#LEARN IN DEPTH

#Be professional in embedded system

Embedded Linux (PART 7)

- GNU CROSS-PLATFORM DEVELOPMENT TOOLCHAIN
- BOOTLOADERS
- THE BOOT SEQUENCE ON (DRAM SYSTEM)
- 3 PHASES BOOT SEQUENCE INSIDE BOOTLOADERS
- THE U-BOOT BOOTLOADER
- BOOTING FROM SDCARD / TFTP SERVER
- U-BOOT ENVIRONMENT VARIABLES

ENG.KEROLES SHENOUDA

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Part6 assignment Solution

CREATE MAKEFILE FOR BAREMETAL SW

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

#LEARN IN DEPTH

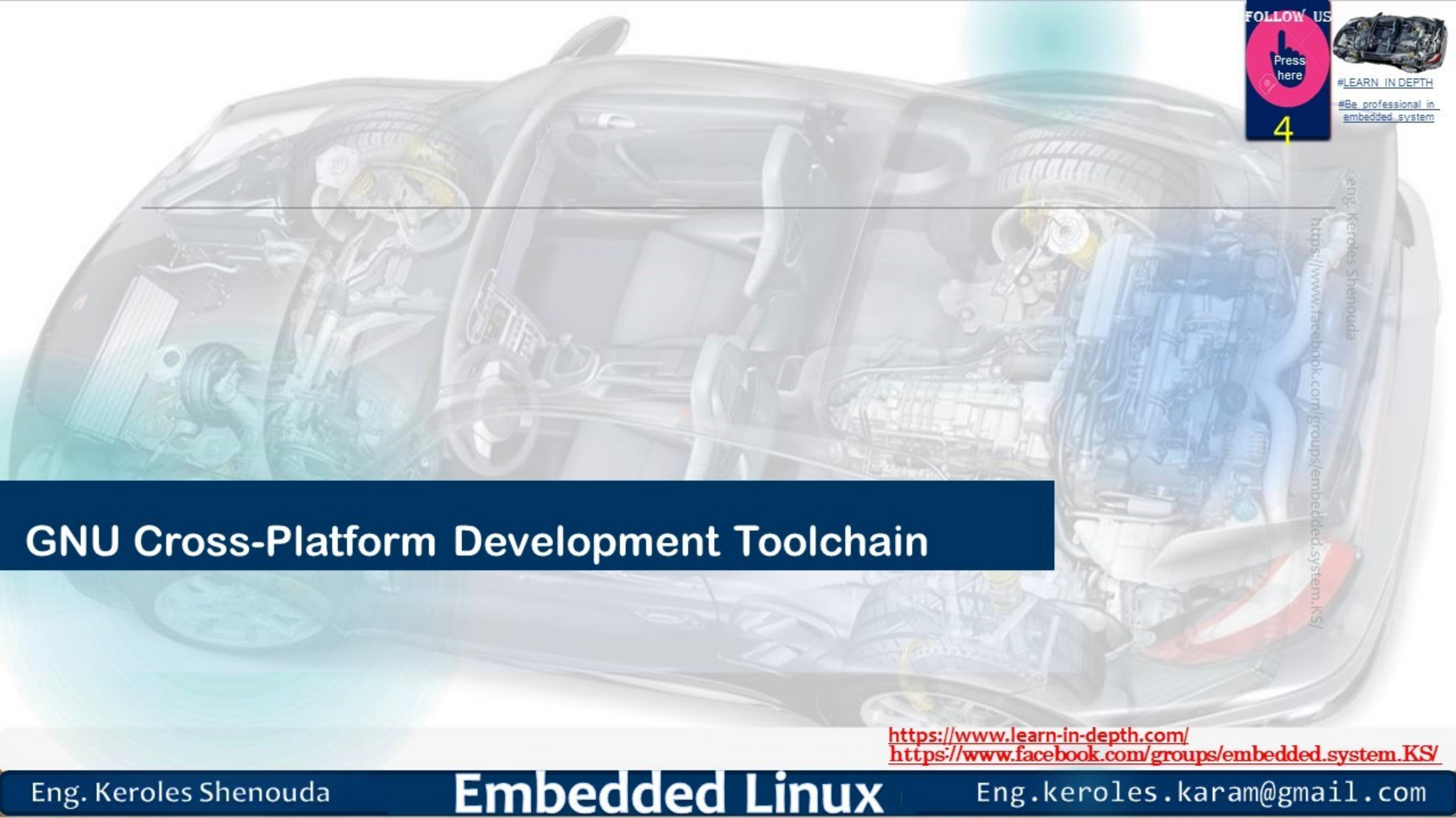
#Be professional in embedded system

Create MakeFILE for Baremetal SW

embedded_system_ks@embedded-KS: ~/toolchain_labs/lab1

```
#prepared Eng Keroles Shenouda
#lern-in-depth
CC=arm-none-eabi
CFLAGS= -I . -g
CPU= -mcpu=arm926ej-s
OBJ=test.o startup.o main.o

%.o: %.c
	$(CC)-gcc -c $(CPU)  $(CFLAGS)  $< -o $@

startup.o: startup.s
	$(CC)-as  $(CPU)  $(CFLAGS)  $<  -o  $@


test.elf: $(OBJ)
	$(CC)-ld -T test.ld $(OBJ) -o $@

test.bin: test.elf
	$(CC)-objcopy -O binary $< $@

all: test.bin
	echo "finished"


clean:
	rm test.o startup.o test.elf test.bin

.PHONY: test.bin

~
                                                    1,1           All
```


https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/



GNU Cross-Platform Development Toolchain

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

#LEARN IN DEPTH

#Be professional in embedded system

GNU follow a standardized format

cpu-manufacturer-kernel-os

The system's chip architecture. Where both a big-endian and little-endian variant exists,

A specific maker or family of boards using the aforementioned CPU. As this rarely has any effect on the toolchain itself

Used mainly for GNU/Linux systems

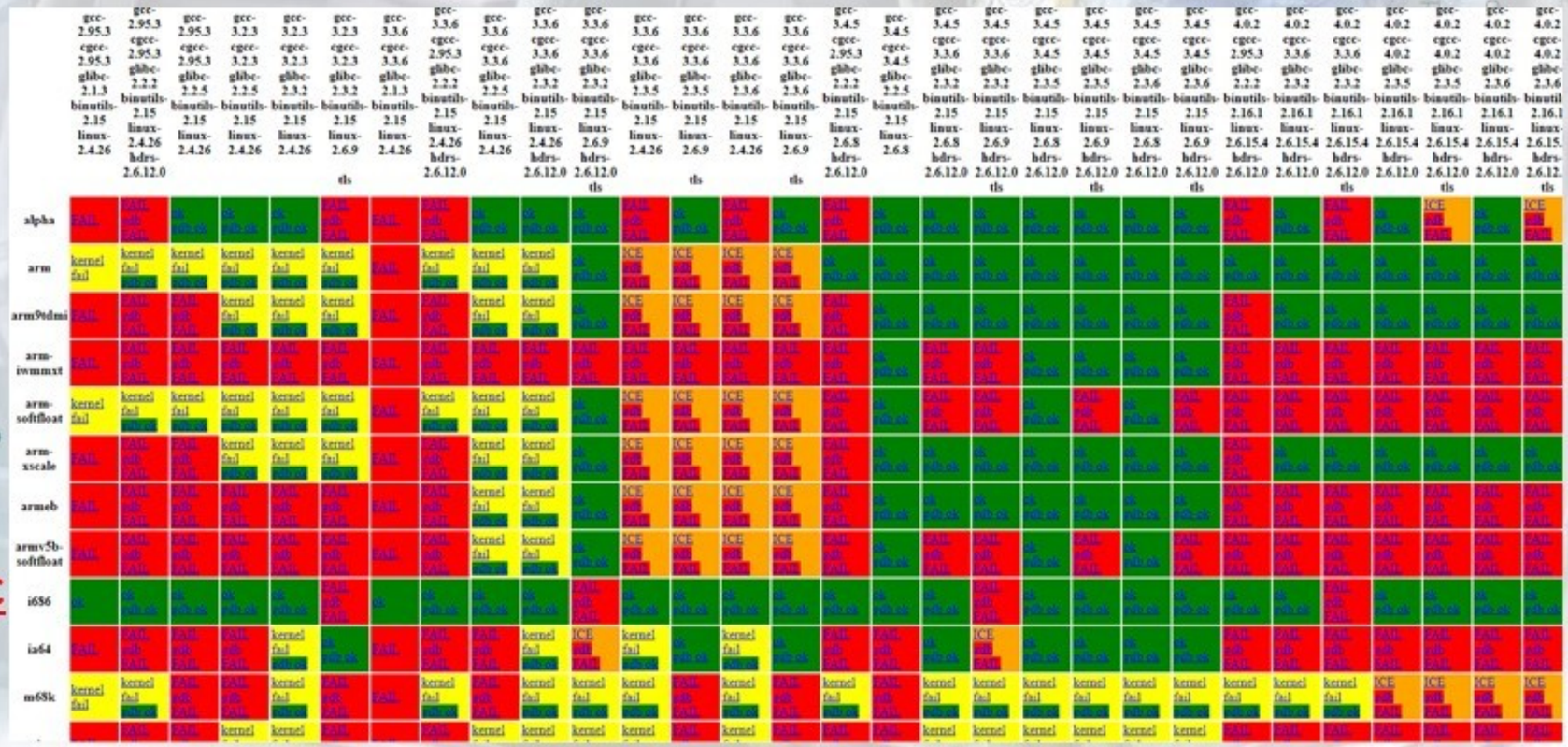
The name of the operating system (or ABI) used on the system. Configuration names may be used to describe all sorts of systems, including embedded systems that do not run any operating system

Some examples of possible host, target, or build triplets follow:

- *i386-pc-linux-gnu*
  - A PC-style x86 Linux system
- *powerpc-8540-linux-gnu*
  - A Freescale 8540 PowerQuickIII Linux system
- *mips-unknown-linux*
  - MIPS Linux system from an unspecified manufacturer

Crosstool build matrix

- The following matrix shows whether the given combinations of gcc, glibc, binutils, and linux kernel headers, lightly patched, can build a cross-toolchain and compile a kernel for the given CPUs
- http://www.kegel.com/crosstool/crosstool-0.43/buildlogs/



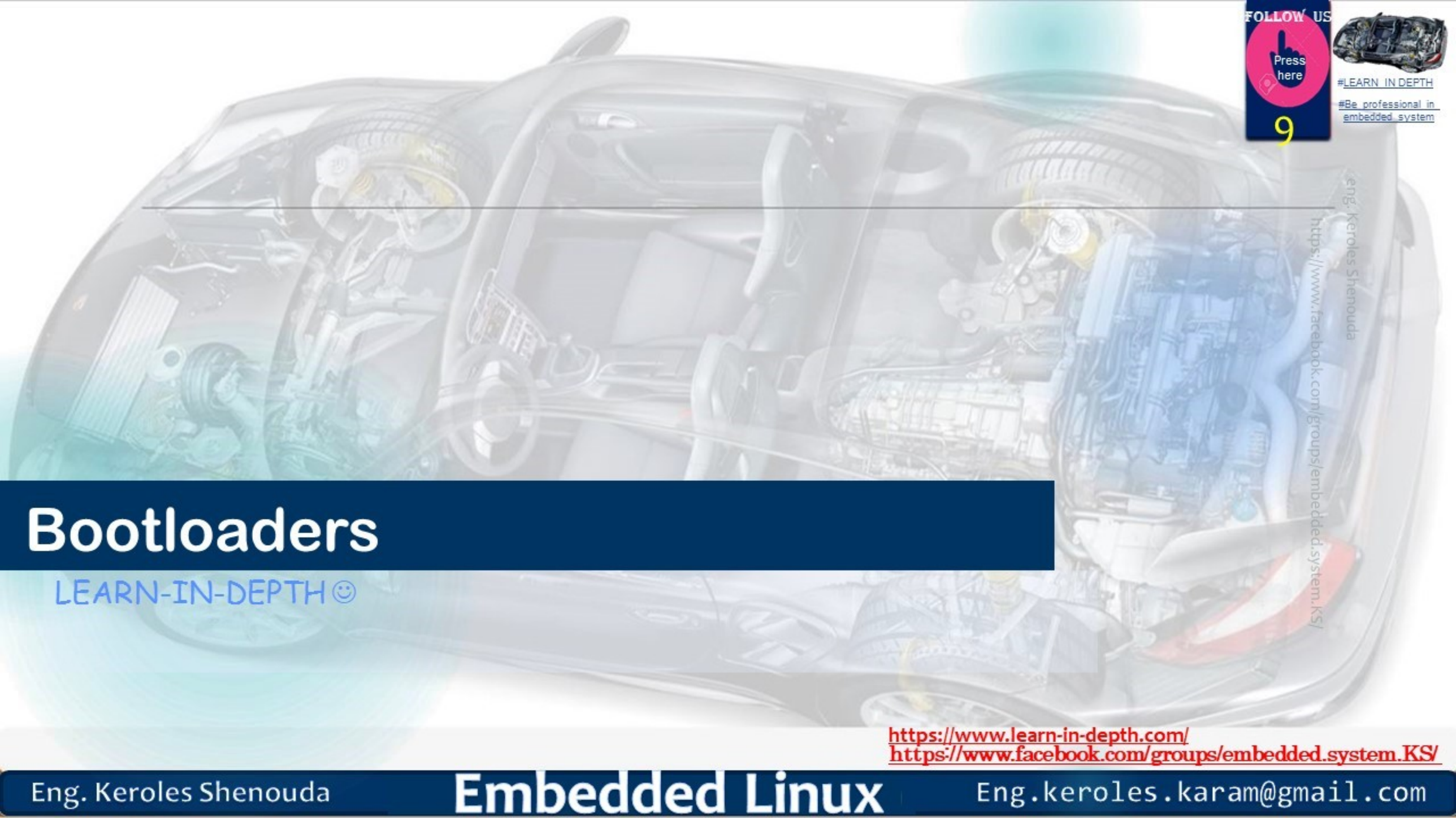
https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Using the Toolchain in makefile

```
# Tool names
CROSS_COMPILE   = ${TARGET}-
AS              = $(CROSS_COMPILE)as
AR              = $(CROSS_COMPILE)ar
CC              = $(CROSS_COMPILE)gcc
CPP             = $(CC) -E
LD              = $(CROSS_COMPILE)ld
NM              = $(CROSS_COMPILE)nm
OBJCOPY         = $(CROSS_COMPILE)objcopy
OBJDUMP         = $(CROSS_COMPILE)objdump
RANLIB          = $(CROSS_COMPILE)ranlib
READELF          = $(CROSS_COMPILE)readelf
SIZE            = $(CROSS_COMPILE)size
STRINGS         = $(CROSS_COMPILE)strings
STRIP            = $(CROSS_COMPILE)strip
```

Utility	Use
as	GNU assembler
ld	GNU linker
gasp	GNU assembler pre-processor
ar	Creates and manipulates archive content
nmu	Lists the symbols in an object file
objcopy	Copies and translates object files
objdump	Displays information about the content of object files
ranlib	Generates an index to the content of an archive
readelf	Displays information about an ELF format object file
size	Lists the sizes of sections within an object file
strings	Prints the strings of printable characters in object files
strip	Strips symbols from object files
c++filt	Converts low-level, mangled assembly labels resulting from overloaded C++ functions to their user-level names
addr2line	Converts addresses into line numbers within original source files

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/



Bootloaders

LEARN-IN-DEPTH ☺

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

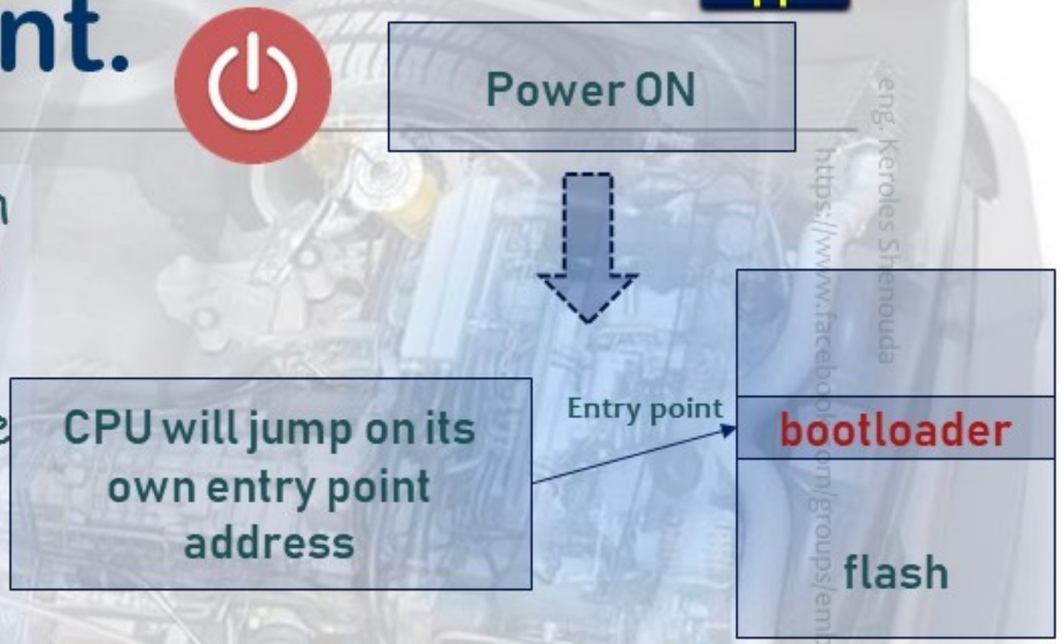
Role of a Bootloader

Power ON

- When ***power*** is first applied to a processor board, many elements of hardware **must be initialized** before even the simplest program can run.

Role of a Bootloader Cont.

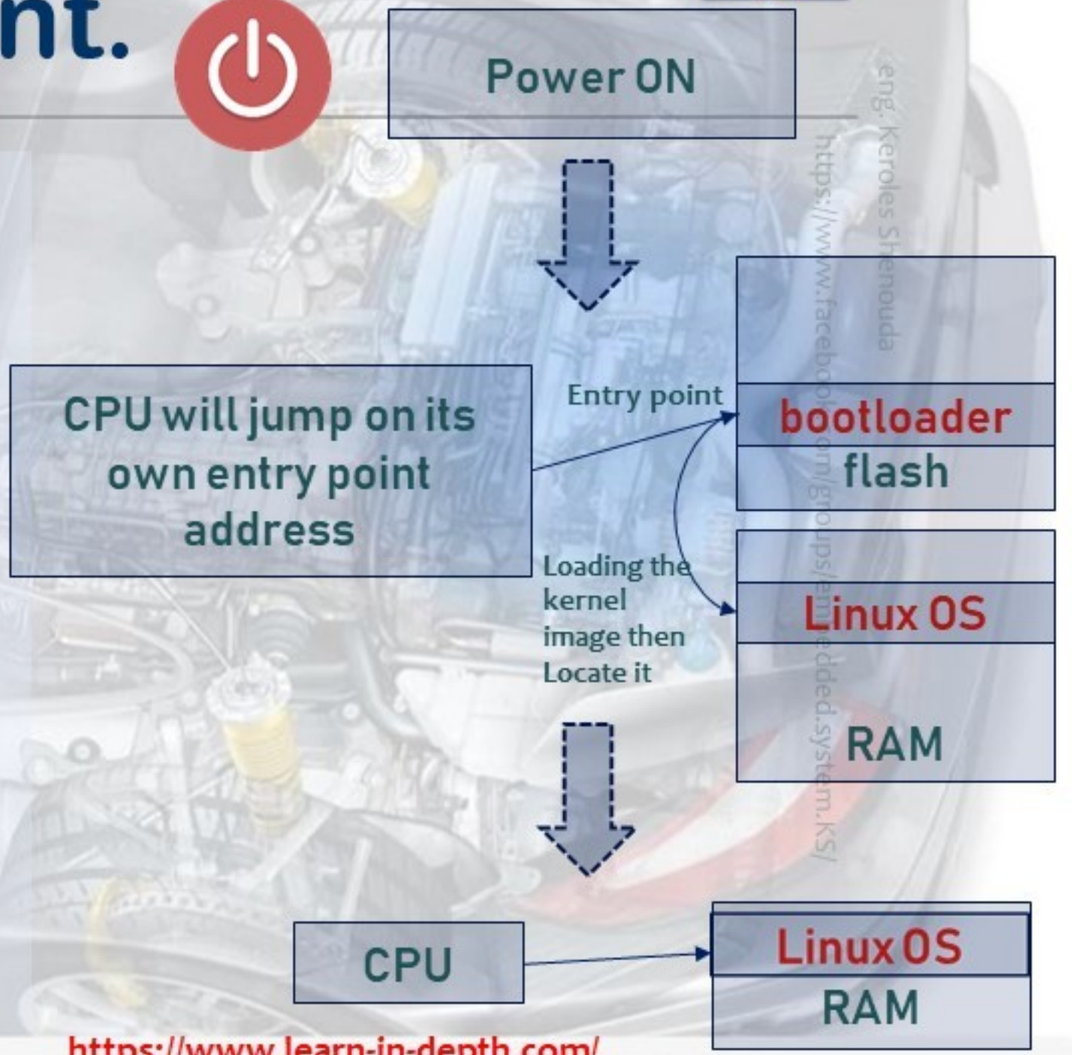
- Most processors have a **default address** from which the first bytes of code are fetched upon application of power and release of reset.
- **Hardware designers** use this information to arrange *the layout of Flash memory on the board* and to select which address range(s) the Flash memory responds to.
- This way, **when power is first applied**, code is fetched from **a well-known and predictable address**, and software control can be established.



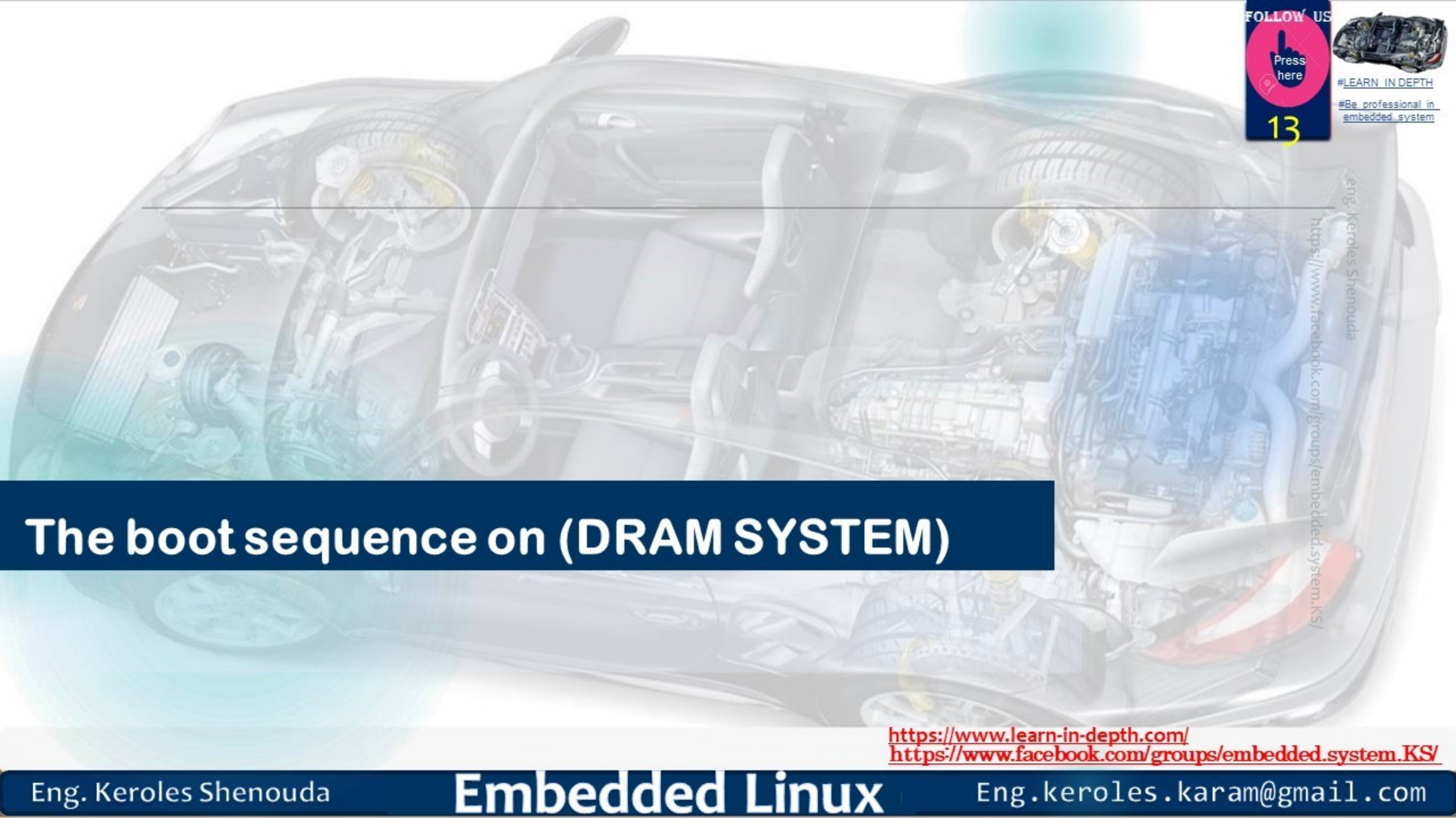
https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Role of a Bootloader Cont.

- The **bootloader** provides this **early initialization code** and is responsible for initializing the board so that other programs can run.
- after the bootloader has performed this basic processor and platform initialization, its primary role becomes booting a full-blown operating system
- It is responsible for **locating, loading, and passing execution** to the primary operating system.



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

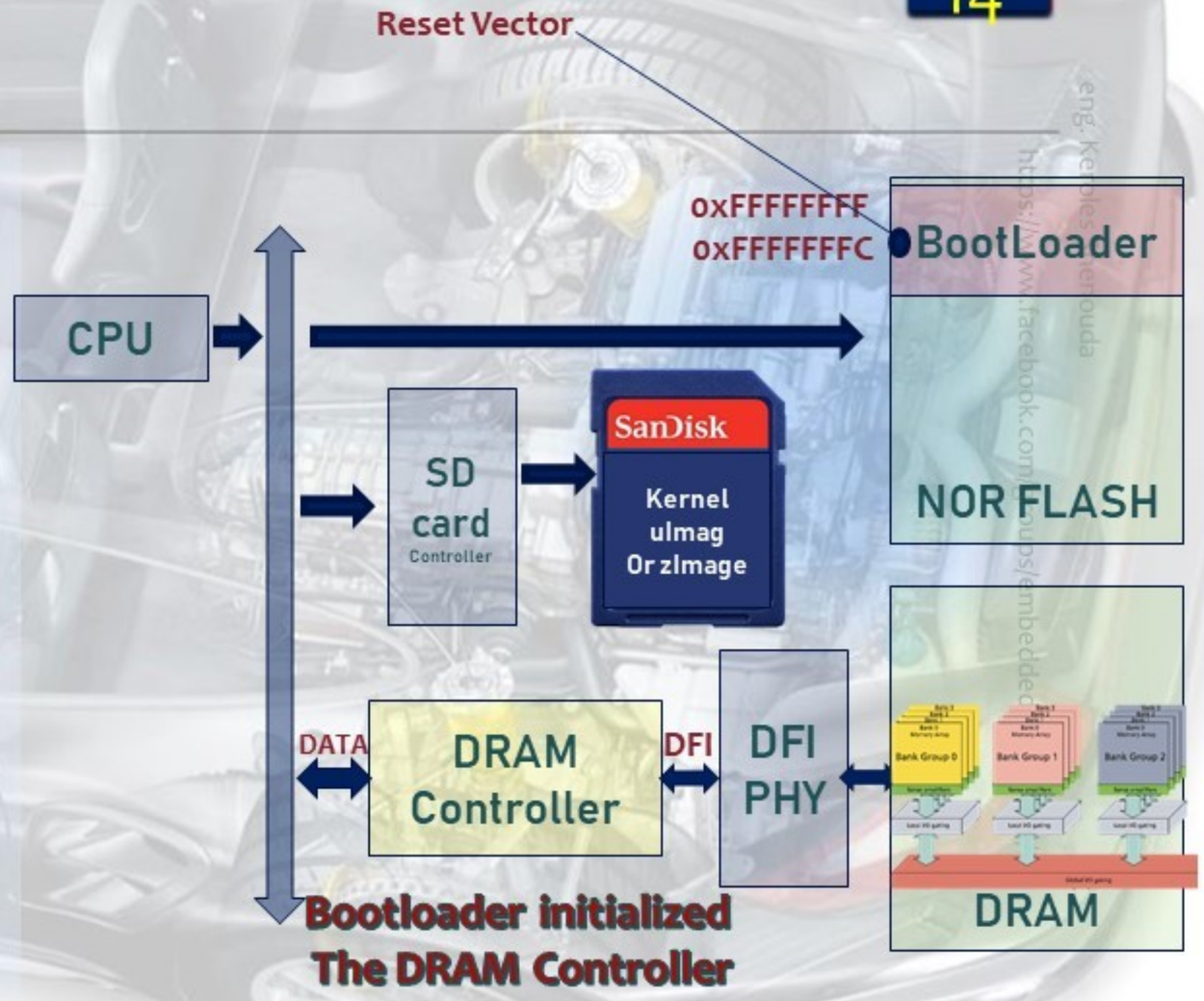


The boot sequence on (DRAM SYSTEM)

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

The boot sequence

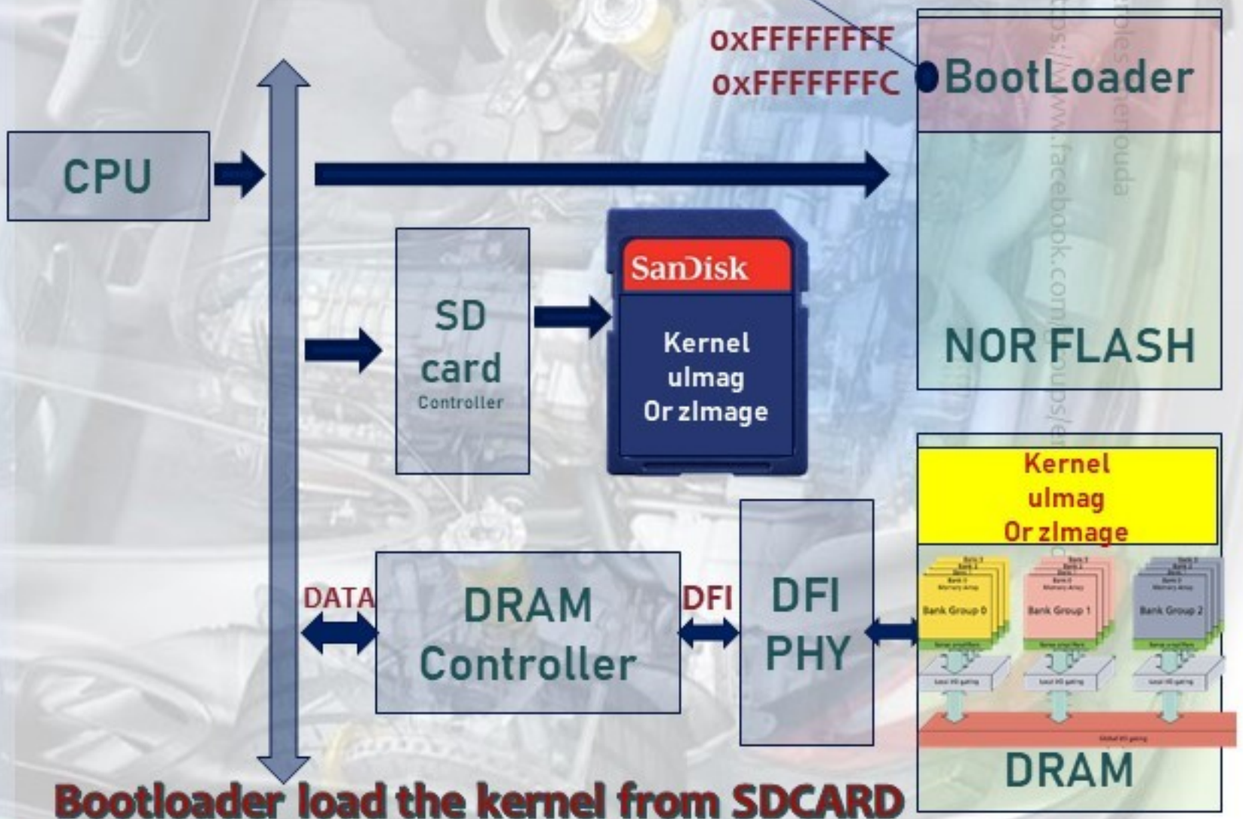
- the bootloader in non-volatile memory at **the reset vector of the processor**.
- From that point, the bootloader code running in NOR flash memory can **initialize** the DRAM controller, so that the main memory, the DRAM, becomes available
- The bootloader can **load** the kernel from **flash memory** into **DRAM** and transfer control to it.



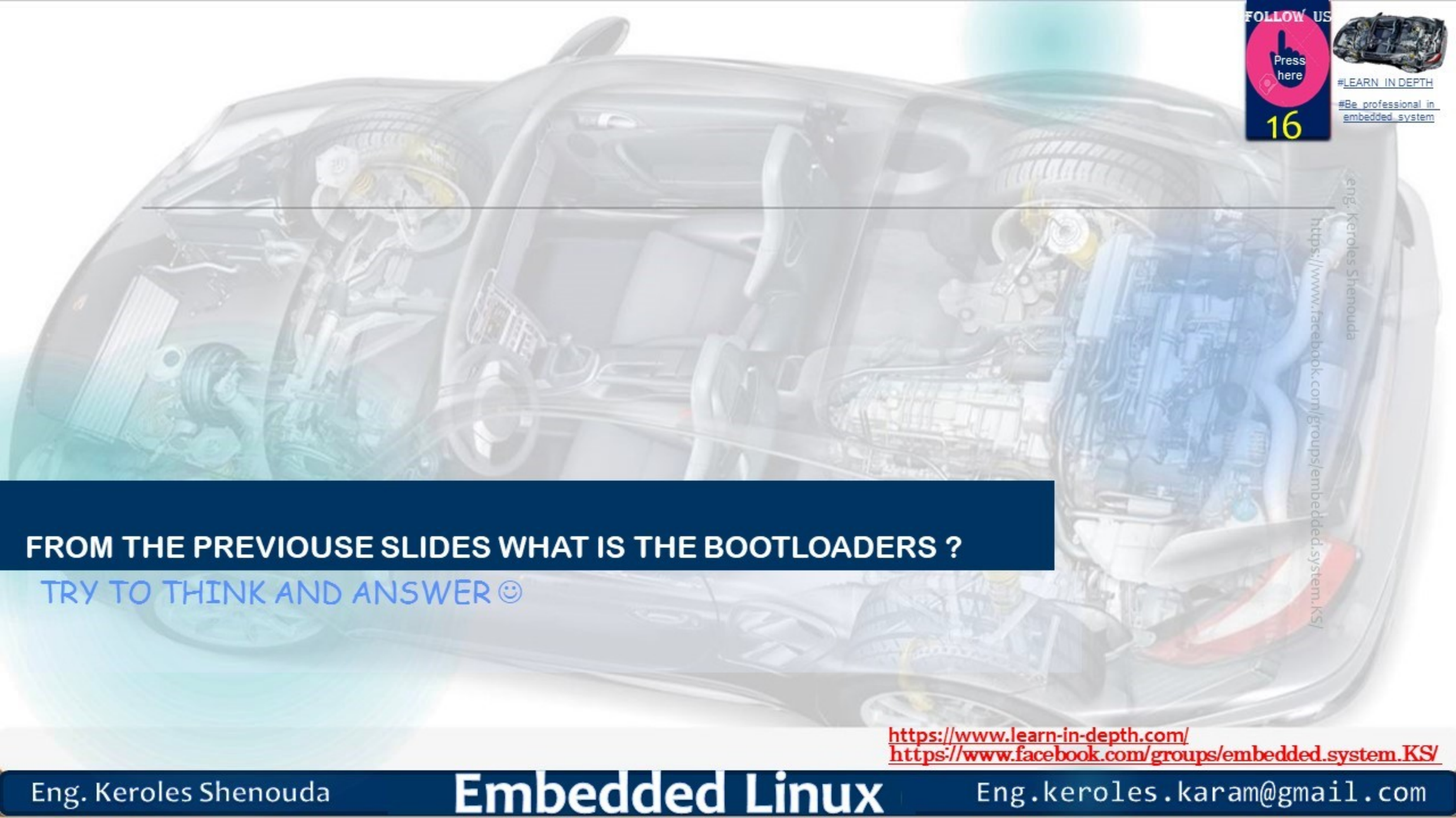
https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

The boot sequence Cont.

- the bootloader in non-volatile memory at **the reset vector of the processor**.
- From that point, the bootloader code running in NOR flash memory can **initialize** the DRAM controller, so that the main memory, the DRAM, becomes available
- The bootloader can **load** the kernel from **flash memory** into **DRAM** and transfer control to it.



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/



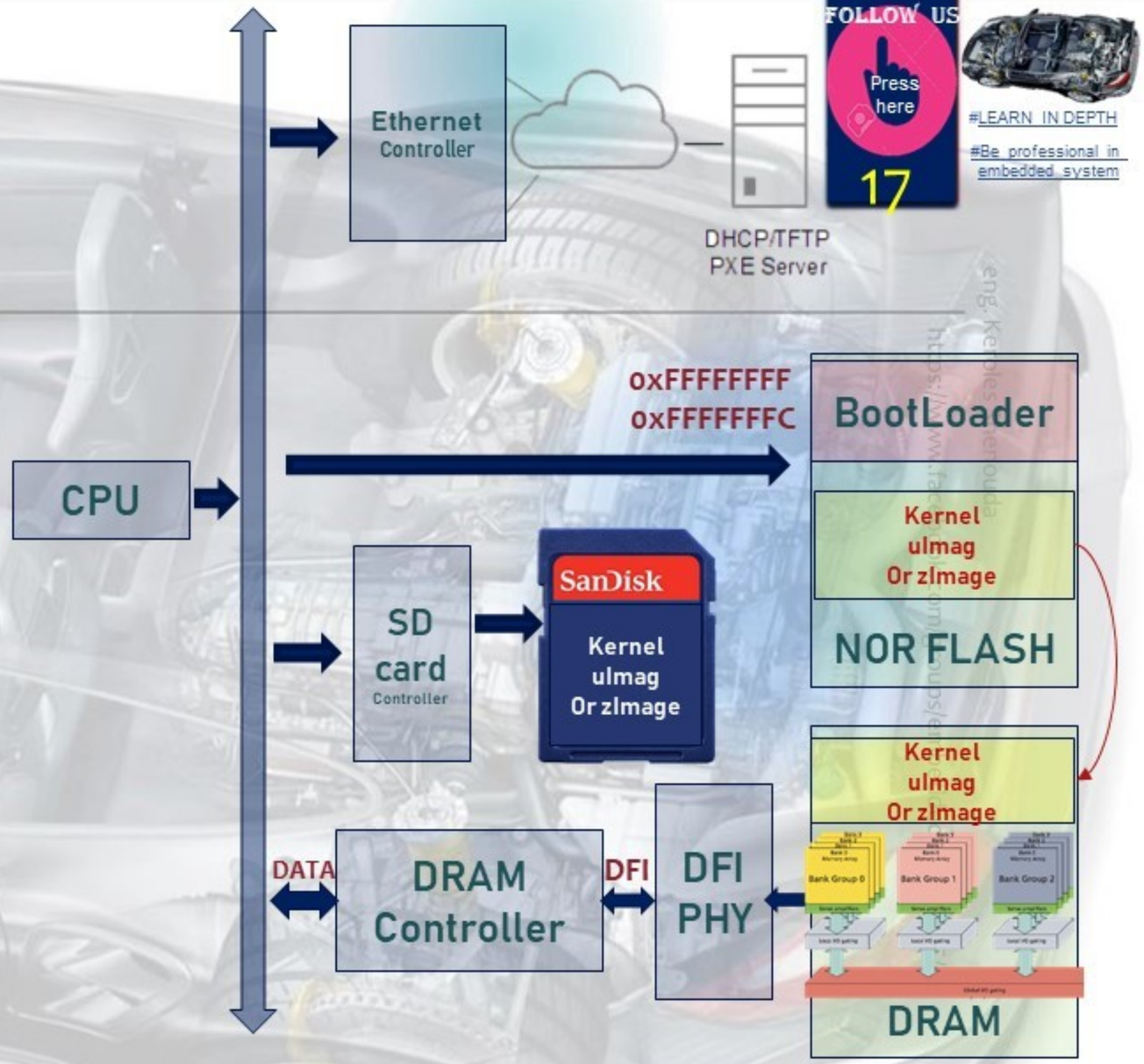
FROM THE PREVIOUSE SLIDES WHAT IS THE BOOTLOADERS ?

TRY TO THINK AND ANSWER ☺

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Bootloaders

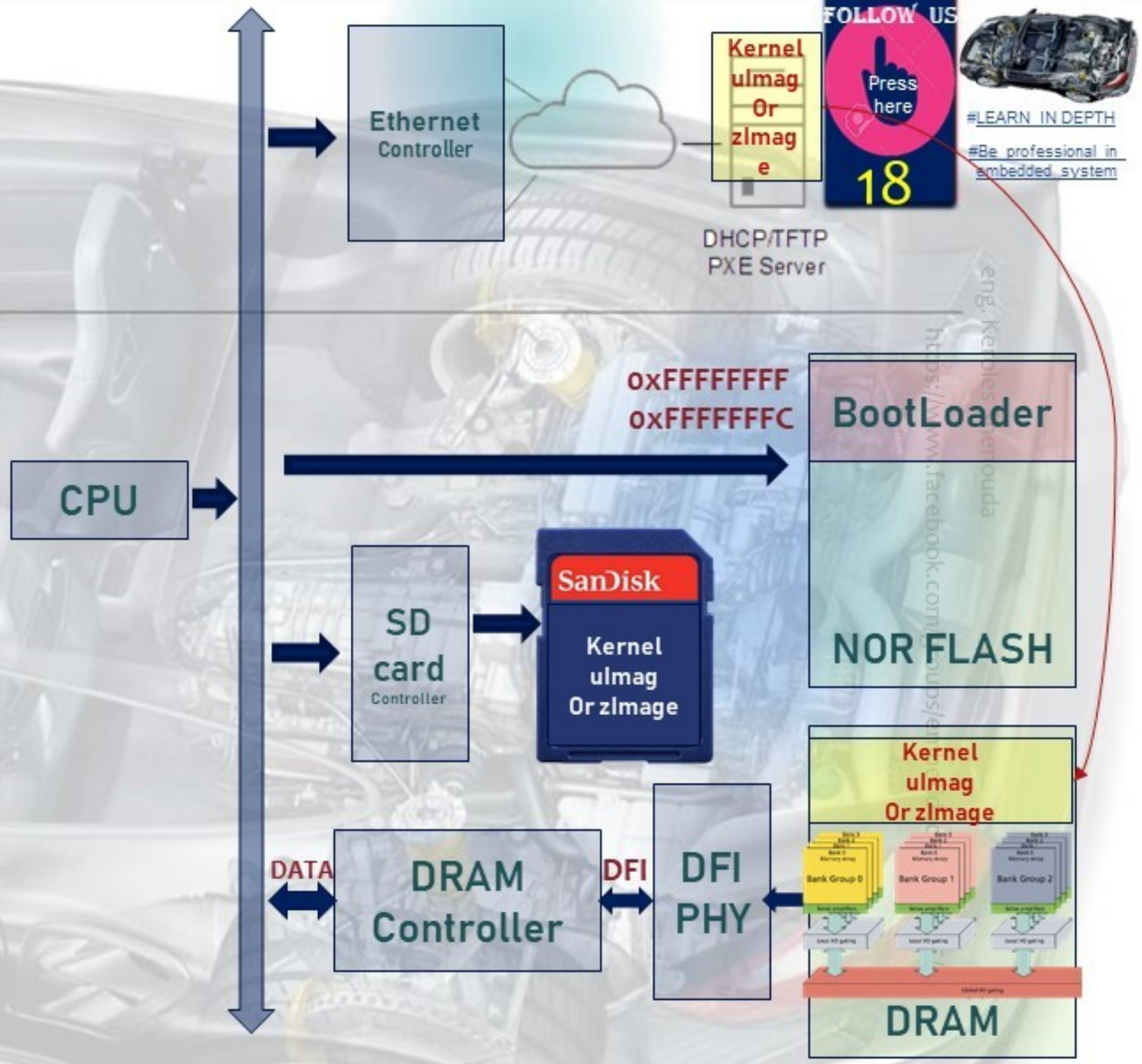
- The bootloader is a piece of code responsible for
  - Basic hardware initialization
  - Loading of an application binary, usually an operating system kernel,
    - from flash storage



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Bootloaders Cont.

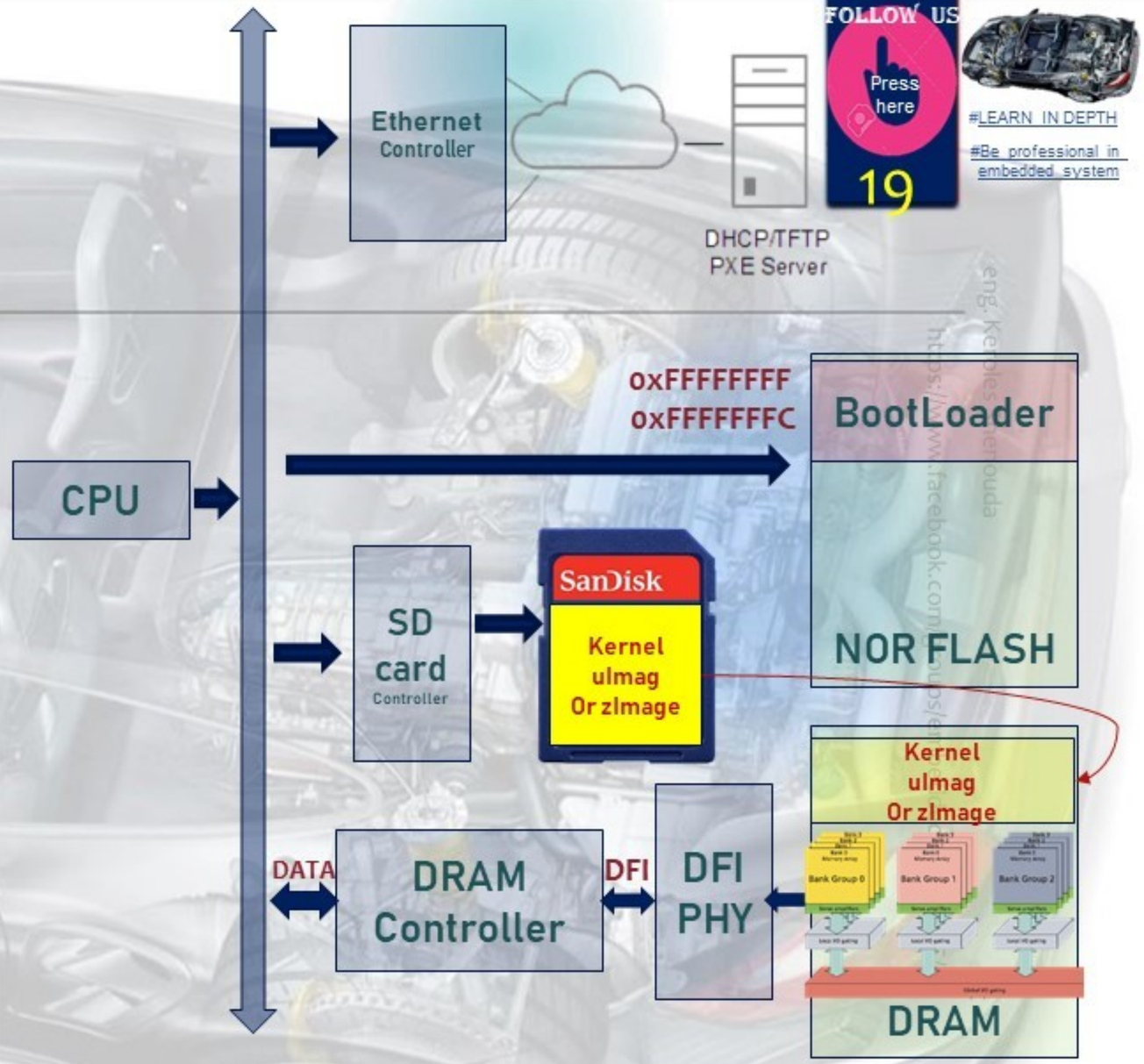
- The bootloader is a piece of code responsible for
  - Basic hardware initialization
  - Loading of an application binary, usually an operating system kernel,
    - from flash storage
    - from the network



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Bootloaders Cont.

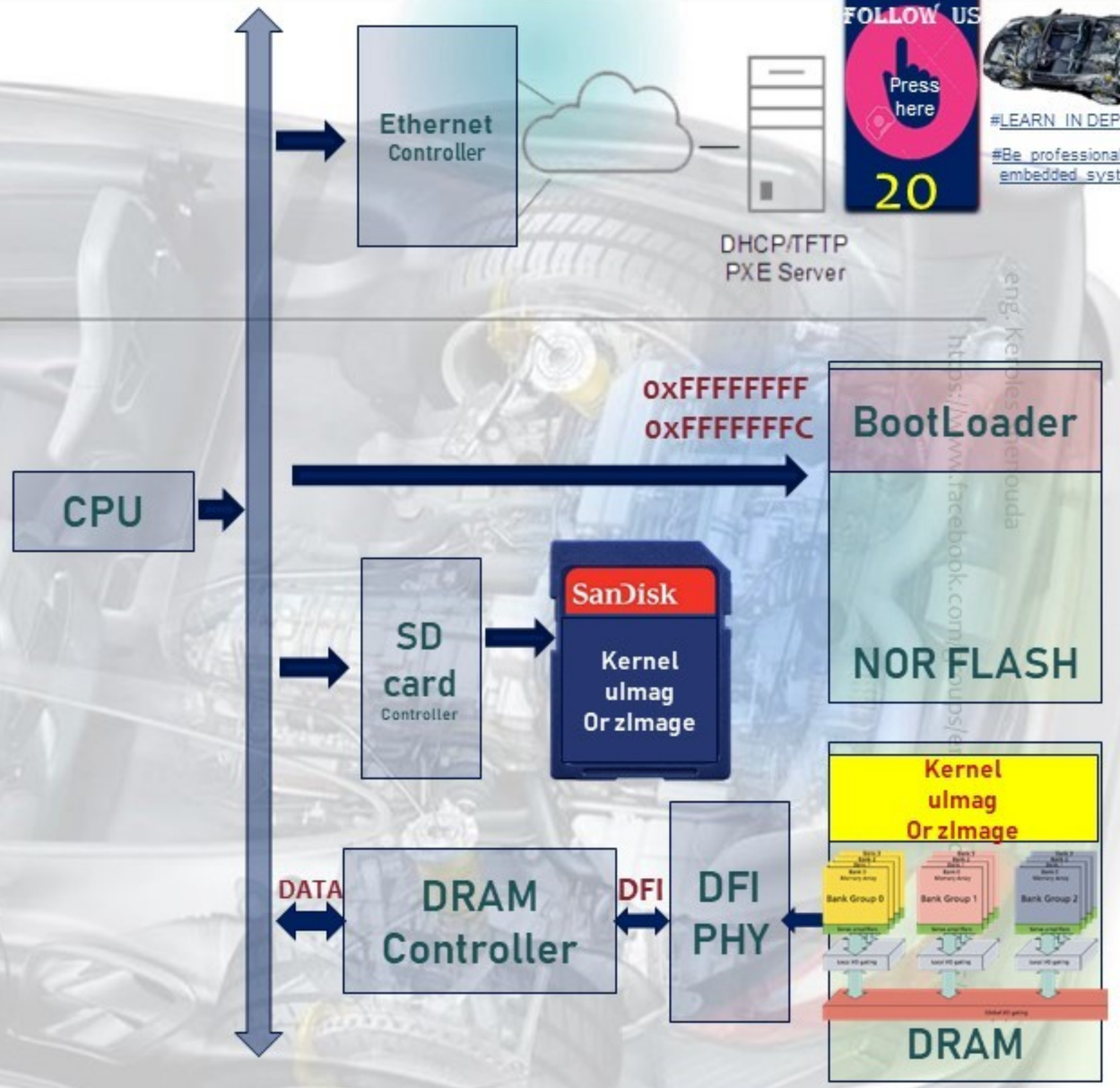
- The bootloader is a piece of code responsible for
  - Basic hardware initialization
  - Loading of an application binary, usually an operating system kernel,
    - from flash storage
    - from the network
    - From SDCard or from another type of non-volatile storage.



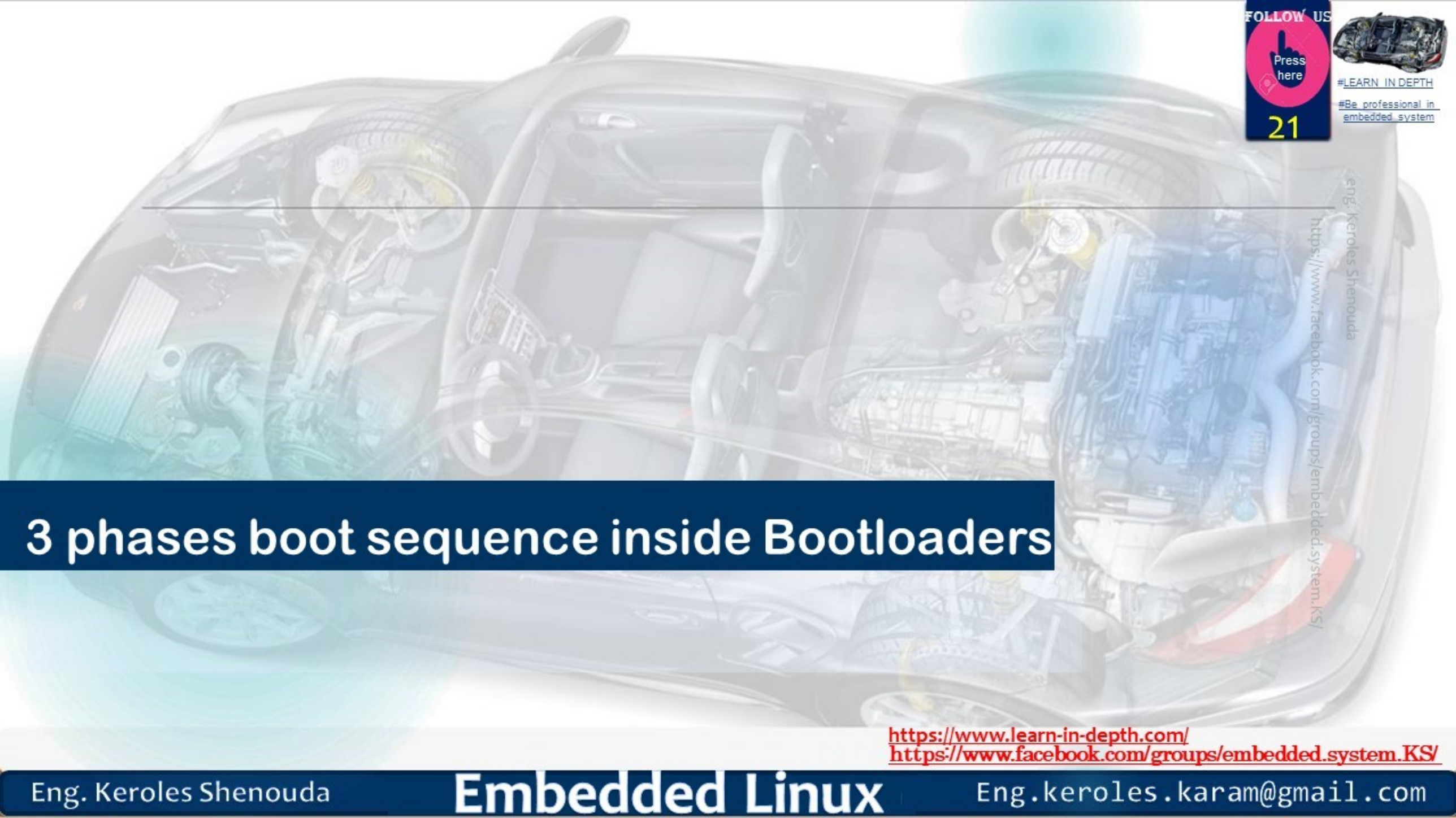
https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Bootloaders Cont.

- The bootloader is a piece of code responsible for
  - Basic hardware initialization
  - Loading of an application binary, usually an operating system kernel,
    - from flash storage
    - from the network
    - From SDCard or from another type of non-volatile storage.
  - Possibly decompression of the application binary
  - Execution of the application
- Besides these basic functions, most bootloaders provide a shell with various commands implementing different operations.
  - Loading of data from storage or network, memory inspection, hardware diagnostics and testing, etc



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

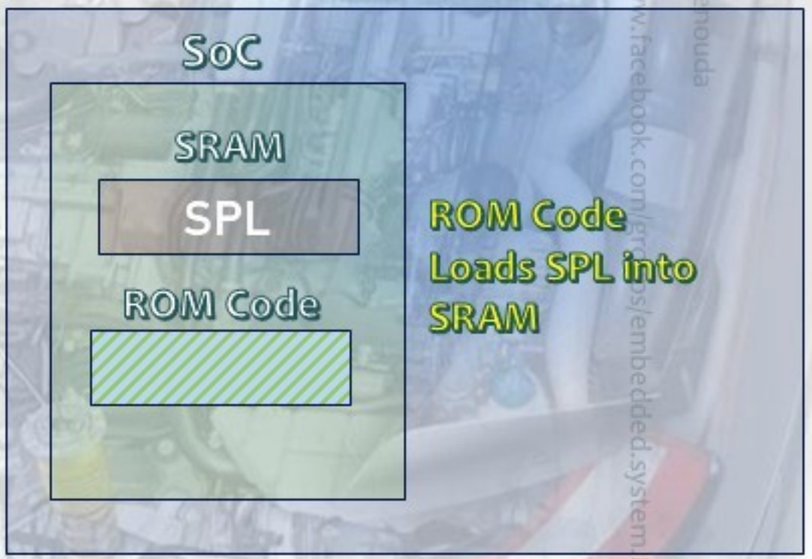


3 phases boot sequence inside Bootloaders

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

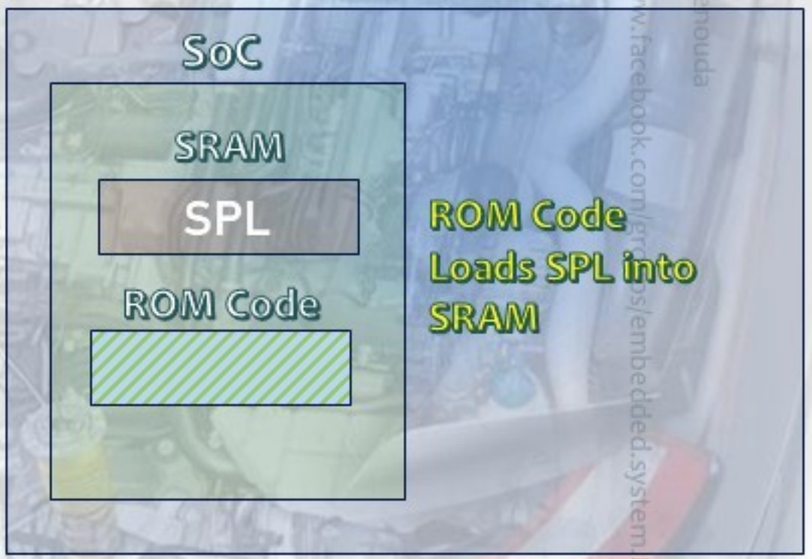
Phase 1 – ROM code

- It is the code that runs immediately after **a reset or power-on** has to be stored on-chip in the SoC;
  - this is known as **ROM code**.
- It is loaded into the chip **when it is manufactured**, and hence the **ROM code** cannot be replaced by an open source equivalent.
- Usually, it does not include code to initialize the memory controller, since DRAM configurations are highly device-specific, and so it can only use **Static Random Access Memory (SRAM)**, which does not require a memory controller.
- Most embedded SoC designs have a small amount of SRAM on-chip, varying in size from as little as 4 KB to several hundred KB:



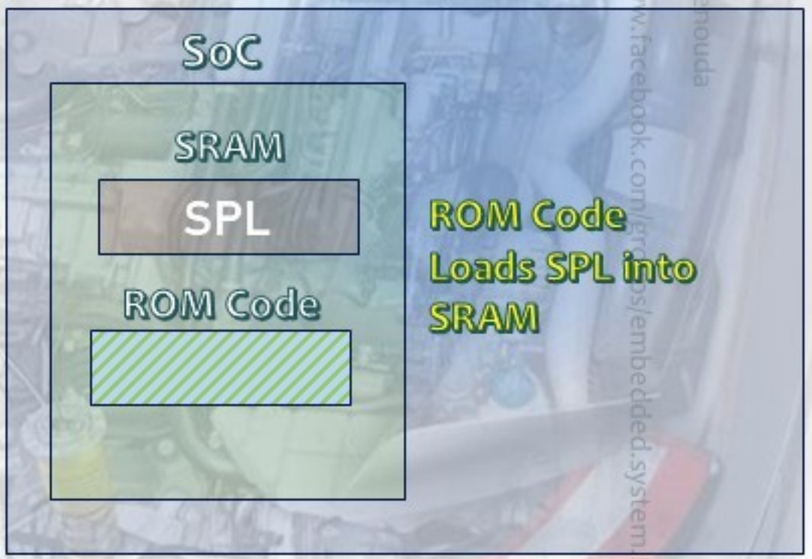
Phase 1 – ROM code Cont.

- The **ROM code** is capable of loading a small chunk of code from one of several pre-programmed locations into the SRAM.
  - As an example, TI OMAP and Sitara chips try to load code from the first few pages of NAND flash memory, or from flash memory connected through a **Serial Peripheral Interface** (**SPI**), or from the first sectors of an MMC device (which could be an eMMC chip or an SD card), or from a file named MLO on the first partition of an MMC device. If
- Most embedded SoCs have **a ROM code** that works in a similar way. In SoCs where the SRAM is not large enough to load a full bootloader like UBoot, there has to be an **intermediate loader** called the secondary program loader, or SPL.



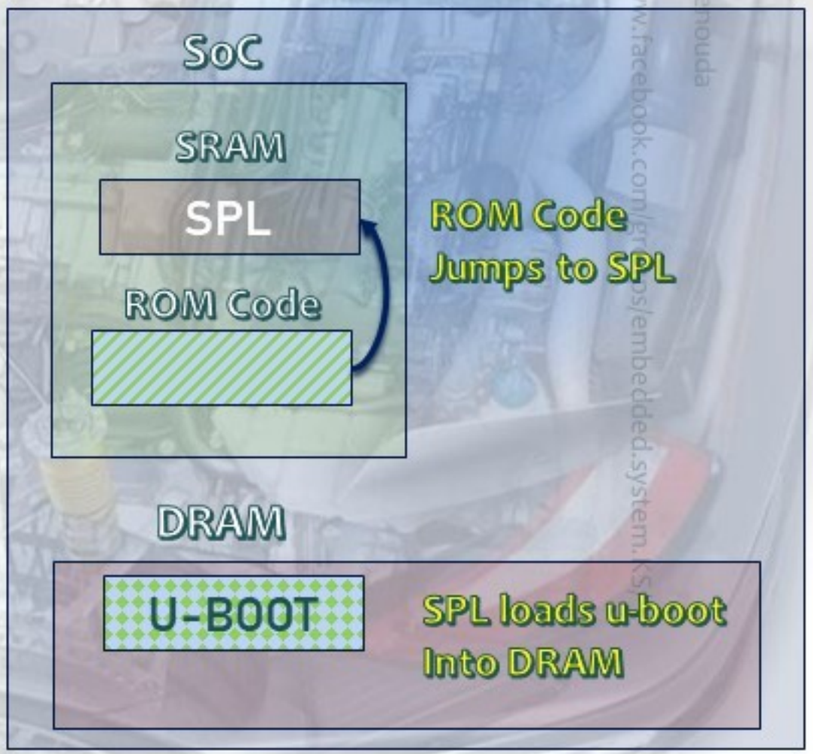
Phase 1 – ROM code Cont.

At the end of the ROM code phase, the SPL is present in the SRAM and the ROM code jumps to the beginning of that code.



Phase 2 – secondary program loader

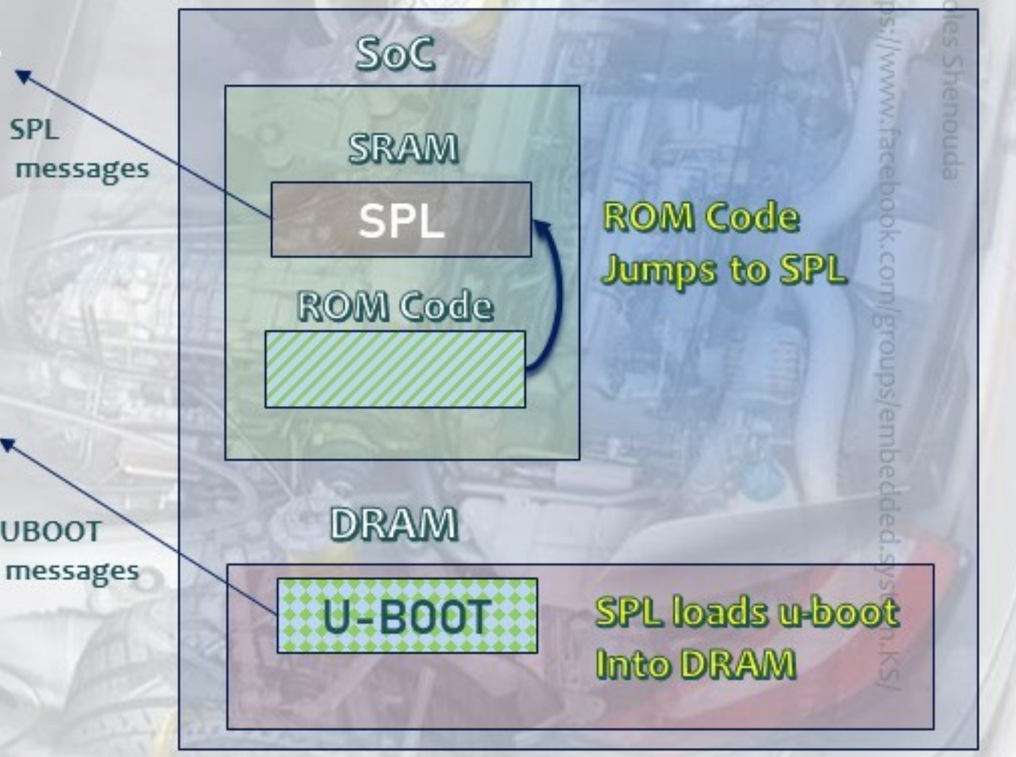
- The SPL must set up the memory controller and other essential parts of the system preparatory to loading the **uboot** into DRAM.
- The functionality of the SPL is limited by the size of the **SRAM**. It can read a program from a list of storage devices.
- system drivers built inside spl, it can read well known file names, such as **u-boot.img** from **a disk partition.**
- The SPL usually doesn't allow for any user interaction, but it may print version information and progress messages, which you can see on the console



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

SPL / UBOOT MESSAGES ON UART

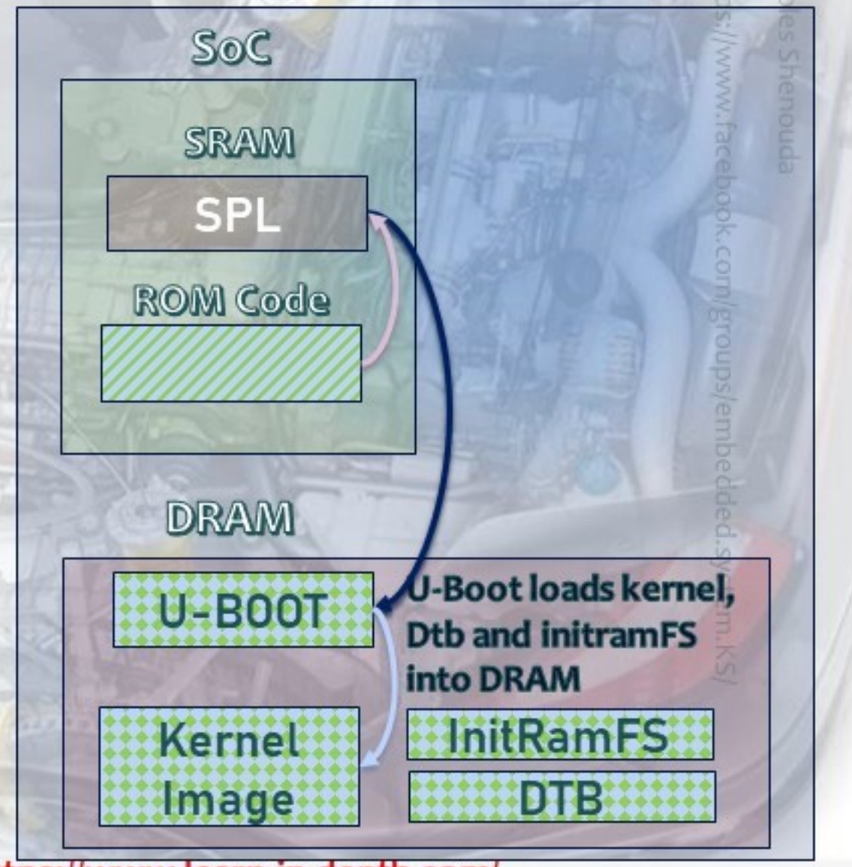
```
=>
U-Boot SPL 2017.09+fslc+ga6a15fedd1a5 (Oct 28 2017 - 13:55:35)
Trying to boot from MMC1


U-Boot 2017.09+fslc+ga6a15fedd1a5 (Oct 28 2017 - 13:55:35 +0200)

CPU:   Freescale i.MX6Q rev1.2 at 792 MHz
Reset cause: POR
I2C:   ready
DRAM:  2 GiB
Can't find PMIC:PFUZE100
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
*** Warning - bad CRC, using default environment

auto-detected panel HDMI
Display: HDMI (1024x768)
In:    serial
Out:   serial
Err:   serial
Board: Wandboard rev C1
Net:   FEC [PRIME]
Hit any key to stop autoboot:  0
=>
```



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Phase 3 – U-BOOT

- Now, at last, we are running a full bootloader, such as U-Boot
- Usually, there is a simple command-line user interface that lets you perform maintenance tasks, such as loading new boot and kernel images into RAM



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

So the Booting Sequence

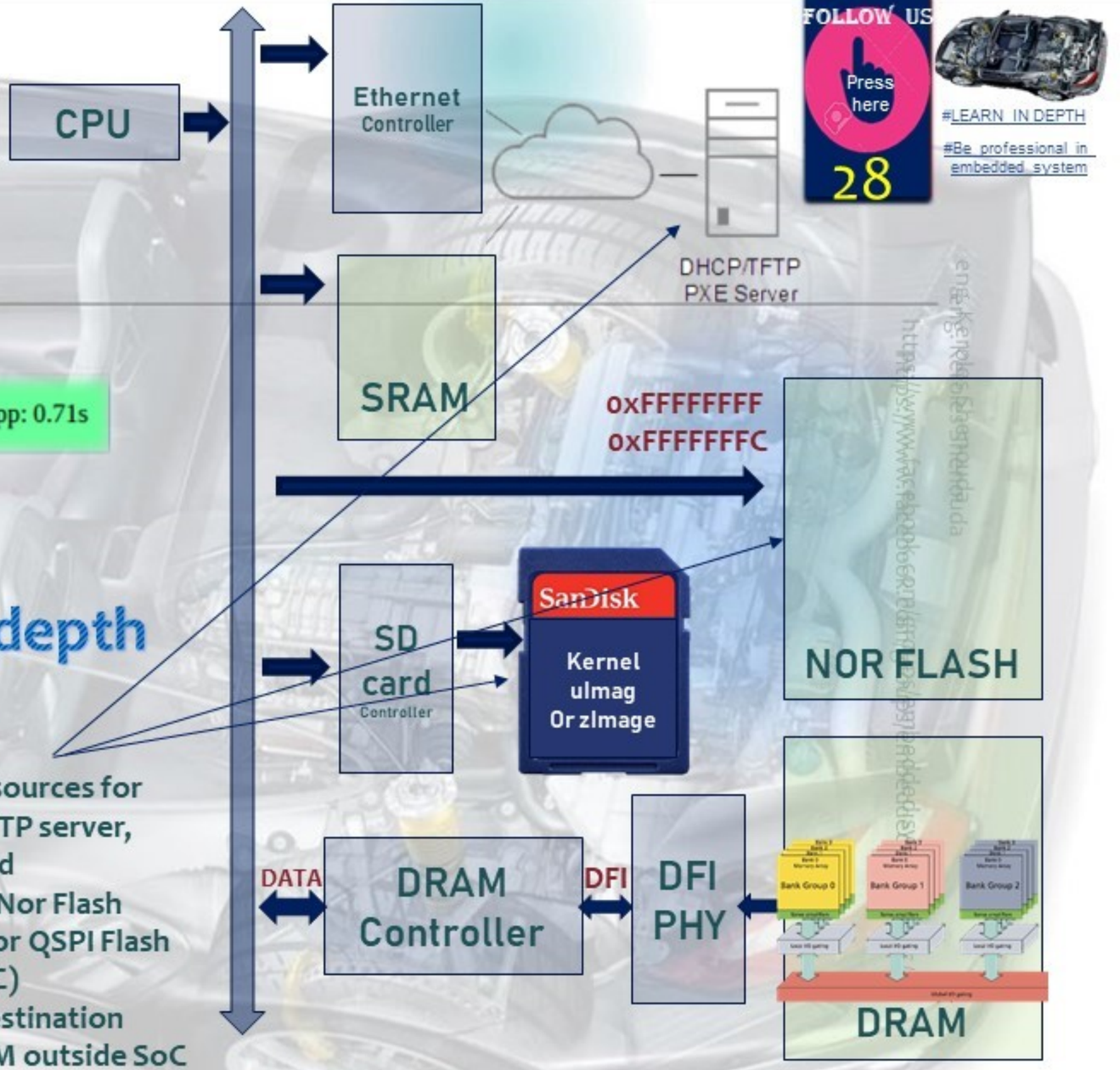
Average time spent on each stage:

BootRom	SPL: 0.264s	U-Boot: 3,377s	Kernel : 4.090s	Init: 3.931s	App: 0.71s

- Power ON
- Boot ROM
- SPL
- UBOOT
- Kernel
- INIT process
- Your APP

Learn-in-depth

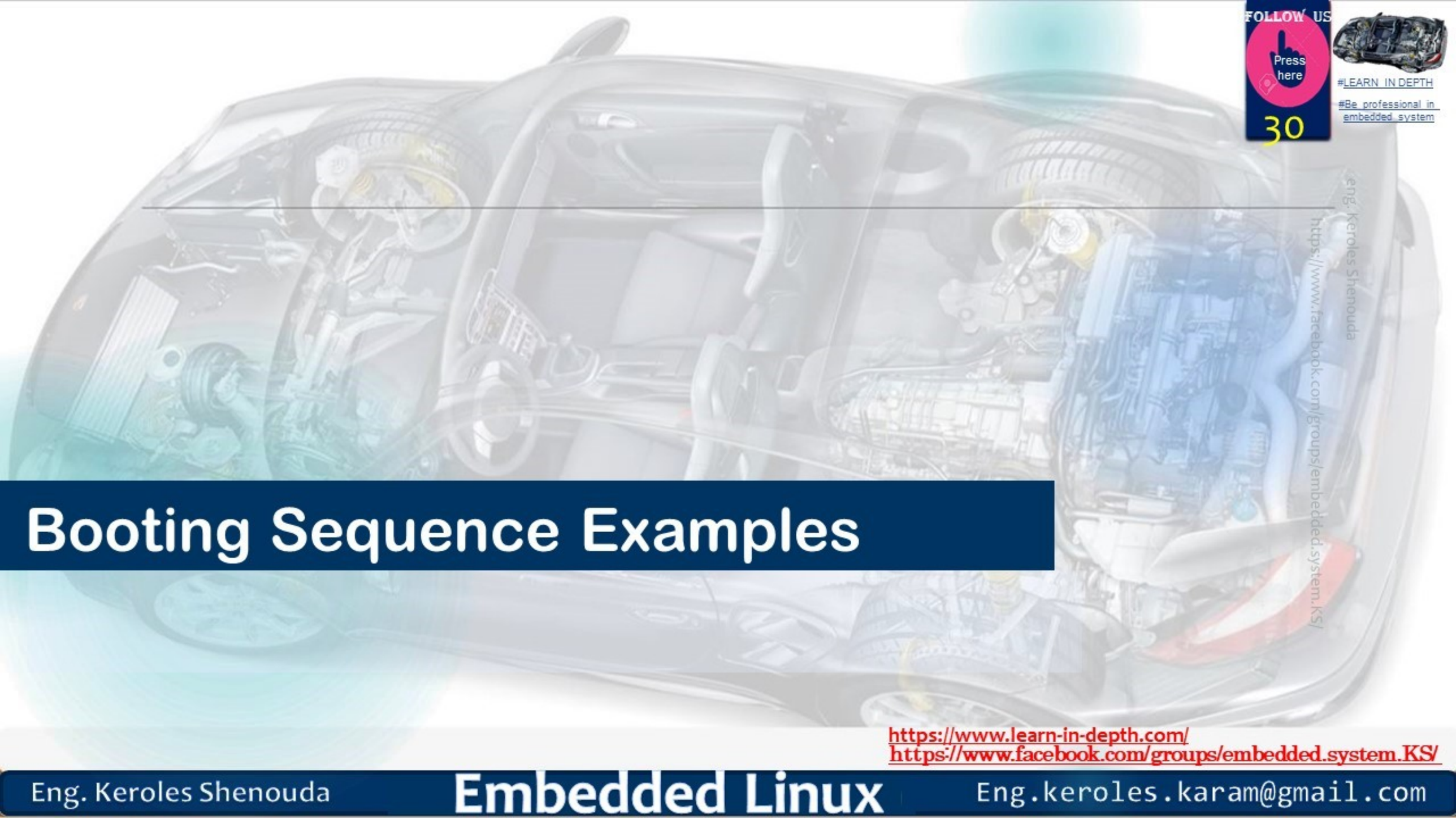
We have 3 sources for loading (TFTP server, SDCARD and Flash even Nor Flash inside SoC or QSPI Flash outside SoC)
And two destination RAM (DRAM outside SoC and SRAM inside SoC)



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Different terminologies

	Terminology 1	Terminology 2
Boot ROM	Primary program loader	N/A
SPL	Secondary program loader	1st stage bootloader
UBOOT	Secondary program loader	2st stage bootloader

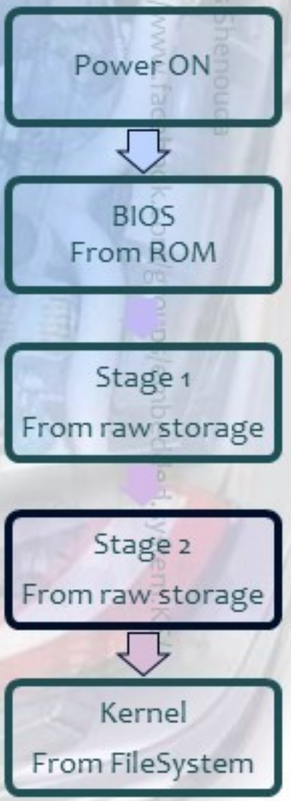


Booting Sequence Examples

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

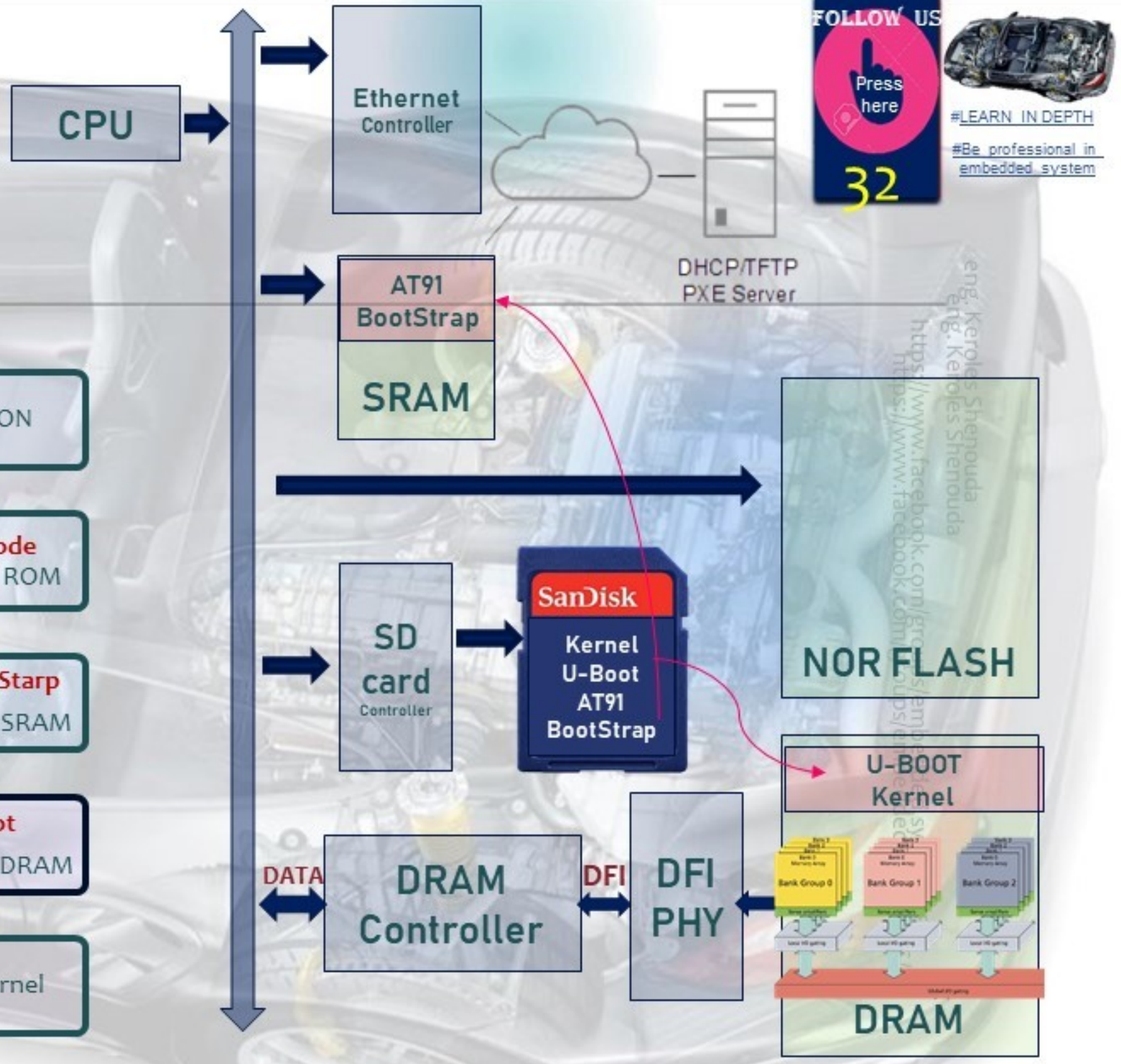
Bootloaders on BIOS-based x86 (1)

- The x86 processors are typically bundled on a board with a non-volatile memory containing a program, **the BIOS**.
- On old BIOS-based x86 platforms: **the BIOS** is responsible for basic hardware initialization and loading of a very small piece of code from non-volatile storage.
- This piece of code is typically a **1st stage bootloader**, which will load the full bootloader itself.
- It typically understands filesystem formats so that the kernel file can be loaded directly from a normal filesystem.
- This Sequence on old BIOS-based x86 platforms
- The most famous/powerful Bootloader in x86 is GRUB, Grand Unified Bootloader. http://www.gnu.org/software/grub/



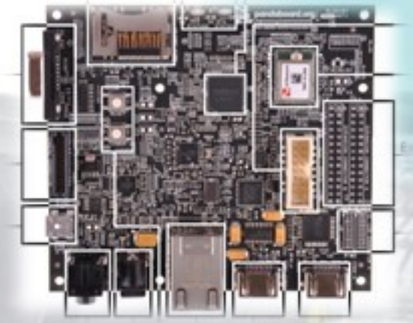
Booting on ARM Microchip AT91

- Rom Code:
  - tries to find a valid **bootstrap image** from various storage sources, and load it into SRAM (DRAM not initialized yet). Size limited to 4 KB. No user interaction possible in standard boot mode.
- AT91Bootstrap:
  - runs from SRAM. **Initializes the DRAM**, the NAND or SPI controller, and loads the **secondary bootloader** into RAM and starts it. No user interaction possible.
- U-Boot:
  - runs from RAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to RAM and starts it. Shell with commands provided.
- Linux Kernel:
  - runs from RAM. Takes over the system completely (the bootloader no longer exists).

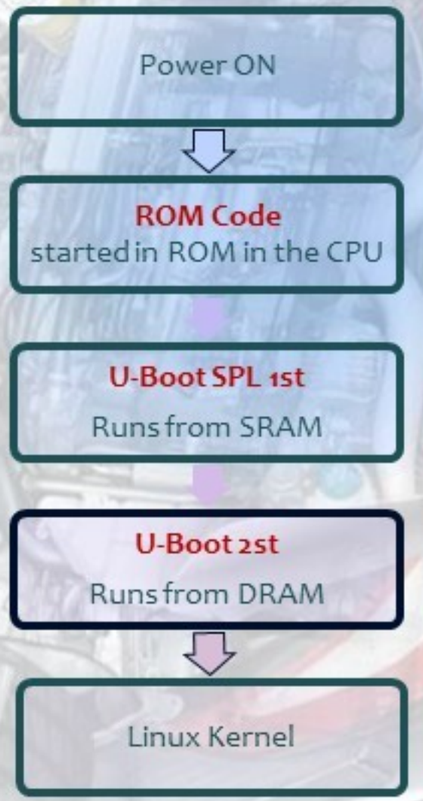


https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Booting on ARM TI OMAP2+ / AM33xx



- Rom Code:
  - tries to find a valid bootstrap image from various storage sources, and load it into SRAM or RAM (RAM can be initialized by ROM code through a configuration header). Size limited to < 64 KB. No user interaction possible.
- U-Boot SPL:
  - runs from SRAM. Initializes the DRAM, the NAND or MMC controller, and loads the secondary bootloader into RAM and starts it. No user interaction possible
- U-Boot:
  - runs from RAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to RAM and starts it. Shell with commands provided. File called **u-boot.bin or u-boot.img**.
- Linux Kernel:
  - runs from RAM. Takes over the system completely (the bootloader no longer exists).



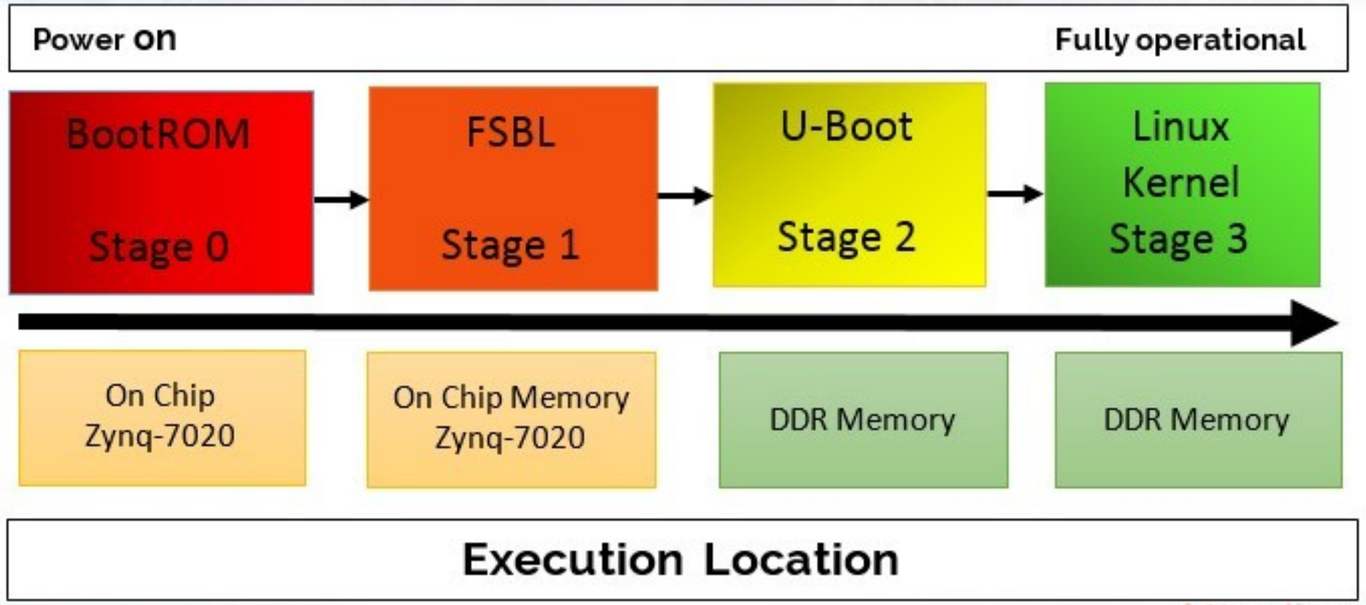
https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Booting on Marvell SoCs

- **ROM Code:** tries to find a valid bootstrap image from various storage sources, and load it into RAM. The RAM configuration is described in a CPU-specific header, prepended to the bootloader image.
- **U-Boot:** runs from RAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to RAM and starts it. Shell with commands provided. File called u-boot.kwb.
- **Linux Kernel:** runs from RAM. Takes over the system completely (bootloaders no longer exists).

Zynq-7020 SoC from Xilinx

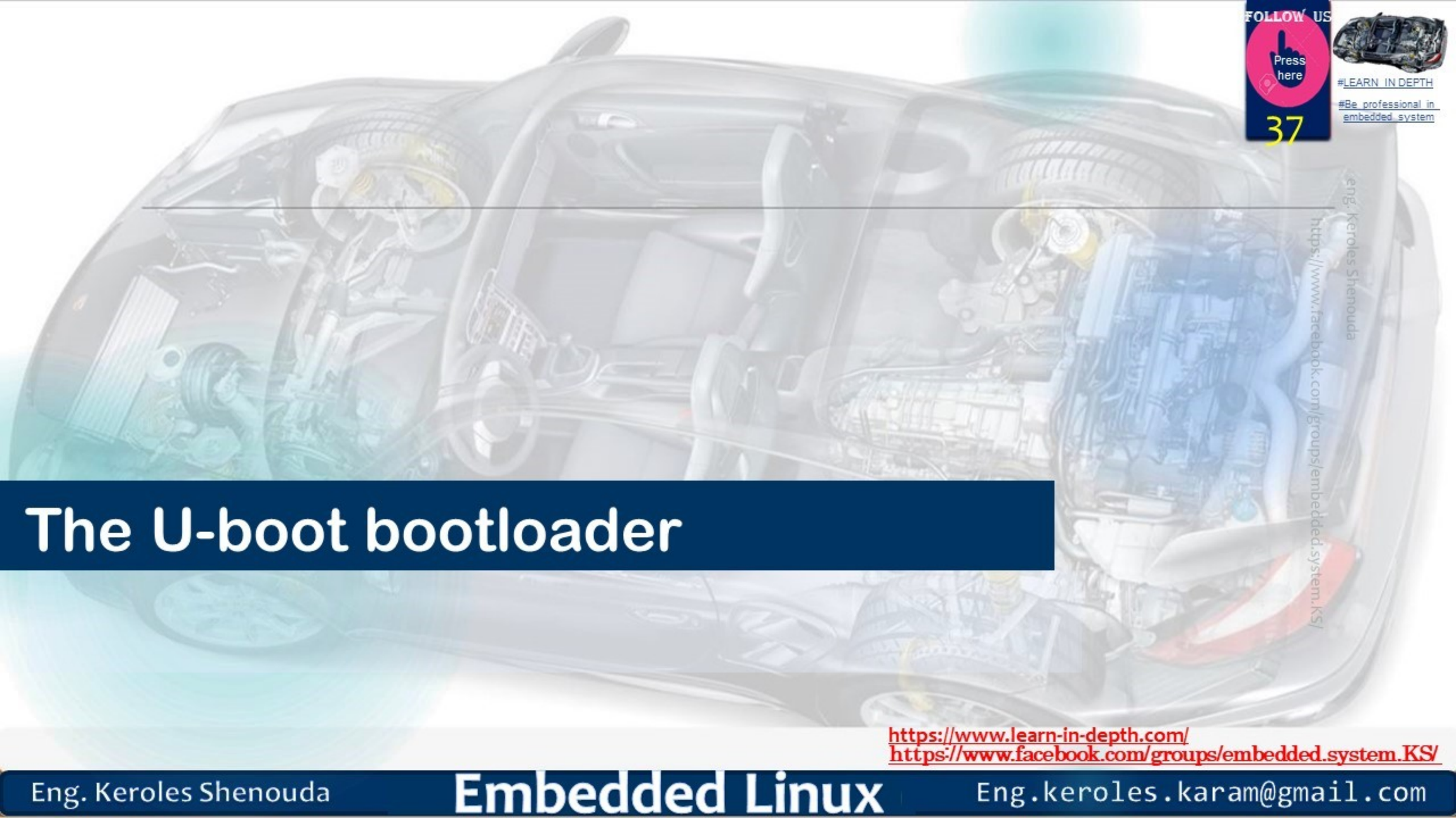
- The BootROM is located within the SoC and not configurable by the user. The BootROM determines whether the boot is secure or non-secure, performs some initialization of the system and clean-up. Next the ROM code determines (usually via switch settings) the desired boot mode (NAND, QSPI, JTAG) and loads the First Stage Boot Loader (FSBL) from the selected peripheral/interface into the On-Chip Memory (OCM) and jumps to the starting location of the FSBL.
- The FSBL code does the hardware initialization, including the external memory, multiplex IO (MIO) and system clocks. This initialization is necessary to move to stage 2 where U-Boot would be loaded. So, it would appear the first point where the hardware is running from a functional aspect is after the FSBL at the end of Stage 1



Generic bootloaders for embedded CPUs

- There are several open-source generic bootloaders.
- Here are the most popular ones:
  - **U-Boot**, the universal bootloader by Denx The most used on ARM, also used on PPC, MIPS, x86, m68k, NIOS, etc. The de-facto standard nowadays. We will study it in detail. http://www.denx.de/wiki/U-Boot
  - **Barebox**, an architecture-neutral bootloader, It doesn't have as much hardware support as U-Boot yet http://www.barebox.org
- There are also a lot of other open-source or proprietary bootloaders, often architecture-specific
  - RedBoot, Yaboot, etc

Name	Main architectures supported
Das U-Boot	ARC, ARM, Blackfin, Microblaze, MIPS, Nios2, OpenRiec, PowerPC, SH
Barebox	ARM, Blackfin, MIPS, Nios2, PowerPC
GRUB 2	X86, X86_64
Little Kernel	ARM
RedBoot	ARM, MIPS, PowerPC, SH
CFE	Broadcom MIPS
YAMON	MIPS



The U-boot bootloader

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

U-Boot

- **U-Boot** is a typical free software project
  - License: GPLv2 (same as Linux)
  - Freely available at http://www.denx.de/wiki/U-Boot
  - Documentation available at http://www.denx.de/wiki/U-Boot/Documentation
  - Development and discussions happen around an open mailing-list http://lists.denx.de/pipermail/u-boot/

Lab1: build/RUN u-boot for vexpress_ca9 SoC

THE EXPLANATION WILL BE **INSIDE** THE LAB TO BE MORE EFFICIENT

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

U-Boot configuration

```
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/u-boot$ tree -L 1
.
├── api
├── arch
├── board
├── cmd
├── common
├── config.mk
├── configs
├── disk
├── doc
├── Documentation
├── drivers
├── dts
├── env
├── examples
├── fs
├── include
├── Kbuild
├── Kconfig
├── lib
├── Licenses
├── MAINTAINERS
├── Makefile
├── net
├── post
├── README
├── scripts
├── spl
├── test
└── tools

23 directories, 6 files
```

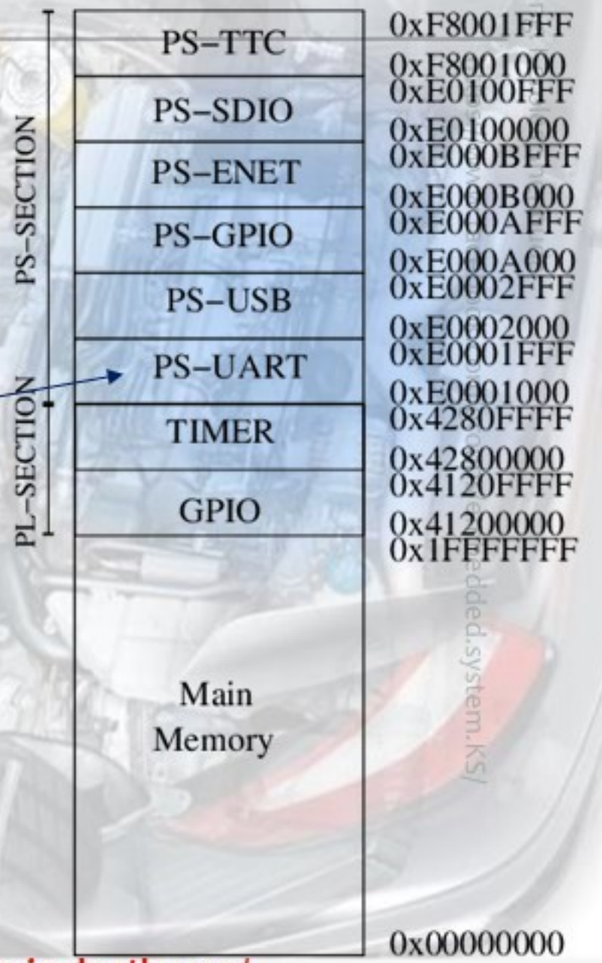
- Get the source code from the website, and uncompressed it
- The configs/ directory contains one configuration file for each supported board
  - It defines the CPU type, the peripherals and their configuration, the memory mapping, the U-Boot features that should be compiled in, etc

For example ZYNQ 7000 SoC in Zedboard


```
embedded_system_ks@embedded-KS: /media/embedded_system_ks/Embedded_Linux/u-boot
[2]+  Done                    gedit configs/zynq_zed_defconfig
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/u-boot$
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/u-boot$ ls  configs/zynq_zed_defconfig
configs/zynq_zed_defconfig
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/u-boot$ gedit configs/zynq_zed_defconfig &
[2] 7664
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/u-boot$
```

zynq_zed_defconfig

```
CONFIG_ARM=y
CONFIG_SPL_SYS_DCACHE_OFF=y
CONFIG_ARCH_ZYNQ=y
CONFIG_SYS_TEXT_BASE=0x4000000
CONFIG_SPL_STACK_R_ADDR=0x200000
CONFIG_SPL=y
CONFIG_DEBUG_UART_BASE=0xe0001000
CONFIG_DEBUG_UART_CLOCK=50000000
CONFIG_DEBUG_UART=y
CONFIG_DISTRO_DEFAULTS=y
CONFIG_SYS_CUSTOM_LDSCRIPT=y
CONFIG_SYS_LDSCRIPT="arch/arm/mach-zynq/u-boot.lds"
CONFIG_FIT=y
CONFIG_FIT_SIGNATURE=y
CONFIG_FIT_VERBOSE=y
CONFIG_LEGACY_IMAGE_FORMAT=y
CONFIG_USE_PREBOOT=y
CONFIG_SPL_STACK_R=y
CONFIG_SPL_OS_BOOT=y
CONFIG_SPL_SPI_LOAD=y
CONFIG_SYS_SPI_U_BOOT_OFFS=0x100000
# CONFIG_BOOTM_NETBSD is not set
CONFIG_CMD_THOR_DOWNLOAD=y
CONFIG_CMD_DFU=y
# CONFIG_CMD_FLASH is not set
```



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Configuring and compiling U-Boot

- U-Boot must be configured before being compiled
  - $ make BOARDNAME_defconfig
  - Where BOARDNAME is the name of a configuration, as visible in the **configs/** directory
  - You can then run "$ make menuconfig" to further customize U-Boot's configuration!
- Make sure that the cross-compiler is available in PATH
- Compile **U-Boot**, by specifying the **cross-compiler** prefix. Example, if your cross-compiler executable is arm-linux-gcc: **$ make CROSS_COMPILE=arm-linux-**
- The main result is a **u-boot.bin** file, which is the U-Boot image.

Configuring and compiling U-Boot vexpress_ca9

- ARM Development boards are the ideal platform for accelerating the development and reducing the risk of new SOC designs. The combination of ASIC and FPGA technology in ARM boards delivers an optimal solution in terms of speed, accuracy, flexibility and cost.
- **ARM development boards are often used to:**
  - Evaluate, benchmark and start software development on the latest ARM processors Prototype, validate and develop software drivers for new SoC IP blocks - for example, a modem or video engine Test custom logic blocks or system IP in an FPGA, connected to an ARM core running at ASIC speed
- **Key Features:**
  - Support for a wide range of ARM processors from deeply embedded to multimedia applications Large system memory and rich set of peripheral interfaces: Ethernet, USB, LCD, UART and more

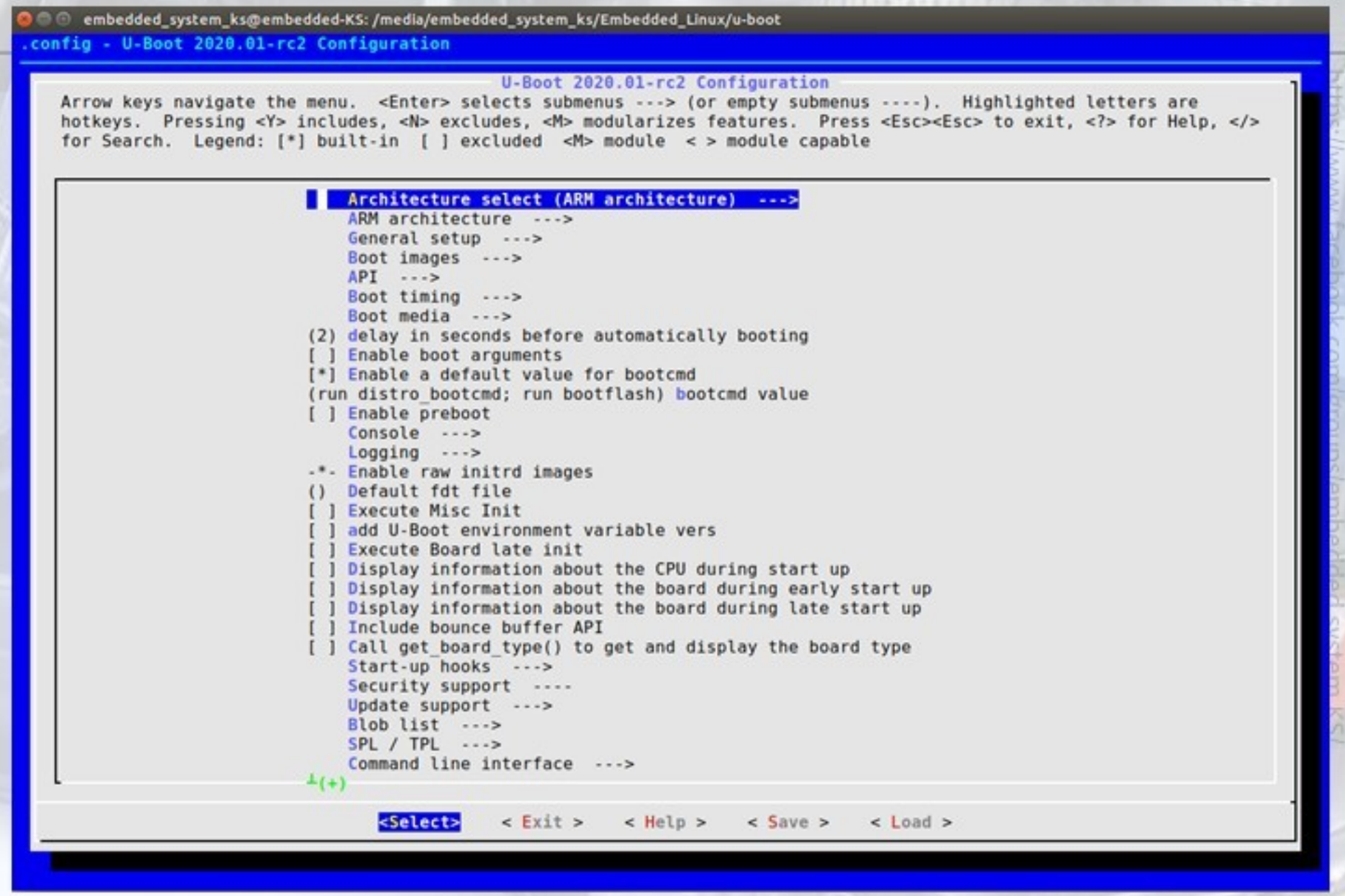
Configuring and compiling U-Boot vexpress_ca9 Cont.

```
embedded_system_ks@embedded-KS: /media/embedded_system_ks/Embedded_Linux/u-boot
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/u-boot$ export CROSS_COMPILE=arm-linux-gnueabi-
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/u-boot$ export ARCH=arm
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/u-boot$ make vexpress_ca9x4_config
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/kconfig/conf.o
  YACC    scripts/kconfig/zconf.tab.c
  LEX     scripts/kconfig/zconf.lex.c
  HOSTCC  scripts/kconfig/zconf.tab.o
  HOSTLD  scripts/kconfig/conf
#
# configuration written to .config
#
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/u-boot$
```

Configuring and compiling U-Boot vexpress_ca9 Cont.

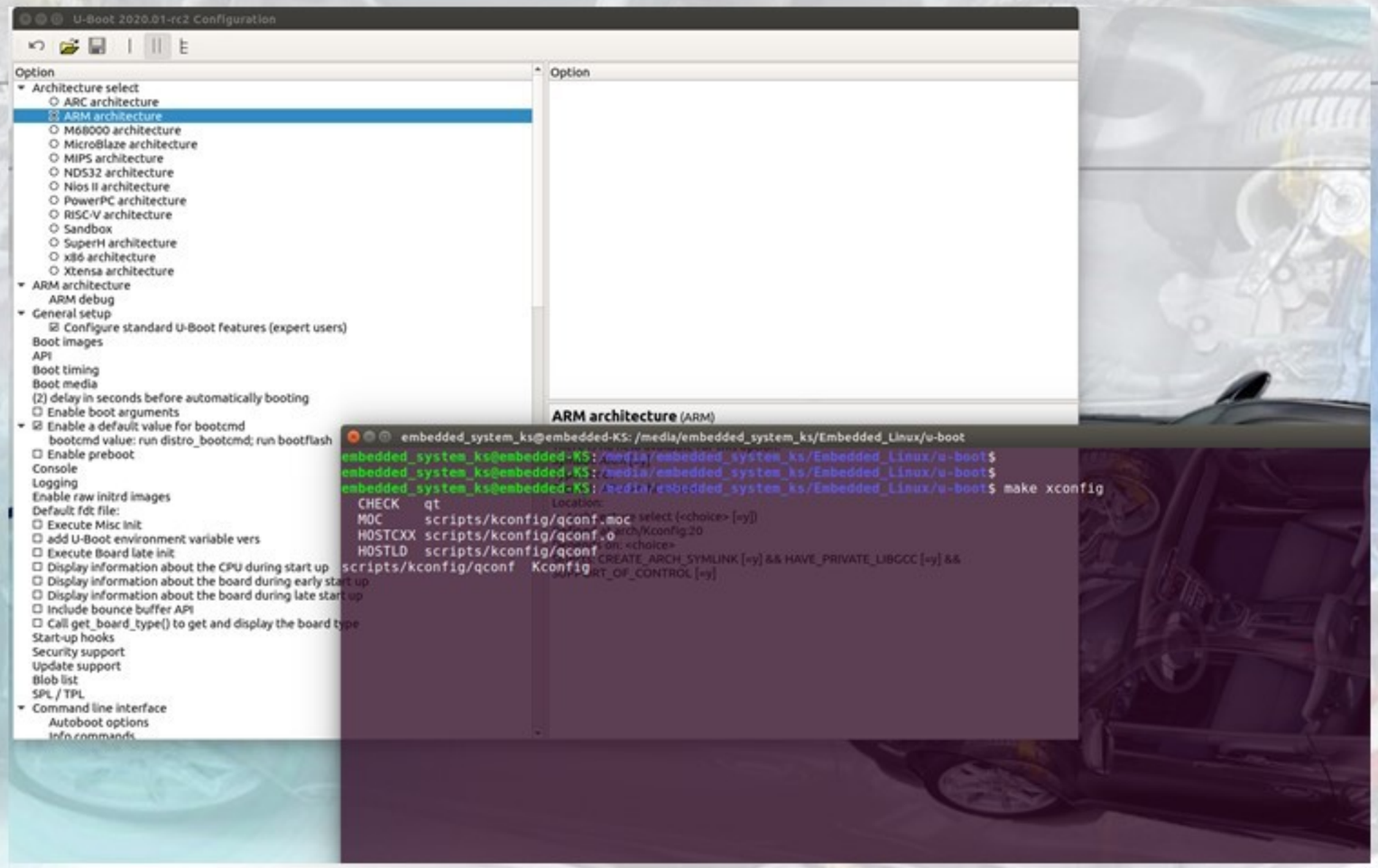
- $ make mnuconfig

We will Learn the configuration options in the next part



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

$ make xconfig



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

$ make xconfig

vexpress_ca9x4_defconfig (Embedded_Linux /media/embedded_system_ks/Embedded_Linux/u-boot/configs) - gedit

```
14 # CONFIG_CMD_LOADS is not set
15 CONFIG_CMD_MMC=y
16 # CONFIG_CMD_ITEST is not set
17 # CONFIG_CMD_SETEXPR is not set
18 # CONFIG_CMD_NFS is not set
19 # CONFIG_CMD_MISC is not set
20 CONFIG_CMD_UBI=y
21 CONFIG_ENV_IS_IN_FLASH=y
22 CONFIG_MTD_NOR_FLASH=y
23 CONFIG_MTD_DEVICE=y
24 CONFIG_FLASH_CFI_DRIVER=y
25 CONFIG_SYS_FLASH_USE_BUFFER_WRITE=y
26 CONFIG_SYS_FLASH_PROTECTION=y
27 CONFIG_SYS_FLASH_CFI=y
28 CONFIG_SMC911X=y
29 CONFIG_SMC911X_BASE=0x4e000000
30 CONFIG_SMC911X_32_BIT=y
31 CONFIG_BAUDRATE=38400
```

U-Boot 2020.01-rc2 Configuration

Option
- Memory commands
- Compression commands
- Device access commands
- Shell scripting commands
- Android support commands
- [x] Network commands
- Misc commands
- TI specific command line interface
- Power commands
- Security commands
- Firmware commands
- Filesystem commands
- Debug commands

Partition Types
Device Tree Control
Path to dtc binary for use within mkimage: dtc
Environment
[x] Networking support
Device Drivers
- Generic Driver Options
- SATA/SCSI device support
- [ ] AXI bus drivers
- [ ] Enable support for checking boot count
- Cache Controller drivers
- Clock
- Hardware crypto devices
- Demo for driver model
- [ ] Device Information
- DFU support
- DMA Support
- Fastboot support
- FPGA support
- GPIO Support
- Hardware Spinlock Support
- I2C support
- LED Support
- Mailbox Controller Support
- Memory Controller drivers
- Multifunction device drivers
- MMC Host controller Support
- MTD Support
  - [ ] Raw NAND Device Support
  - SPI Flash Support
  - UBI support
- [ ] Ethernet PHY (physical media interface) support
- [ ] NXP PFE Ethernet driver
- [x] Network device support

Option
- Enable G
- Broadcor
- Intel PRO
- Synopsys Designware Ethernet MAC
- OpenCores 10/100 Mbps Ethernet MAC
- Freescale FMan ethernet support
- Ftmac100 Ethernet Support
- Enable RGMII
- Enable MII

Search Config

Find: SMC Search

Option
- SMC1
- SMC2
- [ ] Support for ARM SMC Calling Convention (SMCCC)

Support the 'smc' command
- [x] SMSC LAN911x and LAN921x controller driver
- ( ) Enable 16-bit interface
- (•) Enable 32-bit interface
- (0x4e000000) SMC911X Base Address
- () Console Rx buffer length

SMC911X Base Address (SMC911X_BASE)

CONFIG_SMC911X_BASE:

Define this to hold the physical address
of the device (I/O space)

Symbol: SMC911X_BASE [=0x4e000000]
Type : hex
Prompt: SMC911X Base Address
Location:

Vexpress A9 ARM SPECS

```
101 [VE_SERIALDVI] = 0x10016000,
102 [VE_RTC] = 0x10017000,
103 [VE_COMPACTFLASH] = 0x1001a000,
104 [VE_CLCD] = 0x1001f000,
105 /* CS0: 0x40000000 .. 0x44000000 */
106 [VE_NORFLASH0] = 0x40000000,
107 /* CS1: 0x44000000 .. 0x48000000 */
108 [VE_NORFLASH1] = 0x44000000,
109 /* CS2: 0x48000000 .. 0x4a000000 */
110 [VE_SRAM] = 0x48000000,
111 /* CS3: 0x4c000000 .. 0x50000000 */
112 [VE_VIDEORAM] = 0x4c000000,
113 [VE_ETHERNET] = 0x4e000000,
114 [VE_USB] = 0x4f000000,
115 [VE_NORFLASHALIAS] = -1, /* not present */
116 };
117
```

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

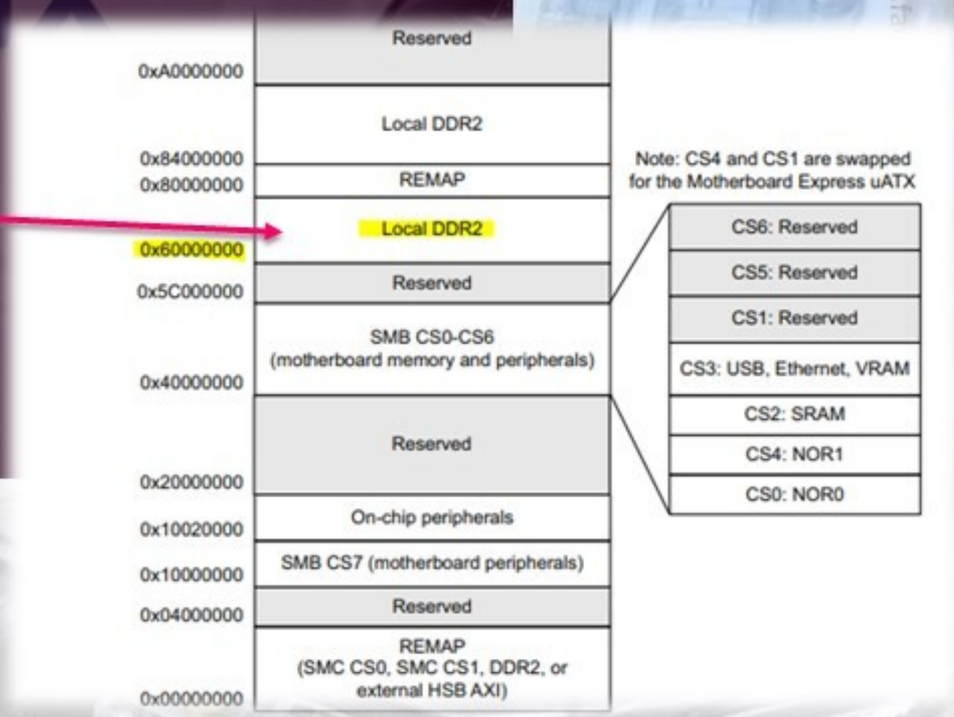
Configuring and compiling U-Boot vexpress_ca9 Cont.

- Build the U-Boot: $ make -j 8

```
embedded_system_ks@embedded-KS: /media/embedded_system_ks/Embedded_Linux/u-boot
embedded_system_ks@embedded-KS: /media/embedded_system_ks/Embedded_Li...  ×  embedded_system_ks@embedded-KS: /media/embedded_system_ks/Embedded_Li...  ×
  CC      lib/linux_compat.o
  CC      lib/linux_string.o
  CC      lib/lmb.o
  CC      lib/membuff.o
  CC      lib/slre.o
  CC      lib/string.o
  CC      lib/tables_csum.o
  CC      lib/time.o
  CC      lib/hexdump.o
  CC      lib/uuid.o
  LD      lib/libfdt/built-in.o
  CC      lib/panic.o
  CC      lib/vsprintf.o
  CC      lib/strto.o
  CC      lib/strmhz.o
  LD      fs/ubifs/built-in.o
  LD      lib/built-in.o
  LD      fs/built-in.o
  CC      examples/standalone/hello_world.o
  CC      examples/standalone/stubs.o
  CC      examples/standalone/smc911x_eeprom.o
  LD      examples/standalone/libstubs.o
  LD      examples/standalone/hello_world
  OBJCOPY examples/standalone/hello_world.srec
  OBJCOPY examples/standalone/hello_world.bin
  LD      examples/standalone/smc911x_eeprom
  OBJCOPY examples/standalone/smc911x_eeprom.srec
  OBJCOPY examples/standalone/smc911x_eeprom.bin
  LD      u-boot
  OBJCOPY u-boot.srec
  OBJCOPY u-boot-nodtb.bin
  SYM     u-boot.sym
  COPY    u-boot.bin
=================== WARNING ====================
```

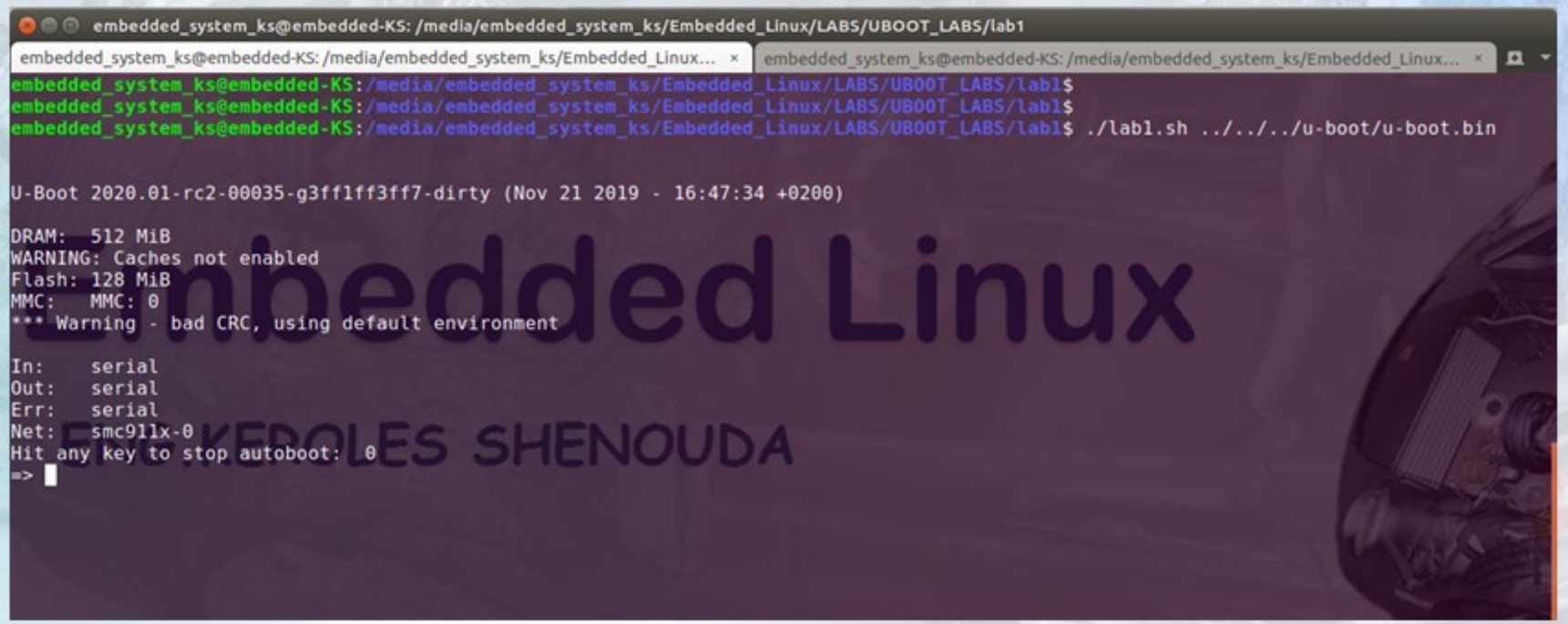
U-boot image

```
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/u-boot$ ls u-boot*
u-boot  u-boot.bin  u-boot.cfg  u-boot.cfg.configs  u-boot.lds  u-boot.map  u-boot-nodtb.bin  u-boot.srec  u-boot.sym
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/u-boot$ arm-none-eabi-readelf -h u-boot
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x60800000
  Start of program headers:          52 (bytes into file)
  Start of section headers:          3357992 (bytes into file)
  Flags:                             0x5000200, Version5 EABI, soft-float ABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         3
  Size of section headers:           40 (bytes)
  Number of section headers:         28
  Section header string table index: 27
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/u-boot$
```



Run U-Boot on vexpress based on ARM Cortex A9 Virtual Platform in QEMU

- As we are using Virtual platform, we can abstract the ROM_Boot Code and we can run the U-Boot Directly

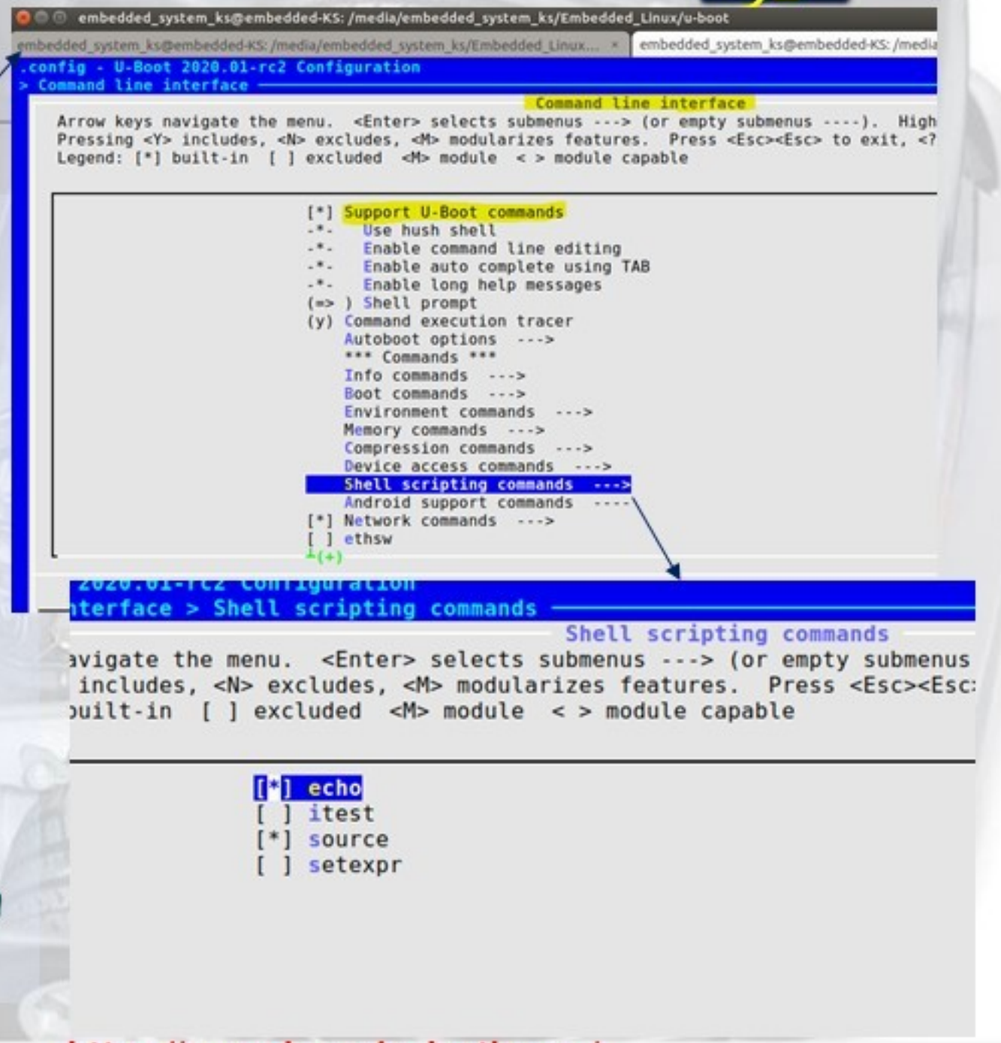


U-boot prompt

- The U-Boot shell offers a set of commands. We will study the most important ones, see the documentation for a complete reference or the help command.

Important commands

- The exact set of commands depends on the U-Boot configuration
  - **help** and help command
  - **ext2load**, loads a file from an ext2 filesystem to RAM
    - And also ext2ls to list files, ext2info for information
  - **fatload**, loads a file from a FAT filesystem to RAM
    - And also fatls and fatinfo
  - **tftp**, loads a file from the network to RAM
  - **ping**, to test the network
  - **boot**, runs the default boot command, stored in *bootcmd*
  - **bootz** , *starts a kernel image loaded at the given address in RAM*

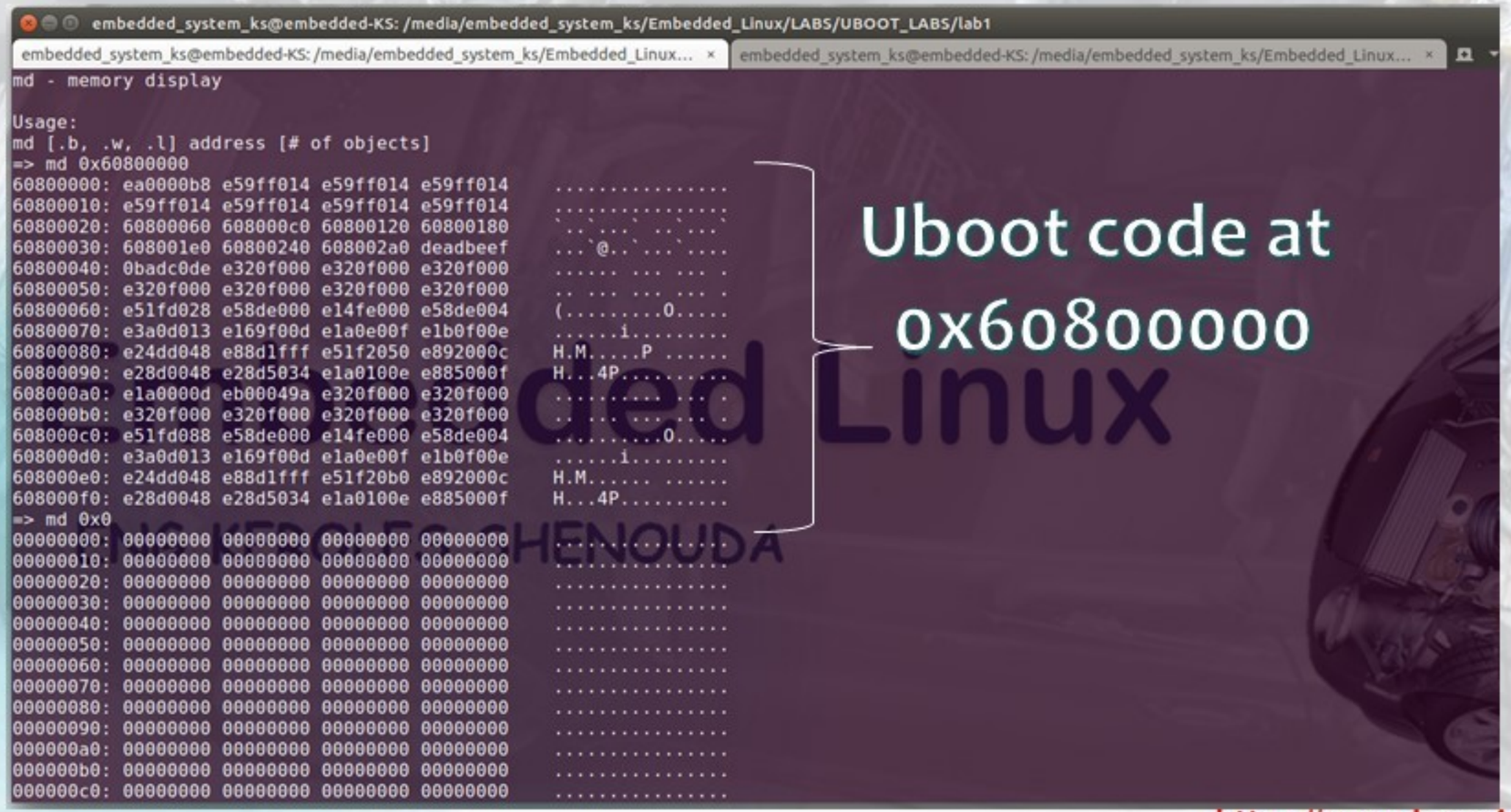


https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Important commands Cont.

- **loadb, loads, loady,** load a file from the serial line to RAM
- **usb**, to initialize and control the USB subsystem, mainly used for USB storage devices such as USB keys
- **mmc**, to initialize and control the MMC subsystem, used for SD and microSD cards
- **nand**, to erase, read and write contents to NAND flash
- **erase, protect, cp,** to erase, modify protection and write to NOR flash
- **md,** displays memory contents. Can be useful to check the contents loaded in memory, or to look at hardware registers.
- **mm**, modifies memory contents. Can be useful to modify directly hardware registers, for testing purposes.

For example md



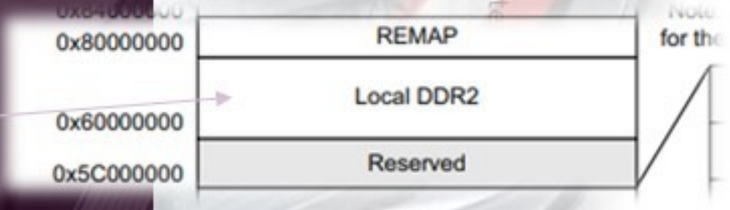
Another examples ☺

That means you didn't connect sdcard On your board

We can see that the default Address for linux kernel In Nor Flash 1 but the ramfile system in DDR2

```
=> mmc list
MMC: 0
=> fatls mmc 0:1
Card did not respond to voltage select!
=> print bootcmd
bootcmd=run distro_bootcmd; run bootflash
=> print distro_bootcmd
distro_bootcmd=for target in ${boot_targets}; do run bootcmd_${target}; done
=> print boot_targets
boot_targets=mmc1 mmc0 pxe dhcp
=> print target
## Error: "target" not defined
=> print bootflash
bootflash=run flashargs; cp ${ramdisk_addr} ${ramdisk_addr_r} ${maxramdisk}; bootm ${kernel_addr} ${ramdisk_addr_r}
=> print kernel_addr
kernel_addr=0x44100000
=> print ramdisk_addr_r
ramdisk_addr_r=0x61000000
=>
```

```
101 [VE_SERIALDVI] = 0x10016000,
102 [VE_RTC] = 0x10017000,
103 [VE_COMPACTFLASH] = 0x1001a000,
104 [VE_CLCD] = 0x1001f000,
105 /* CS0: 0x40000000 .. 0x44000000 */
106 [VE_NORFLASH0] = 0x40000000,
107 /* CS1: 0x44000000 .. 0x48000000 */
108 [VE_NORFLASH1] = 0x44000000,
109 /* CS2: 0x48000000 .. 0x4a000000 */
110 [VE_SRAM] = 0x48000000,
111 /* CS3: 0x4c000000 .. 0x50000000 */
```



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

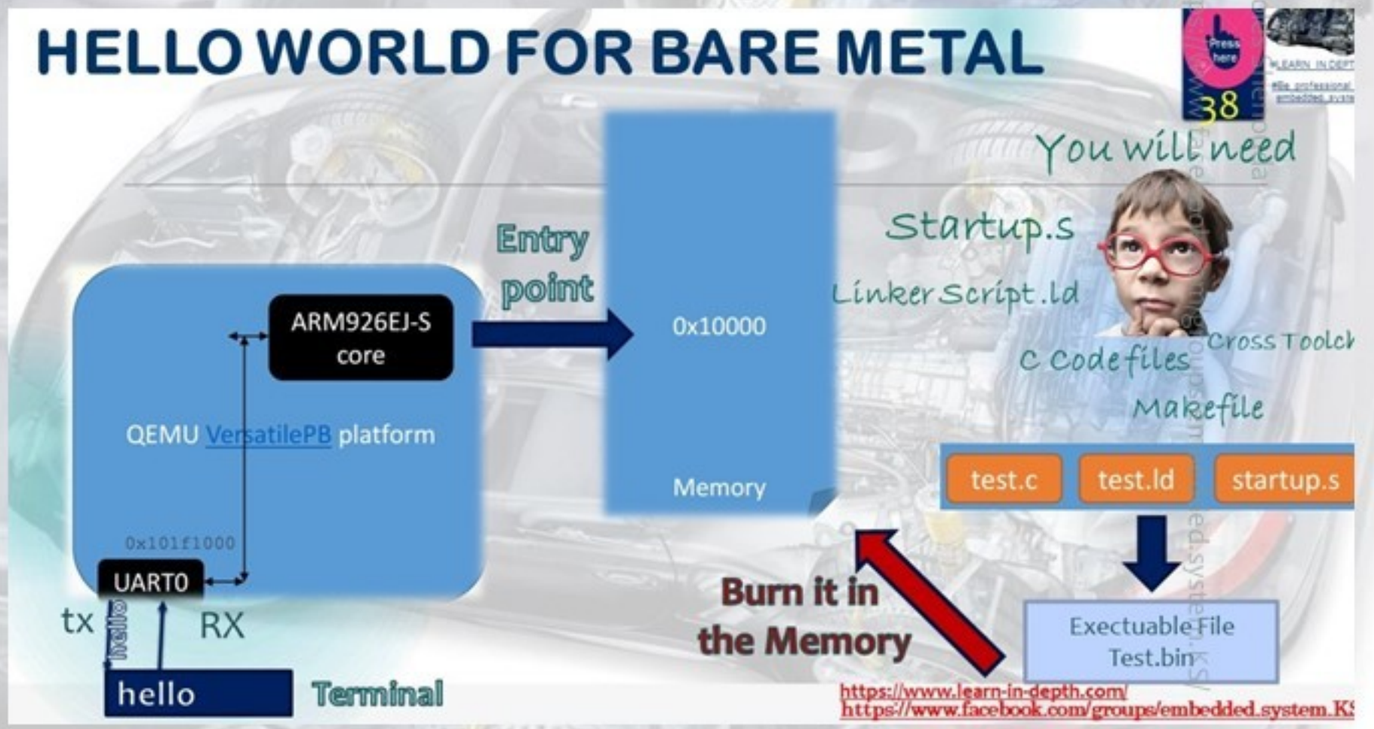
Lab2: run the test application on top of u-boot

KINDLY NOTE THE TEST BAREMETAL APPLICATION IS CREATED ON PART6 LAB1 (CROSS TOOLCHAIN)

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

You already created on PART5

First
We need to run
Without any modification
On Vexpress A9 instead of VersatilePB



Running test.bin without any modifation as a baremetal application

```
embedded_system_ks@embedded-KS: /media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS/lab2
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS/lab2$
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS/lab2$ ./lab2.sh ~/toolchain_labs/lab1/test.bin
qemu-system-arm: Trying to execute code outside RAM or ROM at 0x04000000
This usually means one of the following happened:

(1) You told QEMU to execute a kernel for the wrong machine type, and it crashed on startup (eg trying to run a raspberry pi kernel on a versatilepb QEMU machine)
(2) You didn't give QEMU a kernel or BIOS filename at all, and QEMU executed a ROM full of no-op instructions until it fell off the end
(3) Your guest kernel has a bug and crashed by jumping off into nowhere

This is almost always one of the first two, so check your command line and that you are using the right type of kernel for this machine.
If you think option (3) is likely then you can try debugging your guest with the -d debug options; in particular -d guest_errors will cause the log to include a dump
of the guest register state at this point.

Execution cannot continue; stopping here.

embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS/lab2$
```


Thin in depth ☺

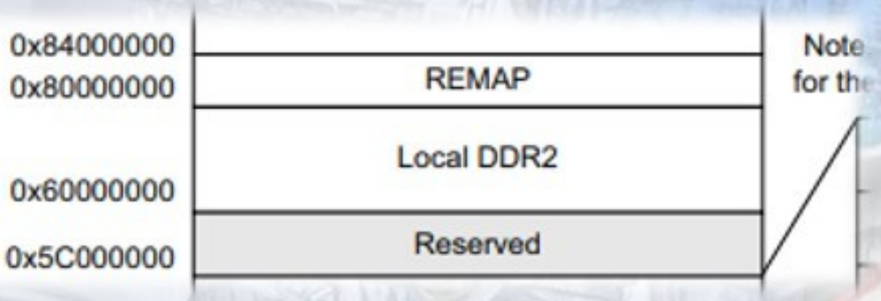
Why this happen ?

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

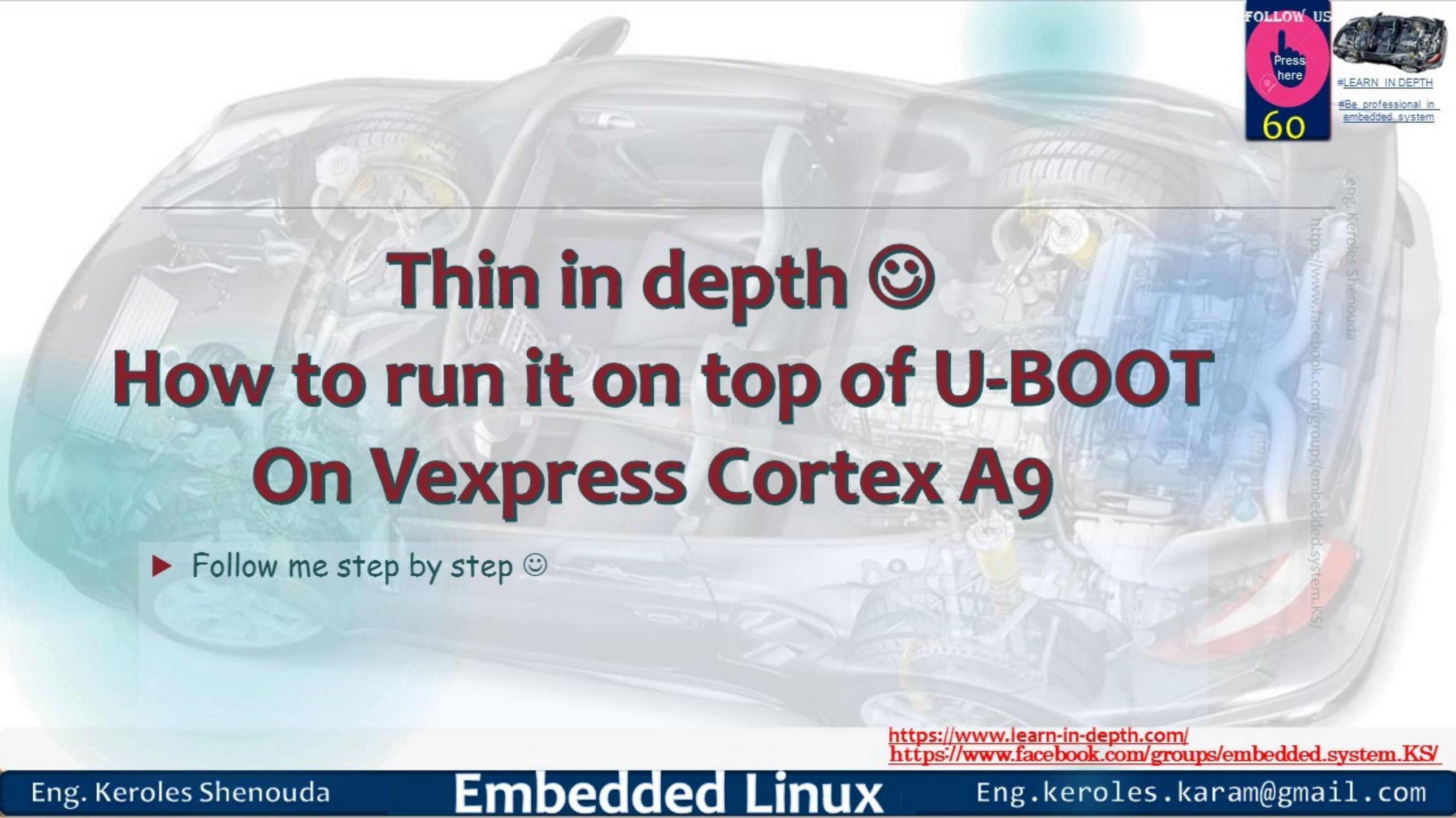
Running test.bin without any modifation as a baremetal application Cont.

- Why this happen ?
- The VersatilePB have a memory on 0x10000
- But in Vexpress A9 we have

```
105  /* CS0: 0x40000000 .. 0x44000000 */
106  [VE_NORFLASH0] = 0x40000000,
107  /* CS1: 0x44000000 .. 0x48000000 */
108  [VE_NORFLASH1] = 0x44000000,
109  /* CS2: 0x48000000 .. 0x4a000000 */
110  [VE_SRAM] = 0x48000000,
111  /* CS3: 0x4c000000 .. 0x50000000 */
```



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/



Thin in depth ☺
How to run it on top of U-BOOT On Vexpress Cortex A9

- Follow me step by step ☺

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Modify the address on the linker script

- $ gedit ~/toolchain_labs/lab1/test.ld &

test.ld (~/toolchain_labs/lab1) - gedit

Open

lab2.sh

```
1 ENTRY(_Reset)
2 SECTIONS
3 {
4 . = 0x48000000 ;
5 .startup . : { startup.o(.text) }
6 .text : { *(.text) }
7 .data : { *(.data) }
8 .bss : { *(.bss COMMON) }
9 . = ALIGN(8);
10 . = . + 0x1000; /* 4kB of stack memory */
11 stack_top = .;
12 }
13
```

```
105 /* CS0: 0x40000000 .. 0x44000000 */
106 [VE_NORFLASH0] = 0x40000000,
107 /* CS1: 0x44000000 .. 0x48000000 */
108 [VE_NORFLASH1] = 0x44000000,
109 /* CS2: 0x48000000 .. 0x4a000000 */
110 [VE_SRAM] = 0x48000000,
111 /* CS3: 0x4c000000 .. 0x50000000 */
```

- This address will be changed on loading by u-boot.
- $ arm-none-eabi-ld test.o startup.o -T test.ld -o a.out

#LEARN IN DEPTH

#Be professional in embedded system

Modify the UART Base address on test.c

test.c (Embedded_Linux /media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS/lab3/SW) - gedit

Open

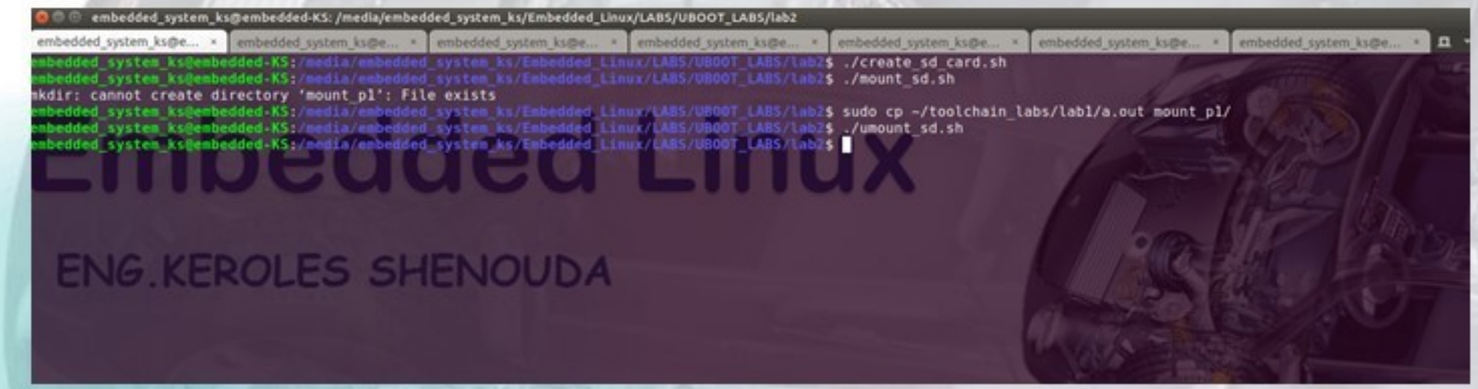
lab3.sh × | create_sd_card.sh × | lab3.sh × | test.ld × | test.c × | Makefile

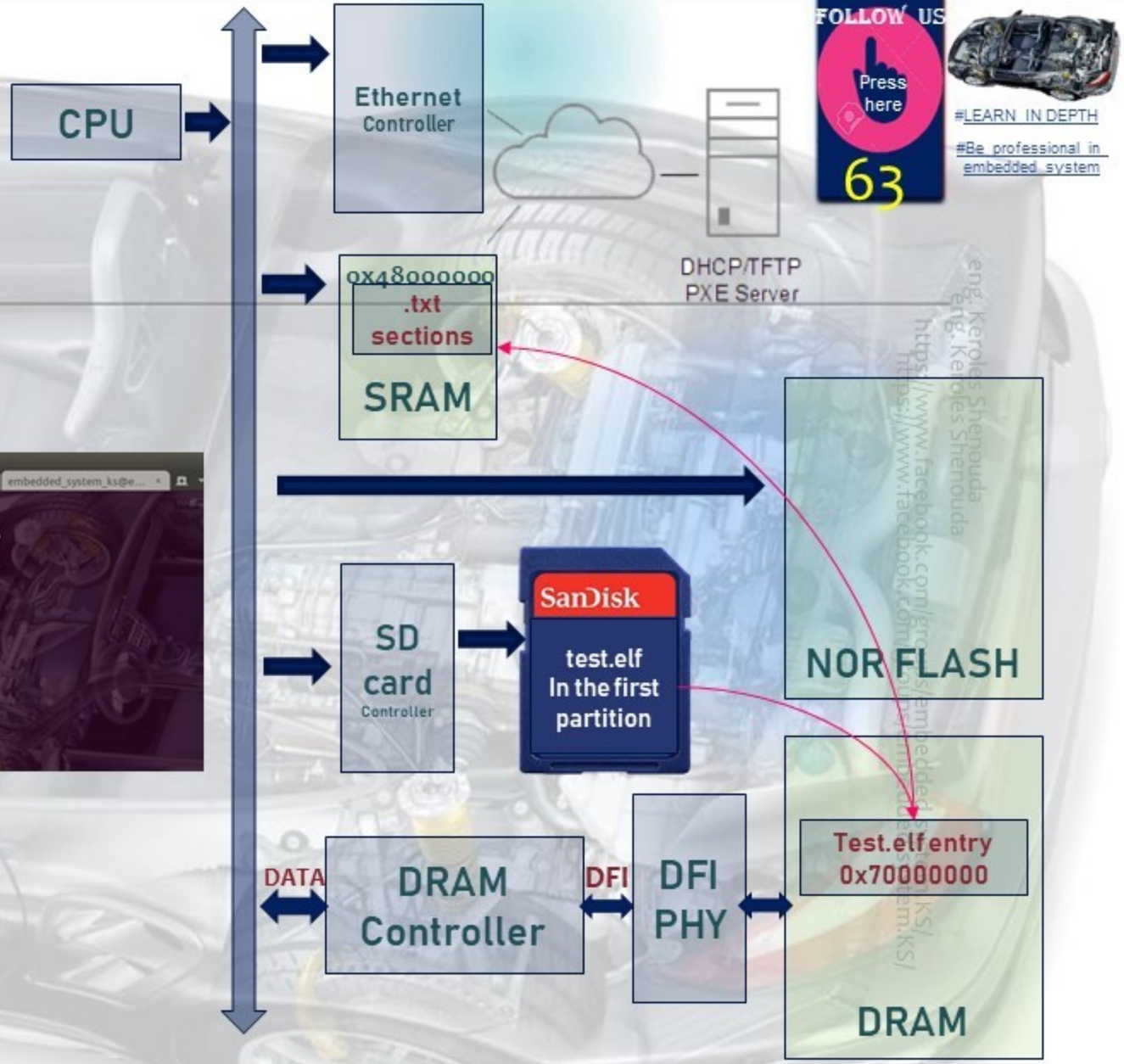
```
1
2 volatile unsigned int * const UART0DR = (unsigned int
  *)0x10009000;
3 void print_uart0(const char *s) {
4         while(*s != '\0') { /* Loop until end of string */
5         *UART0DR = (unsigned int)(*s); /* Transmit char */
6         s++; /* Next char */
7         }
8 }
9 void c_entry() {
10 print_uart0("Hellooooooooooooooooooooooooooo world!\n");
11 }
12
```

```
93  [VE_UART0] = 0x10009000,
94  [VE_UART1] = 0x1000a000,
95  [VE_UART2] = 0x1000b000,
96  [VE_UART3] = 0x1000c000,
```


https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Creating SD CARD

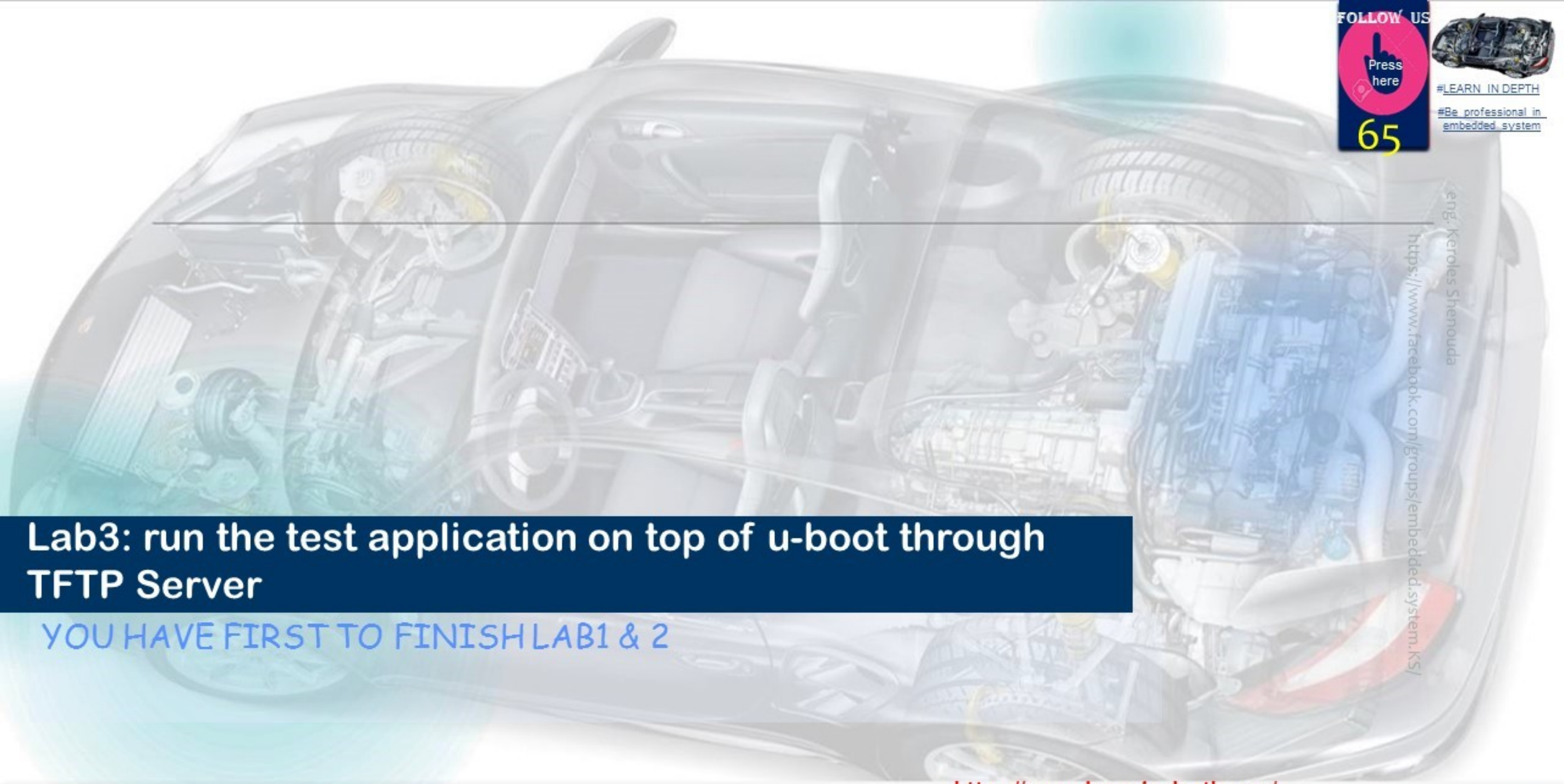




https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Running a.out top of uboot

```
embedded_system_ks@embedded-KS: /media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS/lab2
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS/lab2$ ./lab2.sh ../../../u-boot/u-boot KS_SD_512M.img


U-Boot 2020.01-rc2-00035-g3ff1ff3ff7 (Jan 10 2020 - 22:28:09 +0200)

DRAM:  512 MiB
WARNING: Caches not enabled
Flash: 128 MiB
MMC:   MMC: 0
*** Warning - bad CRC, using default environment

In:    serial
Out:   serial
Err:   serial
Net:   smc911x-0
Hit any key to stop autoboot:  0
=> mmcinfo
Device: MMC
Manufacturer ID: aa
OEM: 5859
Name: QEMU!
Bus Speed: 6250000
Mode: SD Legacy
Rd Block Len: 512
SD version 1.0
High Capacity: No
Capacity: 512 MiB
Bus Width: 1-bit
Erase Group Size: 512 Bytes
=> fatl
  fatload fatls
=> fatls mmc 0:1
    67828   a.out

1 file(s), 0 dir(s)

=> fatload mmc 0:1 0x70000000 a.out
67828 bytes read in 100 ms (662.1 KiB/s)
=> bootelf 0x70000000
## Starting application at 0x48000000 ...
Hello world!
```

Lab3: run the test application on top of u-boot through TFTP Server

YOU HAVE FIRST TO FINISH LAB1 & 2

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Lab2: run the test application on top of u-boot through TFTP Serve

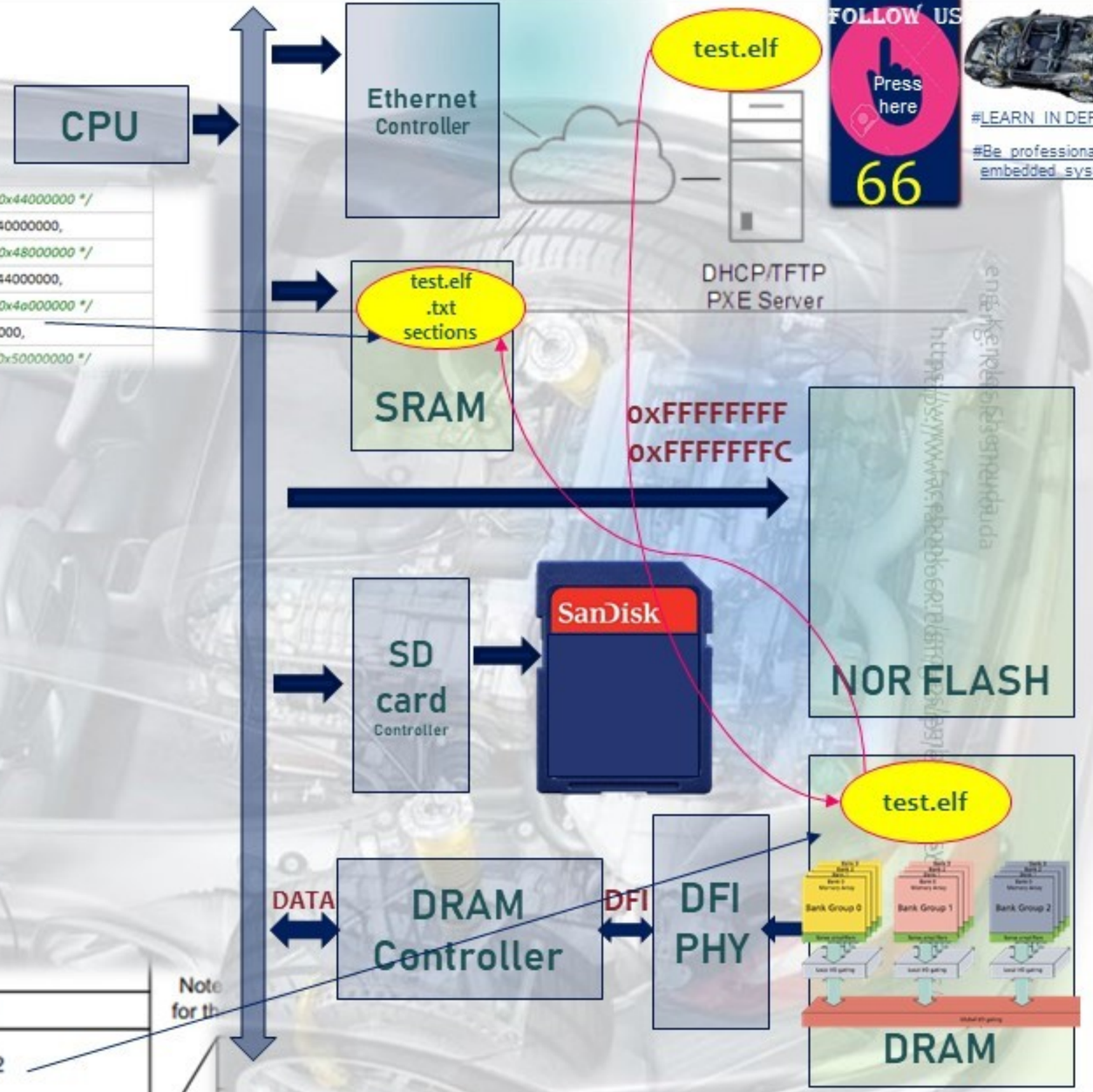
- On this Lab you will create a tftp server on your host machine

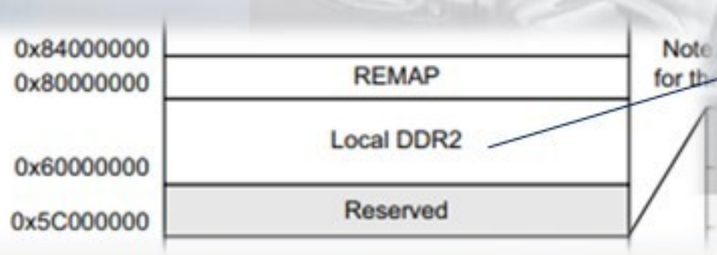
Then you will put your bare-metal SW test.elf in the server mount point.

- After that you Loading the test.elf from tftp server through Uboot Network Command line.
- At runtime we will load the test.elf to 0x70000000
- The linker scripst put the beginning memory layout in SRAM

Ar address 0x48000000

```
105 /* CS0: 0x40000000 .. 0x44000000 */
106 [VE_NORFLASH0] = 0x40000000,
107 /* CS1: 0x44000000 .. 0x48000000 */
108 [VE_NORFLASH1] = 0x44000000,
109 /* CS2: 0x48000000 .. 0x4a000000 */
110 [VE_SRAM] = 0x48000000,
111 /* CS3: 0x4c000000 .. 0x50000000 */
```





https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Installing and Testing TFTP Server in Ubuntu

- Install following packages.
  - $ sudo apt-get install xinetd tftpd tftp
- Create "sudo gedit /etc/xinetd.d/tftp" and put this entry
- Create a folder /tftpboot
  - $ sudo mkdir /tftpboot
  - $ sudo chmod -R 777 /tftpboot
  - $ sudo chown -R nobody /tftpboot
- Restart the xinetd service

```
service tftp
{
protocol        = udp
port            = 69
socket_type     = dgram
wait            = yes
user            = nobody
server          = /usr/sbin/in.tftpd
server_args     = /tftpboot
disable         = no
}
```

```
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS/lab3$ sudo /etc/init.d/xinetd stop
[ ok ] Stopping xinetd (via systemctl): xinetd.service.
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS/lab3$ sudo /etc/init.d/xinetd start
[ ok ] Starting xinetd (via systemctl): xinetd.service.
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS/lab3$
```


https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Testing our tftp server

```
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS/lab3/SW
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Lin...  embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Lin...
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS
/lab3/SW$ make all
arm-none-eabi-gcc -c -mcpu=arm926ej-s  -I . -g  test.c -o test.o
arm-none-eabi-as  -mcpu=arm926ej-s  -I . -g  startup.s  -o  startup.o
arm-none-eabi-ld -T test.ld test.o startup.o  -o test.elf
arm-none-eabi-objcopy -O binary test.elf test.bin
echo "finished"
finished
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS
/lab3/SW$ sudo cp test.elf /tftpboot/
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS
/lab3/SW$ ifconfig enp0s3
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.45  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::a42e:c39f:8e3f:a916  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:a8:79:b7  txqueuelen 1000  (Ethernet)
        RX packets 28449  bytes 2046034 (2.0 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 246  bytes 30249 (30.2 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS
/lab3/SW$ tftp 192.168.1.45
tftp> get tes.elf
Error code 1: File not found
tftp> get test.elf
Received 67905 bytes in 0.0 seconds
tftp>
```

Build Your Application as lab2

Copy the SW to /tftpboot/

Get the IP address for the FTP Server

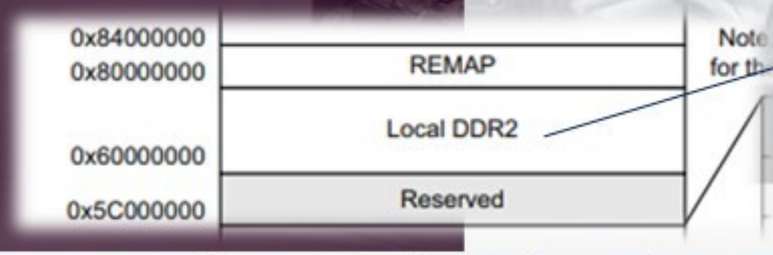
Connect with the tftp

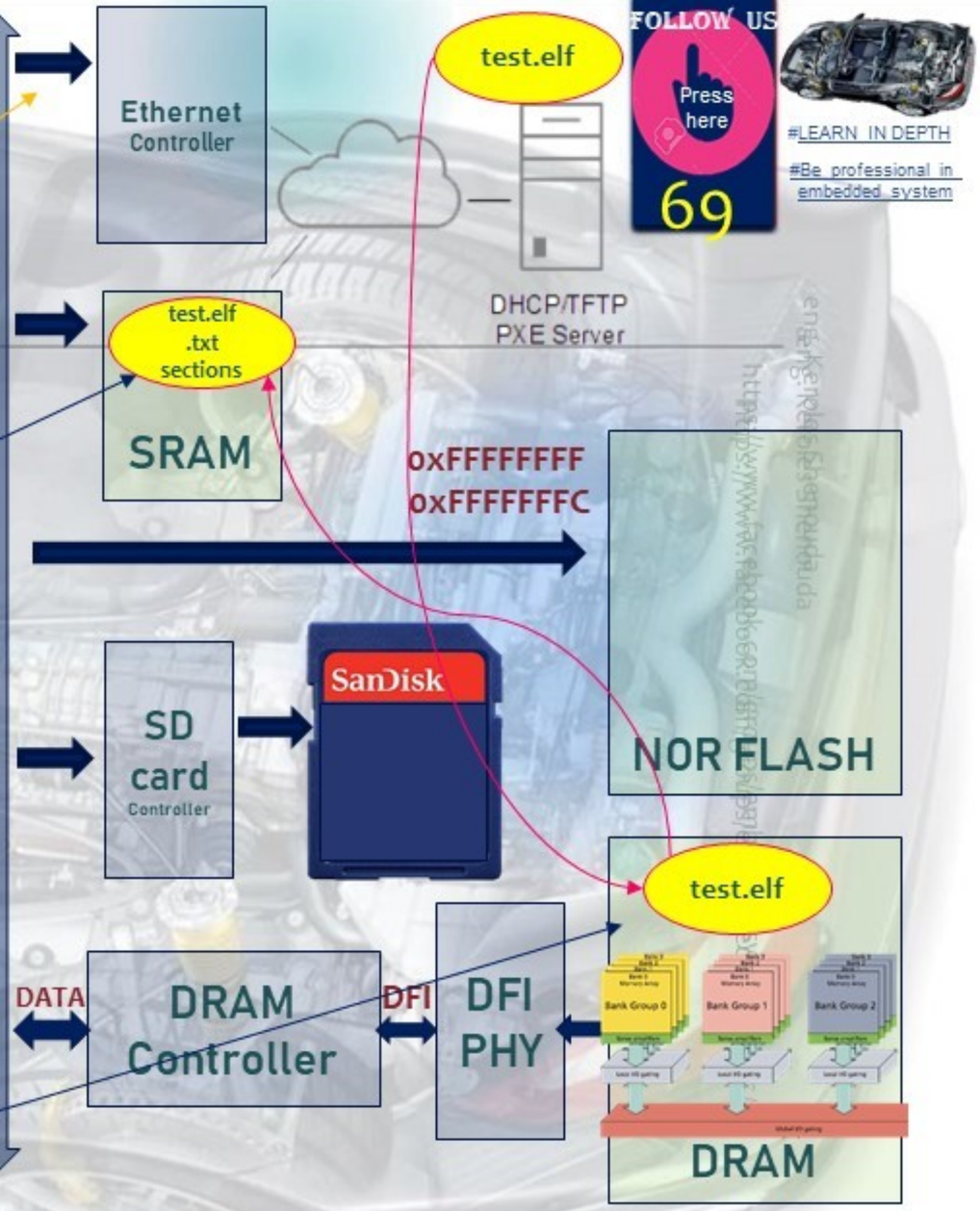
Download the SW from the tftp server

Now run the Board ☺

```
embedded_system_ks@embedded-KS: /media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS/lab3
Net:   smc911x-0
Hit any key to stop autoboot:  0
=> dhcp &
syntax error
=> dhcp
smc911x: MAC 52:54:00:12:34:56
smc911x: detected LAN9118 controller
smc911x: phy initialized
smc911x: MAC 52:54:00:12:34:56
BOOTP broadcast 1
DHCP client bound to address 10.0.2.15 (3 ms)
*** Warning: no boot file name; using '0A00020F.img'
Using smc911x-0 device
TFTP from server 10.0.2.2; our IP address is 10.0.2.15
Filename '0A00020F.img'.
smc911x: MAC 52:54:00:12:34:56

TFTP error: trying to overwrite reserved memory...
smc911x: MAC 52:54:00:12:34:56
=> setenv serverip 192.168.1.45
=> tftp 0x70000000 test.elf
smc911x: MAC 52:54:00:12:34:56
smc911x: detected LAN9118 controller
smc911x: phy initialized
smc911x: MAC 52:54:00:12:34:56
Using smc911x-0 device
TFTP from server 192.168.1.45; our IP address is 10.0.2.15; sending through gateway 10.
0.2.2
Filename 'test.elf'.
Load address: 0x70000000
Loading: ##############
         124 KiB/s
done
Bytes transferred = 67896 (10938 hex)
smc911x: MAC 52:54:00:12:34:56
=> bootelf 0x70000000
## Starting application at 0x48000000 ...
Hellooooooooooooooooooooooooooo world!
```

```
105 /* CS0: 0x40000000 .. 0x44000000 */
106 [VE_NORFLASH0] = 0x40000000,
107 /* CS1: 0x44000000 .. 0x48000000 */
108 [VE_NORFLASH1] = 0x44000000,
109 /* CS2: 0x48000000 .. 0x4a000000 */
110 [VE_SRAM] = 0x48000000,
111 /* CS3: 0x4c000000 .. 0x50000000 */
```



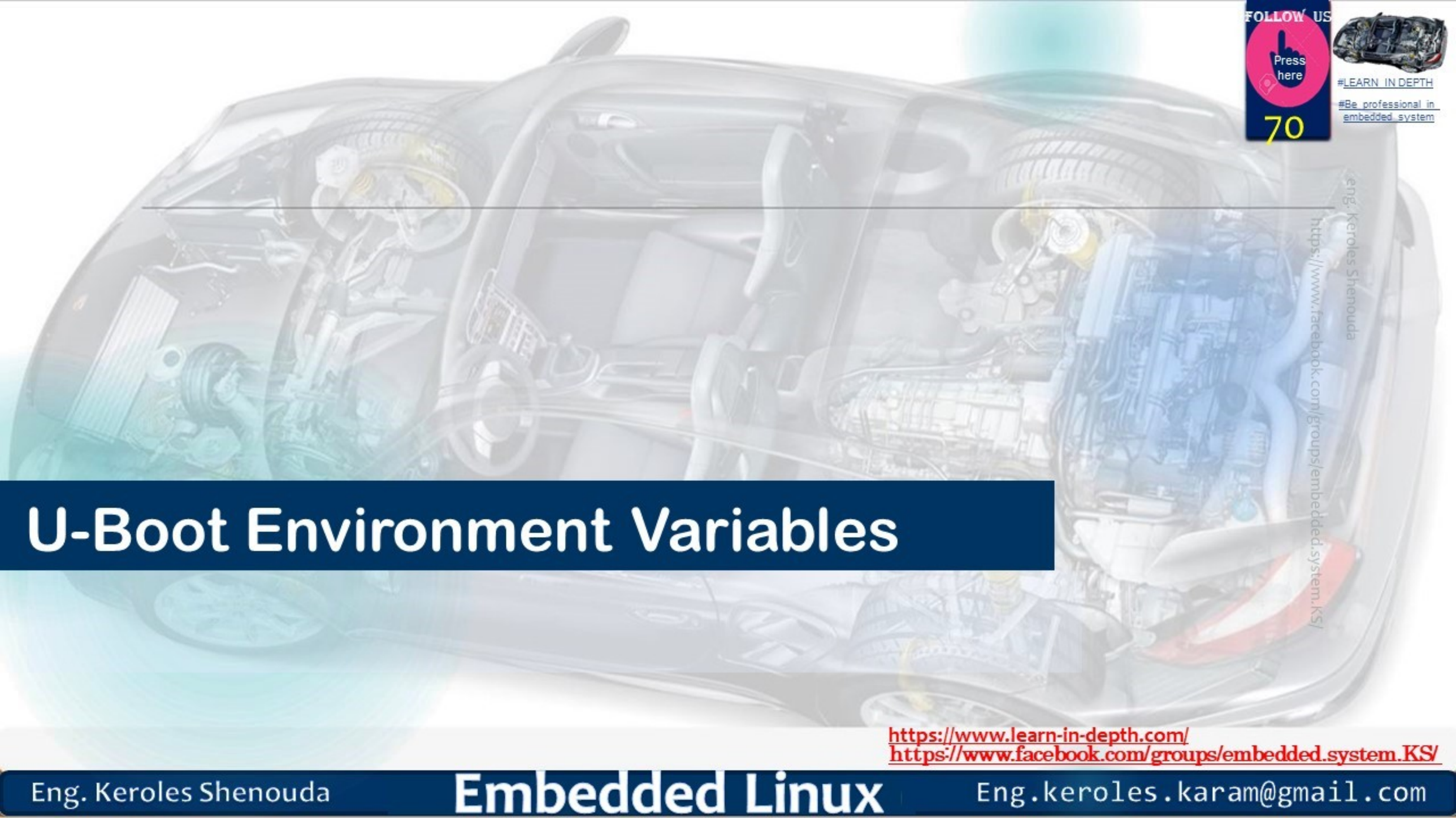


FOLLOW US
Press here
69

#LEARN IN DEPTH

#Be professional in embedded system

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/



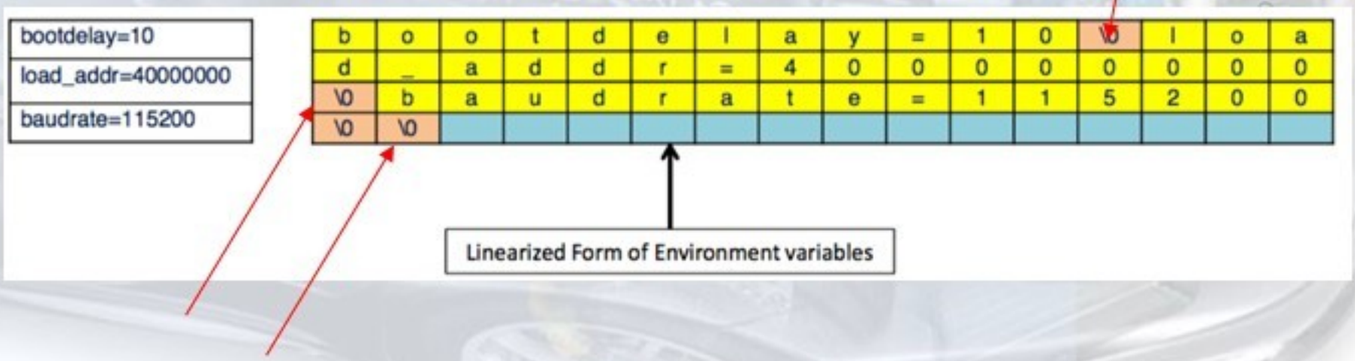
U-Boot Environment Variables

U-Boot Environment variables

- U-Boot can be configured through environment variables
  - Some specific environment variables affect the behavior of the different commands.
  - Custom environment variables can be added, and used in scripts.
- Environment variables are loaded from (flash or in MMC storage) according to uboot configuration to RAM at U-Boot startup, can be modified and saved back to (flash or in MMC storage) for saving.

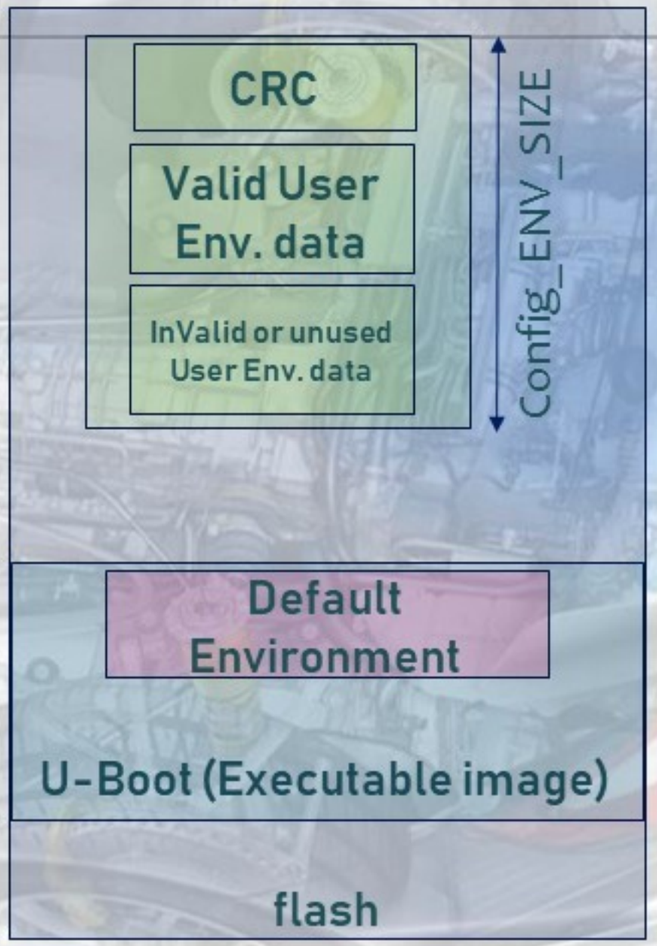
U-Boot Environment variables Cont.

- A collection of **name=value** pairs". Here, **name** refers to the name of the environment variable to which we wish to assign some **value**.
- This "value" could be of any type: string, hexadecimal, boolean etc. Whatever be the type of the value, it is ***converted*** into **string** before being **stored** in a linearized environment data block.
- Each environment variable pair ("name=value") would be stored as a **null terminated string**



How is the environment stored?

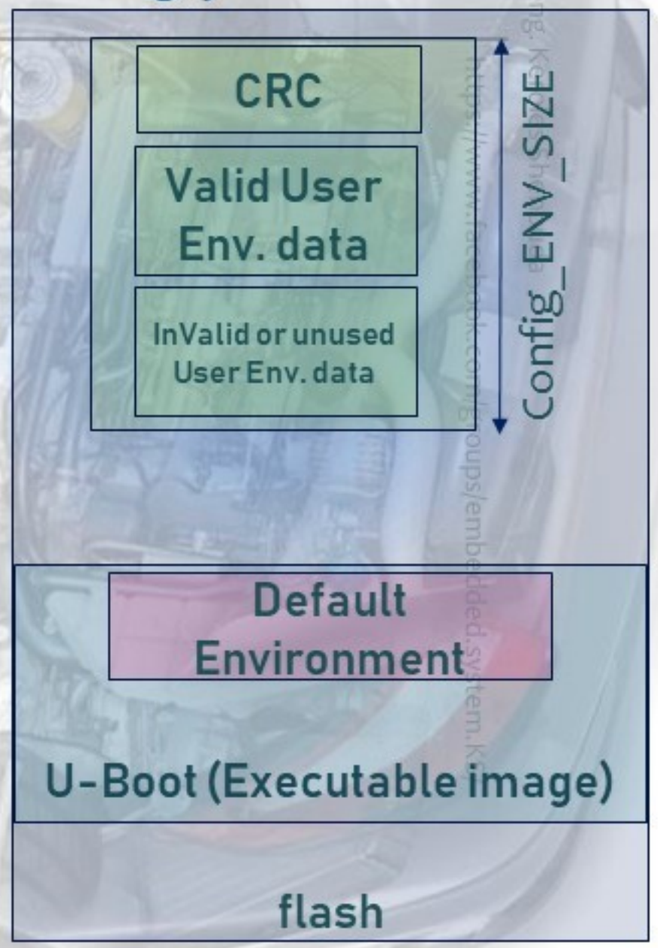
- U-Boot has 2 types of persistent environments
  - Default Environment (Compiled-in, Read-Only)
  - User Supplied Environment (Flashed in external storage, writable)



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Default Environment (Compiled-in, Read-Only)

- Every U-Boot binary has a default built-in environment of its own
- During compilation, a character array called *_defaultenvironment* is embedded in to the U-Boot image
- This character array stores the environment variables as a list of null terminated strings with a double null terminator.
- Environment variables which are commonly used can be enabled by defining the corresponding CONFIGS's in your board config file (include/configs/<YOUR_BOARD>.h)



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

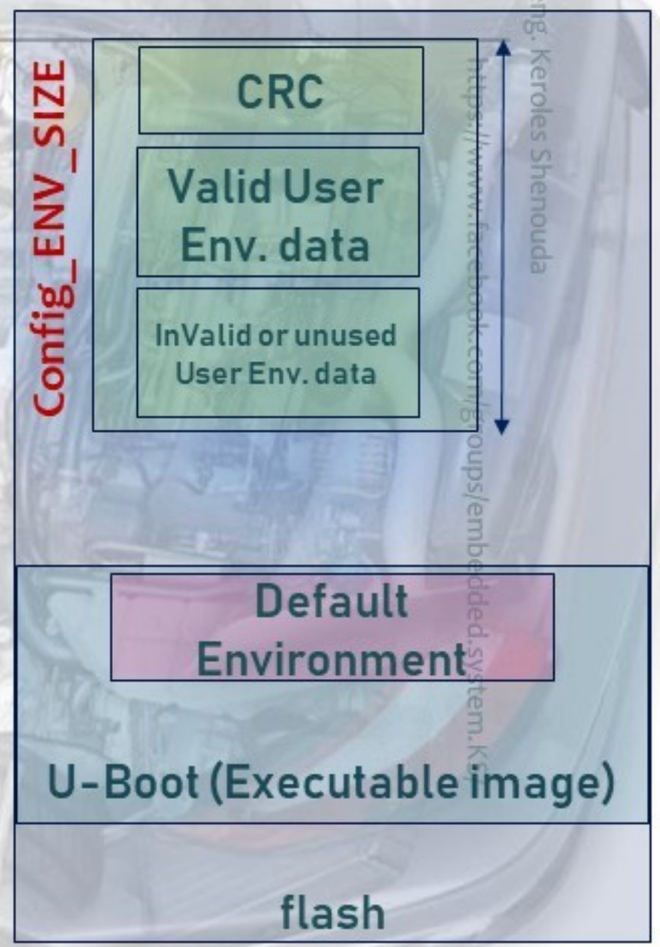
u-boot/include/configs/vexpress_ca9x4.h

vexpress_ca9x4.h | vexpress_common.h

```
/* SPDX-License-Identifier: GPL-2.0+ */
 * (C) Copyright 2011 Linaro

#ifndef __VEXPRESS_CA9X4_H
#define __VEXPRESS_CA9X4_H

#define CONFIG_VEXPRESS_ORIGINAL_MEMORY_MAP
#include "vexpress_common.h"

#endif /* VEXPRESS_CA9X4_H */
```

vexpress_ca9x4.h | vexpress_common.h

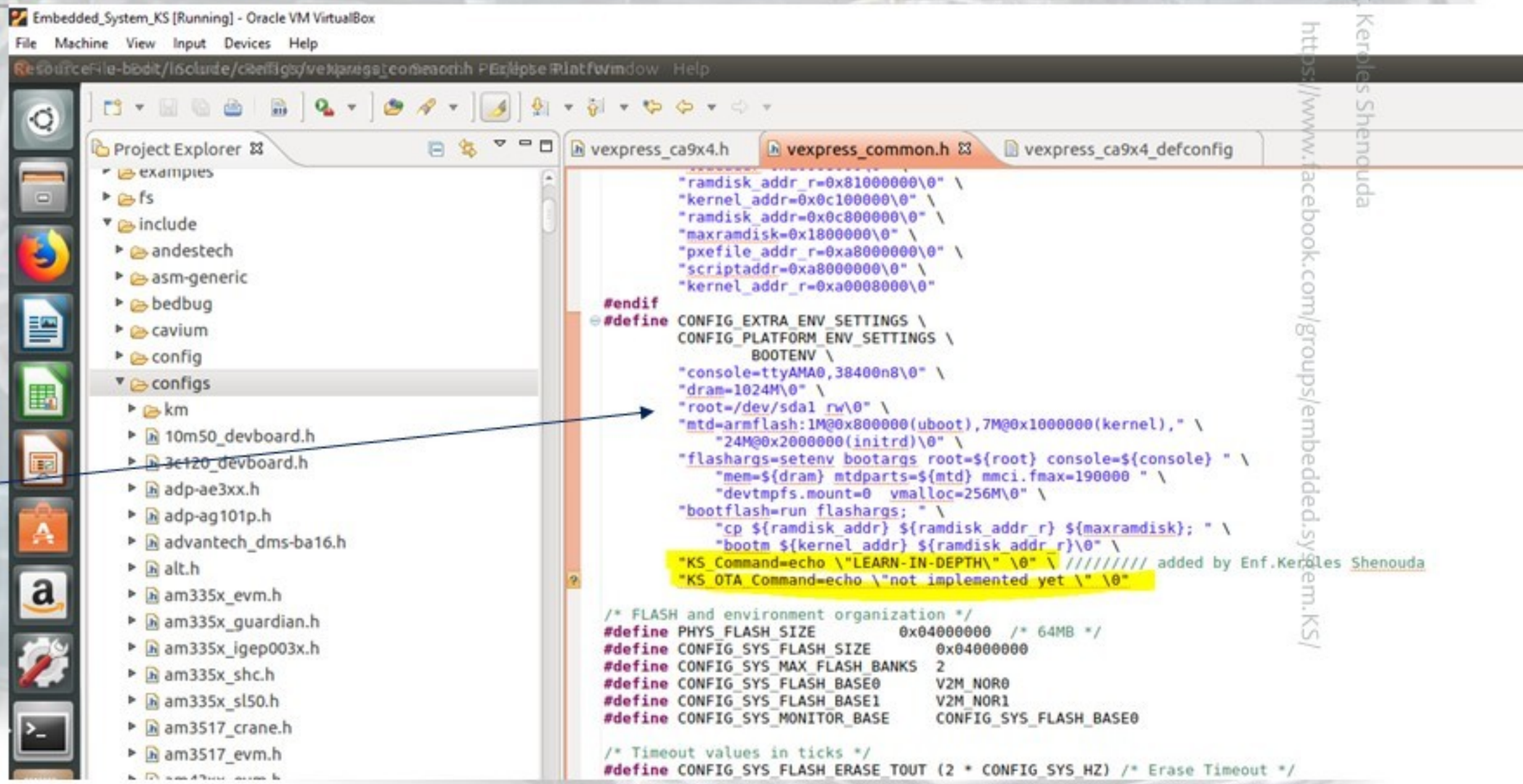
```
        "ramdisk_addr_r=0x81000000\0" \
        "kernel_addr=0x0c100000\0" \
        "ramdisk_addr=0x0c800000\0" \
        "maxramdisk=0x1800000\0" \
        "pxefile_addr_r=0xa8000000\0" \
        "scriptaddr=0xa8000000\0" \
        "kernel_addr_r=0xa0008000\0"
#endif
#define CONFIG_EXTRA_ENV_SETTINGS \
        CONFIG_PLATFORM_ENV_SETTINGS \
            BOOTENV \
        "console=ttyAMA0,38400n8\0" \
        "dram=1024M\0" \
        "root=/dev/sda1 rw\0" \
        "mtd=armflash:1M@0x800000(uboot),7M@0x1000000(kernel)," \
            "24M@0x2000000(initrd)\0" \
        "flashargs=setenv bootargs root=${root} console=${console} " \
            "mem=${dram} mtdparts=${mtd} mmci.fmax=190000 " \
            "devtmpfs.mount=0  vmalloc=256M\0" \
        "bootflash=run flashargs; " \
            "cp ${ramdisk_addr} ${ramdisk_addr_r} ${maxramdisk}; " \
            "bootm ${kernel_addr} ${ramdisk_addr_r}\0"

/* FLASH and environment organization */
#define PHYS_FLASH_SIZE         0x04000000  /* 64MB */
#define CONFIG_SYS_FLASH_SIZE       0x04000000
#define CONFIG_SYS_MAX_FLASH_BANKS  2
#define CONFIG_SYS_FLASH_BASE0      V2M_NOR0
#define CONFIG_SYS_FLASH_BASE1      V2M_NOR1
#define CONFIG_SYS_MONITOR_BASE     CONFIG_SYS_FLASH_BASE0

/* Timeout values in ticks */
#define CONFIG_SYS_FLASH_ERASE_TOUT (2 * CONFIG_SYS_HZ) /* Erase Time
#define CONFIG_SYS_FLASH_WRITE_TOUT (2 * CONFIG_SYS_HZ) /* Write Ti

/* 255 0x40000 sectors + first or last sector may have 4 erase
#define CONFIG_SYS_MAX_FLASH_SECT   259     /* Max sectors */
#define FLASH_MAX_SECTOR_SIZE       0x00040000  /* 256 KB se

/* Room required on the stack for the environment data */
#define CONFIG_ENV_SIZE         FLASH_MAX_SECTOR_SIZE

/*
 * Amount of flash used for environment:
```



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

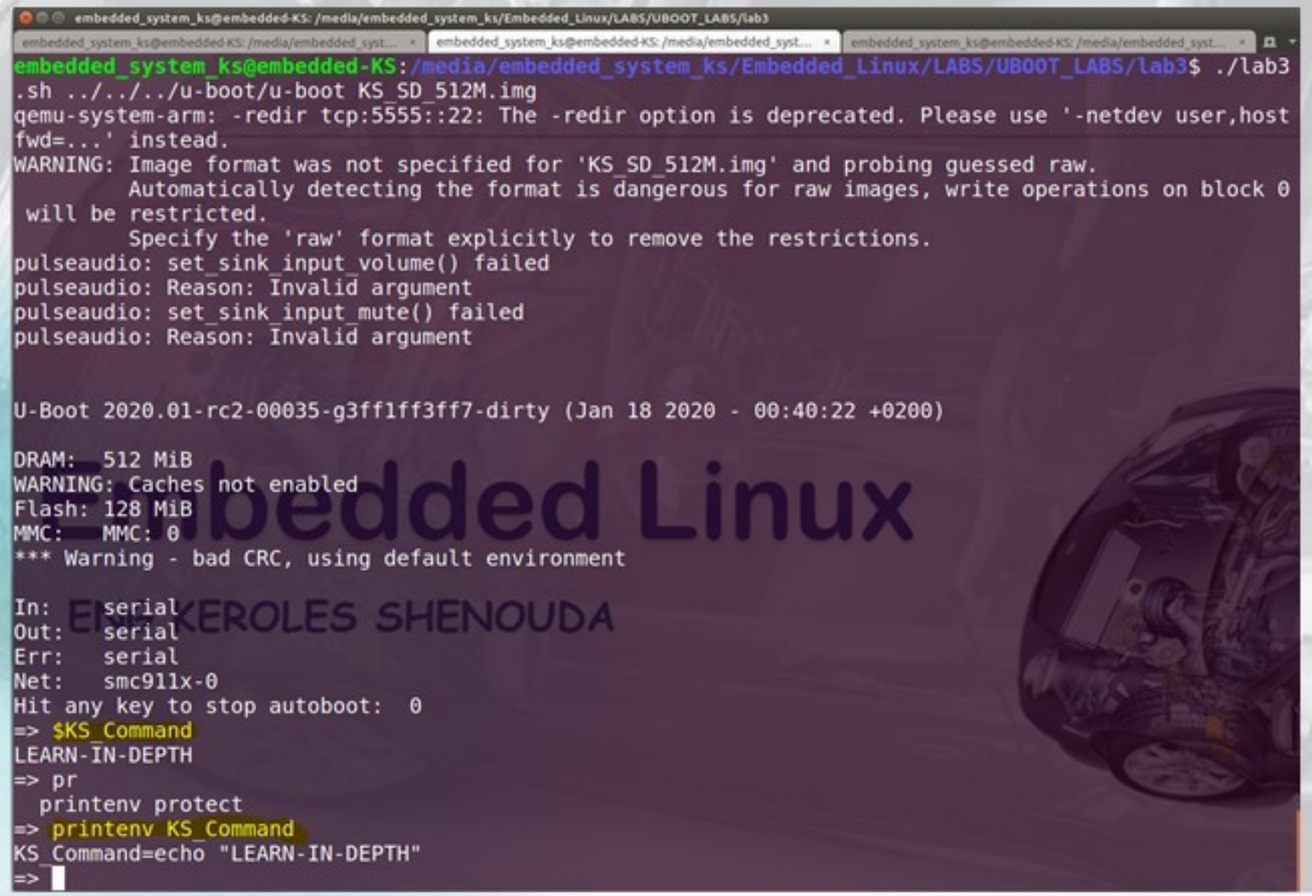
How to add environment variables specific to your board

- You could do this by defining all these variables in a macro called *CONFIG_EXTRA_ENV_SETTINGS* in your board config file.
- You must remember that default environment is "**read-only**" as it is part of U-Boot image itself.
- Vendors normally keep some essential system variables as part of this environment.
- You must not keep too much data in to this default environment as it directly adds to the ***weight*** of the binary. Only keep critical system variables in this environment.



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

How to add environment variables specific to your board Cont.

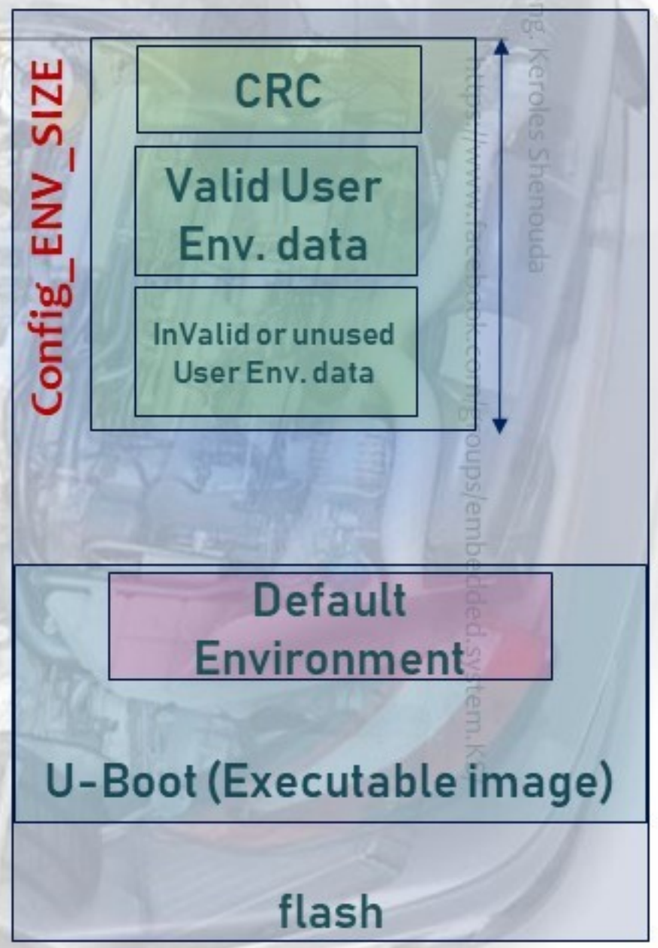


Rebuild and RUN ☺

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

User Supplied Environment (Flashed in external storage, writable)

- The format of this prebuilt environment is the same " linearized list of strings", but there is a 4 byte CRC header prefixed to it.
- This CRC is computed over the environment data.
- Total size of this environment data is fixed to CONFIG_ENV_SIZE during compilation
  - So, if your environment usage exceeds this size you would need to recompile your U-Boot binary after increasing CONFIG_ENV_SIZE
  - If you do not increase the size, U-Boot would refuse to save the environment variables.



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Saving User Supplied Environment Environment

- whenever you modify a variable and issue a saveenv command, that variable would end up in User environment. When you do saveenv, U-Boot would:
  - sort the list of current environment variables
  - convert them to a linearized list of strings
  - compute CRC over this data and burn the environment back at its fixed location in storage.

Commands to manipulate environment variables:

- printenv
  - Shows all variables
- printenv <variable-name>
  - Shows the value of a variable
- setenv <variable-name> <variable-value>
  - Changes the value of a variable, only in RAM
- editenv <variable-name>
  - Edits the value of a variable, only in RAM
- saveenv
  - Saves the current state of the environment to flash

#LEARN IN DEPTH

#Be professional in embedded system

Environment variables commands - Example

```
embedded_system_ks@embedded-KS: /media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS/lab3
embedded_system_ks@embedded-KS: /media/embedded_syst...  embedded_system_ks@embedded-KS: /media/embedded_syst...  embedded_system_ks@embedded-KS: /media/embedded_syst...
  printenv protect
=> printenv KS_Command
KS_Command=echo "LEARN-IN-DEPTH"
=> printenv
KS_Command=echo "LEARN-IN-DEPTH"
arch=arm
baudrate=38400
board=vexpress
board_name=vexpress
boot_a_script=load ${devtype} ${devnum}:${distro_bootpart} ${scriptaddr} ${prefix}${script}; source
${scriptaddr}
boot_extlinux=sysboot ${devtype} ${devnum}:${distro_bootpart} any ${scriptaddr} ${prefix}${boot_sysl
inux_conf}
boot_prefixes=/ /boot/
boot_script_dhcp=boot.scr.uimg
boot_scripts=boot.scr.uimg boot.scr
boot_syslinux_conf=extlinux/extlinux.conf
boot_targets=mmc1 mmc0 pxe dhcp
bootcmd=run distro_bootcmd; run bootflash
bootcmd_dhcp=if dhcp ${scriptaddr} ${boot_script_dhcp}; then source ${scriptaddr}; fi;
bootcmd_mmc0=devnum=0; run mmc_boot
bootcmd_mmc1=devnum=1; run mmc_boot
bootcmd_pxe=dhcp; if pxe get; then pxe boot; fi
bootdelay=2
bootflash=run flashargs; cp ${ramdisk_addr} ${ramdisk_addr_r} ${maxramdisk}; bootm ${kernel_addr} ${
ramdisk_addr_r}
console=ttyAMA0,38400n8
cpu=armv7
distro_bootcmd=for target in ${boot_targets}; do run bootcmd_${target}; done
dram=1024M
ethact=smc911x-0
ethaddr=52:54:00:12:34:56
flashargs=setenv bootargs root=${root} console=${console} mem=${dram} mtdparts=${mtd} mmci.fmax=1900
00 devtmpfs.mount=0  vmalloc=256M
```

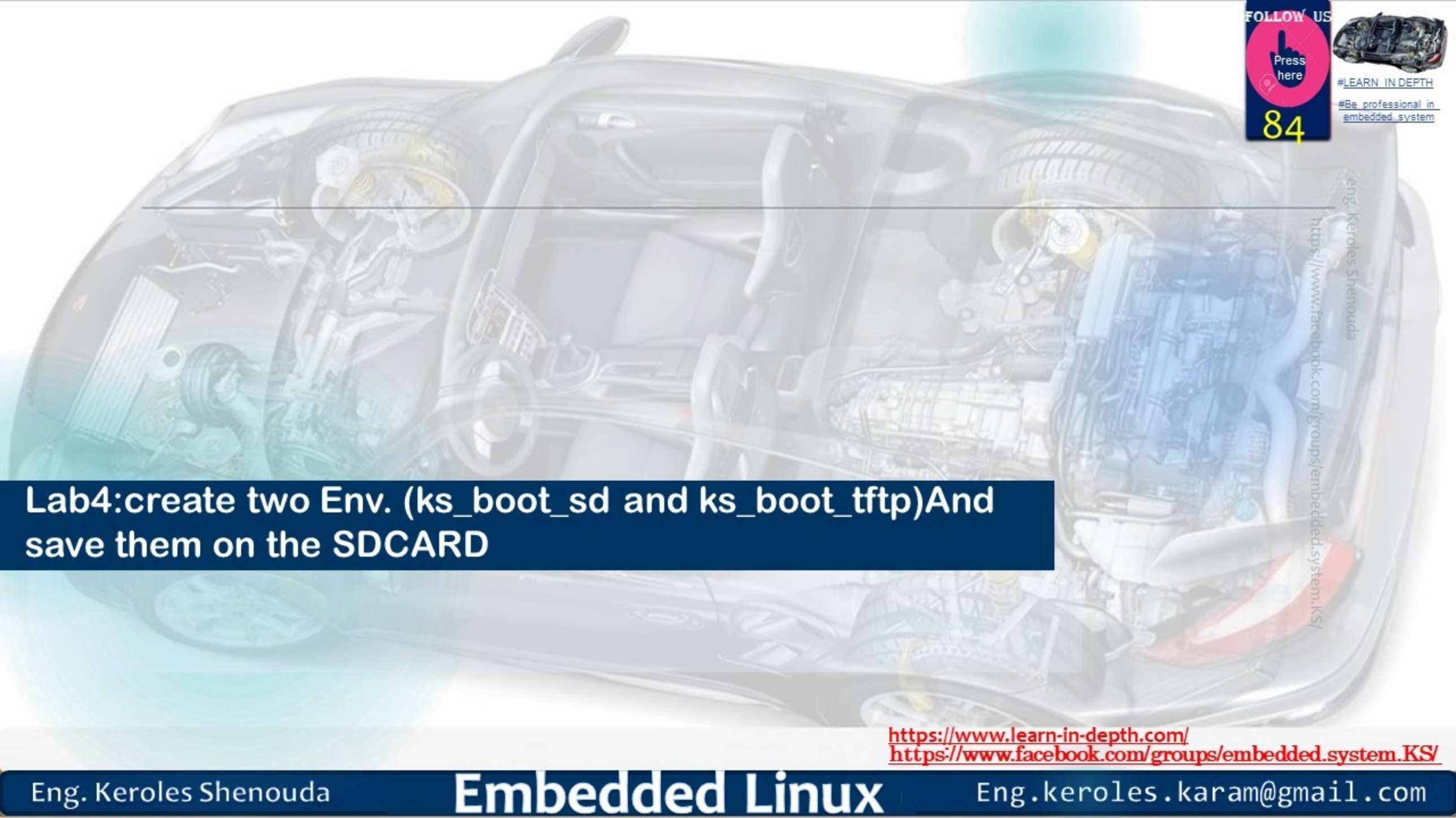

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Important U-Boot env variables

- bootcmd
  - specifies the commands **that U-Boot will automatically** execute at boot time after a configurable delay (bootdelay), if the process is not interrupted.
- Bootargs
  - contains the arguments passed to the Linux kernel
- Serverip
  - the IP address of the server that U-Boot will contact for network related commands
- ipaddr
  - the IP address that U-Boot will use
- netmask, the network mask to contact the server
- ethaddr, the MAC address, can only be set once
- autostart, if set to yes, U-Boot automatically starts an image after loading it in memory (tftp, fatload...)

Scripts in environment variables

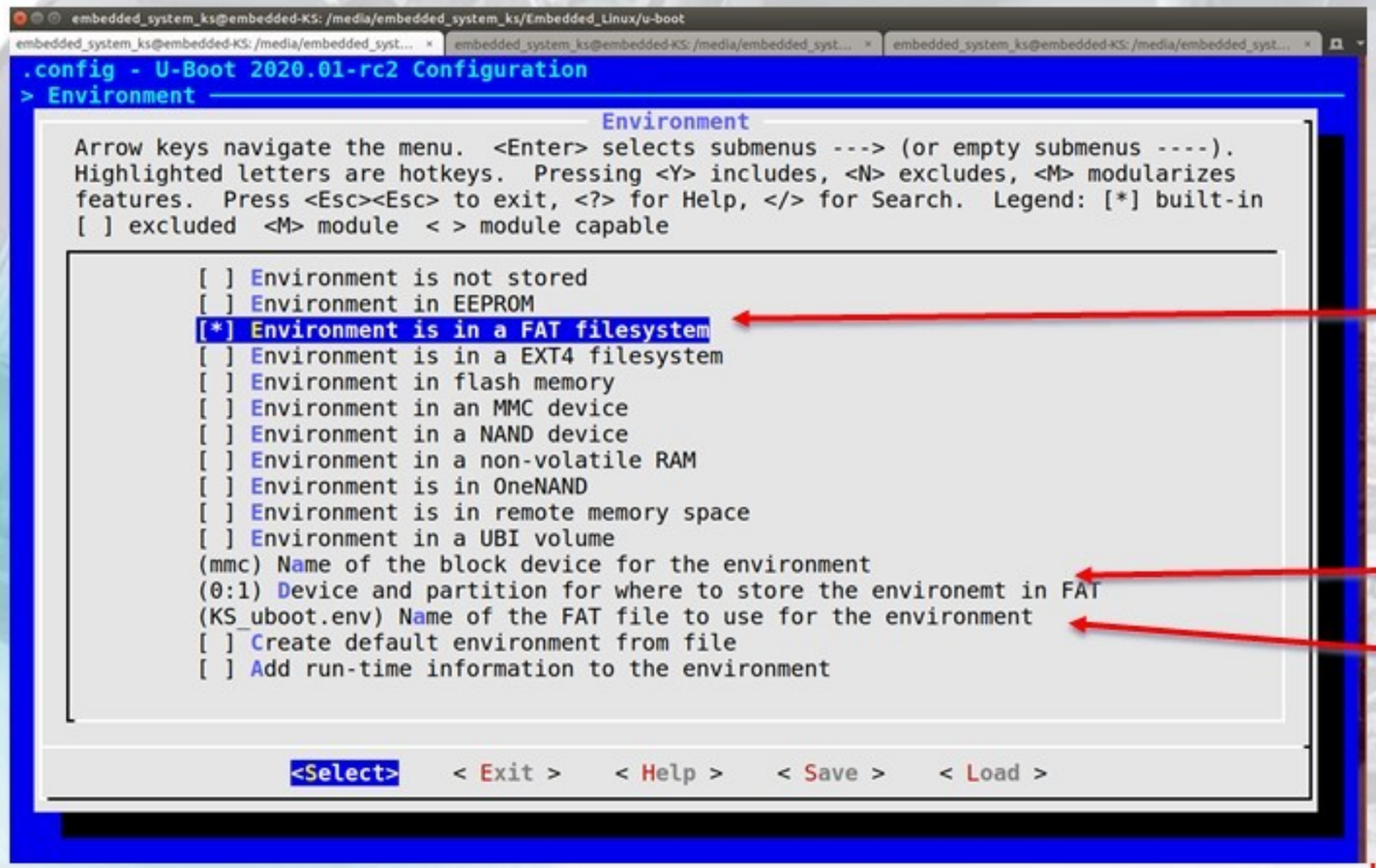
- Environment variables can contain small scripts, to execute several commands and test the results of commands.
  - Useful to automate booting or upgrade processes
  - Several commands can be chained using the ; operator
  - Tests can be done using if command ; then ... ; else ... ; fi
  - Scripts are executed using run <variable-name>
  - You can reference other variables using ${variable-name}
- Example
  - setenv mmc-boot 'if fatload mmc 0:1 0x70000000 test.elf; then bootm 0x70000000; fi



Lab4:create two Env. (ks_boot_sd and ks_boot_tftp)And save them on the SDCARD

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Configure u-boot to save Env. Is mmc fat partition



Mark it

Set mmc device 0 partition 1

Filename will be KS_uboot.env

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

#LEARN IN DEPTH

#Be professional in embedded system

Run it ☺

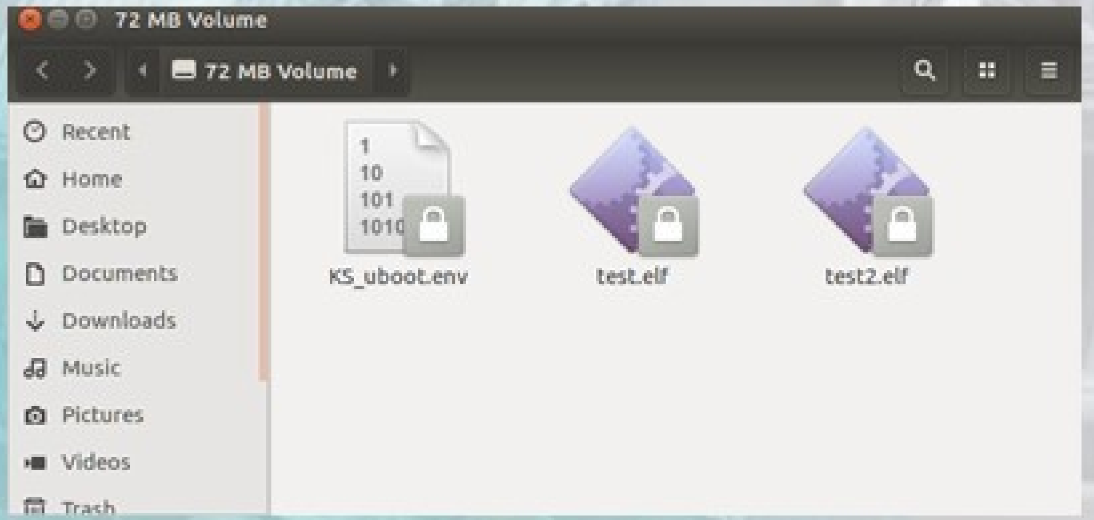
embedded_system_ks@embedded-KS: /media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS/lab3

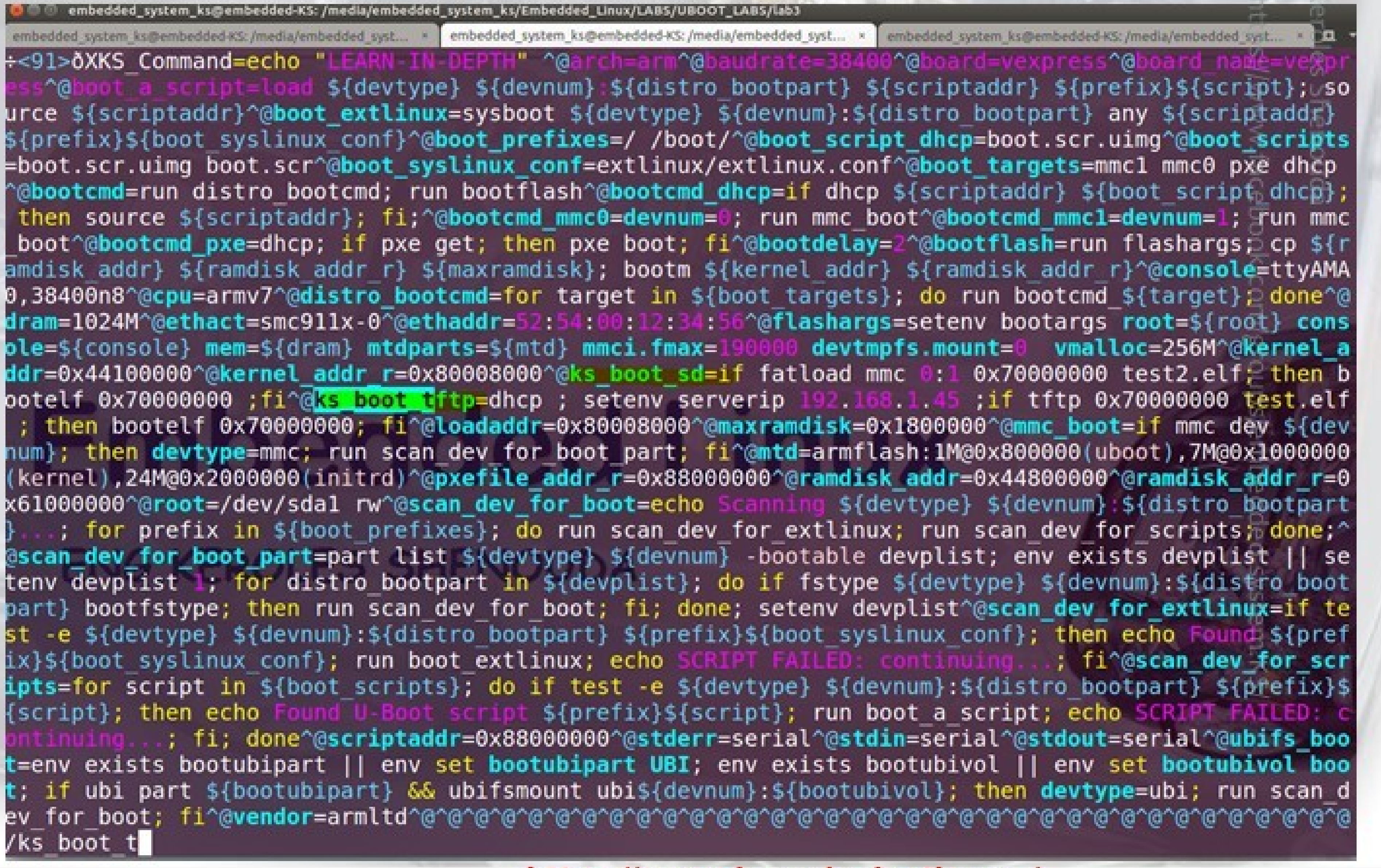
embedded_system_ks@embedded-KS: /media/embedded_syst... | embedded_system_ks@embedded-KS: /media/embedded_syst... | embedded_system_ks@embedded-KS: /media/embedded_syst...

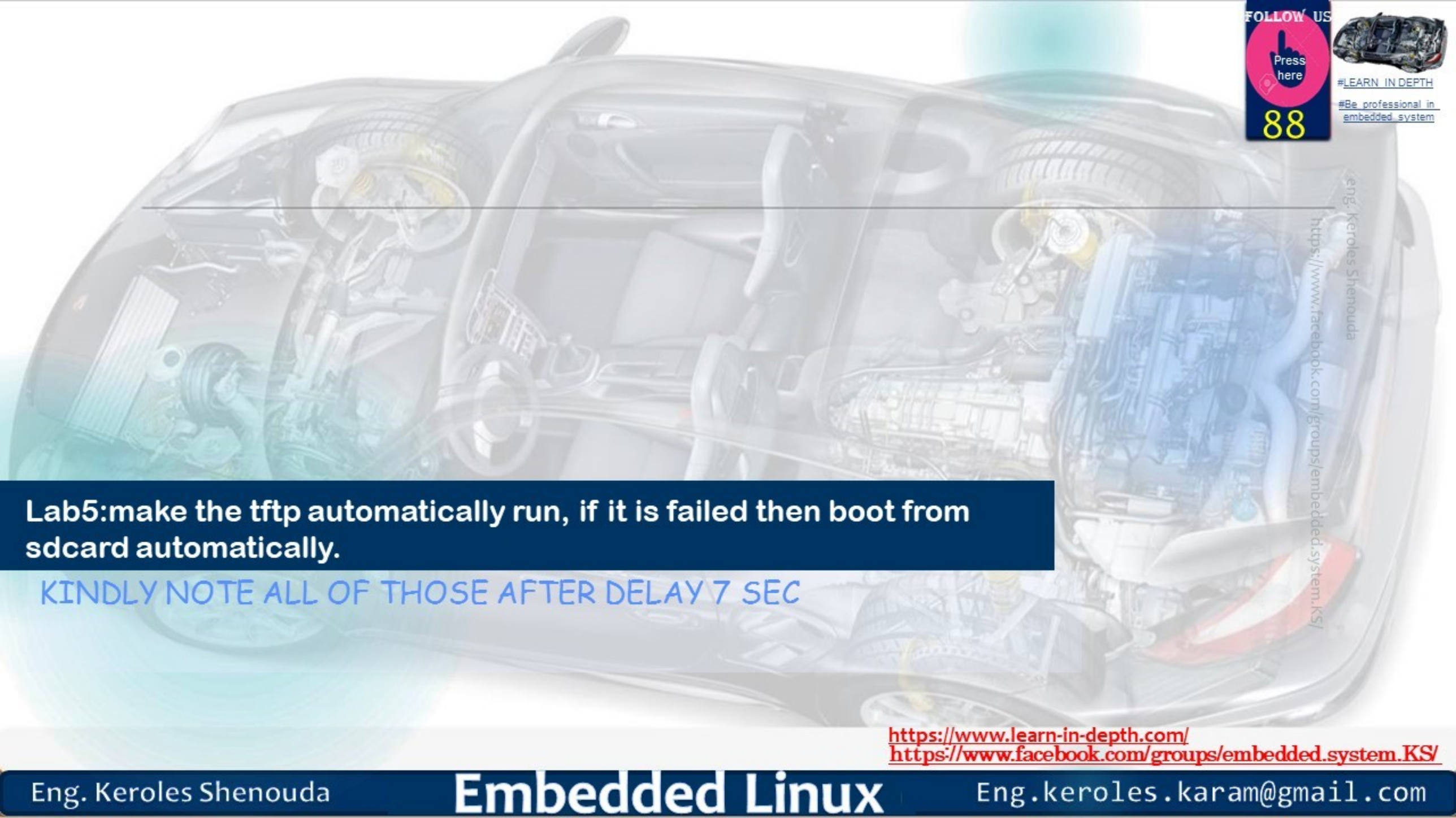
```
U-Boot 2020.01-rc2-00035-g3ff1ff3ff7-dirty (Jan 18 2020 - 02:08:31 +0200)

DRAM:  512 MiB
WARNING: Caches not enabled
Flash: 128 MiB
MMC:   MMC: 0
Loading Environment from FAT... OK
In:    serial
Out:   serial
Err:   serial
Net:   smc911x-0
Hit any key to stop autoboot:  0
=> setenv ks_boot_sd 'if fatload mmc 0:1 0x70000000 test2.elf; then bootelf 0x70000000 ;fi'
=> setenv ks_boot_tftp 'dhcp ; setenv serverip 192.168.1.45 ;if tftp 0x70000000 test.elf ; then boot
elf 0x70000000; fi'
=> saveenv
Saving Environment to FAT... OK
=> run ks_boot_sd
67896 bytes read in 84 ms (789.1 KiB/s)
## Starting application at 0x48000000 ...
Hellooooooooooooooooooooooooooo world!
```

You can now mount the SDCARD and check the ks_uboot.env

$ sudo vi mount_p1/KS_uboot.env







Lab5:make the tftp automatically run, if it is failed then boot from sdcard automatically.

KINDLY NOTE ALL OF THOSE AFTER DELAY 7 SEC

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Modeify bootcmd

```
U-Boot 2020.01-rc2-00035-g3ff1ff3ff7-dirty (Jan 18 2020 - 02:08:31 +0200)

DRAM:  512 MiB
WARNING: Caches not enabled
Flash: 128 MiB
MMC:   MMC: 0
Loading Environment from FAT... OK
In:    serial
Out:   serial
Err:   serial
Net:   smc911x-0
Hit any key to stop autoboot:  0
=> setenv bootcmd 'echo \"Learn-in-depth.com embedded linux prepared by ENg.Keroles \" ; if run ks_b
oot_tftp ; then ; else run ks_boot_sd;fi'
=> setenv bootdelay
=> setenv bootdelay 7
=> saveenv
Saving Environment to FAT... OK
=>
```


https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

#LEARN IN DEPTH

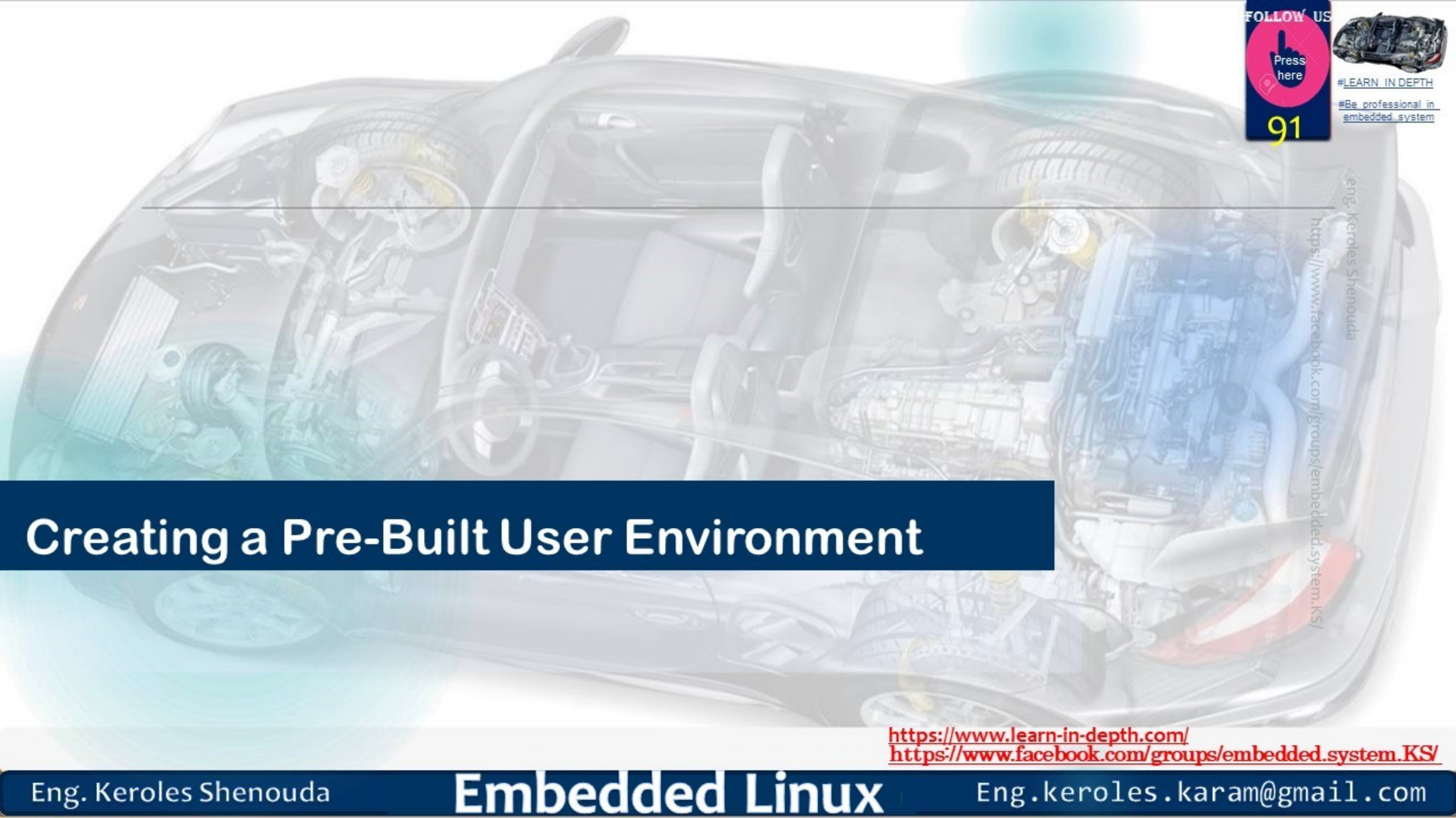
#Be professional in embedded system

Test it

```
U-Boot 2020.01-rc2-00035-g3ff1ff3ff7-dirty (Jan 18 2020 - 02:08:31 +0200)

DRAM:  512 MiB
WARNING: Caches not enabled
Flash: 128 MiB
MMC:   MMC: 0
Loading Environment from FAT... OK
In:    serial
Out:   serial
Err:   serial
Net:   smc911x-0
Hit any key to stop autoboot:  7
```

```
Hit any key to stop autoboot:  0
Learn-in-depth.com embedded linux prepared by ENg.Keroles
smc911x: MAC 52:54:00:12:34:56
smc911x: detected LAN9118 controller
smc911x: phy initialized
smc911x: MAC 52:54:00:12:34:56
BOOTP broadcast 1
DHCP client bound to address 10.0.2.15 (4 ms)
*** Warning: no boot file name; using '0A00020F.img'
Using smc911x-0 device
TFTP from server 10.0.2.2; our IP address is 10.0.2.15
Filename '0A00020F.img'.
smc911x: MAC 52:54:00:12:34:56

TFTP error: trying to overwrite reserved memory...
smc911x: MAC 52:54:00:12:34:56
smc911x: MAC 52:54:00:12:34:56
smc911x: detected LAN9118 controller
smc911x: phy initialized
smc911x: MAC 52:54:00:12:34:56
Using smc911x-0 device
TFTP from server 192.168.1.45; our IP address is 10.0.2.15; sending through gateway 10.0.2.2
Filename 'test.elf'.
Load address: 0x70000000
Loading: ###############
         121.1 KiB/s
done
Bytes transferred = 67896 (10938 hex)
smc911x: MAC 52:54:00:12:34:56
## Starting application at 0x48000000 ...
Hellooooooooooooooooooooooooooooo world!
```


https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

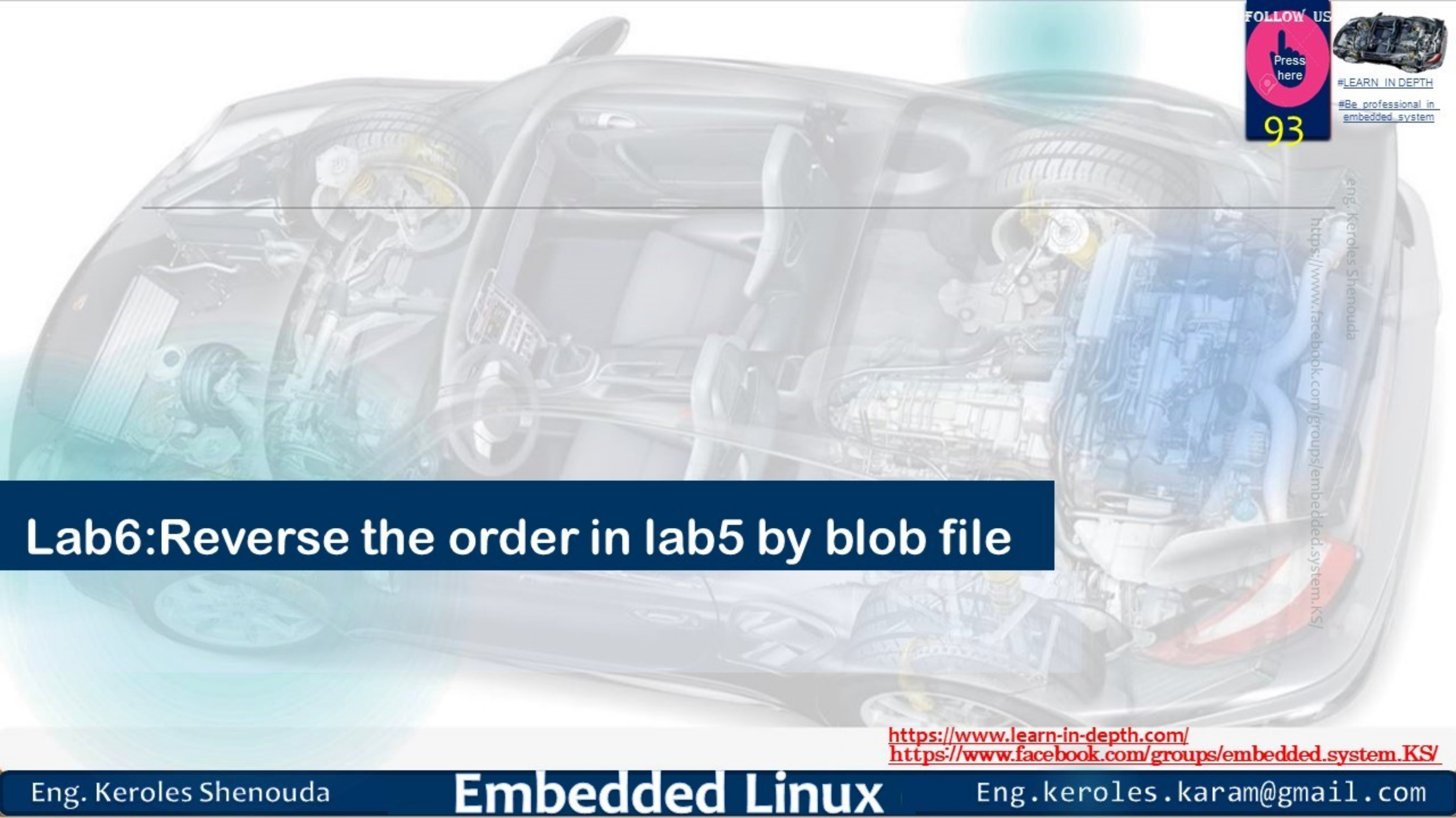


Creating a Pre-Built User Environment

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Creating a Pre-Built User Environment

- U-Boot provides a utility named **mkenvimage** which can be used to generate an environment **blob** suitable to be flashed.
- **mkenvimage** needs at least 2 inputs to create the **blob**:
  - Environment variables in a text file (only one env "name=value" string on each line)
  - Size of the environment blob in bytes (Remember, this must comply with the CONFIG_ENV_SIZE you have defined in your board config).



Lab6:Reverse the order in lab5 by blob file

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

#LEARN IN DEPTH

#Be professional in embedded system

Creating a Pre-Built env

- gedit ks_blob.txt &

ks_blob.txt (Embedded_Linux /media/embedded_system_ks/Embedded_Linux/u-boot) - gedit

Open Save

```
1 bootdelay=12
2 bootcmd='echo "Learn-in-depth.com embedded linux prepared by ENg.Keroles " ; if run ks_boot_tftp ; then ; else
  run ks_boot_sd;fi'
```

Creating a Pre-Built env Cont.

- my env data file is called **ks_blob.txt**, and sizeof desired env blob is 16384 (16 KiB), then i would use the following command:
- $ <UBOOT_PATH>/tools/mkenvimage -s 16384 -o **ks_env_blob** **ks_blob.txt**

you can see the dump of the environment blob using the od command

```
embedded_system_ks@embedded-KS: /media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS/lab3
p ../../../u-boot/ks_
ks_blob.txt  ks_env_blob
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS/lab3$ sudo c
p ../../../u-boot/ks_env_blob mount_p1/
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS/lab3$ sudo
od -t x1c   mount_p1/ks_env_blob
0000000  71  f7  ad  3f  62  6f  6f  74  64  65  6c  61  79  3d  31  32
          q 367 255   ?   b   o   o   t   d   e   l   a   y   =   1   2
0000020  00  62  6f  6f  74  63  6d  64  3d  27  65  63  68  6f  20  22
         \0   b   o   o   t   c   m   d   =   '   e   c   h   o       "
0000040  4c  65  61  72  6e  2d  69  6e  2d  64  65  70  74  68  2e  63
          L   e   a   r   n   -   i   n   -   d   e   p   t   h   .   c
0000060  6f  6d  20  65  6d  62  65  64  64  65  64  20  6c  69  6e  75
          o   m       e   m   b   e   d   d   e   d       l   i   n   u
0000100  78  20  70  72  65  70  61  72  65  64  20  62  79  20  45  4e
          x       p   r   e   p   a   r   e   d       b   y       E   N
0000120  67  2e  4b  65  72  6f  6c  65  73  20  22  20  3b  20  69  66
          g   .   K   e   r   o   l   e   s       "       ;       i   f
0000140  20  72  75  6e  20  6b  73  5f  62  6f  6f  74  5f  74  66  74
              r   u   n       k   s   _   b   o   o   t   _   t   f   t
0000160  70  20  3b  20  74  68  65  6e  20  3b  20  65  6c  73  65  20
          p       ;       t   h   e   n       ;       e   l   s   e
0000200  72  75  6e  20  6b  73  5f  62  6f  6f  74  5f  73  64  3b  66
          r   u   n       k   s   _   b   o   o   t   _   s   d   ;   f
0000220  69  27  00  00  ff  ff  ff  ff  ff  ff  ff  ff  ff  ff  ff  ff
          i   '  \0  \0 377 377 377 377 377 377 377 377 377 377 377 377
0000240  ff  ff  ff  ff  ff  ff  ff  ff  ff  ff  ff  ff  ff  ff  ff  ff
        377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
*
0040000
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/LABS/UBOOT_LABS/lab3$
```

Learn-in-depth.com

References

- Embedded Linux training
  - https://bootlin.com/training/
- Linux kernel and driver development training
- Yocto Project and OpenEmbedded development training
- Buildroot development training
- Building Embedded Linux Systems, 2nd Edition
- Mastering Embedded Linux Programming - Second Edition
- Christopher Hallinan, **Embedded Linux Primer,** Prentice Hall, 2006.
- **Silberschatz**, Galvin, and Gagne's **Operating System Concepts**, **Eighth Edition**.
- Robert love, linux kernel develoepemnt 3 rd edition 2010
- **Advanced Linux Programming By Mark Mitchell, Jeffrey Oldham, Alex Samuel**
- **Linux OS in Embedded Systems & Linux Kernel Internals(2/2)**
  - Memory Management, Paging, Virtual Memory, File system and its implementation, Secondary Storage(HDD), I/O systems

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

References Cont.

- Memory Management article
- Introduction to Operating Systems Class 9 - Memory Management
- Virtual Memory lecture for Introduction to Computer Architecture at Uppsala University.
- OS Lecture Note 13 - Address translation, Caches and TLBs
- CSE 331 Operating Systems Design lectures
- CSE 331 Virtual Memory
- CSE 332 Computer Organization and Architecture
- File Systems: Fundamentals.
- Linux System Programming
- System Calls, POSIX I/O
  CSE 333 Spring 2019, Justin Hsia

#LEARN_IN DEPTH

#Be_professional_in embedded_system

References Cont.

- File Permissions in Linux/Unix with Example
- Introduction to UNIX / Linux - 4
- 12.2 Basic I/O Concepts
- Communicating with Hardware
- I/O Systems I/O Hardware Application I/O Interface
- COMPUTER ARCHITECTURE
- OS
- Linux Operating System
- W4118 Operating Systems, Instructor: Junfeng Yang
- CSNB334 Advanced Operating Systems 4. Process & Resource Management.
- Using the POSIX API
- Linux Memory Management
- Chapter 2: Operating-System Structures
- POSIX Threads Programming
- U-Boot Environment Variables

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

References Cont.

- Using U-boot as production test strategy -- really?
- TFTP Boot using u-boot
- Loading the kernel with TFTP and U-boot
- https://blog.3mdeb.com/2013/2013-06-07-0x5-qemu-network-configuration-and-tftp-for-virtual-development-board/
- Installing and Testing TFTP Server in Ubuntu
- Pthreads: A shared memory programming model https://slideplayer.com/slide/8734550/
- Signal Handling in Linux Tushar B. Kute
- Introduction to Sockets Programming in C using TCP/IP