

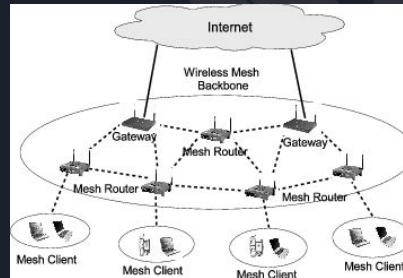
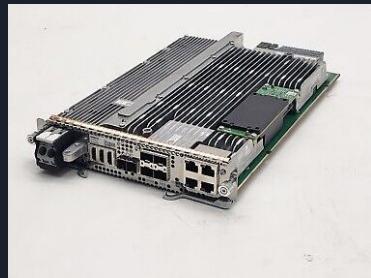


# Embedded Linux: Wprowadzenie

Karol  
Przybylski

# \$ whoami

Senior Software Engineer @ consult.red  
Tech: Embedded Linux, Bash, C, C++



linuxdev





# Uczestnicy

Czym się zajmuję?

Jaki jest dla mnie cel szkolenia?

Co aktualnie wiem o temacie szkolenia?

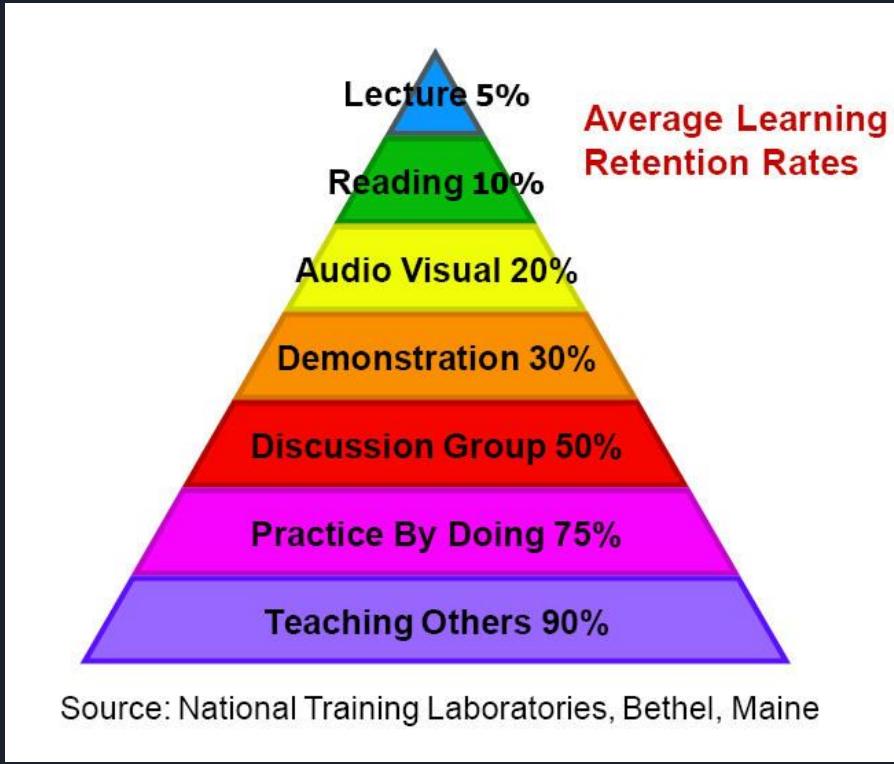
# Informacje organizacyjne

Godziny: 9-16

Przerwa 1: ~10.30

Obiad: ?

Przerwa 2: ?





# Agenda

01 Podstawy Linuxa Embedded - Toolchain, Bootloader, Kernel, Rootfs filesystem

02 Pisanie własnych recept i modyfikacja Buildroota, obsługa sprzętu

03 Optymalizacja, debugowanie, najlepsze praktyki i security



# Droga developera

**OD**

Znam podstawy Linuxa, ale nie czuję się pewnie z tworzeniem obrazów Linuxa Embedded

**DO**

Potrafię zbudować obraz dystrybucji z Buildrootem

Umiem skonfigurować 4 kluczowe elementy - toolchain, bootloader, kernel i rootfs

Wiem jak rozwijać i integrować aplikacje za pomocą Buildroota



# Na zakończenie szkolenia





# Sprawdzenie HW/SW



## Sprawdzenie środowiska - HW

- Raspberry Pi 4b
- Zasilacz
- Karta SD
- Konwerter USB-Serial
- Kabel HDMI
- Kabel ETH

Dodatkowo:

- Monitor
- LEDy
- Kableki, płytka stykowa
- BMP280



# Raspberry Pi 4b

<https://www.raspberrypi.com/products/raspberry-pi-4-model-b/>

Q: Dlaczego nie każde RPi nadaje się do zastosowań przemysłowych?



## Sprawdzenie środowiska - SW

Linux

IDE (e.g. VS Code)

Terminal (e.g. terminator)

Program do obserwowania seriala  
(picocom)

Text editor (jakikolwiek)



# Środowisko developerskie

**Problemy:**

Mnóstwo zależności

Niejednolite środowisko

Możliwości modyfikacji



Maksymalnie prosty

Dostęp do plików na hoście



## Ćwiczenie

Repozytorium:

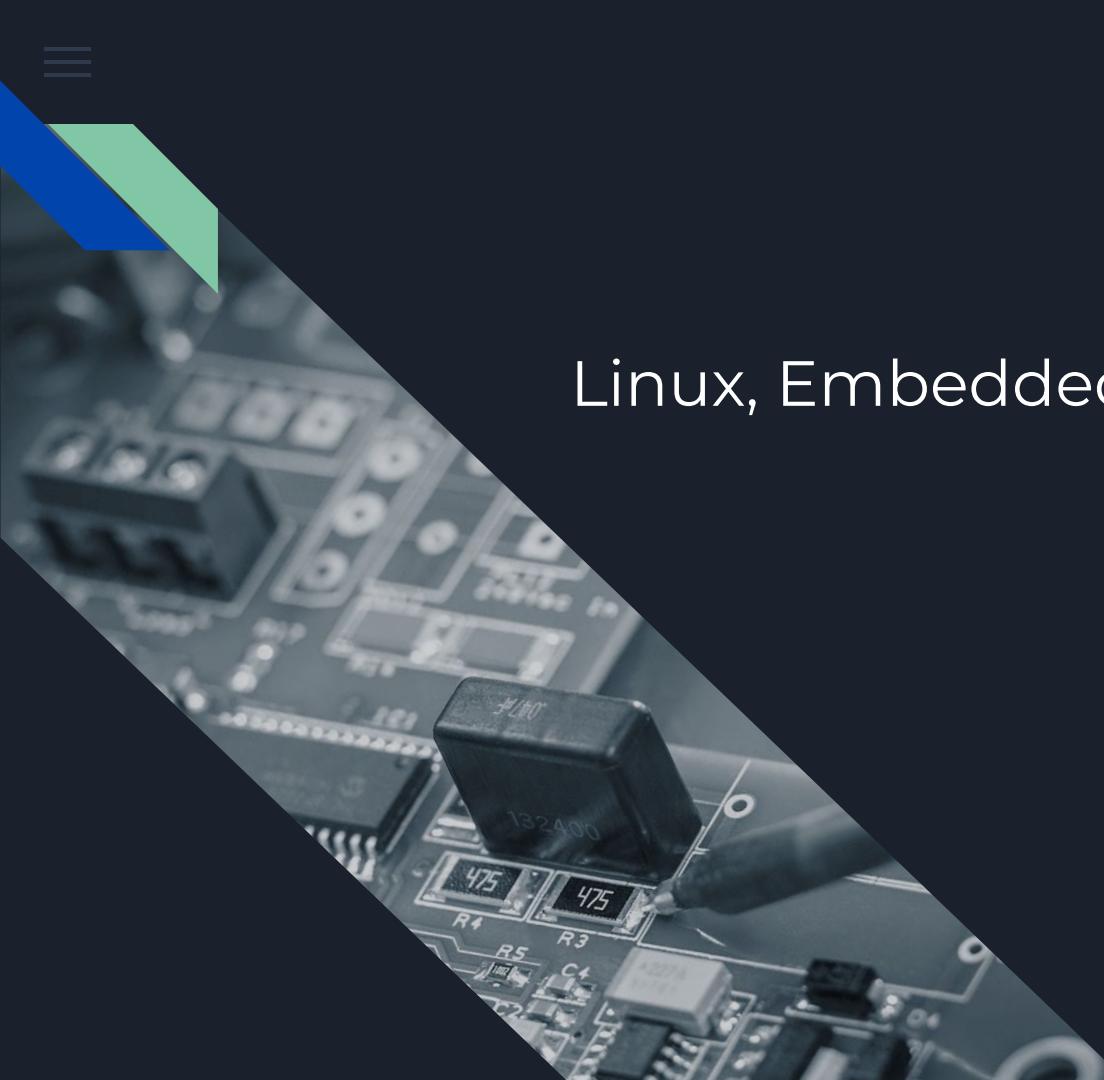
<https://github.com/KarolPWr/builder-intro/tree/master>

Branch: master

Discord:

<https://discord.gg/hY5pQK5d>

## Praca w kontenerze



☰

Linux, Embedded Linux, RTOS...



# Systematyka

“Linux” - linux kernel + distro

GNU/Linux - system operacyjny, kernel + narzędzia GNU (tar, gcc, text editor etc.)

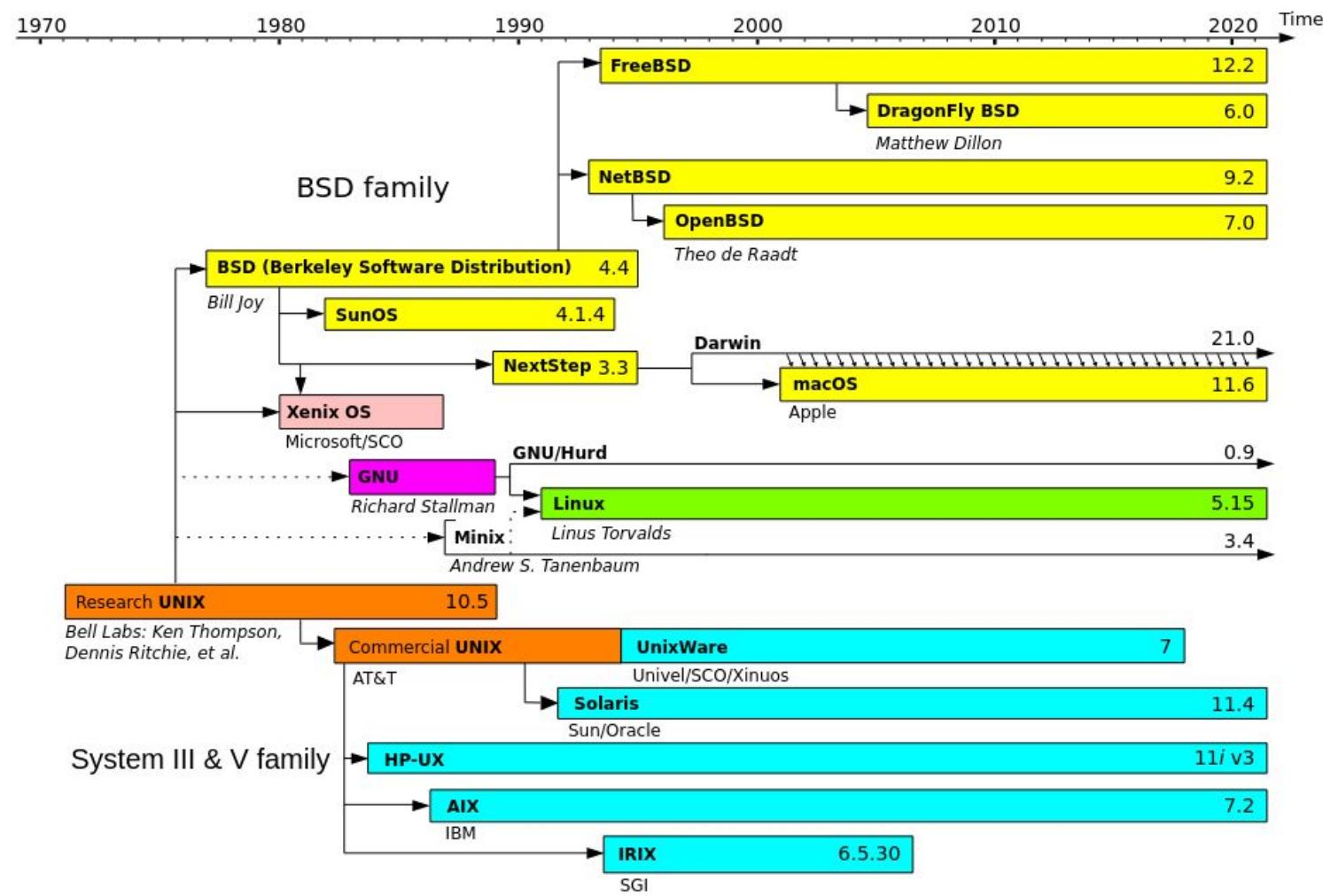
Embedded Linux - linux kernel + wyspecjalizowane distro

QNX - komercyjny, RTOS, microkernel

Wind River

Unix-like - “parasol”







# Dlaczego NIE Linux?

Linux potrzebuje sporo zasobów (m.in. 32 bitowy procesor, pamięć)

RTOS

Certyfikacja (medyczne, automotive itp.)

Diagram: Cortex-M vs Cortex-A



# Dlaczego Linux?

Wsparcie HW 'od strzała'

USB, BT, Network Stack

Wsparcie dla wielu architektur [arch](#)

Modularność

Open source, społeczność

Niezależność od vendora



# Cztery elementy Linuxa Embedded

Toolchain

Bootloader

Kernel

Root filesystem



Bogdan Botezatu  
@bbotezatu



Happy 25th birthday, #Linux! Here's your f-ing #cake, go ahead and compile it yourself.



# Build systems



yocto  
PROJECT

BuildRoot  
Making Embedded Linux Easy

OpenWrt  
WIRELESS FREEDOM



# Co trzeba zrobić?

- Konfiguracja Toolchaina
- Konfiguracja i budowanie rootfs
- Konfiguracja i komplikacja kernela
- Konfiguracja i komplikacja bootloadera
- Integracja tego wszystkiego (**najtrudniejsze!**)
- Stworzenie powtarzalnego środowiska
- I wszystko inne



## Ręczne budowanie (Roll Your Own, [LFS](#))

- Pełna kontrola
- Dopasowanie pod siebie

Ale:

- Bardzo skomplikowane, dependency hell
- Brak zewnętrznego supportu (nawet od community)
- Wymyślanie koła na nowo

# Build systemy

## OpenWRT

Mały, bazuje na  
Buildrootie - dla urządzeń  
sieciowych

## Buildroot

Dobry dla małych i  
średnich produktów

## Yocto

Behemot, dobry dla linii  
produktów i  
skomplikowanych  
systemów

	<b>Pros</b>	Highly modular, flexible, well documented	Complex for beginners	<b>Cons</b>
	<b>Pros</b>	Lightweight, easy to learn	Limited customization options	<b>Cons</b>
	<b>Pros</b>	Excellent for networking devices	Rigid policies, less customization	<b>Cons</b>

[https://pages.ubuntu.com/rs/066-EOV-335/images/Ubuntu\\_Core\\_4\\_pager\\_DS\\_revised\\_may\\_2024.pdf?version=0](https://pages.ubuntu.com/rs/066-EOV-335/images/Ubuntu_Core_4_pager_DS_revised_may_2024.pdf?version=0)

Obrazek: <https://tuxcare.com/blog/which-linux-distro-is-best-for-embedded-development/>



# Buildroot

- + Łatwiejszy niż Yocto
- + Daje spore możliwości
- + Znakomity do nauki Linuxa
- + Dokumentacja
- Dzielenie się kodem w dużych teamach
- Brak package managera (update SW)
- Mniej funkcjonalny, gorszy cache

Częsta ścieżka: początek pracy na Buildroot -> produkt/firma rośnie, dodatkowe modele, przejście na Yocto



# Buildroot z lotu ptaka

- + Konfiguracja - menuconfig
- + Szybki (\*) - dla prostych buildów
- + Napisany w make i bash
- + Generuje małe obrazy, kilka MB
- + 3000+ gotowych paczek do integracji
- + Wspiera wiele architektur
- + Wsparcie społeczności, lista mailowa



# Release management

- Stabilne wersje co 3 miesiące:
  - YYYY.02, YYYY.05, YYYY.08, YYYY.11
- Dościągnięcia jako .tar i z repozytorium githuba
- Wydanie LTS co roku

Dokumentacja: <https://buildroot.org/docs.html>

Oficjalna strona: <https://buildroot.org/>

Git: <https://gitlab.com/buildroot.org/buildroot>

Lista mailowa: <https://lore.kernel.org/buildroot/>



Ćwiczenie

2\_buildroot\_podstawy

# Pierwszy krok



Cztery elementy Linuxa Embedded

Toolchain

Bootloader

Kernel

Root filesystem



# Toolchain



Budowanie systemu z użyciem crosstool-NG



# Proces kompilacji

Preprocesor -> Kompilator -> Assembler -> Linker = plik wykonywalny



# Toolchain

Zawiera:

- Binutils (assembler, linker etc.)
- GCC (GNU Compiler Collection)
- C library (glibc, musl, ulibc)
- Kernel headers  
(`/usr/src/linux-headers-6.8.0-94/include/`)
- GDB



# Co mamy do wyboru

Rozwiązania od vendora, np. Broadcom

Generowany przez build system np. Yocto, Buildroot

Paczka z dystrybucji desktop Linuxa (distribution toolchain)

Rozwiązania open source np. crosstool-ng



Czy jest skonfigurowany tak jak chcemy?

Czy będzie aktualizowany?

Czy posiada zadowalające wsparcie?

Ile potrzebujemy kontroli?

Crosstool-NG: <https://github.com/crosstool-ng/crosstool-ng>



# Co robi C library?

Implementacja standardowych funkcji języka (np. Printf z stdio.h)

Razem z kernel headers daje nam dostęp do funkcji kernela (open, write etc.)

Każdy nasz program będzie z nią linkowany

# Wybór C library

uClibc-ng

Mały rozmiar - gdy mamy  
mało zasobów

glibc

“Stare, dobre”  
Spore community i  
wsparcie  
bloat

musl

“Lepsze” glibc  
Mniejszy bloat  
Dziwne błędy,  
kompatybilność z  
programami



# Toolchain w Buildroocie

Mamy dwie opcje do wyboru:

- Internal toolchain
- External toolchain

# Internal vs external

## Internal

- +Dobrze zintegrowany
- +Buduje tylko co trzeba
- Kosztowny rebuild

## External

- +Więcej kontroli
- +Szybsze rebuildy
- Nasza odpowiedzialność



# crosstool-ng

<https://github.com/crosstool-ng/crosstool-ng?tab=readme-ov-file>

- + Jak poznać czy projekt ma perspektywy (commity, dokumentacja, społeczność)

Introduction:

<https://crosstool-ng.github.io/docs/introduction/>



## Ćwiczenie

Konfiguracja toolchaina:

[3\\_toolchain.md](#)



# Optymalizacja



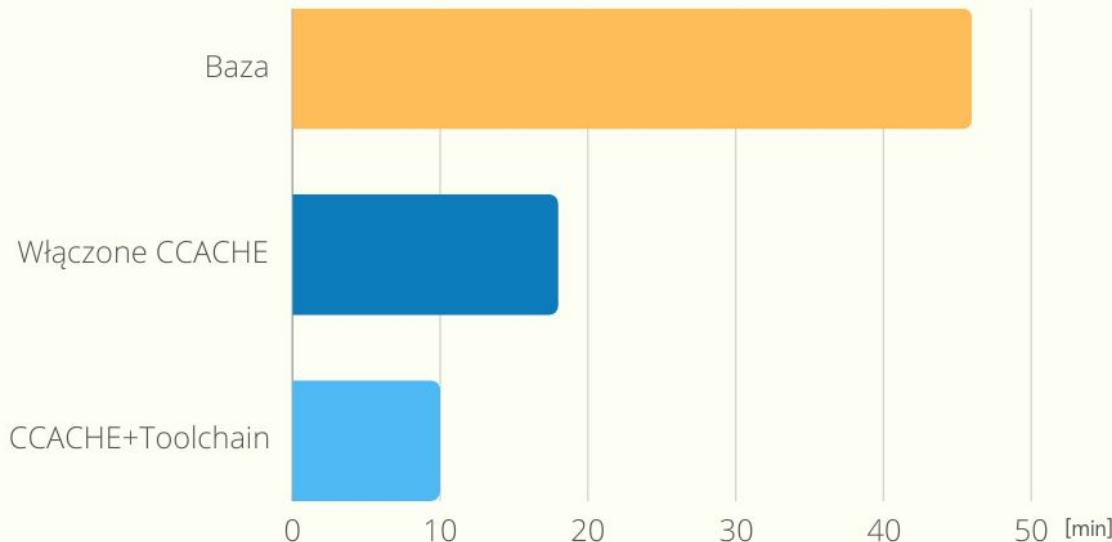


# Jak znacznie zmniejszyć build time?

- Pre-built toolchain
- CCACHE
- Wiedza kiedy trzeba przebudować projekt
- Szybki dysk i dużo RAM
- **Nie korzystanie z NFS**

Lista kiedy należy budować: <https://buildroot.org/downloads/manual/manual.html#full-rebuild>

# Optymalizacja budowania: 46 → 9 minut





## Ćwiczenie

Optymalizacja i pierwszy build

Build - ok 40 min

# Instalacja połączenia szeregowego

Cel: umożliwienie komunikacji z płytą

Po stronie hosta:

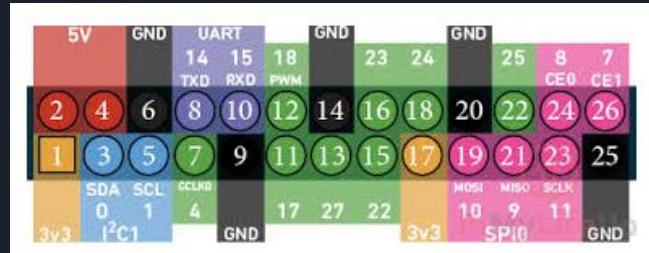
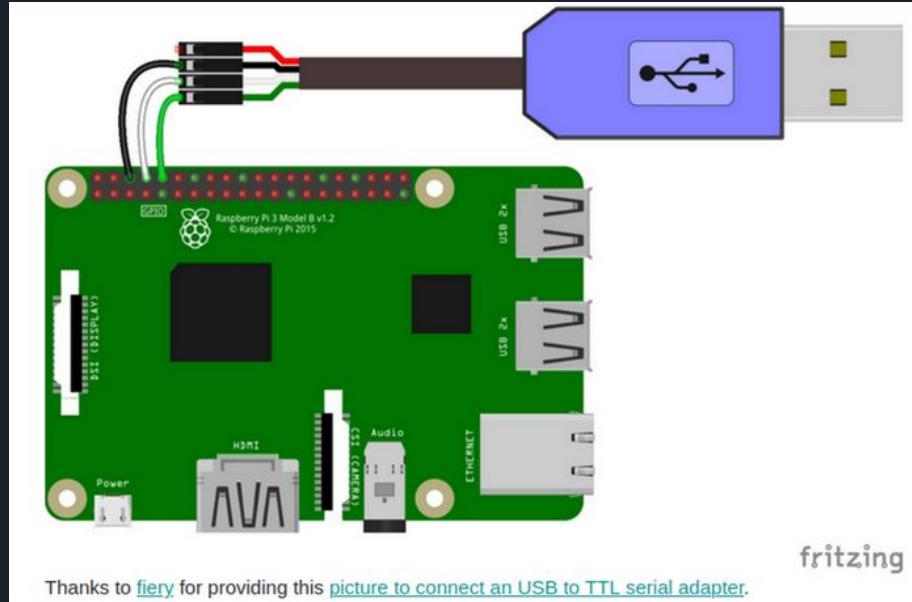
Wpiąć konwerter do gniazda USB

\$ dmesg | tail

\$ picocom -b 115200 /dev/ttyUSBX

UWAGA1: Zworka żółta na 3.3V

UWAGA2: Tx na zmianę z Rx (przemiennie)





# Crosstool - podsumowanie

\$ ct-ng - frontend do toolchaina

\$ ct-ng menuconfig

\$ ct-ng show-config

\$ folder 'x-tools' - lokalizacja wyprodukowanych binarek

Toolchain możemy również używać poza buildrootem.



# Boot process

Zasilanie ->

*Magic!*

-> System



# Kroki

- 1 - ROM code
- 2 - SPL - Secondary Program Loader
- 3 - Główny bootloader (np. u-boot) - ładujemy kernel z devicetree
- 4 - Kernel
- 5 - Initramfs (opcjonalnie)
- 6 - Init manager (np. systemd)



# ROM code

# ROM i początkowy bootloader

## Zasoby

Jeden rdzeń CPU  
On chip memory

## Misja

Zainicjalizować sprzęt  
Załadować SPL do pamięci

## Misje poboczne

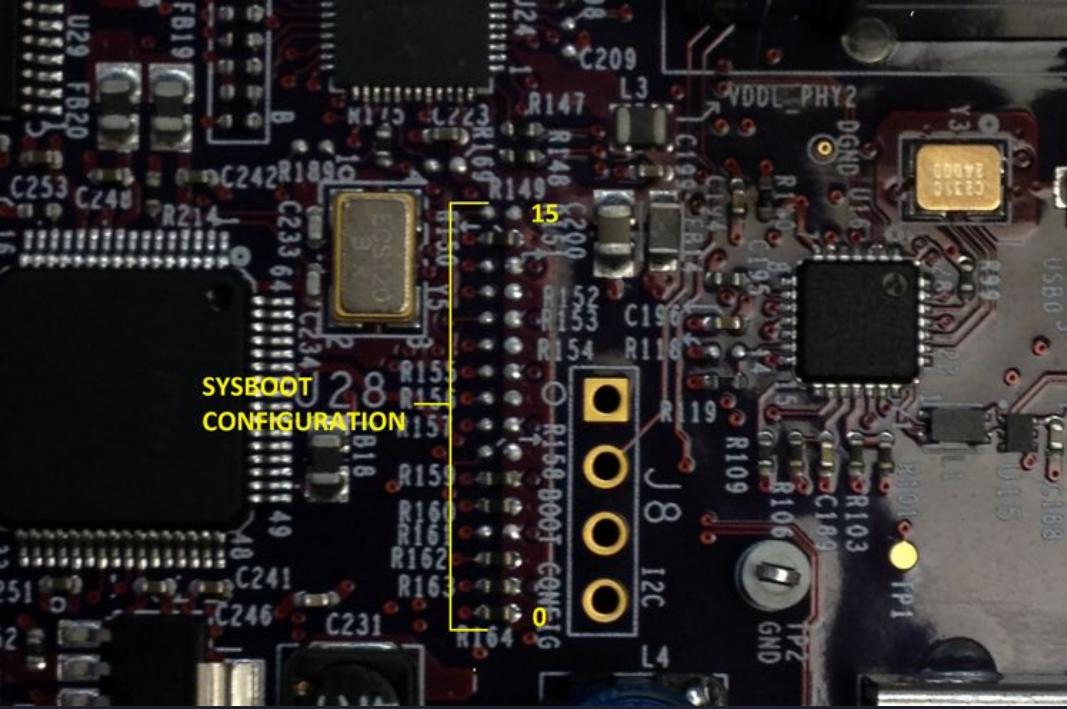
PLL  
System Clock  
Watchdog



# Praktyka

<https://www.ti.com/lit/ug/spruh73q/spruh73q.pdf?ts=1696785774495>

26.1.6 Booting



# Bootloader: Das U-boot





# Bootloadery - SPL/TPL

Dostępne rozwiązania:

U-boot

Custom (od vendora)

Barebox - <https://github.com/barebox/barebox>

Sporo innych, ale najczęściej dwa pierwsze

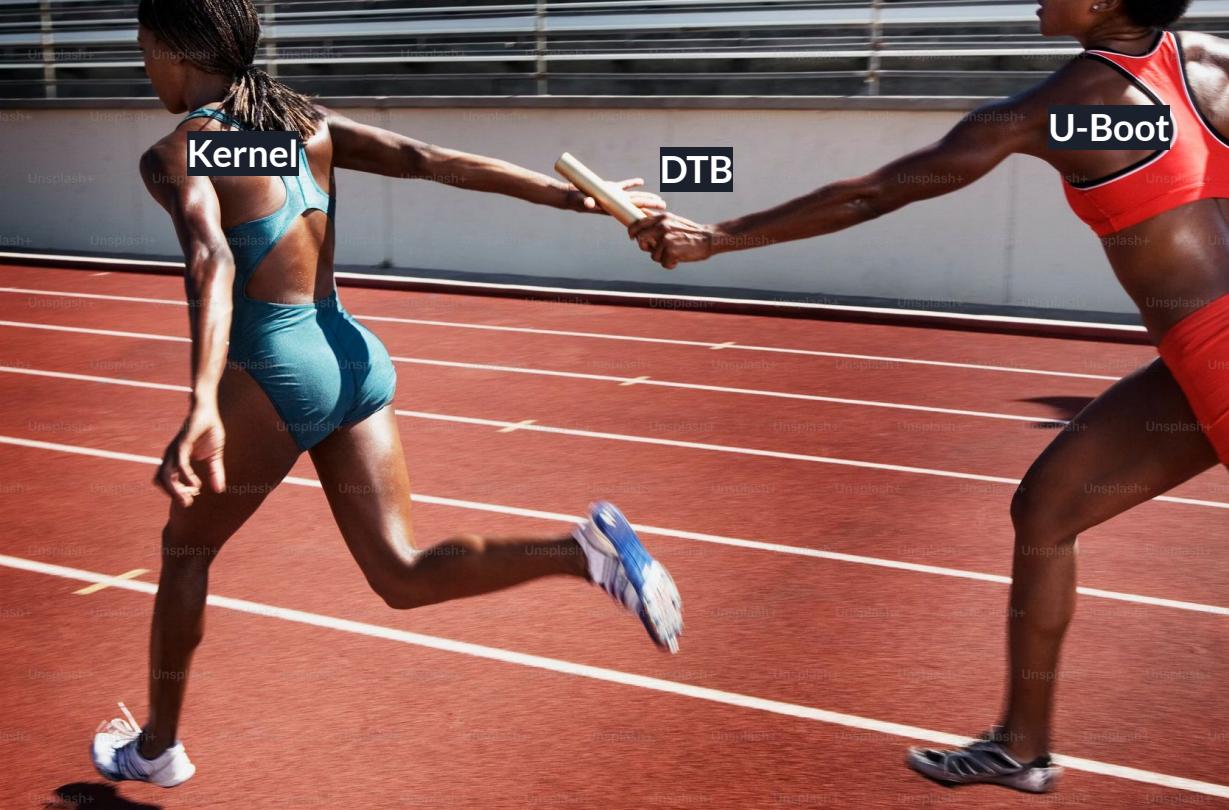


# Wybieramy U-boot'a



"Though there are quite a few other bootloaders, 'Das U-Boot', the universal bootloader, is arguably the **richest, most flexible, and most actively developed** open source bootloader available."

*Building Embedded Linux Systems*, by Karim Yaghmour



**Kernel**

**DTB**

**U-Boot**



# U-boot ciąg dalszy

Pełny program (mini-OS)

Udostępnia podstawowe możliwości:

Ustawianie zmiennych

Połączenie z siecią

Dostęp do pamięci



# Zawartość u-boota

Źródła: buildroot/output/build/u-boot-\*

Ciekawe rzeczy -

<https://github.com/u-boot/u-boot/blob/master/arch/arm/cpu/armv8/cpu.c#L37>

Plik wynikowy - u-boot.bin

Toole pomocnicze -

<https://github.com/u-boot/u-boot/tree/master/tools>



Firmware vendora

Dodatkowy krok inicjalizacji

Zazwyczaj closed source

Dla Raspberry Pi:

Start4.elf

output/images/rpi-firmware



## Ćwiczenie

Konfiguracja u-boota

4\_bootloader.md



# Możliwości u-boota

Pełne zarządzanie bootem - ładowanie kernela, devicetree, ramdysku z różnych źródeł (SD, emmc, USB, NAND, SPI flash)

Uruchamianie różnych obrazów kernela - booti, bootz, bootm

Network boot - TFTP, PXE,

Debugowanie i diagnostyka - printenv, dumpowanie rejestrów HW

Testowanie peryferiów (SPI, usb, UART)

Secure boot

# Diagnostyka

Odczyt rejestrów HW: <https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf>

```
U-Boot> md 0xFE200000 10
```

## Manipulacja GPIO:

```
U-Boot> gpio set 26
```

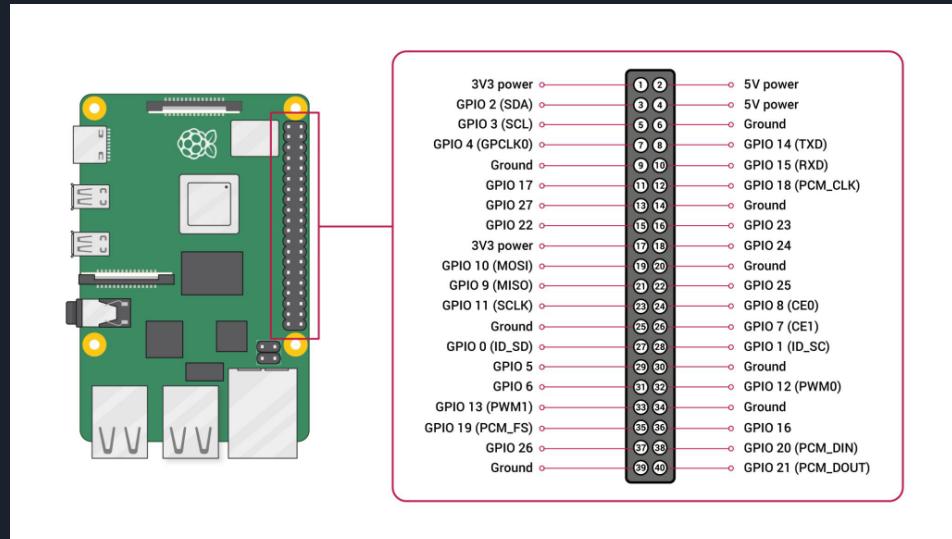
```
U-Boot> gpio clear 26
```

## Wysyłanie znaków przez UART:

```
U-Boot> mw 0xFE201000 0x41 1
```

## Listowanie network devices:

```
U-Boot> net list
```





# Możliwości u-boota

\$ printenv

\$ mmc rescan

\$ setenv ipaddr 192.168.159.42

Definicja nowych zmiennych i komend



# Własne zmiany w buildroot - automatyzacja

3 sposoby na modyfikacje obrazu:

- Zmiana konfiguracji ( pliki **\*config**)
- Skrypty **post-\*** (post-build.sh, post-image.sh)
- Rootfs **overlay** (o nich później)



# Skrypty post-\* (w boards/)

post-build.sh - “run before building the filesystem image”

post-image.sh - “specific actions after all images have been created”

Np. transport rootfs na serwer tftpboot, lista plików na kartę SD

Można dodawać własne skrypty post-image (w menu System configuration)

Przykłady - inne definicje płytek



# Jak trackować swoje zmiany?

Dwa podejścia:

- Osobny branch od buildroota, nasze zmiany trzymamy w głównym drzewie
- Osobne repozytorium, trzymamy wszystko w osobnym folderze (tzw. BR2\_EXTERNAL)

Zalety i wady?



## Ćwiczenie

Automatyzacja kompilacji u-boota

[4a\\_bootloader\\_automat.md](#)