



Embedded Linux: Wprowadzenie

Karol
Przybylski

NobleProg



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



Na zakończenie szkolenia





Sprawdzenie HW/SW





Sprawdzenie środowiska - HW

- Raspberry cm4
- Cm4 devboard
- Zasilacz
- Kabel micro usb
- Konwerter USB-Serial
- Kabel HDMI
- Kabel ETH

Dodatkowo:

- Monitor
- LEDy
- Kableki, płytka stykowa
- BMP280



Raspberry Pi CM4

<https://www.raspberrypi.com/products/compute-module-4/?variant=raspberry-pi-cm4001000&resellerType=education>

Q: Dlaczego CM4 nadaje się do zastosowań przemysłowych (a zwykłe rpi tak sobie)?



Sprawdzenie środowiska - SW

Linux

IDE (e.g. VS Code)

Terminal (e.g. terminator)

Program do obserwowania seriala
(picocom)

Text editor (jakikolwiek)



Uniwersalne środowisko

Problemy:

Mnóstwo zależności

Niejednolite środowisko

Możliwości modyfikacji



Maksymalnie prosty

Dostęp do plików na hoście



Ćwiczenie

Repozytorium:

<https://github.com/KarolPWr/builder-intro/tree/cm4>

Branch: cm4

Praca w kontenerze



Linux, Embedded Linux, RTOS...





Systematyka

“Linux” - linux kernel + distro

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

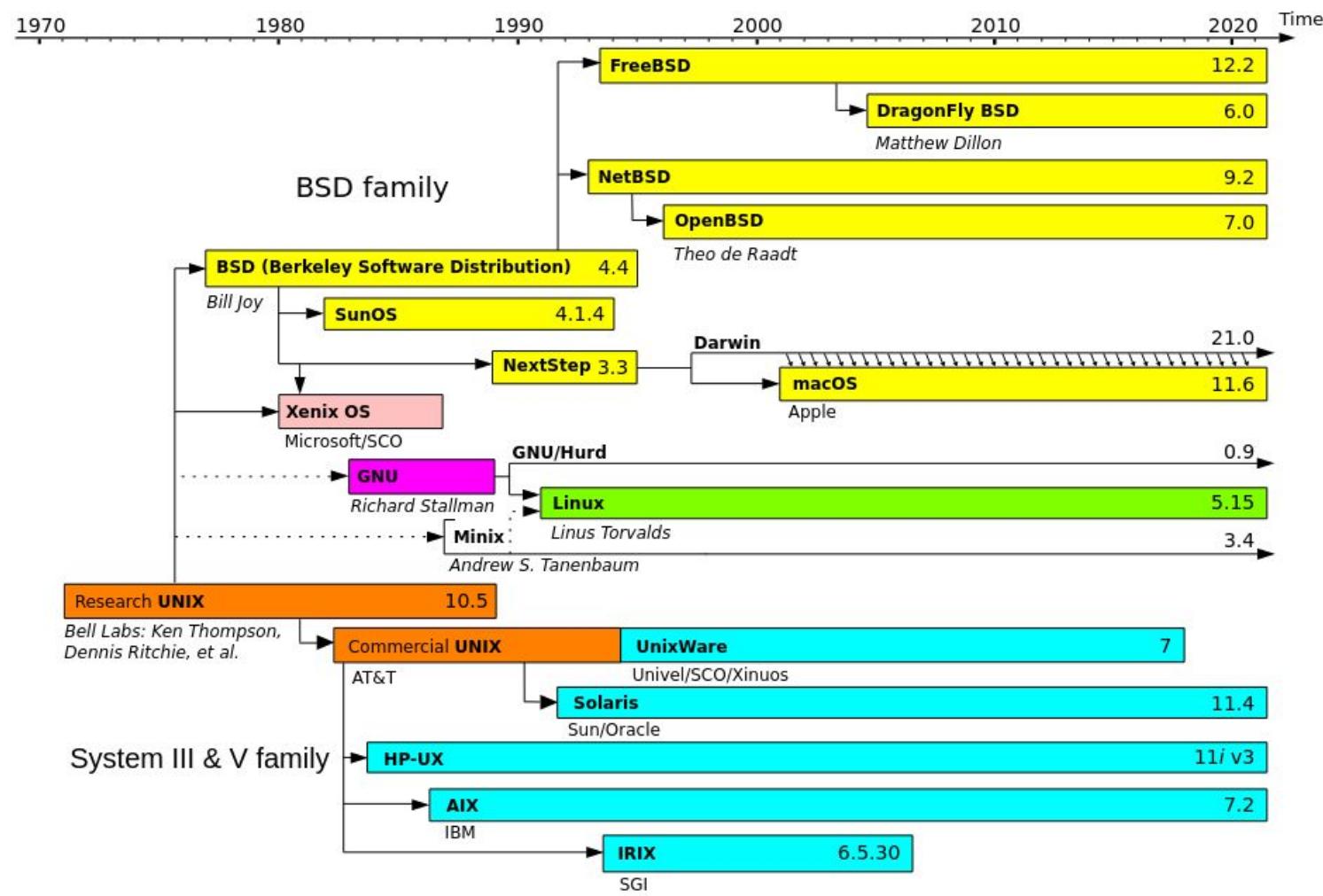
Embedded Linux - linux kernel + wyspecjalizowane distro

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, wojskowe)

Diagram: MMU



Dlaczego Linux?

Wsparcie HW 'od strzała'

USB, BT, Network Stack

Wsparcie dla wielu architektur [arch](#)

Modularność

Open source, społeczność [przykład](#)

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





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

	Pros	Highly modular, flexible, well documented	Cons
	Pros	Lightweight, easy to learn	Cons
	Pros	Excellent for networking devices	Cons

https://pages.ubuntu.com/rs/066-EOV-335/images/Ubuntu_Core_4_pager_DS_revised_may_2024.pdf?version=0

<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 (i wspierana przez rok)

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

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

Github: <https://github.com/buildroot/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
- 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



Optymalizacja



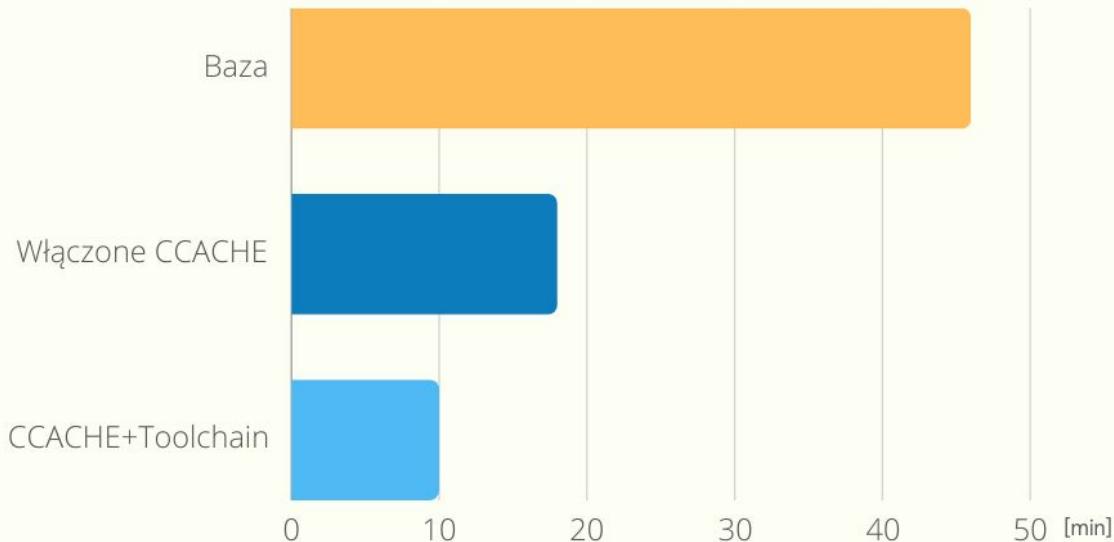


Jak znacznie zmniejszyć build time?

- Pre-built toolchain
- CCACHE
- Wiedza kiedy trzeba przebudować projekt
- Szybki dysk i nie korzystanie z NFS

Lista kiedy należy budować

Optymalizacja budowania: 46 → 9 minut





Ćwiczenie

Optymalizacji i pierwszy build

Build - ok 40 min

Instalacja połączenia szeregowego

Cel: umożliwienie komunikacji z płytą

Po stronie hosta:

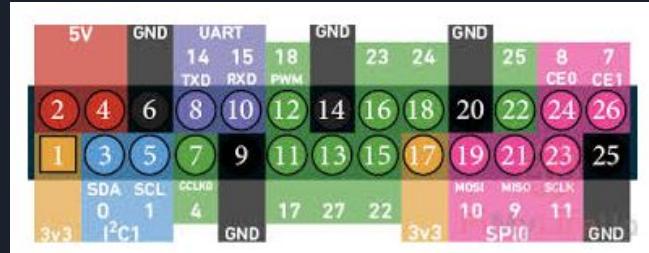
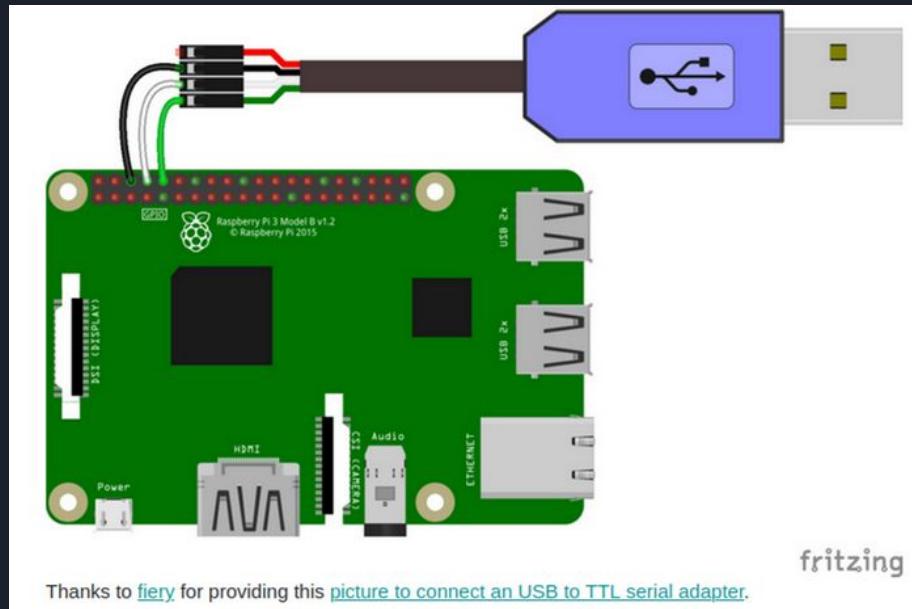
Wpisać konwerter do gniazda USB

\$ dmesg | tail

\$ picocom -b 115200 /dev/ttyUSBX

UWAGA1: Zworka żółta na 3.3V

UWAGA2: Tx na zmianę z Rx (przemiennie)



<https://www.raspberrypi.com/documentation/computers/raspberry-pi.html>



Crosstool - podsumowanie

\$ ct-ng - frontend do toolchainia

\$ 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 pomocne

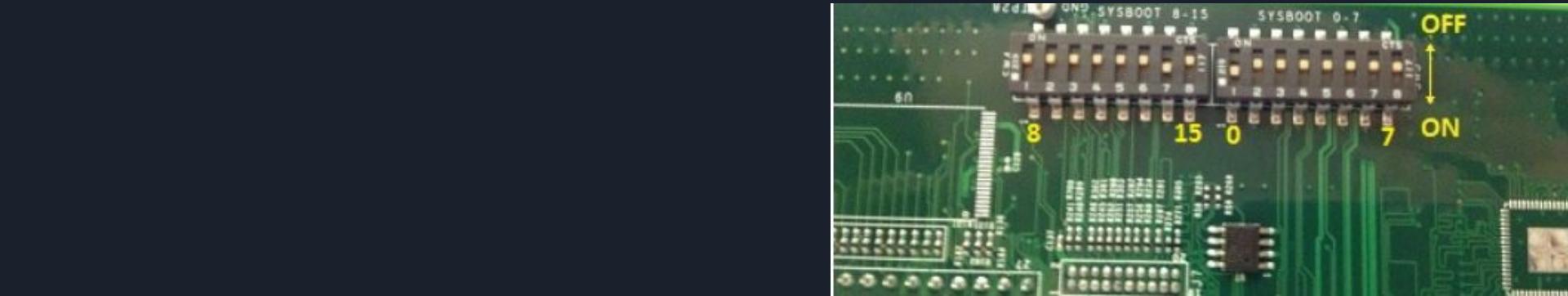
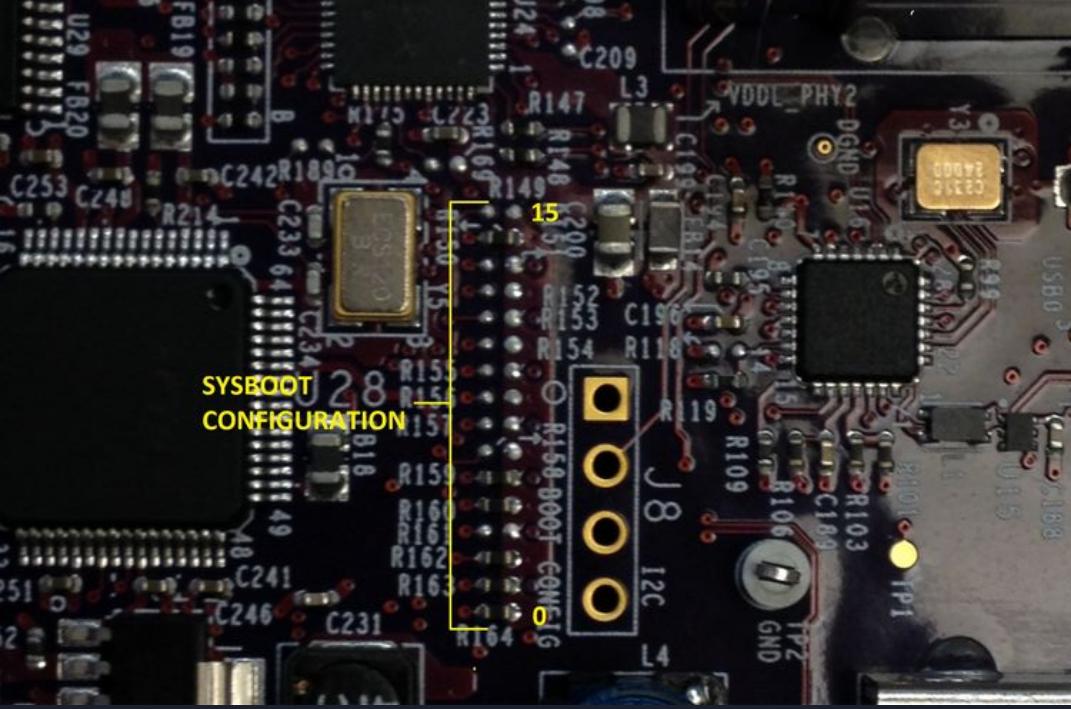
PLL
System Clock
Watchdog



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





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-2024.10

Ciekawe rzeczy - arch/arm/cpu/armv8/cpu.c

Plik wynikowy - u-boot.bin

Tools pomocnicze



Firmware vendora

Dodatkowy krok inicializacji

Zazwyczaj closed source

Dla Raspberry Pi:

Start4.elf

output/images/rpi-firmware



Ćwiczenie

Konfiguracja u-boota



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 23
```

```
U-Boot> gpio clear 23
```



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

A close-up photograph of a pile of yellow and white popcorn kernels. The kernels are scattered across the frame, with some showing their white, papery husks and others being the smooth, yellowish-orange kernels themselves.

Kernel

1. a softer, usually edible part of a nut, seed, or fruit stone contained within its shell.
"the kernel of a walnut"
2. the central or most important part of something.
"this is the kernel of the argument"



Q: Co robi kernel?





Główne zadania kernela

Przy bootie:

- Inicjalizacja reszty HW na podstawie DT (devicetree)
- Inicjalizacja systemu
- Przekazanie pałeczki init managerowi, który robi resztę

W czasie pracy:

- Memory/resource management
- Scheduling
- HW management



Kernel przy boocie

Co jest przekazywane kernelowi?

- Podstawowe informacje o HW (n.p. Clock speed)
- Kernel command line (cmdline)

<https://www.kernel.org/doc/html/v4.14/admin-guide/kernel-parameters.html>

cmdline: console=serial0,115200 console=tty1 root=/dev/mmcblk0p2 rootfstype=ext4

Co robi kernel przy inicio?

Ustawia wektory przerwań, sygnały, arch itp.

Na końcu, szuka programu ‘init’

```
kernel: Booting Linux on physical CPU 0x0
```



Kernel 2

Główny kod inicjalizacji:

<https://github.com/torvalds/linux/blob/master/init/main.c#L896> - start_kernel

Związany z architekturą:

<https://github.com/torvalds/linux/blob/3006adf3be79cde4d14b1800b963b82b6e5572e0/arch/arm64/kernel/setup.c#L293>



Kernel w Buildroocie

Kilka możliwości konfiguracji:

- Latest
- Latest CIP SLTS (co to jest? `?` w menuconfig,
<https://www.cip-project.org/about/linux-kernel-core-packages>)
- Latest CIP RT SLTS
- **Custom version**
- Custom tarball
- Custom git/mercurial/subversion repo

Zalecana wersja: custom, by nie wiązać wersji Buildroota z wersją kernela (upgrade Buildroota nie równa się upgradowi kernela)

Konfiguracja

Kernel używa (jak BR) kconfig do konfiguracji

```
menu "Kernel"

config BR2_LINUX_KERNEL
    bool "Linux Kernel"
    select BR2_PACKAGE_HOST_KMOD # Unconditional, even if modules not enabled
    select BR2_PACKAGE_HOST_IMAGEMAGICK if BR2_LINUX_KERNEL_CUSTOM_LOGO_PATH != ""
    help
        Enable this option if you want to build a Linux kernel for
        your embedded device
```

BR nie potrafi skopiować wszystkich opcji do swojego menuconfiga -> inny sposób

Podobnie to działa w przypadku innych paczek używających kconfig (busybox, uboot etc)



Konfiguracja c.d.

3 możliwości konfiguracji:

- Gotowy defconfig (np. Dla raspberry pi)
- Customowy config (np. Na podstawie defconfiga)
- Defconfig dla danej architektury (np. arm64)

Powyższe konfiguracje można nadpisywać tzw. "Fragmentami"

Przykład 1: buildroot/configs/bananapi_m2_berry_defconfig (custom version)

Przykład 2 (custom config): chromebook_elm_defconfig (linux.config)

3: A jak to robi RPi? (custom tarball) <https://github.com/raspberrypi/linux>



Konfiguracja c.d.

Menu: make linux-menuconfig

Załaduje zdefiniowany defconfig albo nasz customowy plik

Zmiany widoczne tylko w buildroot/build/linux-version (make clean je usuwa)

Aby zapisać:

```
$ make linux-update-config (cały .config)
```

```
$ make linux-update-defconfig (tylko minimalny defconfig)
```

Działa tylko gdy używamy opcji “custom config file”



Ćwiczenie

Konfiguracja kernela

Rootfs





Co powinno być w środku?

- Init
- Shell
- Demony
- Shared libraries
- Pliki konfiguracyjne
- Device nodes
- proc/ i sys/



Foldery

https://refspecs.linuxfoundation.org/FHS_3.0/fhs-3.0.pdf

Można jak się 'podoba' - ale lepiej nie...

- /bin
- /dev
- /etc
- /home
- /lib
- /proc
- /sbin
- /sys
- /tmp
- /usr
- /var



Rootfs w buildroocie

Diagram kroków



Skeleton

Skeleton - paczka zawierająca szablon do standardowego rootfs'a

Jeśli nie chcemy standardowego, tworzymy nowy jako opcja
skeleton-custom

Domyślny skeleton jest dobry dla większości zastosowań



buildroot/system/skeleton

Baza dla obrazu

Demo



Permission table

system/device_table.txt

Domyślnie wszystkie pliki są własnością użytkownika root, a uprawnienia, z którymi zostały zainstalowane w \$(TARGET_DIR) są zachowane.

Aby dostosować własność lub uprawnienia zainstalowanych plików, można utworzyć jedną lub kilka tabel uprawnień BR2_ROOTFS_DEVICE_TABLE zawierających oddzieloną spacjami listę plików tabeli uprawnień plików.

Domyślnie używany jest plik system/device_table.txt

Czym są minor i major numbers?

system/device_table_dev.txt - stary sposób, gdy węzły sprzętowe są tworzone statycznie

Zarządzanie użytkownikami: users_table

BR2_ROOTFS_USERS_TABLES - lista user table definiująca użytkowników

Składnia:

username	uid	group	gid	password	home	shell	groups	comment
----------	-----	-------	-----	----------	------	-------	--------	---------

Przykład:

foo -1 bar -1 !=blabla /home/foo /bin/sh alpha,bravo Foo user

Źródło: <https://buildroot.org/downloads/manual/manual.html#makeuser-syntax>



Instalacja paczek

Wszystkie skonfigurowane packages są komplikowane i kopiowane do szkieletu



Cleanup

Czyścimy workspace z niepotrzebnych rzeczy żeby zmniejszyć rozmiar rootfsa

- Headery
- Pkg-config
- Man pages, documentation
- Czyszczenie programów i bibliotek przez wywołanie \$ strip

Dla konkretnych paczek mogą być dodatkowe kroki - np. Usuwanie niepotrzebnych plików pythona



Rootfs overlay

Nakładka na gotowy rootfs ("patch")

W łatwy sposób możemy dodawać własne pliki i foldery

```
karol.przybylski:~/prv/buildroot_rpi/buildroot$ grep ^BR2_ROOTFS_OVERLAY .config
```

```
BR2_ROOTFS_OVERLAY="board/raspberrypi4/rootfs_overlay"
```



Init systems

Program, który przejmuje kontrolę po inicjalizacji kernela. Zarządza userspace

Przykłady:

- Busybox (bardzo prosty)
- SysV
- systemd



Ćwiczenie

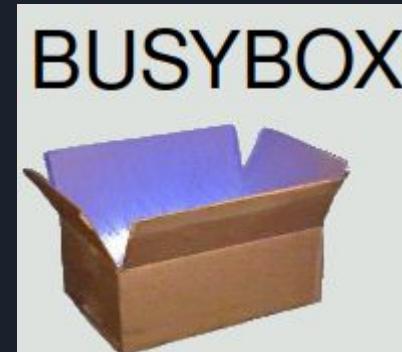
Rootfs

Skąd wziąć wszystkie programy?

Opcja 1 - wszystko robimy sami

Shell: bash, ash, hush

Programy użytkowe: grep, vi etc.... Łącznie ok 50





Jak działa busybox

Nie wszystkie funkcje są wspierane

Np. \$ grep nie posiada każdego przełącznika

Wszystkie narzędzia są jedną binarką

```
$ busybox cat file.txt
```

```
[applet]_main
```

```
› ls -la cat
```

```
lrwxrwxrwx 1 root root 7 wrz 27 22:45 cat -> busybox
```





Boot process - podsumowanie





ROM code

Firmware (SPL - Secondary Program Loader)

Bootloader (U-boot)

Kernel

Init (busybox/rootfs)



Devicetree





Devicetree

Q: Jaki mamy hardware?

Devicetree - struktura opisująca podłączony sprzęt

DTB - *device tree blob*

DTS - *device tree structure*

DTBO - *device tree overlay*

DTC - *device tree compiler*



Demo

<https://github.com/torvalds/linux/blob/3006adf3be79cde4d14b1800b963b82b6e5572e0/arch/arm64/kernel/setup.c#L184> - kernel

Struktura devicetree

<https://github.com/raspberrypi/linux/blob/rpi-6.6.y/arch/arm/boot/dts/broadcom/bcm2711-rpi-cm4.dts>

Kompilacja



Devicetree overlays

Jedno raspberry wspiera mnóstwo HW a ma ograniczone interfejsy

Różne DT na każdą okazję - chaos

Rozwiązanie - devicetree overlay, "nakłada się" na główne DT

Jedno DT + mnóstwo dtbo = wsparcie dla większości HW

Dodatkowo: firmware RPI patchuje samodzielnie devicetree, wiec w zasadzie tylko overlaye zostają



Ćwiczenie

Modyfikowanie i pisanie devicetree



Własne paczki





Kompilacja aplikacji

Cross-compilation - kompilacja na inny system niż system hosta (laptop: x86, target: ARM64)

Źródła aplikacji -> zewnętrzne repozytorium

Workflow in-dev:

Kompilacja -> Deploy -> Test

Gdy działa to dodajemy do Build systemu i integrujemy z całą resztą (automatyzacja)

Workflow in-prod:

Budowanie obrazu -> Deploy na platformę -> Testy



Recepty

☰

Minimum

- Odpowiednia struktura w packages/<NAZWA>

```
.  
├── Config.in  
├── hello.mk  
└── src  
    ├── hello.c  
    └── Makefile
```

- Wpis w packages/Config.in



Config.in

```
config BR2_PACKAGE_LIBFOO
    bool "libfoo"
    help
        This is a comment that explains what libfoo is. The help text
        should be wrapped.
        http://foosoftware.org/libfoo/
```



Recepta - plik .mk

Wspierane build systemy:

- Generic Makefiles
- CMake
- Autotools
- Lua
- Python



Kod .mk

```
#####
# Recepta
#####

# Replace 'your-package' with the actual name of your package.
HELLO_VERSION = 1.0
HELLO_SITE = $(call github,KarolPWr,aahed,v$(HELLO_VERSION))

# Build config
define HELLO_BUILD_CMDS
    $(MAKE) CC="$(TARGET_CC)" LD="$(TARGET_LD)" -C $(@D)
endef

# Install target for your built binaries
define HELLO_INSTALL_TARGET_CMDS
    $(INSTALL) -D -m 0755 $(@D)/src/hello $(TARGET_DIR)/usr/bin
endef

# Register the package
$(eval $(generic-package))
```



Kod aplikacji

Dowolność - źródła + build system (Makefile, cmake itp.)

Q: Z jakich build systemów korzystaliście, jakie znacie?



Ćwiczenie

Pisanie recepty

Integracja paczki w systemie



Obsługa sprzętu w Linuxie





Kernel i sterowniki

Kernel sprawuje kontrolę nad inicjalizacją i zarządzaniem HW.

W runtime albo przez devicetree

Typy driverów:

- Character devices (np. /dev/ttyS0)
- Block devices (np. Karta SD)
- Network devices (Eth0, wlan0)



Ekspozycja informacji - sysfs, dev/, procfs, ioctl

Katalog /dev - device nodes:

- Daje dostęp do “zawartości pudełka”
- Pozwala programom userspace na dostęp do danych w urządzeniu, np. Zapisywanie danych na dysku, odczyt z seriala
- Udev, mdev - automatyczne zarządzanie device nodes (wygodne)

Sysfs (/sys/class/)

- Daje dostęp do “opakowania”
- Pozwala sprawdzić wyniki pomiarów, czy device jest włączony, jaki ma adres etc.
- Pliki w sys/ to linki oraz zwykłe pliki. Wpisywanie / odczyt danych wyzwala określone funkcje w kernelu



Procfs, ioctl

Procfs - kiedyś drivery, teraz raczej tylko odczyt systemowy

cat /proc/cpuinfo

cat /proc/meminfo

ioctl - interfejs programistyczny, raczej dziś niemodny

```
.... // Open the block device (Requires root privileges)
.... fd = open("/dev/sda", O_RDONLY);
.... if (fd == -1) {
....     perror("Failed to open /dev/sda");
....     return 1;
.... }

.... // Get disk geometry using ioctl
.... if (ioctl(fd, HDIO_GETGEO, &geo) == -1) {
....     perror("ioctl error");
....     close(fd);
....     return 1;
.... }
```



Przykład - driver bmp280

<https://github.com/torvalds/linux/blob/master/drivers/iio/pressure/bmp280-i2c.c#L8>

Bmp280-i2c -> bmp280core -> iio.h -> industrialio-core.c -> char_dev.c



Ćwiczenie

Obsługa drivera BMP280



Dobór sprzętu

Q: Co jest najważniejsze w projekcie?

A: Koszt.

Tani czujnik - DHT11

Jest wsparcie w wersji bit-banging. Możliwe że zadziała słabo

BMP280 - też tani, ale ma lepsze wsparcie. Mniej pracy i więcej pewności że zadziała

Jak sprawdzić czy jest wsparcie?

<https://www.kernelconfig.io/index.html>

Security





Ćwiczenie

Security hardening w Buildroot



Jaki wybrać?

Choosing the Right Embedded Linux OS

Real-Time Needs

Consider RTLinux for precise timing

Network Focus

Opt for OpenWRT/LEDE

Quick Deployment

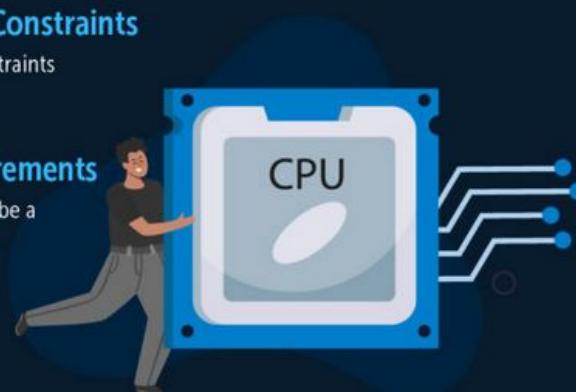
Ubuntu for faster development

Resource Constraints

Resource Constraints

GUI Requirements

Android might be a good fit





Pamiątki

Repozytorium z instrukcjami i rozwiązań:

<https://github.com/KarolPWr/buildroot-intro/tree/cm4>

Docker przystosowany do buildroota:

<https://github.com/KarolPWr/docker-buildroot>

Bonus - przydatne linki: w repozytorium

Prezentacja - push po szkoleniu

Co dalej?



Build systemy - Yocto

Aplikacje - web serwer, stacja pogodowa,

Wizytówka - <https://www.thirtythreeforty.net/posts/2019/12/my-business-card-runs-linux/>

Kernel drivers - pisanie prostego drivera

Aplikacje graficzne - dorzucenie GUI do Buildroota

Języki: Bash, C, C++, Linux