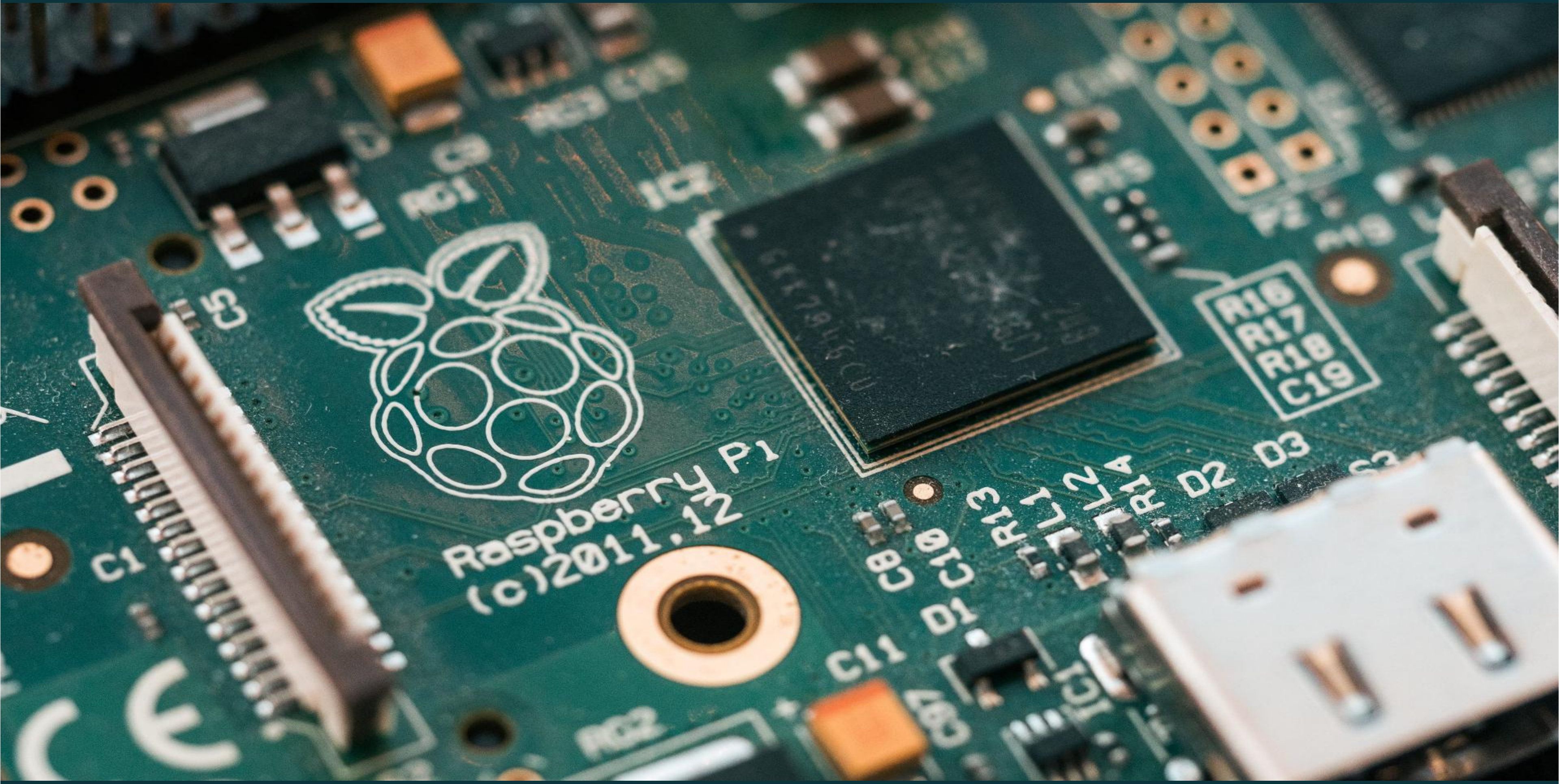
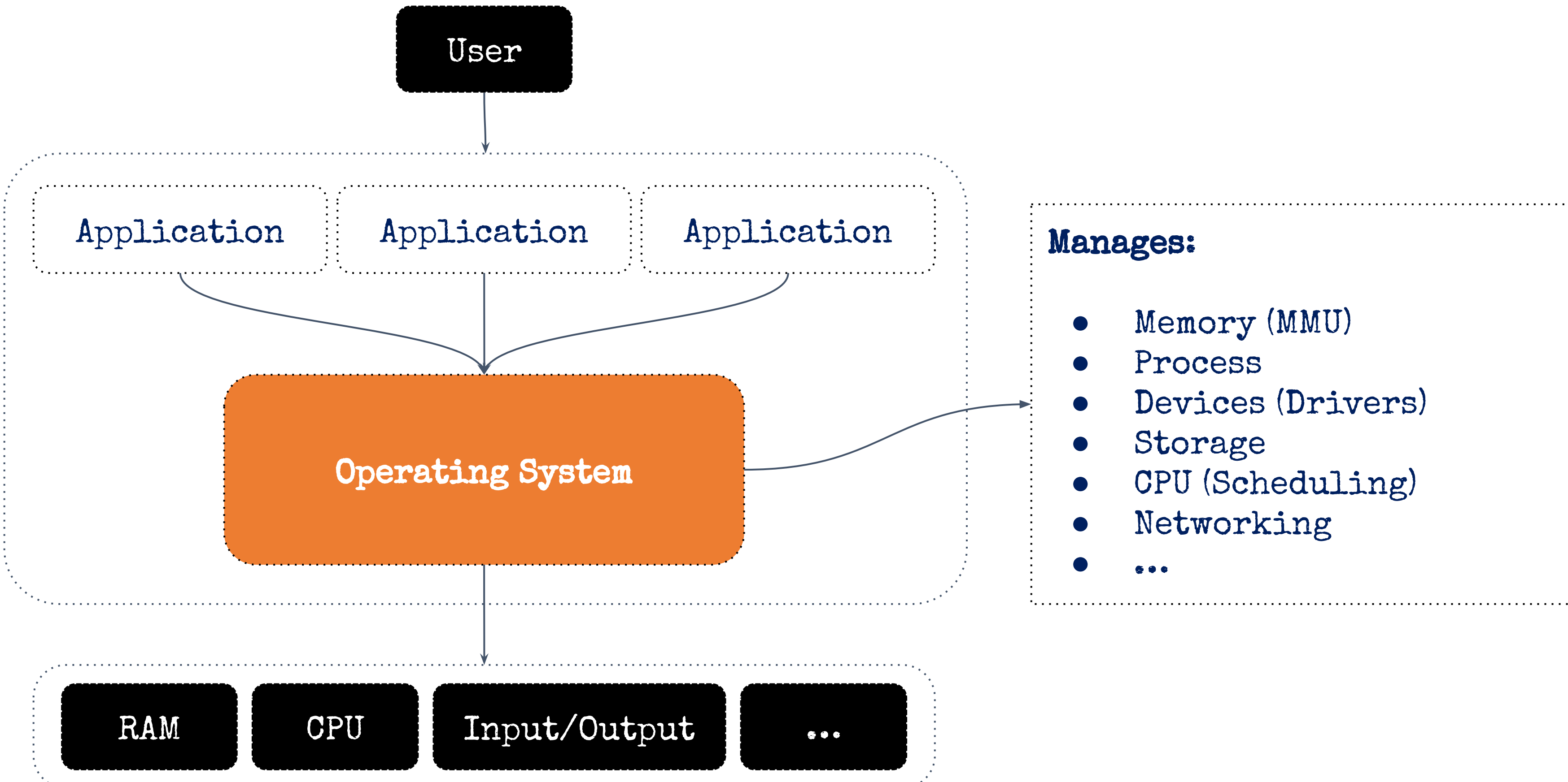
The Golden Notes and Roadmap for Embedded Linux



Talel BELHAJSALEM § bhstalel@gmail.com § linkedin.com/in/bhstalel/

Operating System



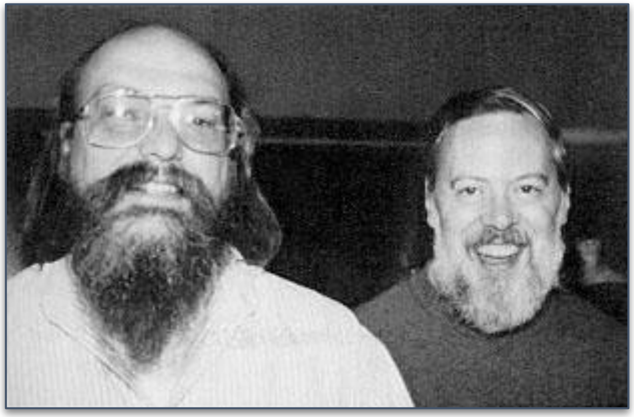
Unix

Multics

Unix

1960s

- **Multics** (**Mult**iplexed **I**nformation and **C**omputer **S**ervices)
- **Unix** (**Un**iplexed **I**nformation and **C**omputer **S**ervices) (Unics)
- Multics had many problems that Unics solved
- Unix provides:
  - Hierarchical file system
  - Processes
  - Command line interface
  - More utilities



Ken Thompson - Dennis Ritchie

POSIX

- **POSIX:** Portable **O**perating **S**ystem **I**nterface
- IEEE 1003.1 standard, 1980s
- Defines the language interface between app programs and UNIX OS
- Provides portability
- Defines:
  - **System interfaces and headers**
  - Commands and utilities

C Library

GNU

Multics — Unix — GNU

1960s — 1983

- **GNU:** GNU's **N**ot **U**nix
- Present the Free Software concept:
    - Freedom to run the software
    - Freedom to study and change the software
    - Freedom to redistribute the software
    - GNU **G**eneral **P**ublic **L**icense (**GPL**)
- Goal: create a whole free-software operating system
- Collection of free-software projects:
    - shell, coreutils (ls, ..) , compilers, libraries (C Lib), ..

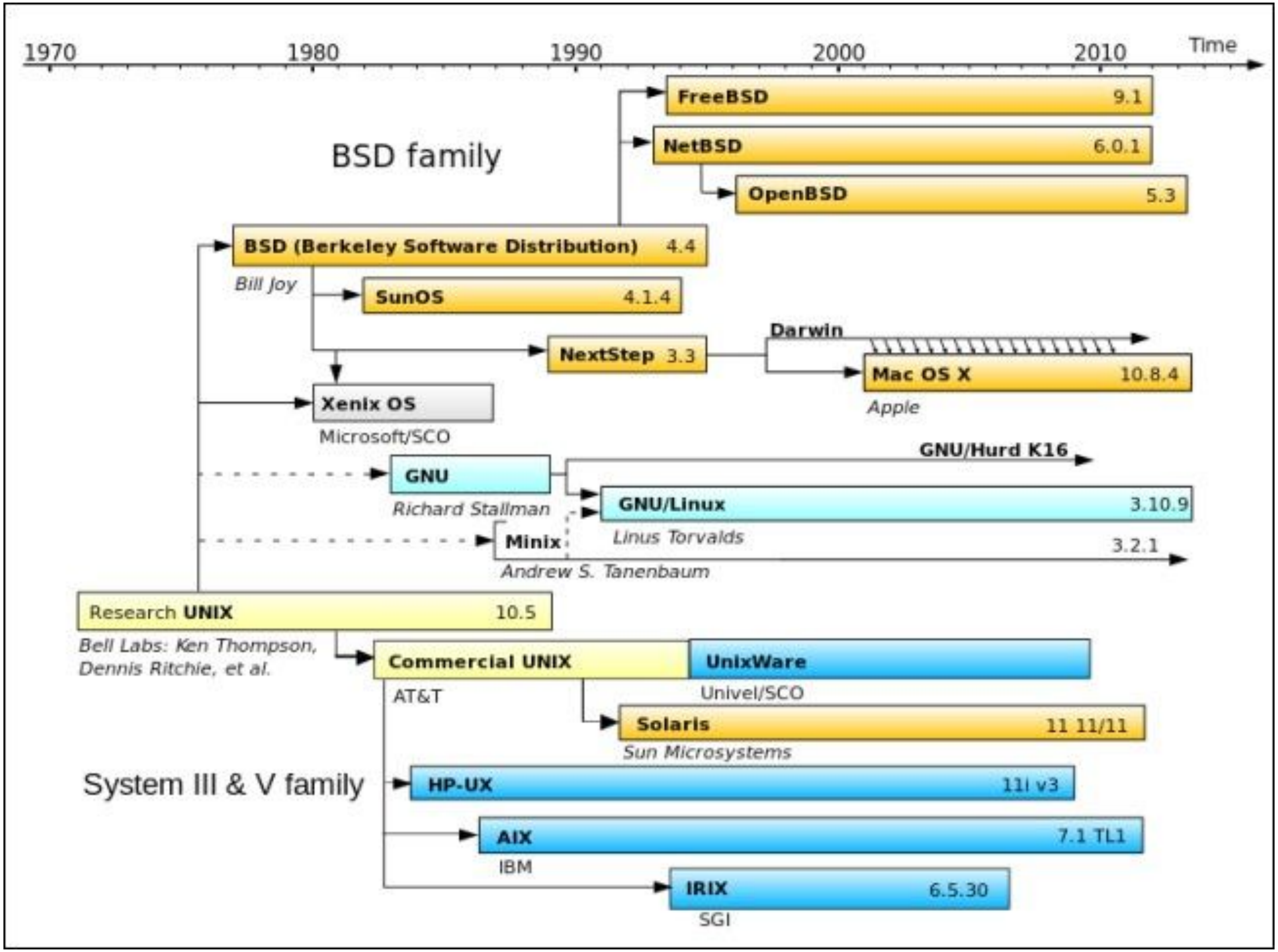
Linux

Multics Unix GNU Linux

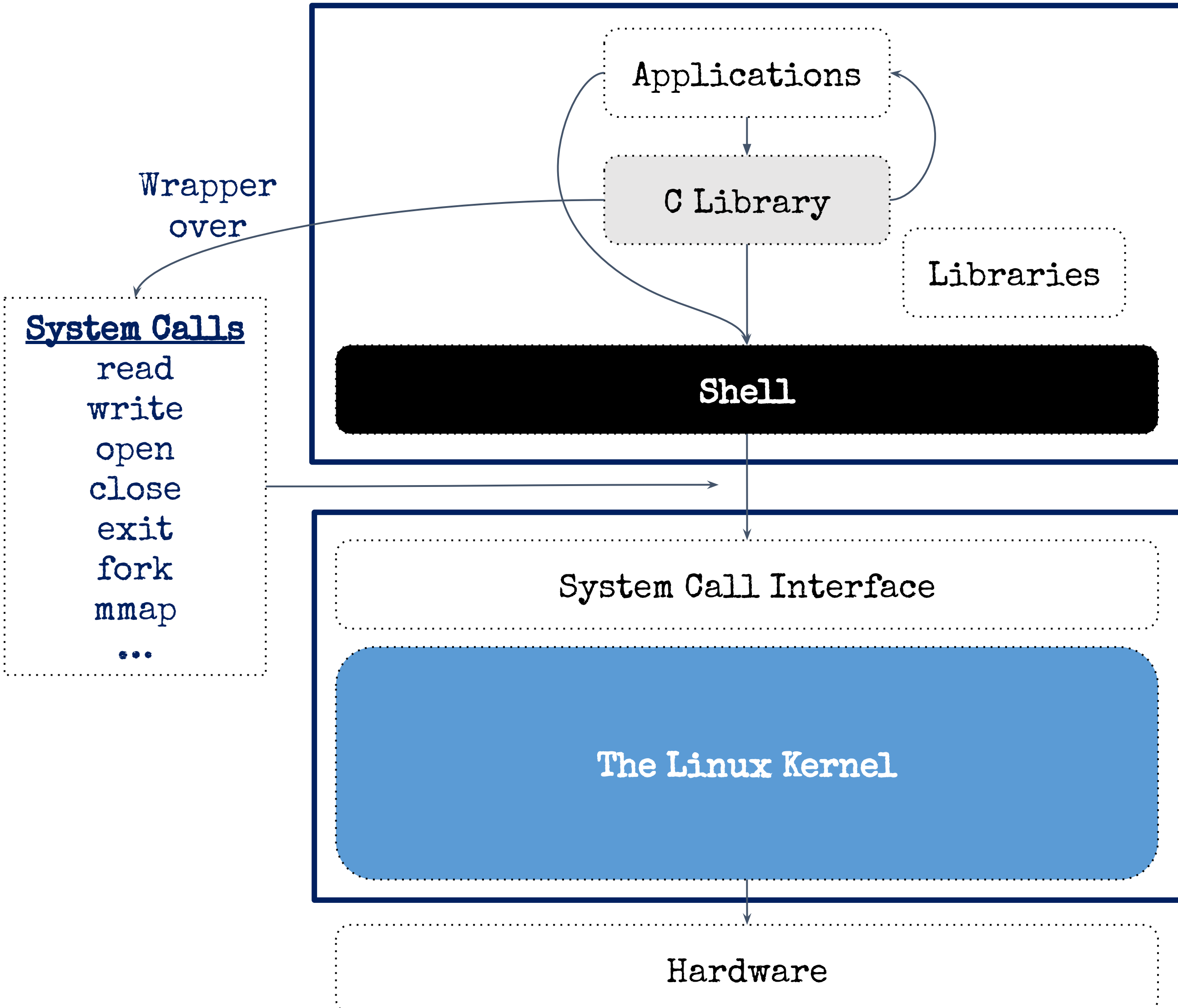
1960s 1983 1991

- Introduced by **Linus Torvalds**
- Licensed under version 2 of GPL (**GPLv2**)
- Used GNU **GCC** for compilation
- Advantages:
    - Low cost, full control
    - Community support
- **Unix-like** operating system
- **The kernel that GNU project needed**

Unix


1970
1980
1990
2000
2010
Time
BSD family
FreeBSD 9.1
NetBSD 6.0.1
OpenBSD 5.3
BSD (Berkeley Software Distribution) 4.4
Bill Joy
SunOS 4.1.4
Darwin
NextStep 3.3
Mac OS X 10.8.4
Apple
Xenix OS
Microsoft/SCO
GNU/Hurd K16
GNU
Richard Stallman
GNU/Linux 3.10.9
Linus Torvalds
Minix 3.2.1
Andrew S. Tanenbaum
Research UNIX 10.5
Bell Labs: Ken Thompson, Dennis Ritchie, et al.
Commercial UNIX
AT&T
UnixWare
Univel/SCO
Solaris 11 11/11
Sun Microsystems
System III & V family
HP-UX 11i v3
AIX 7.1 TL1
IBM
IRIX 6.5.30
SGI

Architecture



User Space:

- Unprivileged mode
- Run User Apps
- **Syscalls** used to switch to Kernel mode
- Organized in a **Root filesystem** and **pseudo-filesystems**

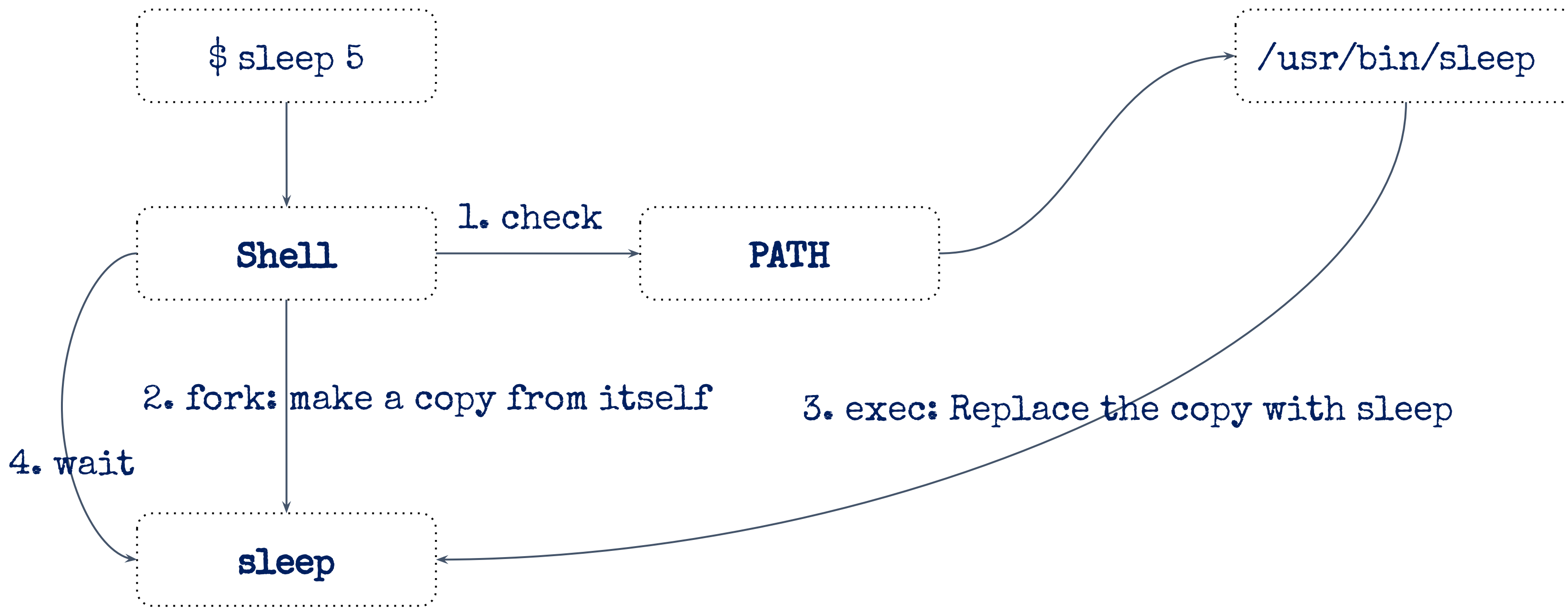
Kernel Space:

- The OS
- Privileged mode
- Run drivers to talk to the HW
- Allow user apps to run

Shell

- An application that runs other applications
- Since it is already running it has permissions to invoke system calls
- Running an application does not mean running it's instructions, it means it has to inform the Kernel to run it using the **exec*** system calls
- The shell is usually referred to as "**Terminal**" in which you write commands
- Shell started from Unix specification, implementations are various:
    - sh
    - bash (Bourne Again Shell)
    - zsh
    - fish
    - ...
- They all serve the same purpose, they differ in syntax and interpretations

Shell | Example



What's next?

- System calls are defined in **ABI: A**pplication **B**inary **I**nterface, read about it
- It is recommended to understand some Assembly because it is arch-specific
  - Understand how the Kernel handles the system call
  - Read about **vDSO**
  - Try to develop something without using the C Library

- Understand the **PATH** variable and other **environment variables** in Shell

- Understand the **ELF: E**xecutable & **L**inkable **F**ormat binary format in Linux
  - Understand the sections and headers
  - It has **.data, .text, .bss** (You probably heard of that before)
  - Learn how the Kernel runs an application using a **shebang**

- Learn about the **Root file system (rootfs)** and all folders under it (bin, sbin, etc, var, boot, usr, ...) and pseudo-filesystems (Essentially **procfs** and **sysfs, ...**)

What's next?

- Learn about the known shell utilities, known as **Coreutils** (ls, cd, mkdir, cp, mv, ...)

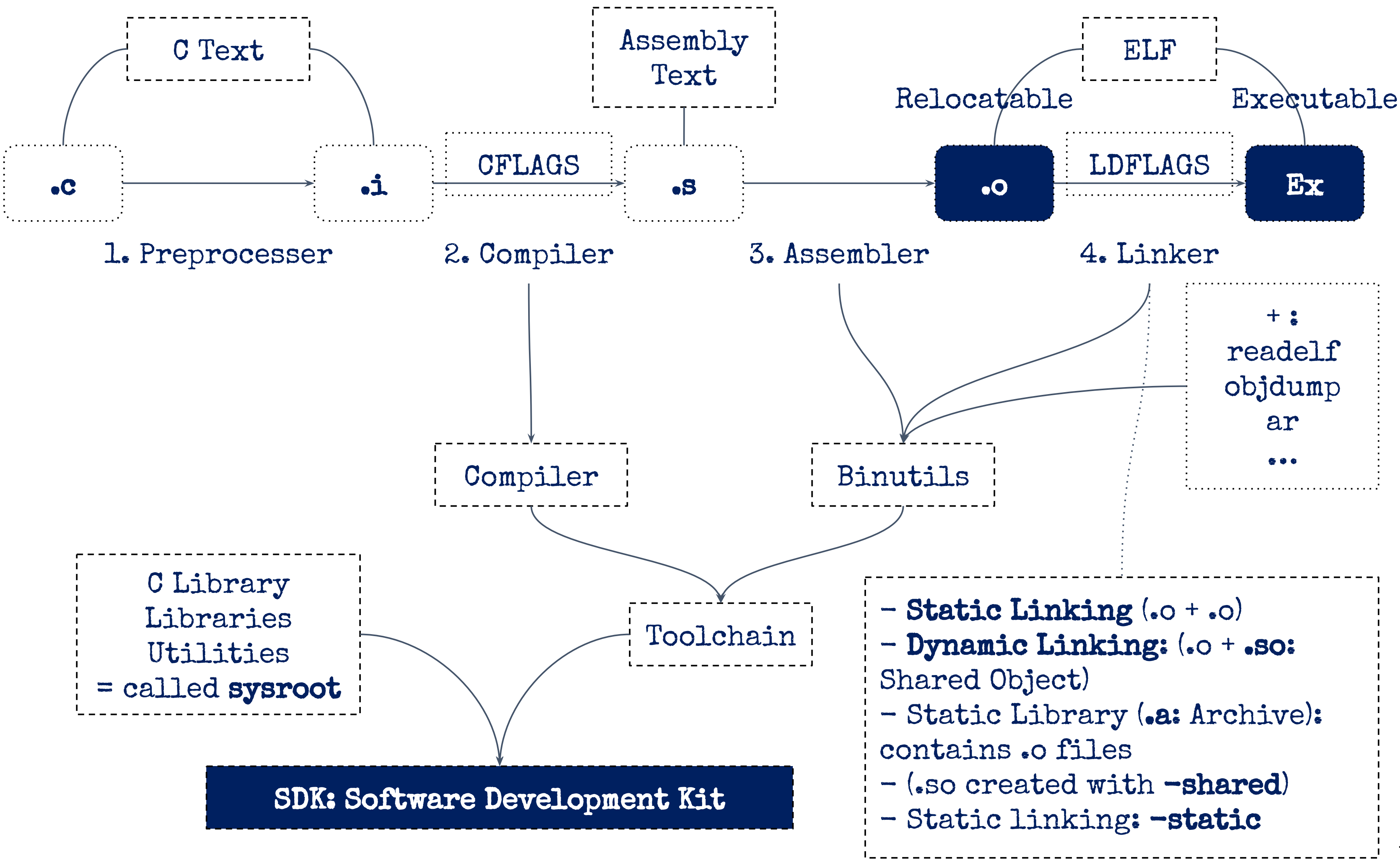
- Learn about shell programming:
  - **if, while, case, for**
  - **test** command and its options (-d, -f, -lt, -gt, ...)
  - Variables and how to access them
  - String **substitution** (Important topic)
  - **Functions**
  - **Arguments** passing and handling
  - The **set** command

- Learn about Environment variables:
  - What is **source** command
  - What is **export** command
  - What is a **shebang**

- Learn about IMPORTANT stuff:
  - Input and Output redirecting (**<, <<, >, >>**)
  - Piping (**|**)

- Learn advanced commands: **awk, sed, grep, ...**

Compilation



What's next?

- GNU has developed a compiler collection: **GCC GNU Compiler Collection**
  - It has: gcc (GNU C Compiler) , g++, ...

- Manipulate the compilation process manually:
  - Preprocessor: (**cpp**) or (**gcc -E**)
  - Compiler: **gcc -S**
  - Assembler: (**as *.i**) or (**gcc -c**)
  - Linker: (**ld:** not recommended as it is complex) or (**gcc *.o**)

- Generate **.a** files with (**ar**) and link with them

- Generate **.so** files and link with them
  - Learn about **"ldd", LD_LIBRARY_PATH, ldconfig** and **SONAME.**

Types of compilation

BUILD		HOST		TARGET	
	=		=		~Native
	=		!=		~Cross
	!=		=		~Cross-Native
	!=		!=		~Canadian

- **Build:** The architecture of the part that prepares the compiler

- **Host:** The architecture of the part that runs the compiler

- **Target:** The architecture of the part that will run the compiled binary

- Example:
  - Building on an intel i7 **x86-64** with **gcc** and runs on the same PC: **Native**
  - Building on an intel i7 **x86-64** with **arm-gcc** and runs on **RPI: Cross**
  - Building a gcc on intel i7 **x86-64** to run on **RPI** and build for **RPI: Cross-Native**
  - Canadian is not really used, or really ?

Cross Compilation

- To cross compile for a Linux target system, you need to answer 4 questions:
  - What **C Library** used in the target system?
  - What **Architecture**?
  - What **ABI** is used for the target architecture?
  - Is the target CPU has **FPU** (**F**loating **P**oint **U**nit)

- Answering the questions will lead to the following pattern:
  - **<arch>-linux-<Clib><abi><fpu>**

- Examples:
  - **arm-linux-gnueabihf** (ARM, GNU CLib, EABI, HF: Hardware Float)
  - **arm-linux-musleabi** (ARM, Musl CLib, EABI, No FPU)

- You need a full SDK for cross compilation, essentially a Toolchain:
  - Example: **gcc-arm-linux-gnueabihf** and **binutils-arm-linux-gnueabihf**

- If no toolchain found for your combination, then you need to create one using:
  - **crosstool-ng**
  - **Yocto**
  - ...

What's next?

- Learn about other architectures Assembly (**ARM** and **RISCV**)
- Download and install a cross toolchain
- Do some cross compilation and examine the generated ELF file with **file** command
- Examine the Assembly output differences (To master registers and low level CPU stuff)
- Learn about **Qemu** to simulate the cross compiled binary

Build Systems

- Build systems are frameworks that help you automate the build process.
- How can you generate 1000 .o files from .c and link them manually?
  - Running `gcc -c f1.c` to 1000?
- How can you handle dependencies?
- How can you detect when to recompile a .c file (Always or only on modification?)
- How can you support linking process?
- And more and more questions are answered by build systems like: **make** and **cmake**

Build Systems | make

- Knowing about make is enough for starters.
- **make** is based on an input file, generally, called **Makefile** (it can support custom name)

The general rule of Makefile

```
target: dependencies
<TAB> command1
<TAB> command2
```

Makefile

```
main: main.c
	gcc main.c -o main
```

```
$ make main
gcc main.c -o main
$ ./main
```

Can be written as the following.

DON'T BE AFRAID OF LEARNING ADVANCED STUFF !

```
EXEC=main
CC=gcc
CFLAGS=
LDFLAGS=
$(EXEC): main.o
	$(CC) $< $(LDFLAGS) -o $@
%.o: %.c
	$(CC) -c $< $(CFLAGS)
```

What's next?

- Explore more about **Makefiles**:
  - How to handle all .c and .h files in the project automatically
  - Advanced techniques like functions, .PHONY and other

- Learn about **cmake** as it is a wrapper over make and other build systems

- Document about other build systems (*DO NOT BE AFRAID, THEY SERVE THE SAME PURPOSE*)
  - bazel
  - ninja
  - meson
  - conan
  - vcpkg
  - ...

- Or, create your own?
  - I have developed one in Rust and Python
  - Rust: https://github.com/bhstalel/rmake-demo
  - Python: https://github.com/bhstalel/pymake-demo

POSIX Programming

- Any running program is in fact, when not running, an ELF file
- When it gets running it becomes a: **Process** that has a unique ID: **PID**
- A Process is a context of a running ELF file
- A **thread** is just a sequence of instructions
- A Process has at least one thread which is the **main thread**
- A Process can have multiple threads, so it is called: **Multithreading**
- A Process can invoke another Process called its Child: **Multiprocessing**
  - Multiprocessing is the same concept of shell: fork+exec*

POSIX Programming

- Parent and Child processes share the same global data
- Multiple threads in one Process share everything in the context except the stack
- This sharing case needs a synchronization mechanism like:
  - Mutex
  - Semaphore
  - Manual locking
  - Other, ...
- POSIX provides library that handles Multithreading and Multiprocessing
  - Example: <pthread.h> for C
- To access files in the HW, system calls need to be invoked for the Kernel (open, ...)
  - This topic is "**File handling**" in Linux

Inter Process Communication

- Since Process is in a separate context of other processes, it cannot communicate with other processes, unless, you use one of so called “**Inter Process Communication**”:
    - Shared memory
    - File sharing
    - Sockets (Unix, UDP, TCP, ...)
    - RPC: Remote Procedure Call (gRPC, xmlrpc, ...)
    - D-Bus
    - Other, ...

- All so called “**microservices**” are just processes talking to each other via IPC.

What's next?

- Develop a program that runs a Child process and check their PIDs
  - Develop your custom C Shell that takes input and invokes fork, exec* and wait
- Develop multithreading programs and check their **TIDs** (Thread ID)
- Implement Mutex and other sync mechanisms
- Try IPC mechanisms and learn about them deeply

Kernel | Kbuild

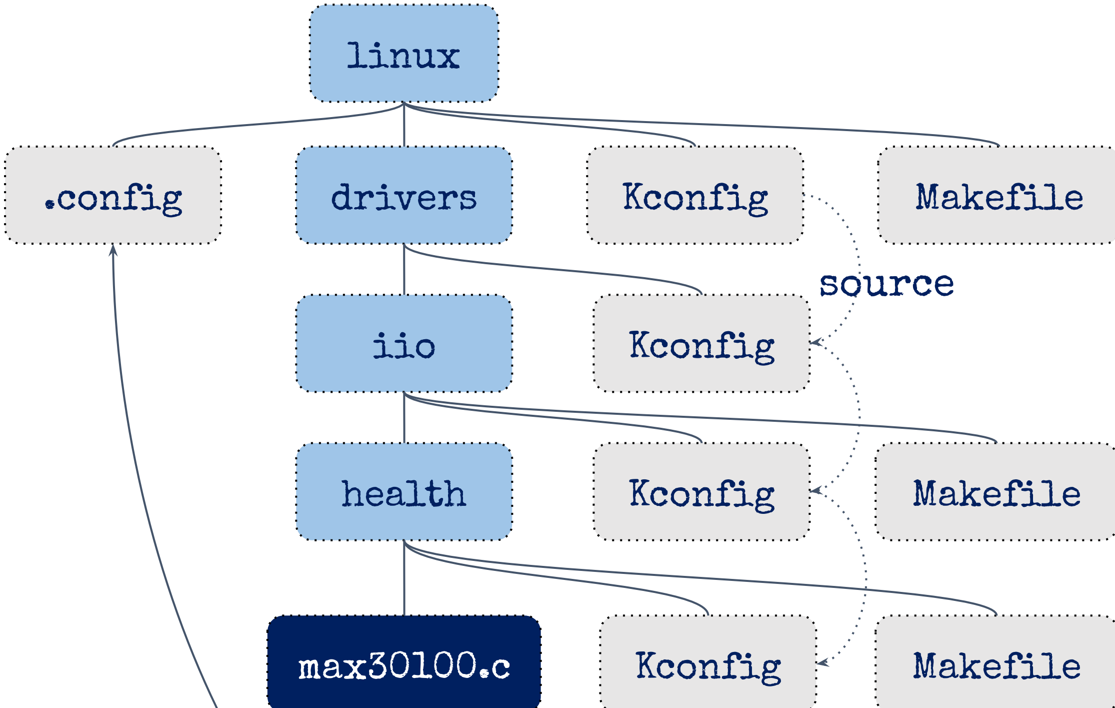
- Kbuild system is a way to compile and manage Kernel components in a way that make the Kernel so modular and can be adapted to any need.
  - Example: No need for the Kernel to know WIFI if no WIFI used in the project

- Kbuild is based on 4 parts:
  - **Symbol:** A component that has a name, description, type and dependencies
  - **Kconfig:** A file that holds the symbols
  - **.config:** A file that holds your choice of the symbols
  - **Makefile:** Main build file that has all build targets

- Think of this as a restaurant:
  - It provides recipes which are symbols
  - All recipes are listed in a menu that is Kconfig
  - You need to choose something that the waiter will note down that is .config
  - It has a kitchen and tools to cook that is Makefile

- Managing the choosing process manually is not recommended due to dependencies
  - So, a menu tool is developed to help you with that
  - The purpose is to show you all symbols and it will adapt .config automatically

Kernel | Kbuild



obj-$(**CONFIG_MAX30100**) += max30100.o

This variable will come from **.config**, so at the end it is one of:

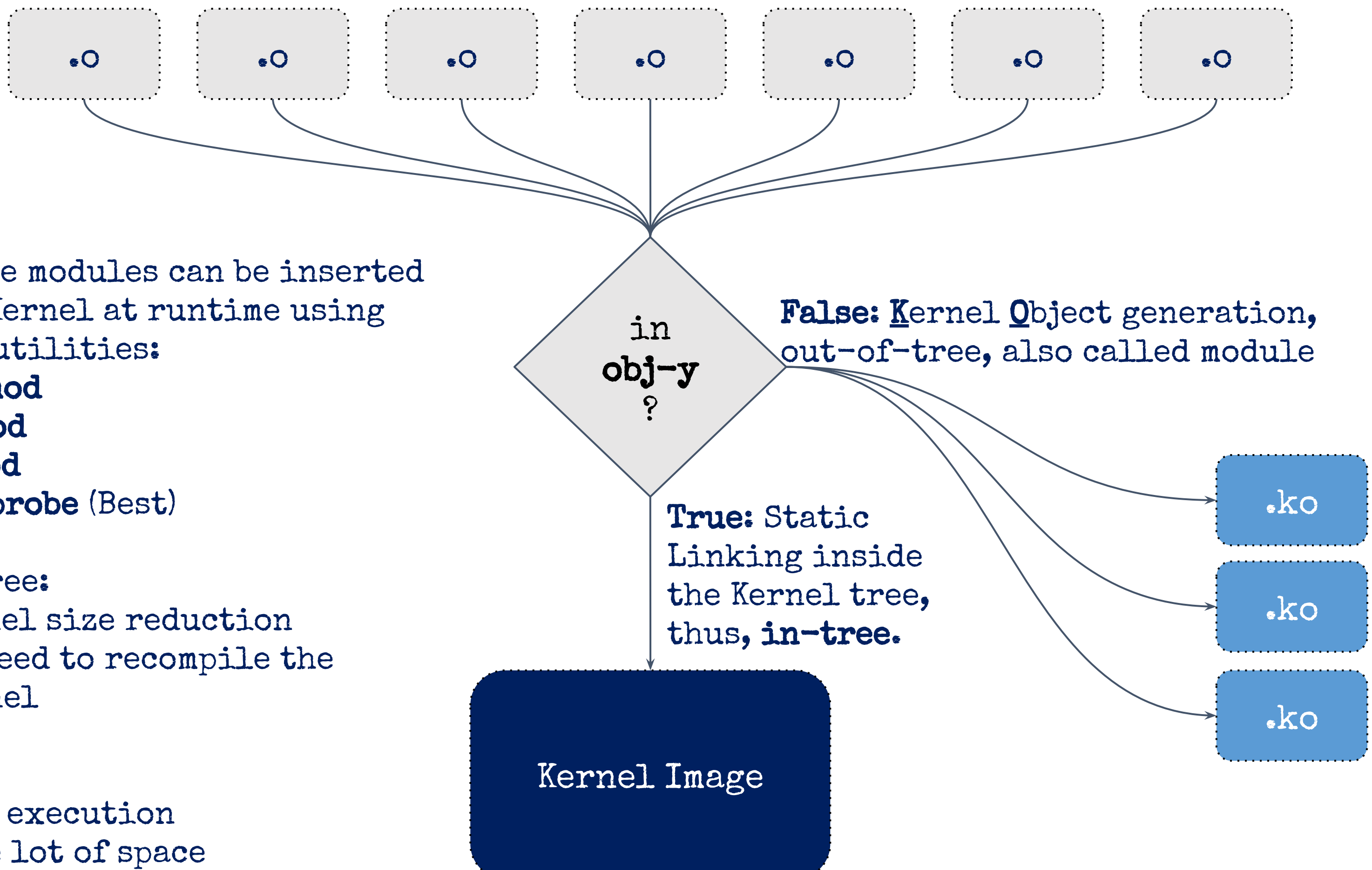
obj-y += max30100.o
obj-m += max30100.o
obj- += max30100.o

CONFIG_MAX30100=?
Since type is tristate, the "?" can be:
y (*In tree*)
m (*Module*)
Or the full line is commented

```
config MAX30100
        tristate "MAX30100 heart rate and pulse oximeter sensor"
        depends on I2C
        select REGMAP_I2C
        select IIO_BUFFER
        select IIO_KFIFO_BUF
        help
          Say Y here to build I2C interface support for the Maxim
          MAX30100 heart rate, and pulse oximeter sensor.

          To compile this driver as a module, choose M here: the
          module will be called max30100.
```

Kernel | Kbuild

Assuming **.o** are not in **obj-** meaning that they are not disabled.



Out of tree modules can be inserted into the Kernel at runtime using the **kmod** utilities:

- **insmod**
- **rmmod**
- **lsmod**
- **modprobe** (Best)

out-of-tree:

- Kernel size reduction
- No need to recompile the kernel

in-tree:

- Fast execution
- Take lot of space
- Need to recompile the kernel

Kbuild | menuconfig

- **menuconfig** is one of the Makefile targets that compiles an ncurses application and runs the root **Kconfig** file on it and thus you get a menu that handles **.config** automatically.

- It makes a backup for **.config** named **.config.old** before doing any saving.
  - This helps using **diffconfig** utility to show the difference between the two
  - That is called: **Kernel Configuration Fragment (.cfg)**
  - Used to automatically apply a configuration on a preset of .config

- When working with a fresh Linux sources, you need to create a **.config** before working with menuconfig
  - This is usually done via **<name>_defconfig** target that tells Makefile to get a saved and ready defconfig file and copy it as **.config.**
  - This is usually saved in: **linux/arch/<ARCH>/configs**

- Example:
  - # Setup for cross compilation:
  - $ export ARCH=arm
  - $ export CROSS_COMPILE=arm-linux-gnueabihf-
  - $ make defconfig   # Prepare the .config file
  - $ make menuconfig   # Opens the menu utility
  - $ make modules   # Compile only out-of-tree modules
  - $ make   # Compile the full Kernel Image

Kernel | Development

- There is no C Library in the Kernel, it has its own library
- It is up to the Kernel to pass information to userspace on what its doing
  - **procfs** (Process management information)
  - **sysfs** (Information about drivers and modules)
- Each subsystem has its own API and they are all well developed and documented
- Most development in the Kernel is in Device Drivers section, it's +95% of source code.
- Two parts of development:
  - **Core development:** Memory Management, Process Management, VFS, Networking, ...
  - **Device Drivers Development:** I²C, SPI, USB, IIO, Regmap, ...
- The Kernel provides lot of features and libraries (API).
  - You should focus on one subsystem at a time to understand it.
- Device Drivers development requires understanding how the core works (Memory, ...)

What's next?

- Talking about the Kernel internals will take writing a full book.

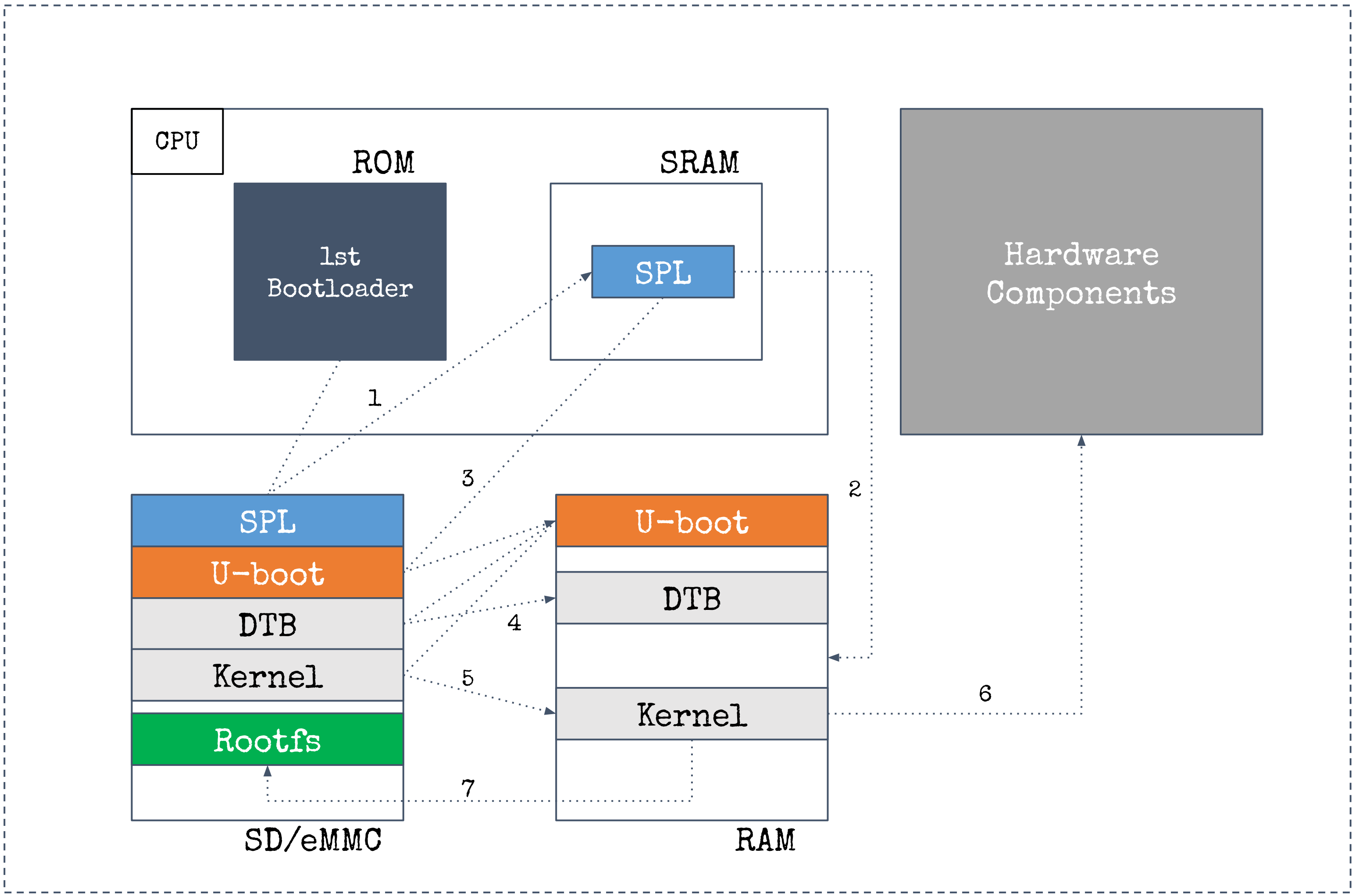
- Learn about **Memory Management:**
  - How kernel manages the **MMU** (**M**emory **M**anagement **U**nit)
  - How kernel organizes its Kernelspace
  - What is the API for memory (de)allocation? (**kmalloc, kzalloc, GFP:** **G**et **F**ree **P**age?)
  - How the kernel sees the memory via (**Zones, Pages, Frames**)
  - What is the **Slab** allocator?

- Learn about **Process Management**
  - How an ELF file gets a context (Process) and then its **.text** gets executed.
  - How Kernel **schedules** all processes

- Learn a simple Hello world Kernel out-of-tree module
  - Compile it
  - Insert it and examine its output in the userspace
  - Remove it and manipulate it

- Learn about common stuff between device drivers:
  - **Char device drivers** (**Minor** and **Major** numbers, **File operations, ...**)
  - **Classes**
  - **I²C, SPI, UART, ..., Regmap**

Bootloader | Boot process



Bootloader | Boot process

- **SPL:** Second Program Loader: Initializes the RAM and loads TPL
- **TPL:** Third Program Loader (Infamous **U-boot,** or other): Load Kernel and DTB
- **DTB:** Device Tree Blob
    - Describes the full Hardware buses, components, ...
    - Used by the Kernel to know where to find stuff and how to deal with them
    - Used only by non-memory-mapped-io systems like ARM (Not x86-64)
    - DTB begins as Device Tree Source (**DTS**) and get compiled by Device Tree Compiler (**DTC**)
    - The Device Tree utility is part of the Linux sources itself
- At the end, the Kernel loads the first program (**init**) from the rootfs
    - There are other programs before init, but it is up to you to go that deep.
- The init program starts running other programs (fork+exec) until reaching the shell
- Usually it invokes what is called an **Init Manager** (systemd, sysvinit, busybox-init, ..)

Bootloader | Boot process

- Device trees are vendor specific (Arch-specific as well)
- Usually located under: **linux/arch/<ARCH>/boot/dts**
- It can be compiled with: **make dtbs** from root Linux Makefile
- Example:

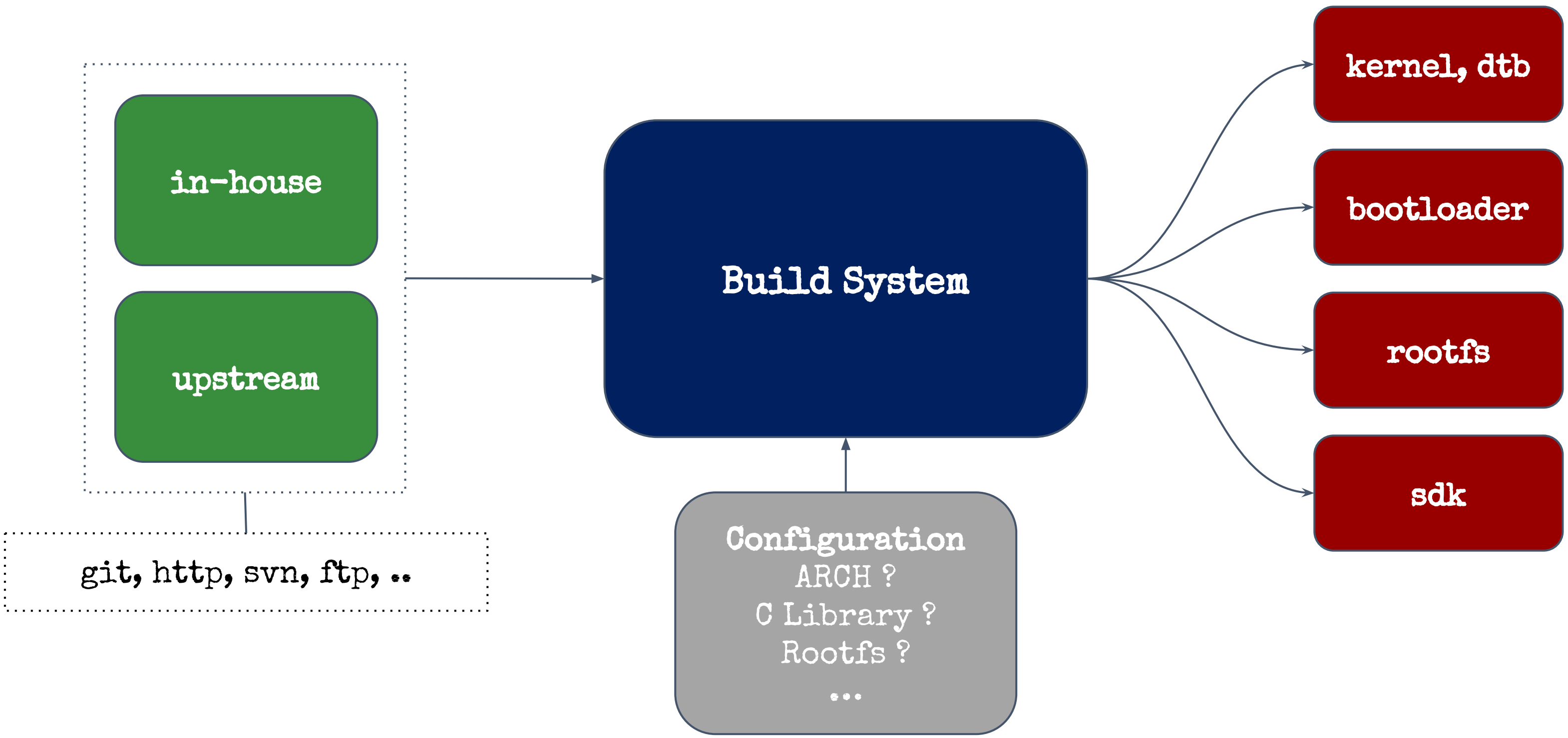
```
lcd_backlight: backlight {
      compatible = "pwm-backlight";                      // What driver to invoke

      pwms = <&pwm5 0 50000>;                            // Parameters
      brightness-levels = <0 4 8 16 32 64 128 255>;      // can be fetched from the driver
      default-brightness-level = <6>;                    // using the libof "Open Firmware" that created
                                                         // the specification of Device Tree

      status = "okay";                                   // Whether the device is present in the board
};
```

What's next?

- Download and compile Uboot
- Run it using **Qemu** and manipulate its CLI commands
- Create a simple SD card with **dd** and load it with **qemu-system** after Uboot and load files from there
- Learn about Device Tree Source syntax
  - You will need that if you have new board or extra hardware to solder
- Learn about Device Tree Overlay (**DTBO**)
  - What is it for?
  - How to create one?
  - How to load one in Uboot?
- Learn how to do "**Network booting**" to fetch Kernel and DTB from networking (**TFTP, NFS**)

Distribution Build System

- Build systems are used to build full distributions for your need
  - Far better than working with pre-built distros like Ubuntu, Raspbian, ...
- Distro build systems have same idea as Software build systems like make
- Example: Buildroot, Yocto, ...



What's next?

- This topic is huge and the more you master the previous topics of general Embedded Linux the more you understand build systems as:
    - They will fetch, prepare and build your software (Kernel, Uboot, Busybox, ...)
    - Prepare the type of compilation and toolchain, ...
    - Assembly the final image for you
- Learn about **Yocto**
- Prepare simple image for Qemux86-64 as an example and boot it

Embedded Linux Jobs

BSP (Board Support Package) Development

- Device Drivers development
- Kernel configuration and compilation
- Device Tree Source development
- RAM calibration and SPL development
- Assuring Boot Process
- ...

System Integration

- Yocto
- Buildroot
- ...

Software Development

- POSIX Programming
- Programming languages:
  - Shell (MUST)
  - C (MUST)
  - C++ (90% MUST)
  - Python (90% MUST)
  - Rust (50% MUST)
- Design Patterns: Singleton, Mediator, ...
- Inter Process Communication
- Graphics programming: SDL, Qt, ...
- Build systems: make, cmake, ...
- Debugging: GDB, Binary Ninja, ...
- ....

BELIEVE IN YOURSELF
YOU CAN DO ALL
:)

Going Beyond

- Mastering all previous content will make you capable of working in industry themes:
  - **Automotive:** Adds some protocols: **CAN, SOME/IP, ...**
  - **IoT:** Based on all 3 jobs, adds **MQTT** protocol (TCP), ...
  - **Robotics:** Has **ROS** (Robot Operating System) based on Linux with C++ and Python, ..
  - **Routers:** Based on **openWRT** which is based on **Buildroot,** just learn **Networking**

- Learn and work on Security topics:
  - Encrypting the root filesystem
  - TrustZone
  - Secure Boot

- Work on Boot time optimization:
  - Reducing Kernel image size
  - Choosing faster and smaller init manager

- Learn about **Virtualization** (Docker, LXC, ...)

- Learn about **Cloud** (AWS, Azure, ...)

Going Far Beyond | For Seniors

- Learn about Embedded Android
  - It is basically Embedded Linux with more stuff from Google

- Learn about Machine Learning and AI
  - For Robotics and IoT

- If you are really obsessed with Embedded Linux, be an avatar:
  - Electronics and PCB design
  - Embedded Linux
  - MCU
  - Embedded Android
  - Mechanics and 3D stuff

- This will lead you to be an embedded full stack engineer:
  - Create your own PCB
  - Develop your own drivers and port Linux by yourself
  - Develop user application as you need
  - Prepare appropriate container

- Learn about **Binary Exploitation** and Cyber Security

Resources to never miss

- **Mastering Embedded Linux Programming** Book by Chris Simmonds
- **Bootlin** Free Slides: https://bootlin.com/training/
- **Linux Device Drivers Development** Book by John Madieu
- **Advanced Programming in the UNIX environment** Youtube playlist: https://www.youtube.com/watch?v=QnL4eYpb5Iw&list=PL0qfF8MrJ-jxMfirAdxDs9zIiBg2WugOz
- **CS 361 Systems Programming** Youtube playlist: https://www.youtube.com/watch?v=TavEuAJ4z9A&list=PLhy9gU5W1fvUND_5mdpbNVHC1WCIaABbP
- **Operating Systems** Youtube playlist: https://www.youtube.com/watch?v=eby6bJVx4BA&list=PL1XjRDnU2tOipNUtu22aHUGC4SADqHrYF