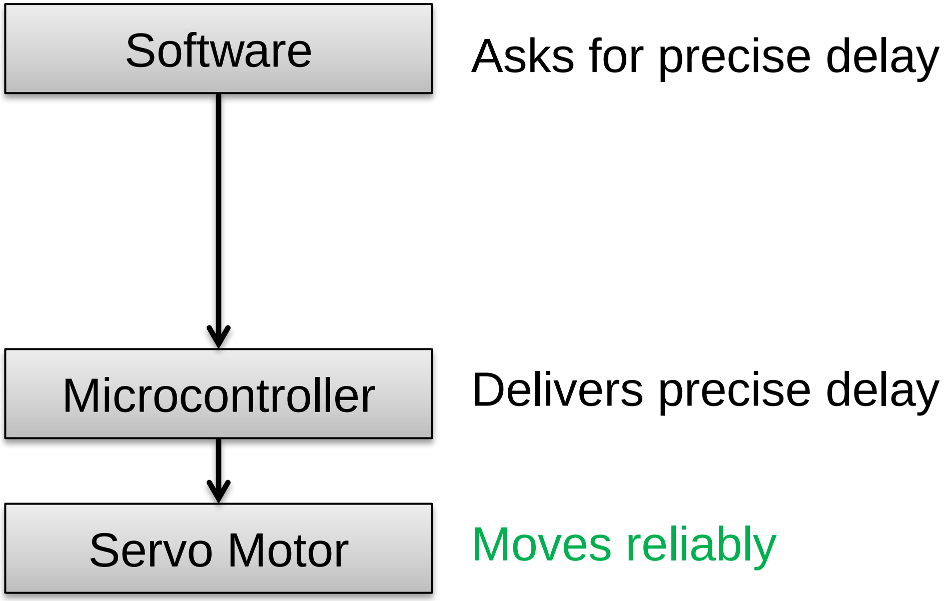
Motivation 1

The unwanted interaction between systems was all originally specified as “Interference” which was interpreted as electrical effects electromagnetic radiation. In fact there was a physical coupling between the systems that wasn’t discovered until late in the design cycle



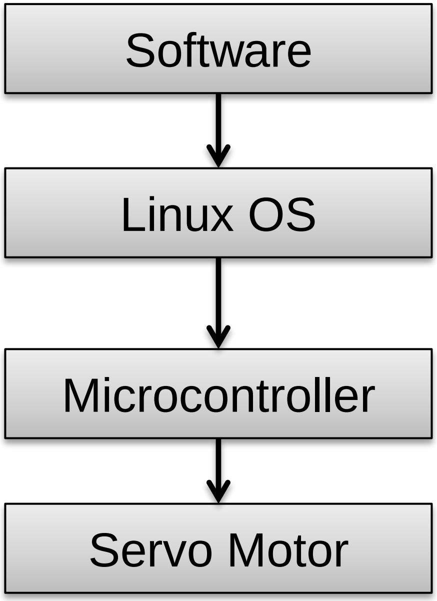
Motivation 2

A robot system was designed with its main microcontroller responsible for generating the precise waveforms required for motor control.



Motivation 2

The specification included constraints on connection but not on time. If there had been such a constraint, it would have been obvious early on that Linux could not fulfil it. As it was, the problem was caught late in the design cycle and incurred significant cost to rectify



Software	Asks for precise delay
Linux OS	Can’t guarantee precise delay
Microcontroller	Delivers imprecise delay
Servo Motor	Rips itself and the robot apart

Embedded Systems as Systems

Embedded Systems are designed like all systems.

1. Identify Functions
2. Identify Interfaces
3. Create a Functional Architecture
4. Assign functions to interacting elements
5. Repeat the above for each element until you get to a point where each remaining element is realizable

Embedded Systems as Systems

Functional Specification

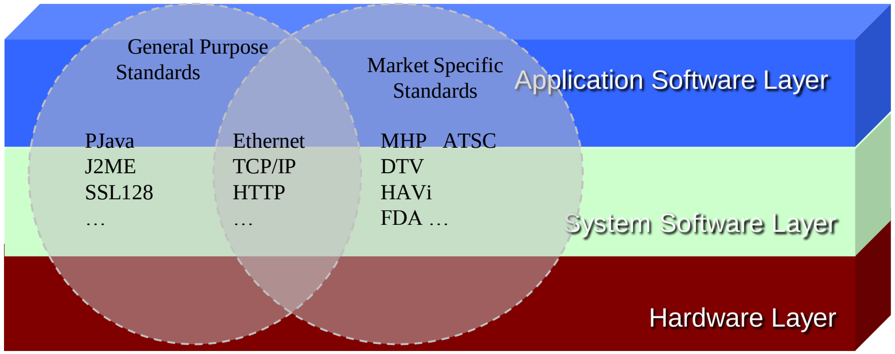
Most of you will have come across the notion of specifying a system in terms of its functions during the Systems Project, System Design or otherwise in the course of your degree. It isn't unique to Embedded Systems and is outside the scope of this course.

System Interfaces

The interfaces in Embedded Systems are certainly within the scope of this course

Don’t re-invent the wheel

Know your standards



- Market-Specific Standards
  → Consumer Electronics, Medical, Industrial Automation & Control, Networking & Communications, Automotive, Aerospace & Defence, Office Automation, …
- General Standards
  → Networking, Programming Language, Security, Quality Assurance, …

System Interfaces

Example: Digital Television Receiver.

Standard Type	Standard
Market-Specific	Digital Video Broadcasting (DVB)
	Java TV
	Home audio/video interoperability (HAVi)
	Digital Audio Video Council (DAVIC)
	High Definition Multimedia Interface (HDMI)
	Infrared Data Association (IrDA)
	.. and many more
General Purpose	Hypertext Transfer Protocol (HTTP)
	Post Office Protocol (POP3)
	Internet Message Access Protocol (IMAP)
	Simple Mail Transport Protocol (SMTP)
	Java
	Networking (TCP/UDP/IP/Ethernet/802.11)
	POSIX

System Interfaces

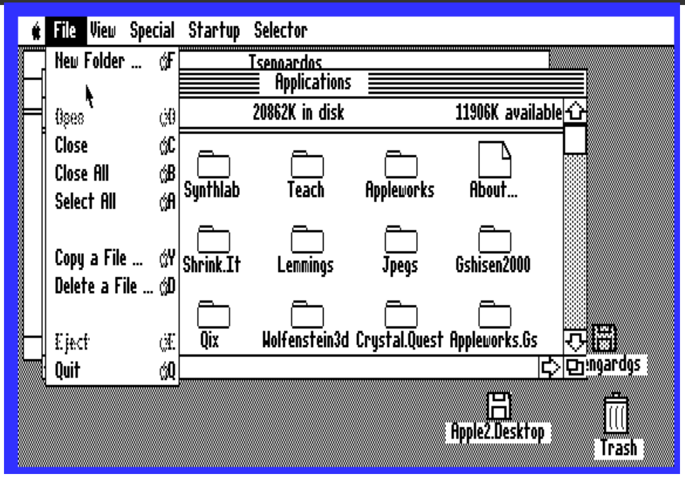
Example: Digital Television Receiver.

Systems interfaces includes things you would expect, the box has to be able to receive the digital video signal, a remote control signal and output some standard format for display on a Television (DVB, HDMI, IrDA).

The device might have functions outside its core role, such as a web interface (HTTP) or email notifications (SMTP) which in turn build on top of other standards (Ethernet/TCP/UDP/IP/802.11)

The systems interfaces also have to cover interfaces to the software as well. The ability for your Embedded System to run the target software package is just as important as any other aspect of interfacing.

System Interfaces



Example: Digital Television Receiver.

Clearly most interfaces in a typical Embedded System interfaces can be implemented against a known standard. This usually speeds up development and improves reliability as reference implementations and test benches are often parts of the standard themselves and not something that has to be implemented from scratch.

If there isn’t an applicable standard, there are often at least standard conventions. Graphical User Interfaces are an example – the notion of icons and windows as interaction metaphors has proven to scale from the Apple computers of old to the latest touchscreen enabled devices.

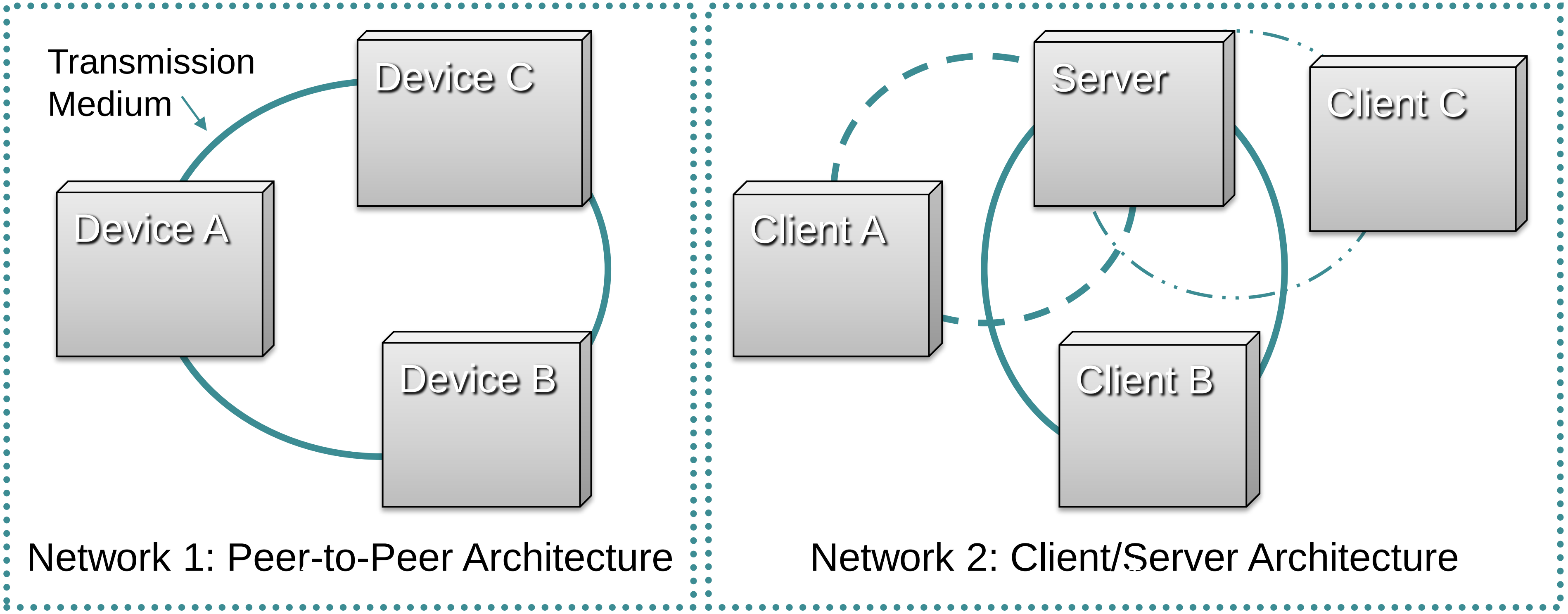
System Interfaces

When not to standardize: When the boss tells you not to! The most typical case for not adhering to some standard protocol is that the market forces are such that interfaces must remain proprietary.

The use of proprietary standards is decreasing as companies realize that interoperability is generally desirable after all, but also that if you close an interface there are plenty of people out there interested in opening it back up. The Microsoft SMB file sharing protocol is one such example, where Andrew Tridgell (now at the ANU) reverse engineered the protocol from scratch, was sued by Microsoft but prevailed in the courts. Simply, it’s often not worth the extra time required to lock down an interface.

Network Interfaces

Network interfaces extend the Embedded System architecture beyond the physical system itself. The devices may be connected in a Client/Server fashion or a Peer-to-Peer manor.

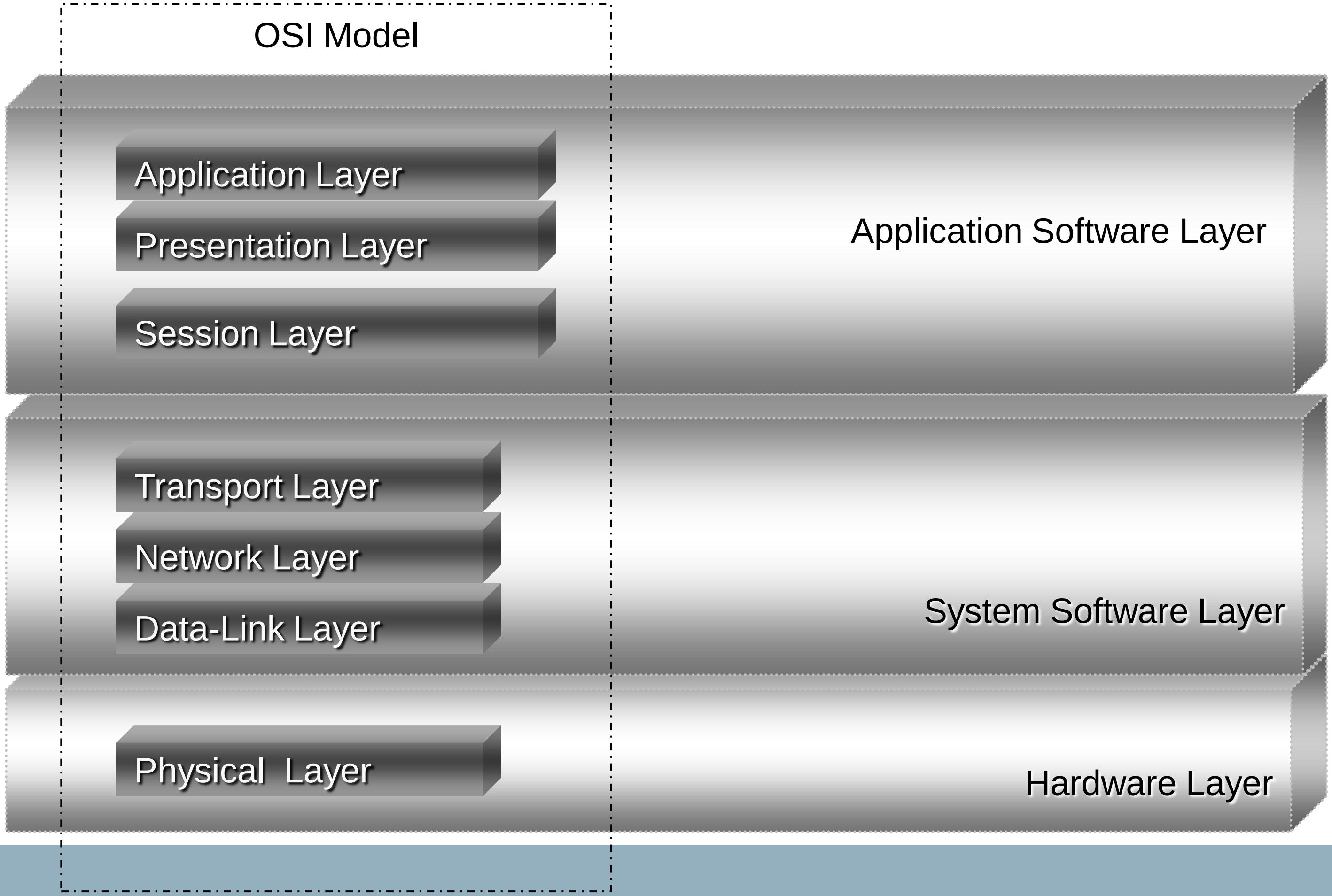


Network 1: Peer-to-Peer Architecture

Network 2: Client/Server Architecture

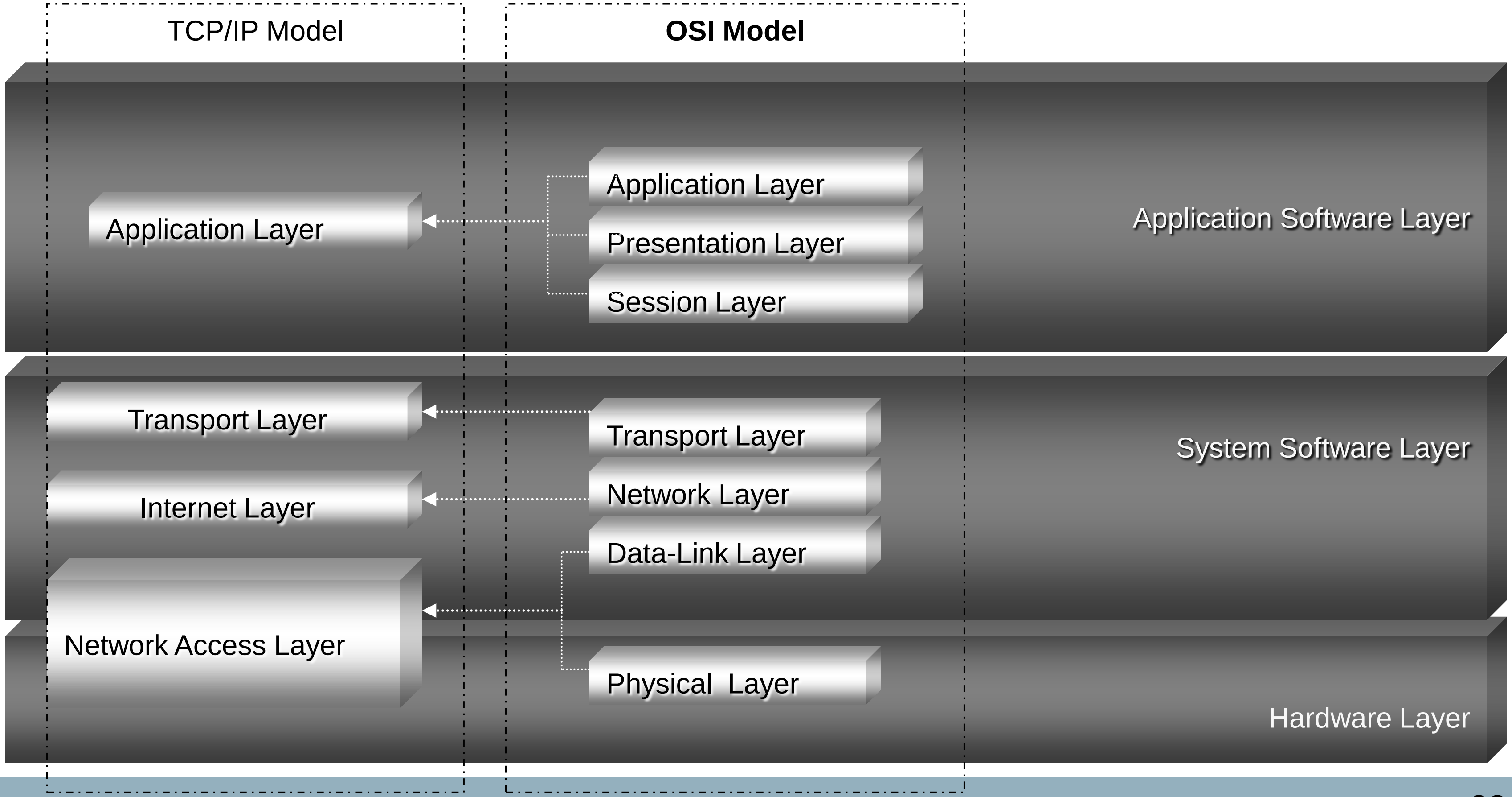
Network Interfaces

Within each device, the network stack can be defined in terms of the 7-layer OSI Model



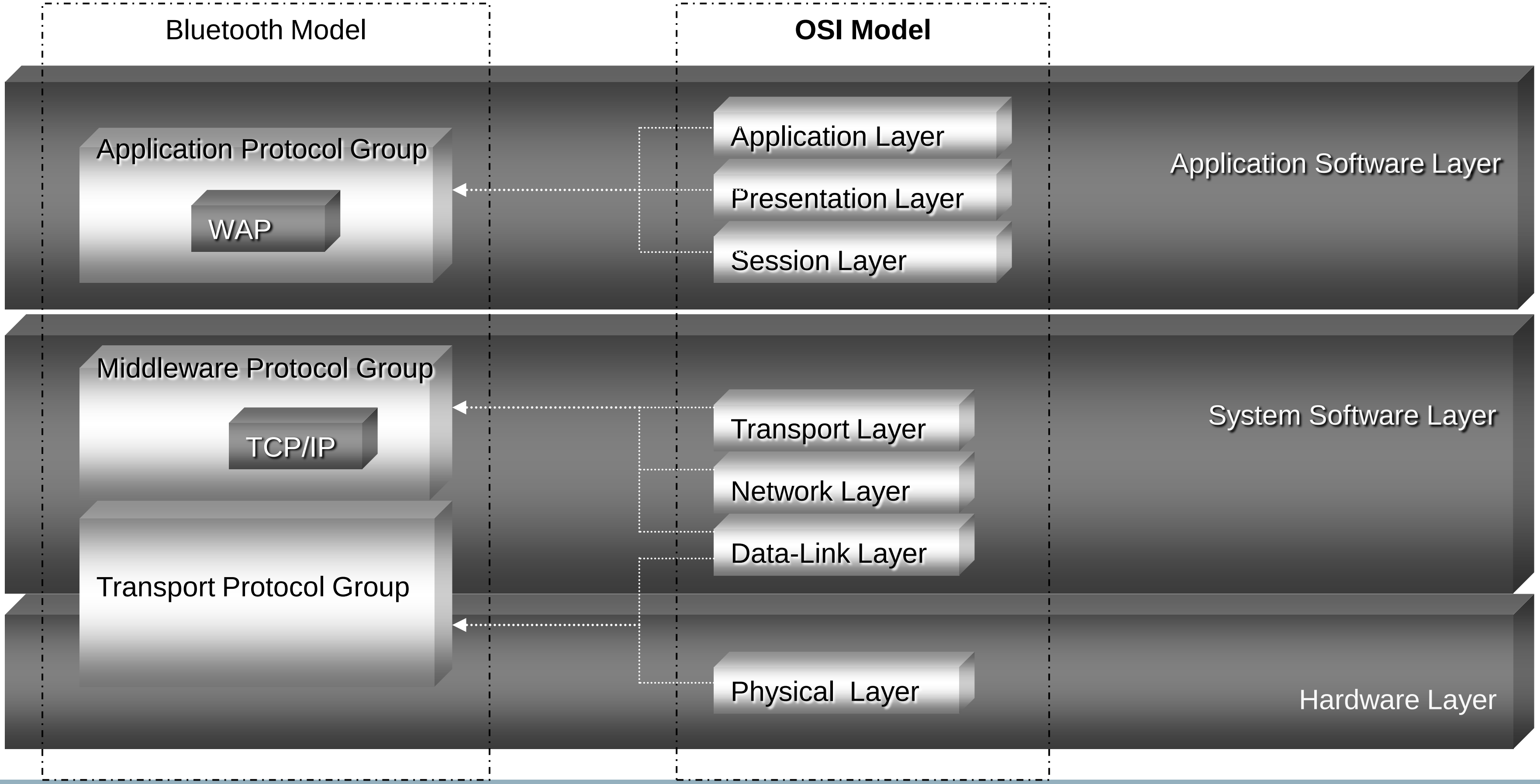
Network Interfaces

Common protocols however generally combine or miss several of the OSI layers



Network Interfaces

Common protocols however generally combine or miss several of the OSI layers



Bluetooth Model

Application Protocol Group

WAP

Middleware Protocol Group

TCP/IP

Transport Protocol Group

OSI Model

Application Layer

Presentation Layer

Session Layer

Transport Layer

Network Layer

Data-Link Layer

Physical Layer

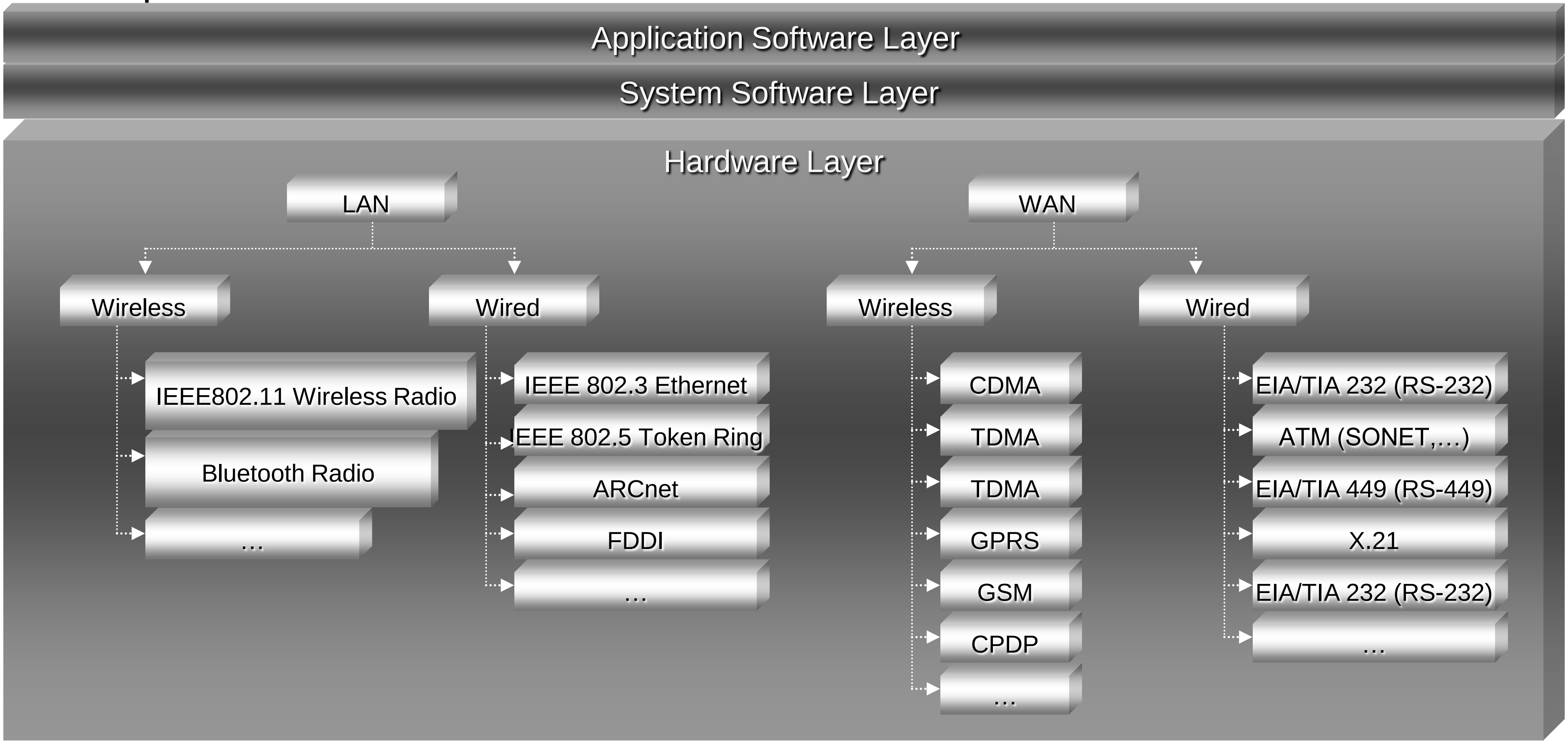
Application Software Layer

System Software Layer

Hardware Layer

Network Interfaces

Six of the OSI layers are often implemented in software but the Physical Layer *must* be implemented in hardware



Application Software Layer

System Software Layer

Hardware Layer

- LAN
  - Wireless
    - IEEE802.11 Wireless Radio
    - Bluetooth Radio
    - …
  - Wired
    - IEEE 802.3 Ethernet
    - IEEE 802.5 Token Ring
    - ARCnet
    - FDDI
    - …
- WAN
  - Wireless
    - CDMA
    - TDMA
    - TDMA
    - GPRS
    - GSM
    - CPDP
    - …
  - Wired
    - EIA/TIA 232 (RS-232)
    - ATM (SONET,…)
    - EIA/TIA 449 (RS-449)
    - X.21
    - EIA/TIA 232 (RS-232)
    - …

Software Interfaces

The software stack is an important part of the system architecture. The way the hardware and software interact is at least as important as any other interface in the system.

Software systems are typically specified from the allocated functionality and hardware systems are selected to support these systems. This isn’t a one-way process, hardware restrictions will always come back and change software system specifications.

We’ll go over software interfaces in some detail, as they are a part of the architecture that would not have been covered in previous Engineering courses and is vital for motivating our next lectures on Embedded Processors, Operating Systems and Real Time constraints.

Software Interfaces

Typical ways in which a software system will influence hardware choices:

- Programming Language
  - Precompiled, Just-in-time compiled, interpreted …
- Operating System, Run-time Environment, Libraries
  - Size, complexity, driver availability …
- Memory Requirements
  - A function of programming language, functions, implementations and more
- Speed Requirements
  - As above, influenced by a large number of software choices
- Real-Time Requirements
  - Not the same as speed requirements as we will see later

Software Interfaces

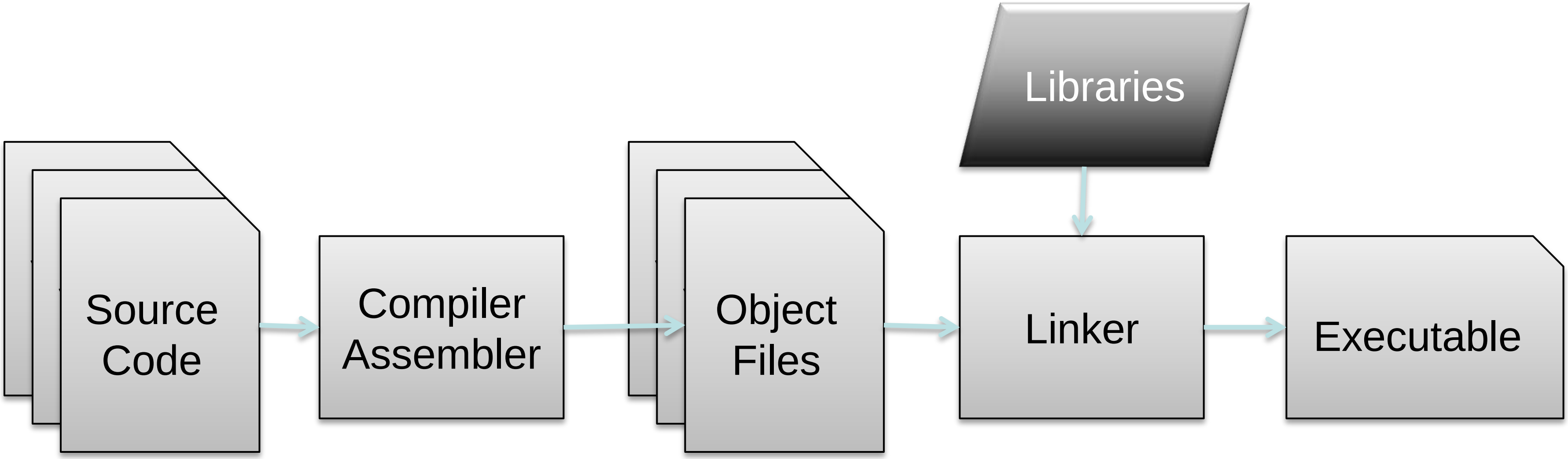
The choice of programming language(s) is perhaps the most fundamental software architecture choice.

A balance must be struck between features, efficiency, reliability, time to market etc.

One of the key points that distinguishes programming languages from each other are whether they are precompiled, Just-in-Time compiled or Interpreted. These three options place restrictions on the locations in which a language may be used and what may be accomplished with code written in that language.

Programming Languages

Precompiled code is typical of programming languages such as C, C++, ADA etc. Precompiled code doesn't incur run-time overhead translating the programmer's wishes to actions, however it offers less flexibility than other options.



Programming Languages

The compiler takes source code and generates machine language assembly mnemonics (written instructions such as MOV, ADD). The assembler takes this assembly, along with any that has been manually written and generates the binary object files. References to memory locations remain in symbolic form – the object file doesn't yet know where in memory everything is going to end up.

The compiler and assembler are typically invoked with a single instruction, the intermediate assembly mnemonics are stored in a temporary file and removed upon successful generation of the object file.

The compiler is responsible for most optimizations and generation of platform-specific code. When choosing hardware, it must be selected to be compatible with the target compiler.

Programming Languages

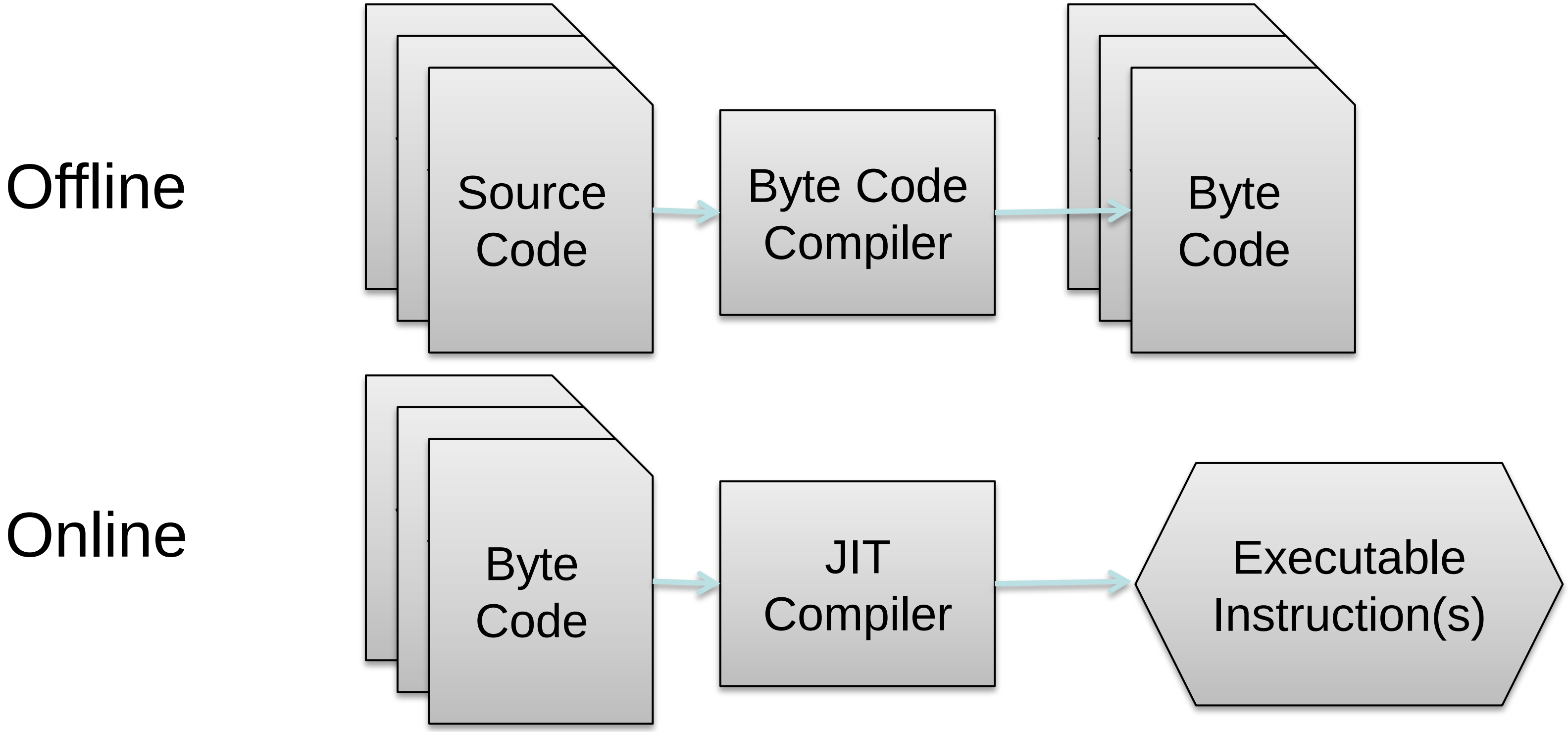
The Linker combines multiple object files (typically one per source file) along with external libraries, joins them all together in a single executable then goes through and resolves all symbols. Symbol resolution is the process of going through the executable, finding instructions that reference memory (load, store, function calls etc.) and inserting the correct memory location in to the instruction.

If static libraries are used, all symbols in the executable are resolved and no other item is required for that binary to be executed. This is typically the case for Embedded Systems.

Dynamic libraries (DLLs in Windows-speak, SOs in Linux) have their symbols left unresolved at Link time. The final memory locations for shared symbols are only resolved when the executable is run. This increases start up times but sharing code between executables can reduce overall size.

Programming Languages

Just-in-time (JIT) compiled code has two processes, one on done before-hand and one done while the program runs. The intermediate representation is called ‘Byte Code’



Programming Languages

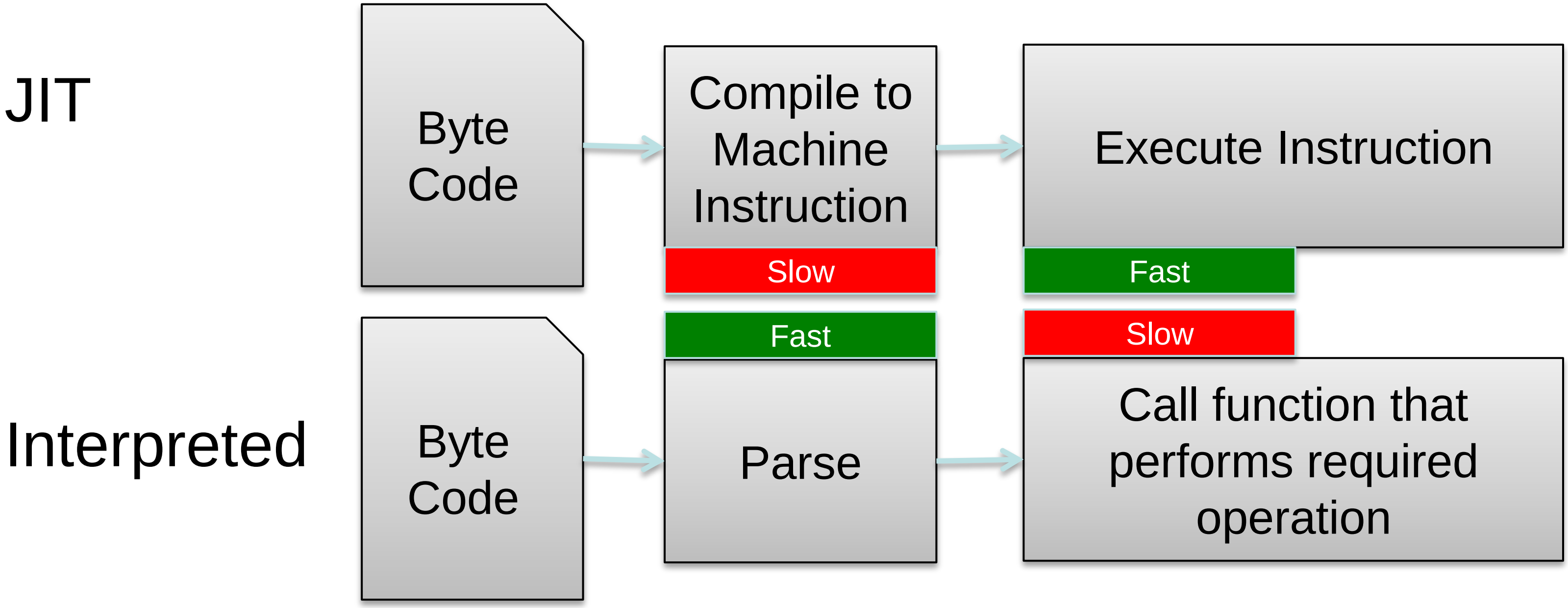
The JIT paradigm adds some overhead to the runtime process as the translation from byte code to machine instructions has to be done before the instruction can actually be executed. This sounds like it would take a lot of overhead, but modern JIT compilers might add just a few percent to the execution time.

JIT code means that optimization and generation of platform-specific codes are left to the platform itself, the same byte code can then be run on multiple machines.   If your Embedded System and your development machine use different architectures, you can still run a single lot of code on both.

The platform itself is also usually in the best position to determine which optimizations to apply, so the overhead of JIT compilation may be reclaimed through smart application of code optimization.

Programming Languages

The first stage of Interpreted code is the same as for JIT – source code is converted to byte code. When run, a special piece of software called a Virtual Machine reads each byte code in turn and decides what to do with it


JIT
Byte Code
Compile to Machine Instruction
Slow
Execute Instruction
Fast
Interpreted
Byte Code
Fast
Parse
Slow
Call function that performs required operation

Programming Languages

Interpreted code has the same portability advantages as JIT code but because the code is run in a completely controlled and isolated fashion (inside a Virtual Machine), there is the possibility for increased security.

JIT compilers are widely recognized as the future of this kind of technology and are slowly replacing run-time interpreters. One exception is on obscure architectures as Virtual Machines are typically written entirely in C and can be compiled for anything supported by that compiler. JIT compilers have to be written, at least partially, specifically for a target architecture as it has to know which machine instructions to output.

Programming Languages

Summary

Precompiled languages: C, C++, ADA etc. Typically older, more established languages but still very commonly used in Embedded Systems. Much low-level code such as the Linux Kernel are written in precompiled languages as there isn’t necessarily any platform upon which the JIT compiler or Virtual Machine can run.

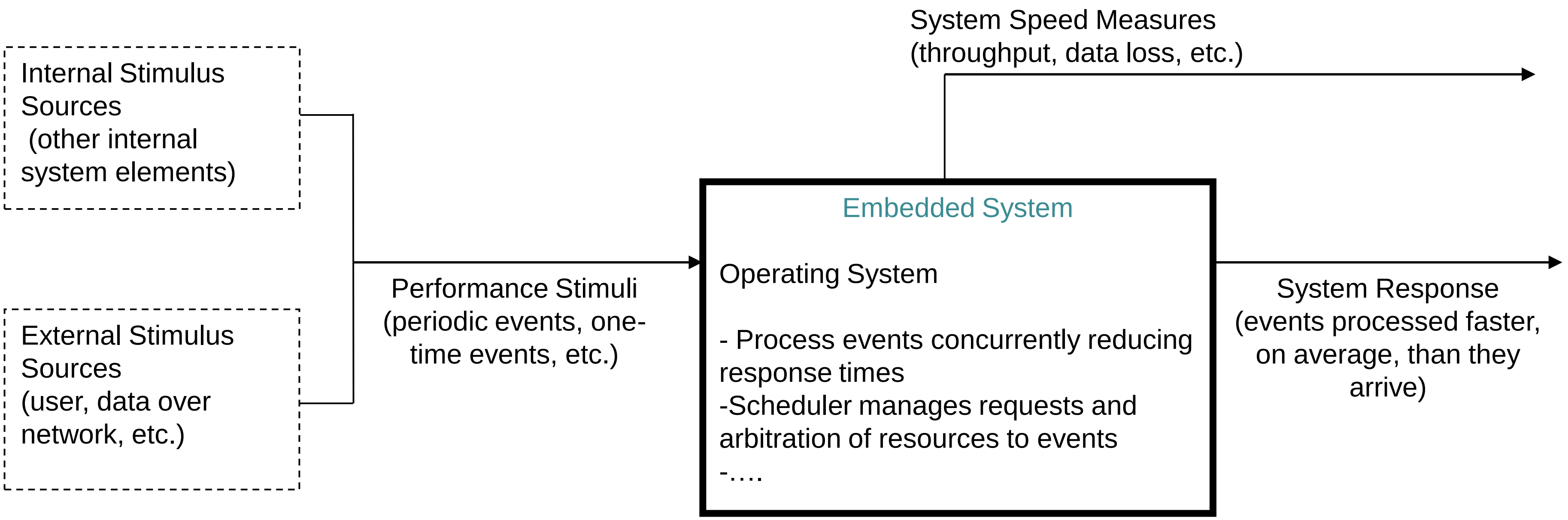
JIT languages: Java, .NET, Dalvik since Android 2.2, Python, Javascript in some browsers. Most modern languages were either designed to be interpreted but have subsequently had JIT compilers written, or were designed for JIT execution from the start.

Interpreted languages: Typically older versions of the JIT languages above.

Speed Requirements

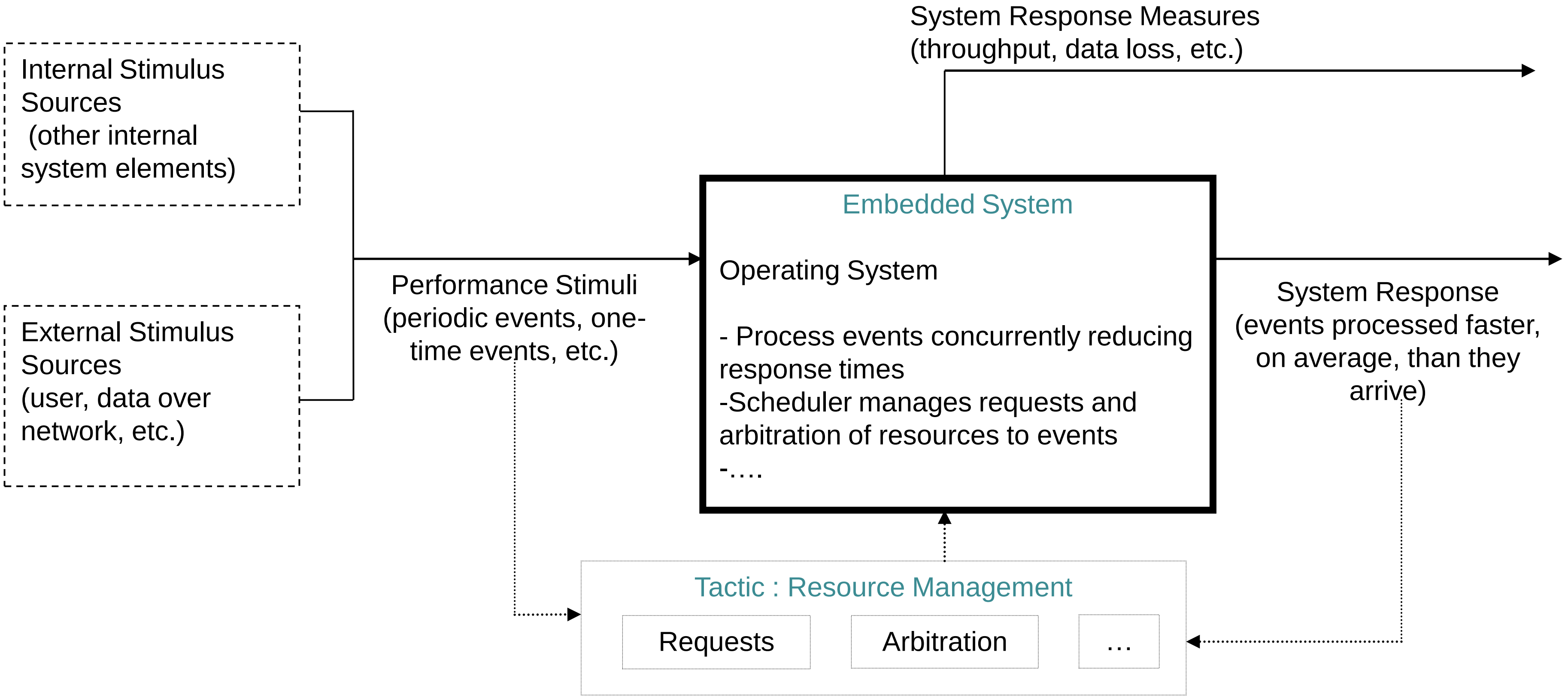
Speed in an Embedded System can mean a number of things. In this course, speed will be used primarily as a reference to throughput. That is, “Instructions per second”, “event responses per second” etc.

A framework for sketching speed requirements is shown below


System Speed Measures
(throughput, data loss, etc.)
Internal Stimulus Sources
(other internal system elements)
External Stimulus Sources
(user, data over network, etc.)
Performance Stimuli
(periodic events, one-time events, etc.)
Embedded System
Operating System
- Process events concurrently reducing response times
-Scheduler manages requests and arbitration of resources to events
-….
System Response
(events processed faster, on average, than they arrive)

Speed Requirements

The most common architectural change to improve speed is to change the management of resources. These might be processor cycles, buffer memory, access to hardware accelerators and so on.

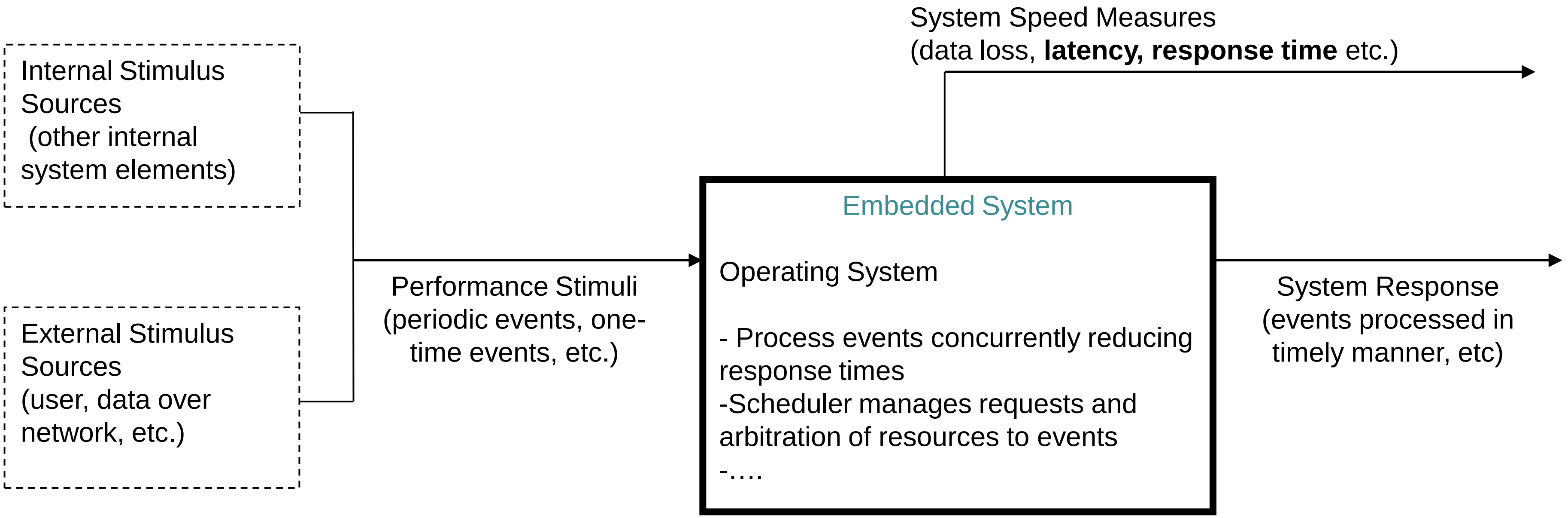


System Response Measures
(throughput, data loss, etc.)
Internal Stimulus Sources
(other internal system elements)
External Stimulus Sources
(user, data over network, etc.)
Performance Stimuli
(periodic events, one-time events, etc.)
Embedded System
Operating System
- Process events concurrently reducing response times
-Scheduler manages requests and arbitration of resources to events
-….
System Response
(events processed faster, on average, than they arrive)
Tactic : Resource Management
Requests
Arbitration
…

Responsiveness Requirements

Responsiveness is related to Real Time concepts as will be presented in a few lectures time. Low latency response, or rather **bounded** latency response, is a key aspect of all Embedded Systems

Responsiveness requirements can be couched in the same terms as speed but with slightly different metrics


System Speed Measures
(data loss, **latency, response time** etc.)

Internal Stimulus Sources
(other internal system elements)

External Stimulus Sources
(user, data over network, etc.)

Performance Stimuli
(periodic events, one-time events, etc.)

Embedded System

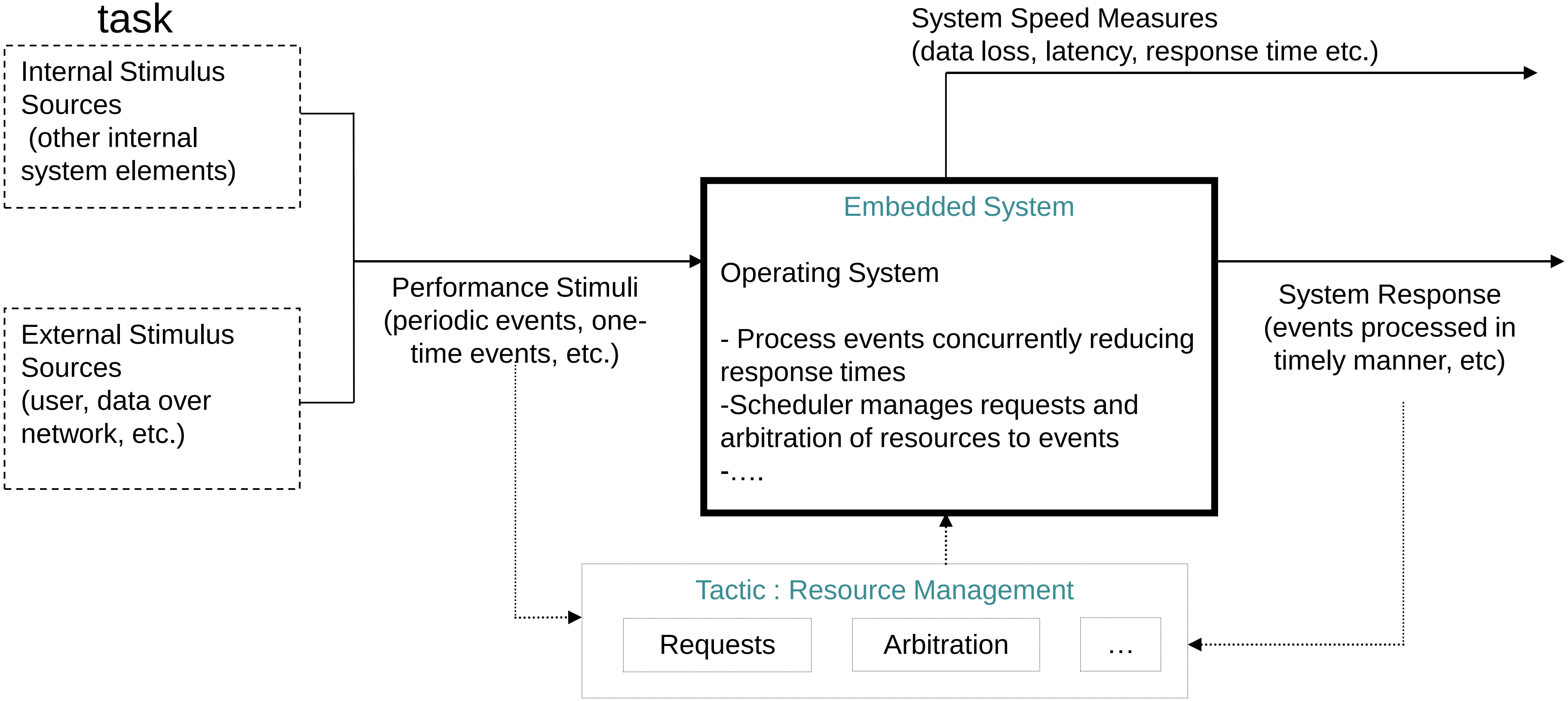
Operating System

- Process events concurrently reducing response times
-Scheduler manages requests and arbitration of resources to events
-….

System Response
(events processed in timely manner, etc)

Responsiveness Requirements

Responsiveness is improved by similar means to speed, but with different goals. Rather than increasing the number of available CPU cycles, one might increase the number of CPU Cycles **reserved** for the particular task


System Speed Measures
(data loss, latency, response time etc.)
Internal Stimulus Sources
(other internal system elements)
External Stimulus Sources
(user, data over network, etc.)
Performance Stimuli
(periodic events, one-time events, etc.)
Embedded System
Operating System
- Process events concurrently reducing response times
-Scheduler manages requests and arbitration of resources to events
-….
System Response
(events processed in timely manner, etc)
Tactic : Resource Management
Requests
Arbitration
…

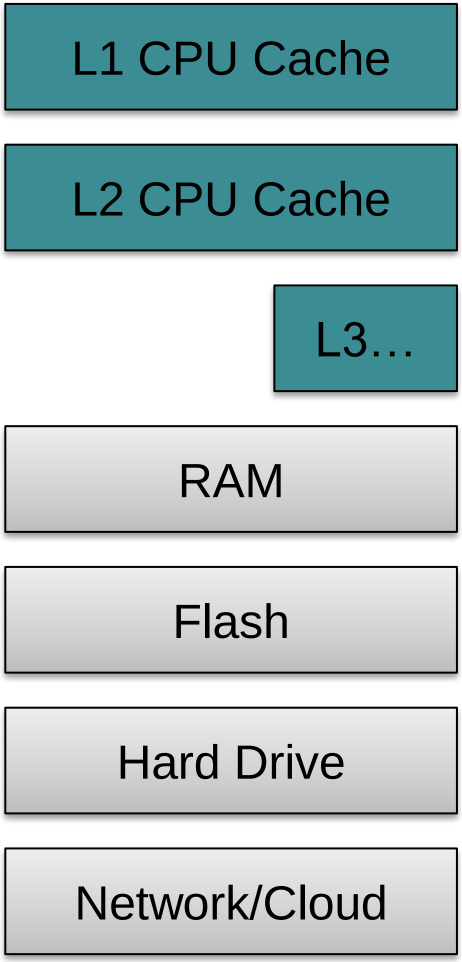
Memory Requirements

Memory is typically one of the last requirements specified. Prototype Embedded Systems may have several times as much memory as will be required in the field to facilitate the orderly development and debugging of code.

Code with debugging information left in may be 2-3x as big as ‘Release’ code.

Memory exists in a hierarchy with larger memory sources typically slower to access.

Memory Requirements



L1 CPU Cache

L2 CPU Cache

L3…

RAM

Flash

Hard Drive

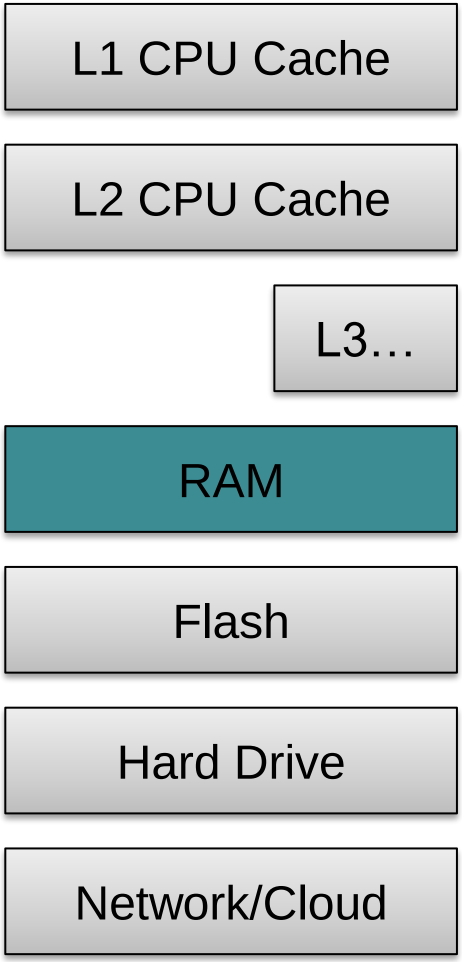
Network/Cloud

Most modern CPUs keep recently-accessed data cached in a small amount of memory tightly coupled to the CPU itself. Code rarely controls the cache itself, but can be optimized for good cache performance.

L1 Cache is typically accessed in 1-2 CPU cycles while lower level caches are proportionally slower.

In multi-core chips, the lower levels of cache may be shared between cores. This increases complexity somewhat as the cache has to remain ‘coherent’ across all cores; that is, because cache data is a copy of RAM data, it exists in at least two places at one and every core must ensure it accesses the most recent copy.

Memory Requirements



L1 CPU Cache

L2 CPU Cache

L3…

RAM

Flash

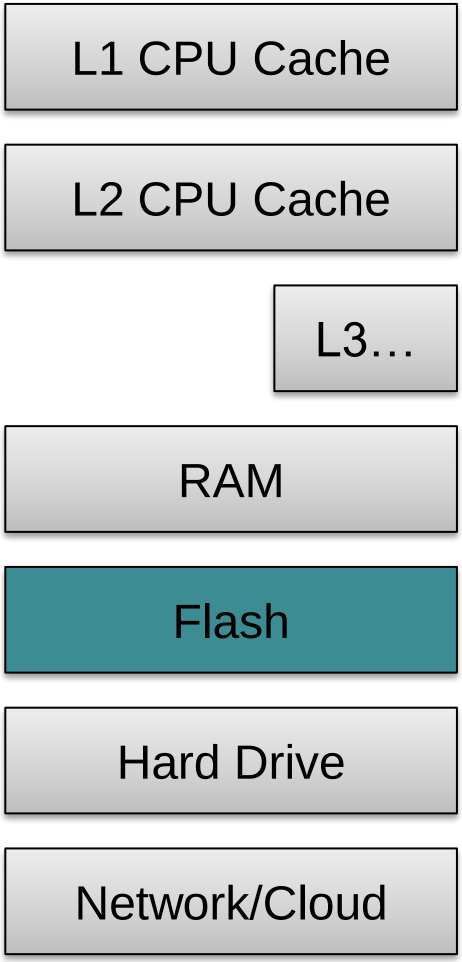
Hard Drive

Network/Cloud

RAM is what we typically think of when we think of memory. In an Embedded System, some base amount is required for program variables and the rest may be used for buffering, trading off cost against throughput.

Larger RAM devices (such as Dynamic RAM) require code to execute in order to be initialized. This causes a “chicken and egg” problem; code needs RAM to execute, but the RAM needs some code to have executed. If a device is designed to use Dynamic RAM, it will typically also include a small amount of Static RAM internal to the device for the sole purpose of supporting this initial execution.

Memory Requirements


L1 CPU Cache

L2 CPU Cache

L3…

RAM

Flash

Hard Drive

Network/Cloud

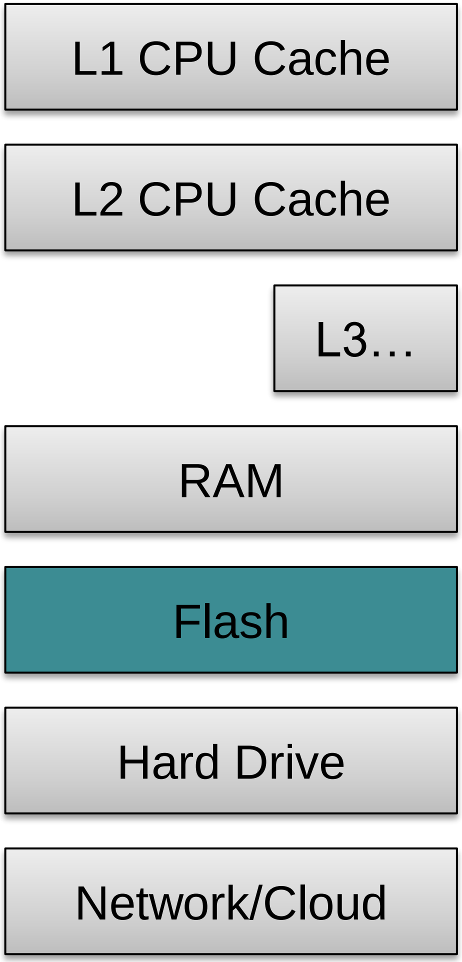
Flash memory is slowly replacing hard drives as the standard for local, non-volatile memory.

It is typically a few orders of magnitude slower than RAM and several orders of magnitude slower than a CPU cache.

Flash can only be written a finite number of times before it wears out, so much of the technology behind dealing with this type of memory goes in to avoiding, detecting and/or correcting errors.

Premium flash devices include Error Correcting Codes, ECC, which are special portions of memory that contain enough information to not just detect, but even fix 1, 2 or several-bit errors.

Memory Requirements



L1 CPU Cache

L2 CPU Cache

L3…

RAM

Flash

Hard Drive

Network/Cloud

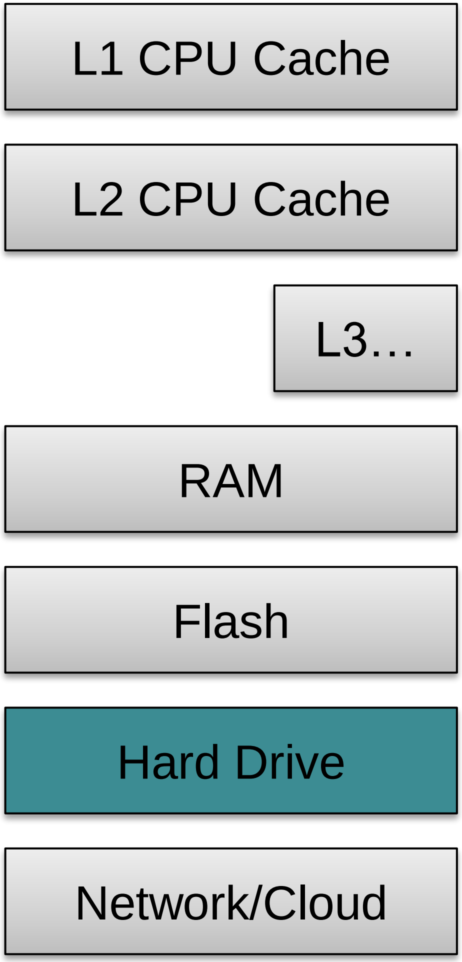
Aside

Modern flash memory cards such as SD cards may cheat with their wear leveling algorithms.

The FAT filesystem that is typically used on such cards writes data across the card in something approaching a round-robin fashion – in essence it does its own wear leveling. The exception to this is the File Allocation Table block which is written every time any other piece of the card is written. Given that the FAT gets written many more times than any other piece, the memory may choose to only wear level the section of the card where it thinks the FAT will sit.

This means that using a filesystem other than FAT on an SD card may reduce its lifespan 10s or 100s of times!

Memory Requirements

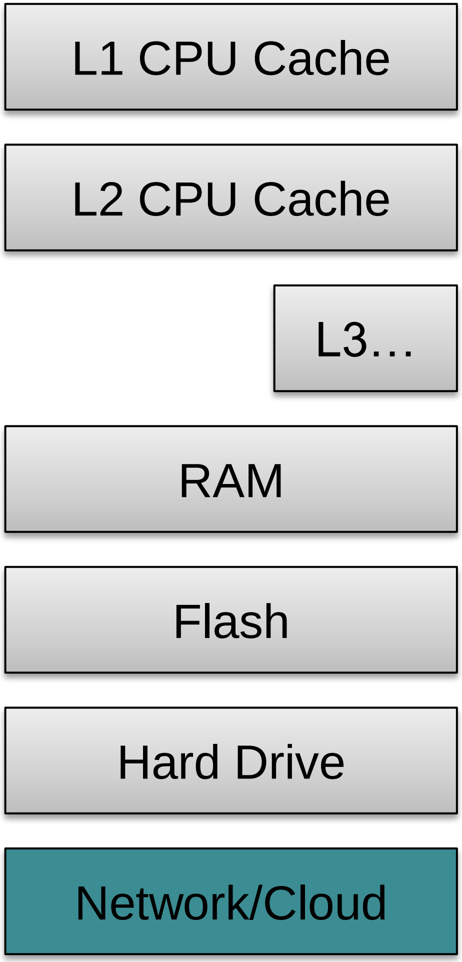


Hard Drives, or magnetic storage generally, have been around almost as long as computers themselves.

They are still the de-facto standard for high volume, non-volatile storage.

They are not often used in Embedded Systems as they are relatively power-hungry and fragile, at least when they are spinning. They also have non-deterministic response time as the magnetic heads need to physically move over the surface of the disk with each new request.

Memory Requirements



Network storage is becoming more fashionable as markets become less about the Embedded Device itself and more about the service it can offer. Having the service state data live externally to the system itself streamlines upgrades and can drive down system costs by centralizing much more of the system than would traditionally be the case.

Network storage has the obvious drawbacks though – the network is a weak link, both in terms of uptime and reliability, but also in terms of security.