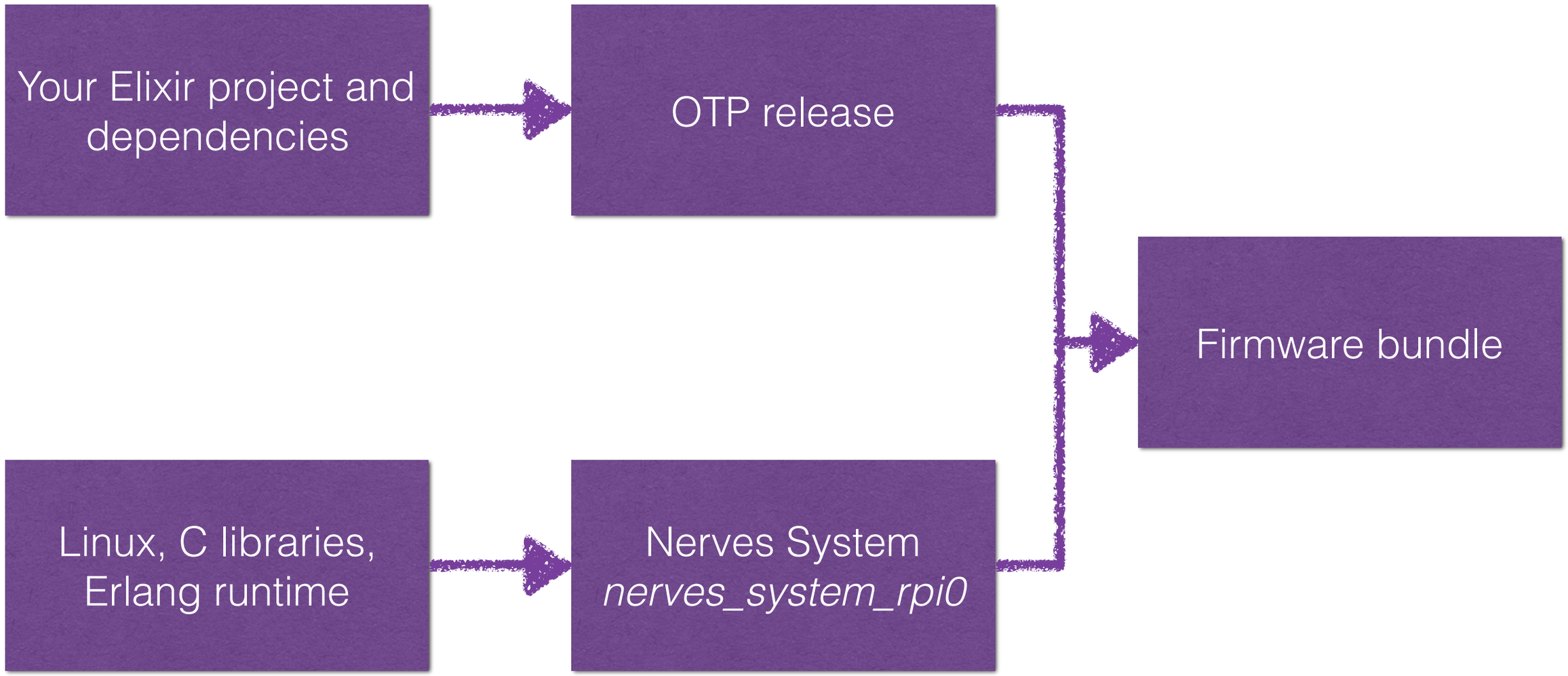
ELIXIRCONF™ 2017

# Nerves Training
# Part 4: Systems

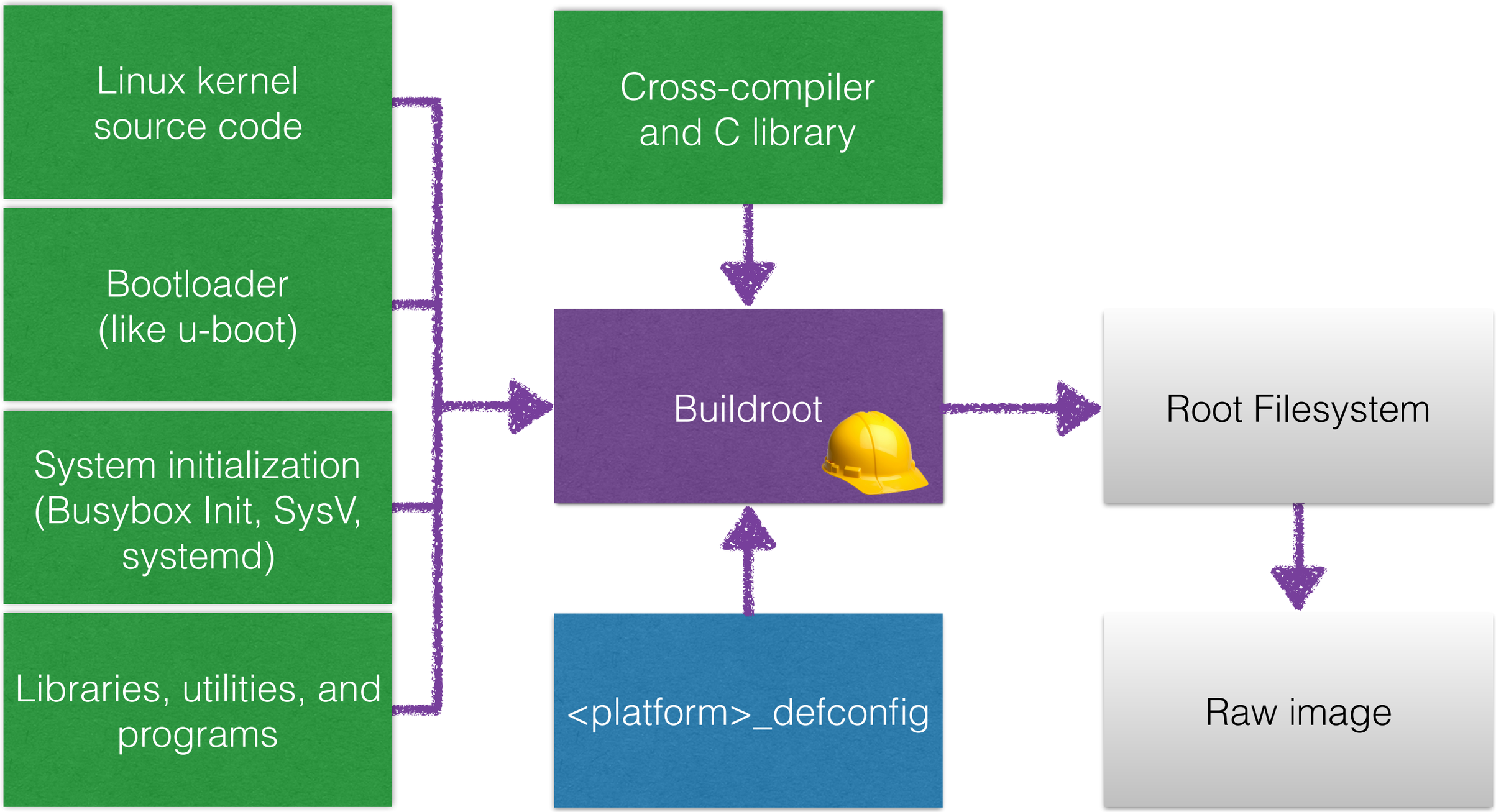
# Goals

- Introduce Buildroot
- Create a custom Nerves System
- Port a C program to your Nerves System
- Bundle up the system to share it with others

# Nerves

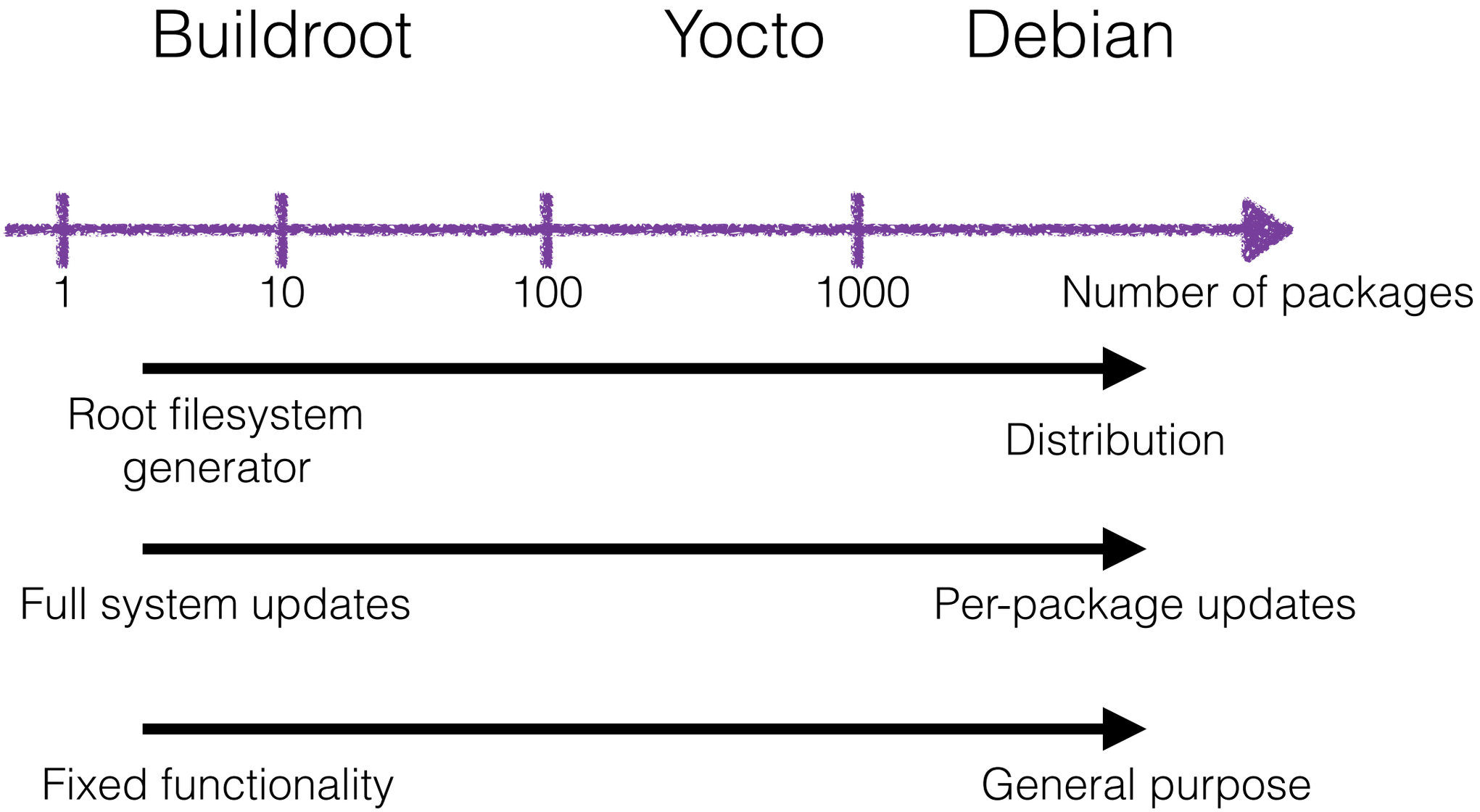

Your Elixir project and dependencies
OTP release
Firmware bundle
Linux, C libraries, Erlang runtime
Nerves System
nerves_system_rpi0

# Buildroot


Linux kernel source code
Bootloader (like u-boot)
System initialization (Busybox Init, SysV, systemd)
Libraries, utilities, and programs
Cross-compiler and C library
Buildroot
<platform>_defconfig
Root Filesystem
Raw image

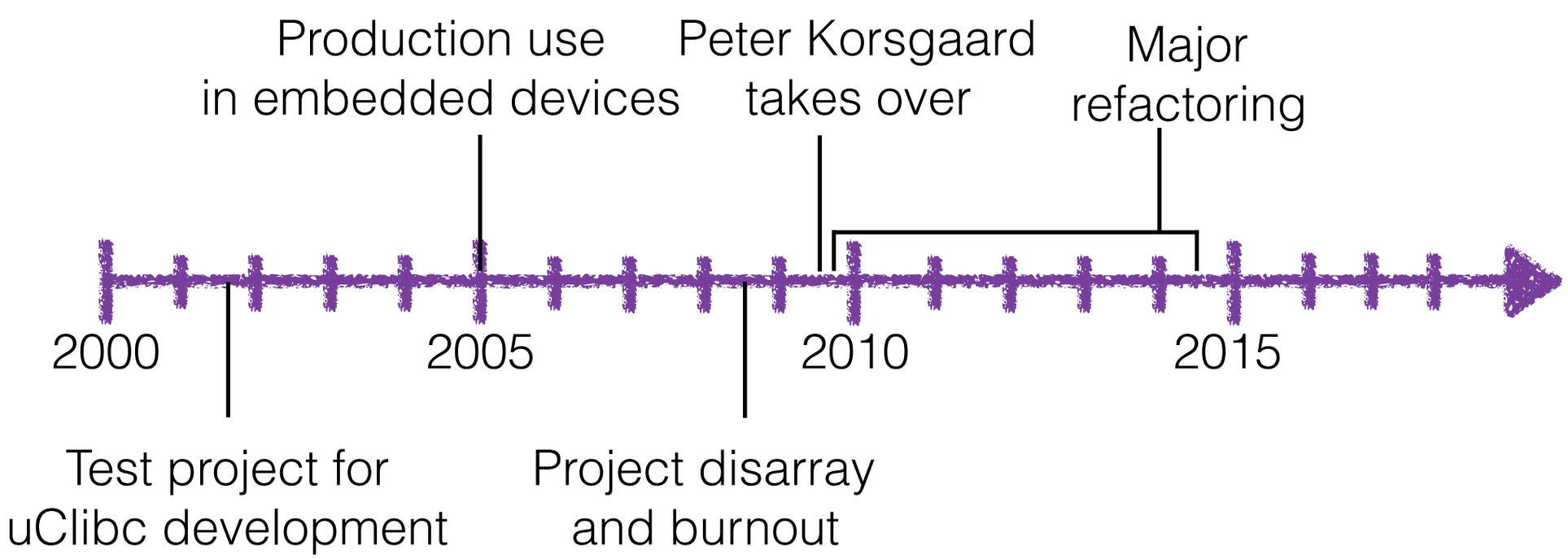
# Why Buildroot?

- Widely used in embedded Linux systems
- Well-supported
  - Large user base and dedicated maintainers
  - Thorough documentation
  - Commercial support available
- Tons of supported libraries and applications




Adapted from http://free-electrons.com/pub/conferences/2012/lsm/buildroot/buildroot.pdf

# Buildroot history




Buildroot currently has 3 committers, a team of patch reviewers
and maintainers, a regular release cycle w/ long term support releases,
regular meetings, and a large and very active mailing list and IRC channel.

# Buildroot challenges

- Minimalist philosophy means some packages don't work out-of-the-box
- Lack of binary package manager means rebuilds - sometimes long rebuilds
- "PR" submission to mailing list is unfamiliar
- Significantly less marketing compared to Yocto

The first three are by design - many have lots of debate on the Buildroot mailing list.

# Toolchains and Systems

- Toolchains
  - Compile anything that's C/C++ related
  - Associated with a target processor architecture (ARM, MIPS, i586, x86_64, etc.)
  - Runs on the host (64-bit Linux, OSX, Raspberry Pi 3)
  - Creating your own toolchain is rare
- Systems
  - Built for a target device
  - Reusable across projects
  - Buildroot-based

Prebuilt components of Nerves

# Use cases for creating systems

- Port Nerves to a new platform
  - Port Buildroot to the new platform
  - Figure out the firmware image layout and how firmware updates should work
- Add C programs or modify the Linux kernel of an existing system
  - Fork and modify an existing Nerves system
  - Most common use case

# Official Nerves Systems

- Raspberry Pi A+/B+, 2, 3, and Zero
- BeagleBone Black, Green, and compatible variants
- Lego EV3
- Linkit Smart Duo
- QEMU ARM

# Some systems in the wild

- Orange Pi Zero - Allwinner ARM SoC
- Vultr - x86_64 system for Nerves deployment to the cloud
- Kiosk x86_64 - Webkit Kiosk on x86_64 PCs
- Kiosk Raspberry Pi 3 - Webkit Kiosk for RPi3's with displays

# Anatomy of a System

- **mix.exs** - Standard Elixir build info + Nerves info
- **nerves_defconfig** - Buildroot configuration
- **linux-x.x.defconfig** - Linux kernel configuration
- **fwup.conf** - Firmware image layout and upgrade configuration
- **rootfs_overlay** - Additional files to put in the read-only root filesystem

Example: https://github.com/nerves-project/nerves_system_rpi0/

# Goal: Create a custom system

# Create a new directory for the custom system and a trivial project

```
$ cd <workspace>
$ mkdir mysystem
$ cd mysystem

copy over the starter project
$ cd starter
$ export MIX_TARGET=rpi0
$ mix do deps.get, firmware
```

**Skip typing**

```
$ git clone \
https://bitbucket.org/fhunleth/nervestraining-mysystem.git mysystem
$ cd mysystem
$ git checkout step1
```

# Create a custom system based off nerves_system_rpi0

```
$ cd mysystem
$ git clone git@github.com:nerves-project/nerves_system_rpi0.git \
    custom_rpi0
$ cd custom_rpi0
$ git checkout v0.17.1

Create an empty `custom_rpi0` repository in your GitHub account
$ git remote rename origin upstream
$ git remote add origin git@github.com:YourGitHubUserName/custom_rpi0.git
$ git push origin master
```

**Skip typing**

```
$ git clone \
https://bitbucket.org/fhunleth/nervestraining-mysystem.git mysystem
$ cd mysystem
$ git checkout step2
```

# Rename, rename, rename

- Update metadata in mix.exs, nerves.exs and nerves_defconfig
- nerves_system_rpi0 -> custom_rpi0
- Be sure to update URLs
- Run `git show step3` to see changes

```
Skip typing
$ git checkout step3
Really, just do this. This is a tedious step.
```

# Update starter to use custom_rpi0

```
  def system("rpi"), do: [{:nerves_system_rpi, ">= 0.0.0", runtime: false}]
  def system("rpi0"), do: [{:nerves_system_rpi0, ">= 0.0.0", runtime: false}]
  def system("custom_rpi0"), do: [{:custom_rpi0, path: "../custom_rpi0", runtime: false}]
  def system("rpi2"), do: [{:nerves_system_rpi2, ">= 0.0.0", runtime: false}]
```

**Skip typing**

```
$ git checkout step4
```

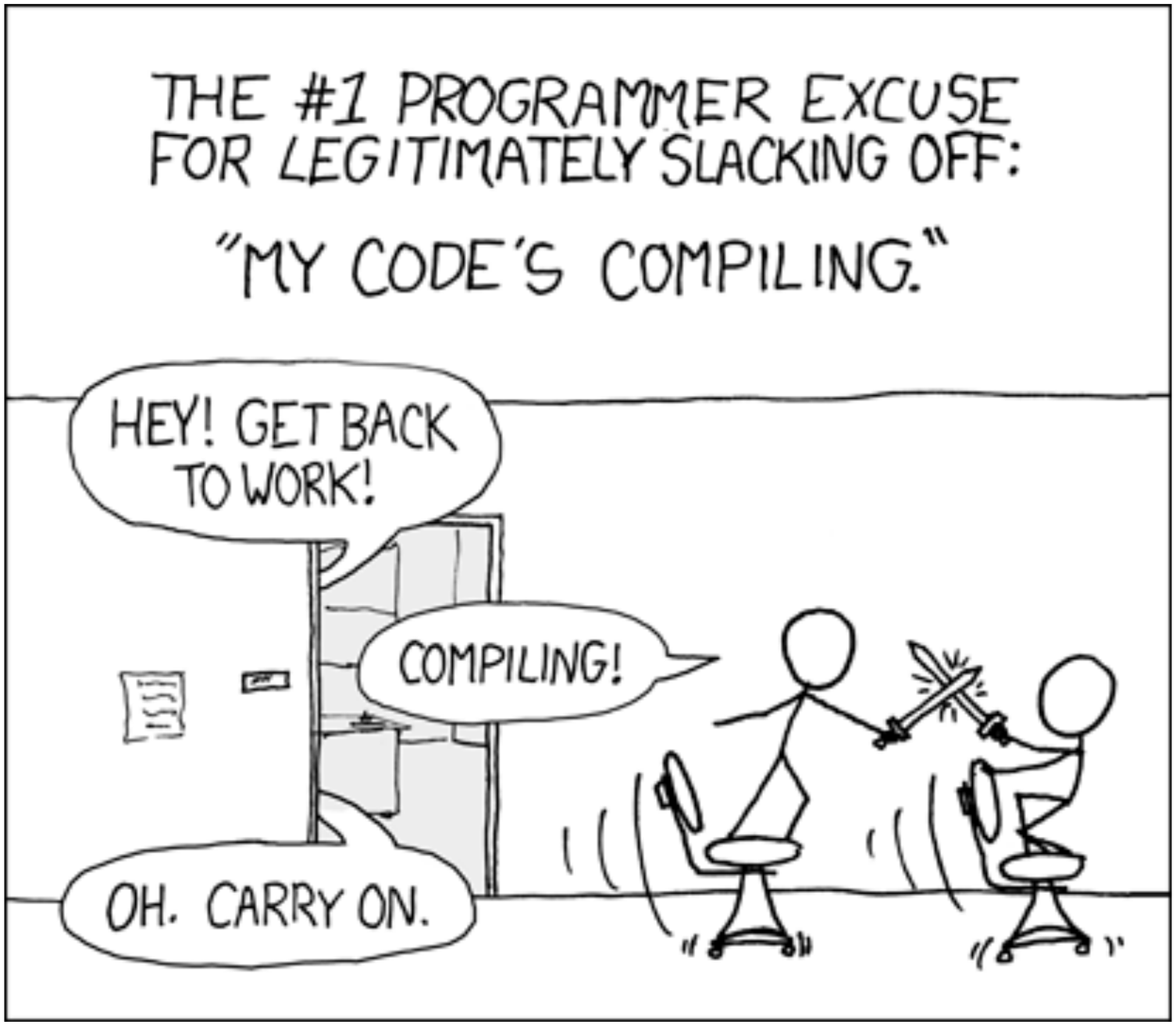
# Build

This part takes ~30 minutes and downloads a lot.

```
$ export MIX_TARGET=custom_rpi0
$ mix deps.get
$ mix firmware
```

**Skip typing**

```
$ git checkout step4
```




https://xkcd.com/303/

# Break

- Before leaving, let the build go for a few minutes
- If you copied over the files to ~/.nerves/dl, you should not see any files being downloaded
- Should be fine to CTRL+C the build and start it up later this evening
- If you get errors, send a message to #nerves_training or arrive a little early tomorrow

# Verify that your custom build works

```
$ cd <workspace>/mysystem/starter
$ mix firmware.push nerves.local
   -or-
$ ./upload.sh

$ picocom /dev/tty.usb*
```

**Skip typing**

```
$ git checkout step4
```

# Goal: Modify the Linux kernel to blink LED on USB activity

# Linux LED Subsystem

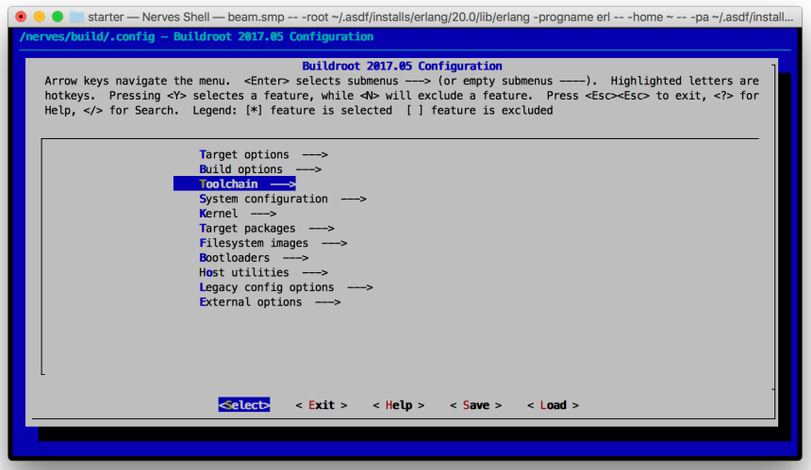
- Linux has built in drivers for controlling LEDs via reads and writes to files in /sys/class/leds
- Pros
  - Blinking an LED is practically free
  - Triggers support blinking on CPU, network, disk, USB activity and a lot more
- Cons
  - Requires device tree configuration
- Luckily, the RPi0 W device tree has the main LED already configured, but need a kernel option enabled to support USB triggers

# nerves.system.shell

- MIX_TARGET=custom_rpi0 mix nerves.system.shell
- Drops you into the directory or Linux container that builds the Nerves System
- Hides a LOT of machinery involved with building Linux on non-Linux machines
- To use CTRL+C you need to set a VM Flag

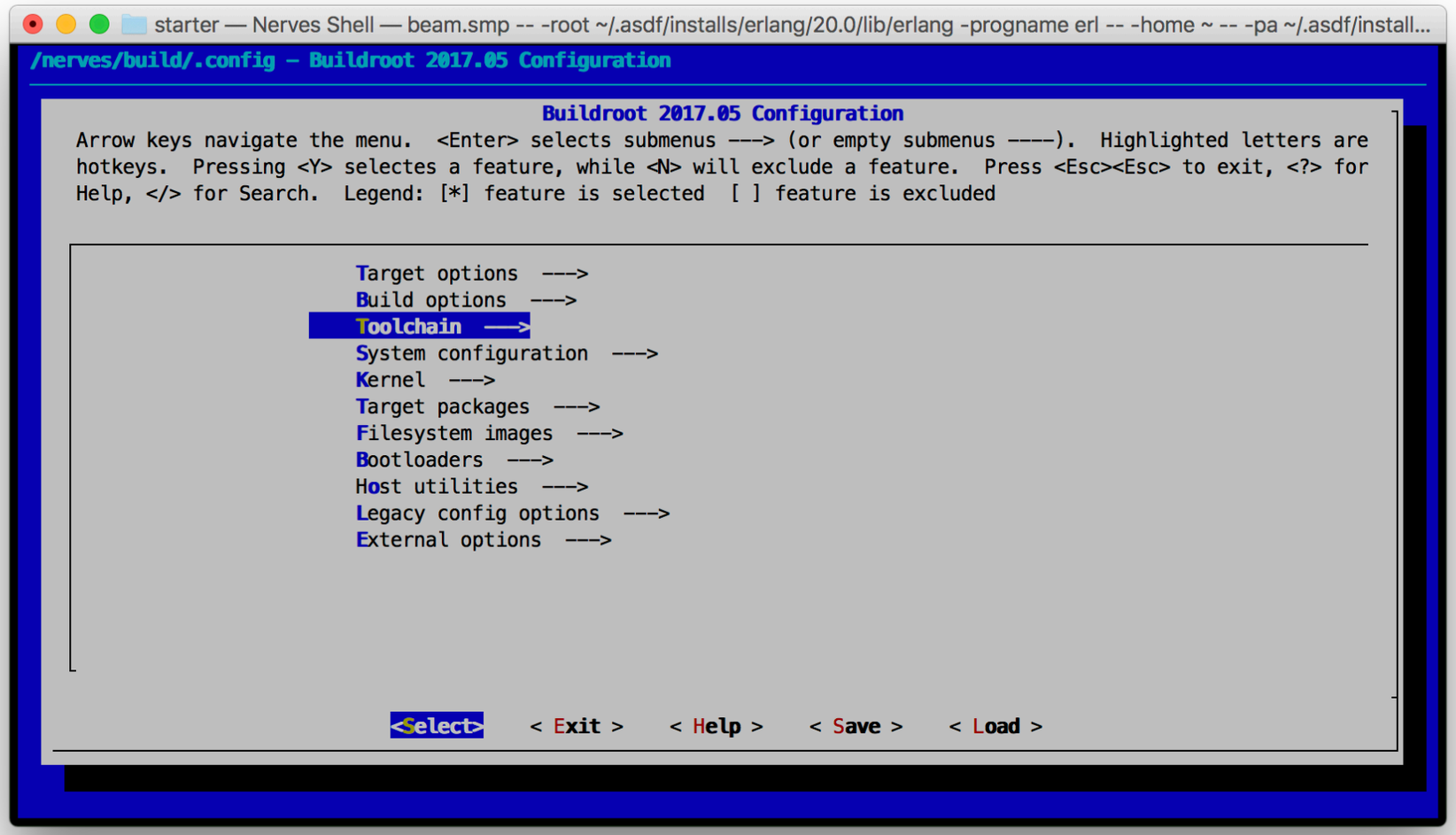
```
ELIXIR_ERL_OPTIONS="+Bc" mix nerves.system.shell
```

# Kconfig




- Kconfig
  - Linux kernel configuration tool and language
  - Reused by many Linux programs including Buildroot
    - "make menuconfig"
- Important files
  - .config - contains all configuration options
  - defconfig - file non-default options; created by "make savedefconfig"
- Usually defconfigs are version controlled

# menuconfig basics




- Exit will prompt whether to save
- Type '/' to search and then type the number of the result to jump to it

# Common menuconfigs

- make menuconfig
  - Modify the Buildroot configuration
  - "make savedefconfig" saves to nerves_defconfig
- make linux-menuconfig
  - Modify the Linux kernel configuration
  - "make linux-savedefconfig" saves to *build/linux-x.y.z/defconfig*
  - Manually copy the *defconfig* to the Nerves System directory

# Enable USB LED triggers

```
$ export MIX_TARGET=custom_rpi
$ ELIXIR_ERL_OPTIONS="+Bc" mix nerves.system.shell

/nerves/build > make linux-menuconfig

Navigate to Device Drivers->USB Support->USB LED Triggers
Press SPACE to select option
Keep pressing ESCAPE and then Yes to save

/nerves/build > make linux-savedefconfig
/nerves/build > cp build/linux-0*/defconfig ../env/custom_rpi0/linux-4.4.defconfig
(on Linux) > cp build/linux-0*/defconfig ../../../linux-4.4.defconfig
/nerves/build > exit # or CTRL+D

See that you did something
$ git diff
```

**Don't skip typing this one, but here's the result**

```
$ git checkout step5
```

# Try it out

```
$ mix firmware
$ mix firmware.push nerves.local
   -or-
$ ./upload.sh

$ picocom /dev/tty.usb*

iex> trigger_path="/sys/class/leds/led0/trigger"
iex> File.read!(trigger_path)
"none ... usb-gadget ... [mmc0] ..."
iex> File.write!(trigger_path, "usb-gadget")

Type on the console and watch the blinking
```

**Skip typing**

```
$ git checkout step5
```

# For more LEDs in Elixir

## nerves_leds 0.8.0

Functions to drive LEDs on embedded systems

### Maintainers

Garth Hitchens, Chris Dutton

### Links

Online documentation (download)
github

### License

MIT

**332** downloads this version
**2** downloads yesterday
**18** downloads last 7 days
**3 986** downloads all time

### Versions (2)

**0.8.0** February 23, 2017 (docs)
**0.7.0** June 24, 2016 (docs)

### Dependencies (0)

### Config

mix.exs

```
{:nerves_leds, "~> 0.8.0"}
```

### Checksum

```
193692767dca1a201b09
```

### Build Tools

mix

### Owners

ghitchens

### Dependents (0)

# Goal: Add a Buildroot package to our system

# Buildroot packages

- Buildroot contains >1500 applications, libraries, drivers and more
- Buildroot's secret sauce is making all of these crosscompile with very little work
- Nerves (nerves_system_br) adds a few more
- Each Nerves System can add even more (useful for proprietary C libraries/applications)

# Add *devmem2*

```
$ export MIX_TARGET=custom_rpi
$ ELIXIR_ERL_OPTIONS="+Bc" mix nerves.system.shell

/nerves/build > make menuconfig

Navigate to Target packages
Select "Show packages that are also provided by busybox"
Navigate to Hardware handling
Select "devmem2"
Keep pressing ESCAPE or CTRL+C and then Yes to save

/nerves/build > make savedefconfig
/nerves/build > exit # or CTRL+D

See that you did something
$ git diff
```

**Don't skip typing this one, but here's the result**

```
$ git checkout step6
```

# Try it out

```
$ mix firmware
$ mix firmware.push nerves.local
   -or-
$ ./upload.sh

$ picocom /dev/tty.usb*

iex> cmd("devmem2")

Usage:  devmem2 { address } [ type [ data ] ]
        address : memory address to act upon
        type    : access operation type : [b]yte, [h]alfword, [w]ord
        data    : data to be written
```

**Skip typing**

```
$ git checkout step6
```

# Goal: Add a custom package to our system

# Reasons to add custom packages

- C library or application not available in Buildroot
- Special customization of a library or application that doesn't make sense to push upstream
- Integrate proprietary libraries and applications

# Buildroot Benefits

- Extremely well-tested rules for building CMake and autotools libraries and applications
- Installation rules make C libraries *just* work
- Build once for the system
- More information about the target than what's available if you use *elixir_make*
- Buildroot developers provide awesome support if some weird cross-compilation issue pops up

# Reasons against

- Alternative is to wrap C code in a mix project and invoke via the port interface

- If C application with a couple files, then *elixir_make* is pretty simple

- *mix deps* can pull in the dependency

- Use code with official Nerves systems

# Anatomy of a Buildroot package

- Config.in
  - Package name and description for Kconfig
  - Contains constraints and dependencies
    - Build options (if any)
- <package name>.mk
  - Download, build and installation instructions
  - Makefile language
- <package name>.hash

# helloworld

- https://bitbucket.org/fhunleth/helloworld/
- Trivial, but representative, autotools-based project
- "Official" release is in the Downloads tab
- You may want the Buildroot docs handy:
  http://buildroot.org/manual.html#_infrastructure_for_autotools_based_packages

# Create directories and glue

```
$ cd custom_rpi0
$ mkdir -p package/helloworld
```

custom_rpi0/Config.in

```
source "$NERVES_DEFCONFIG_DIR/package/helloworld/Config.in"
```

custom_rpi0/external.mk

```
# Include system-specific packages
include $(sort $(wildcard $(NERVES_DEFCONFIG_DIR)/package/*/*.mk))
```

**Skip typing**

```
$ git checkout step7
```

# custom_rpi0/package/helloworld/Config.in

One tab character

```
config BR2_PACKAGE_HELLOWORLD
	bool "helloworld"
	help
	  helloworld is a trivial autoconf/automake project.

	  https://bitbucket.org/fhunleth/helloworld
```

One tab + 2 spaces

**Skip typing**

```
$ git checkout step7
```

# custom_rpi0/package/helloworld/helloworld.mk

```
##############################################################################
#
# helloworld
#
##############################################################################

HELLOWORLD_VERSION = 1.0.0
HELLOWORLD_SITE = https://bitbucket.org/fhunleth/helloworld/downloads
HELLOWORLD_LICENSE = Unlicense
HELLOWORLD_LICENSE_FILES = LICENSE

$(eval $(autotools-package))
```

Buildroot magic sauce

**Skip typing**

```
$ git checkout step7
```

# Try it out

```
$ export MIX_TARGET=custom_rpi
$ cd starter
$ ELIXIR_ERL_OPTIONS="+Bc" mix nerves.system.shell

/nerves/build > make menuconfig

Navigate to External options->User-provided options
Select "helloworld"
Keep pressing ESCAPE or CTRL+C and then Yes to save

/nerves/build > make savedefconfig
/nerves/build > make
/nerves/build > ls target/usr/bin
See that helloworld was installed

$ exit # or CTRL+D
```

**Skip typing**

```
$ git checkout step7
```

# Goal: Publish our system for others

# Why?

- Building systems is slow and tedious
- The goal is to program in Elixir!

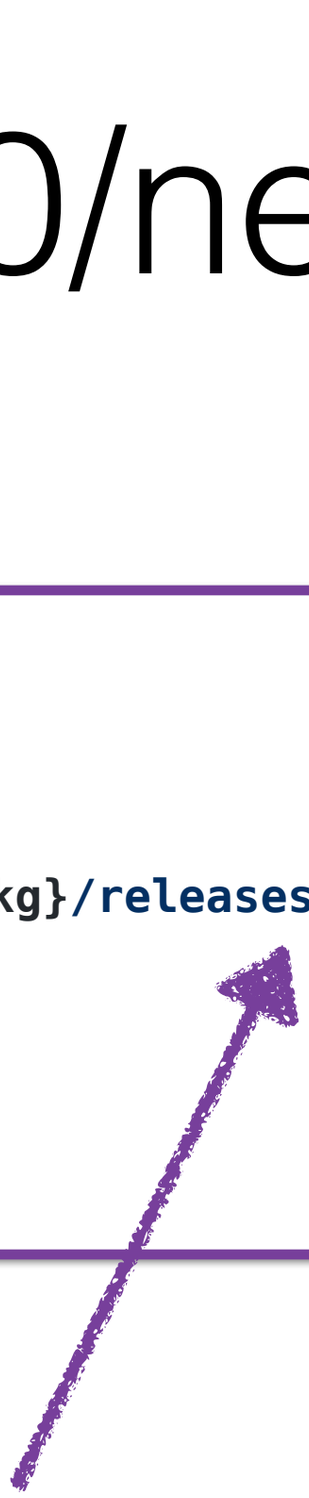
# Publishing the system

```
$ export MIX_TARGET=custom_rpi
$ ELIXIR_ERL_OPTIONS="+Bc" mix nerves.system.shell

/nerves/build > make system

Copy the system for ourselves

/nerves/build > cp custom_rpi0.tar.gz ../host/artifacts/custom_rpi0-v0.17.0.tar.gz

Upload custom_rpi0-v0.17.0.tar.gz to an HTTP server

/nerves/build > exit # or CTRL+D
```

Not done yet... Need to record the URL

# custom_rpi0/nerves.exs

```
config pkg, :nerves_env,
  type: :system,
  version: version,
  compiler: :nerves_package,
  artifact_url: [
    "https://github.com/yourstruly/#{pkg}/releases/download/v#{version}/#{pkg}-
v#{version}.tar.gz",
  ],
  platform: Nerves.System.BR,
  platform_config: [
    defconfig: "nerves_defconfig"
  ],
```



Double check the URL and then it's ready to *hex publish*

# Buildroot Golden Rules

1. make clean

2. make clean

3. mix nerves.clean custom_rpi0

Buildroot doesn't support incremental builds, but they work in so many cases that you'll think that it does. Then something super weird happens.

https://buildroot.org/downloads/manual/manual.html#full-rebuild

# Shipping Nerves - Licensing

- Buildroot infrastructure in Nerves can aid process
- make legal-info
- Other licenses
  - nerves-toolchain
    - gcc
    - crosstool-ng
  - All of your mix dependencies

```
legal-info
├── buildroot.config
├── host-licenses
│   ├── autoconf
│   │   ├── COPYING.EXCEPTION
│   │   └── COPYINGv3
│   └── ...
│       └── README
├── host-licenses.txt
├── host-manifest.csv
├── host-sources
├── licenses
│   ├── busybox
│   │   └── LICENSE
│   ├── erlang
│   │   └── LICENSE.txt
│   └── ...
├── licenses.txt
├── manifest.csv
├── README
└── sources
    ├── boardid-v0.4.0.tar.gz
    ├── busybox-1.24.1.tar.bz2
    ├── ...
    └── zlib-1.2.8.tar.xz
```

# Buildroot Bonus

- *make graph-depends* - see all package dependencies
- *make <pkg>-depends* - see all dependencies for one package
- *make graph-size* - see what takes up space
- *make source* - download everything needed for an offline build
- *ccache*, SSDs, DRAM, >4 cores, and native Linux

# Aside: Debugging HW

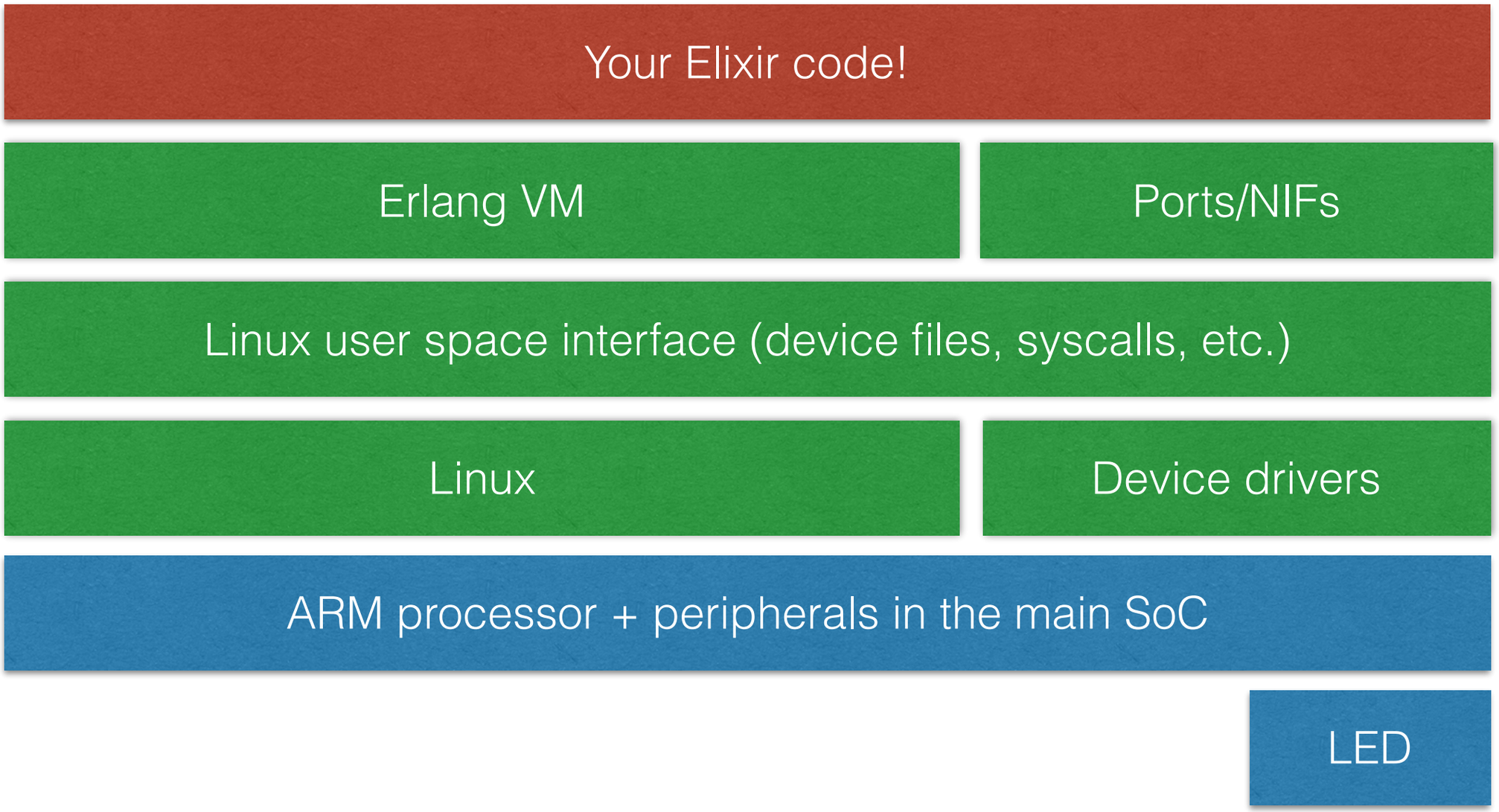
- At some point you'll need help from an Electrical Engineer to debug a HW problem

- EE's generally don't trust software (IMHO)

- EE's mostly trust manufacturer datasheets and specifications

- One debug strategy is to test at the datasheet level

# Cutting through the stack



# Goal: Blink an LED by going directly to the hardware

# Plan

- Find out where LED is attached to the BCM2835
- Find out how the lowest level software controls that interface
- Use *devmem2* from our custom Nerves System to write to this lowest level software interface

# LED connection

- Normally you "just" look at the schematic
  - https://github.com/raspberrypi/documentation/tree/master/hardware/raspberrypi/schematics
  - Unless updated recently, this is a partial schematic and it doesn't show the status LED connection
- Try #2: Device tree

# Status LED connection via Device Tree

https://github.com/raspberrypi/linux/blob/rpi-4.4.y/arch/arm/boot/dts/bcm2708-rpi-0-w.dts

```
&leds {
   act_led: act {
      label = "led0";
      linux,default-trigger = "mmc0";
      gpios = <&gpio 47 0>;
   };
};
```

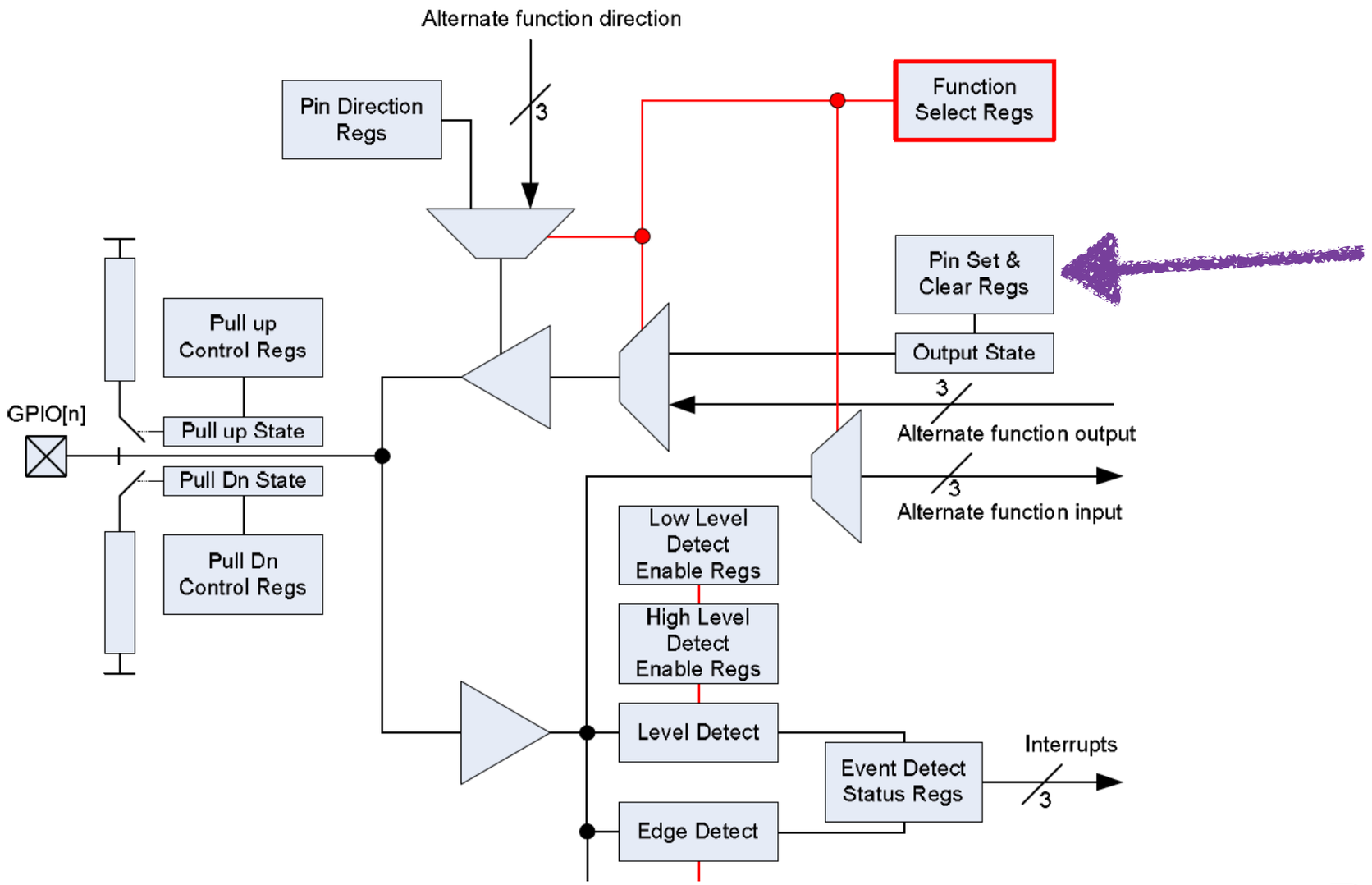
Maybe GPIO 47?

# Get the datasheet

- The SoC on the Raspberry Pi Zero is the Broadcom BCM2835
- Google "bcm2835 arm peripherals" for datasheet
- https://github.com/raspberrypi/documentation/tree/master/hardware/raspberrypi/bcm2835

# Find the GPIO section

The block diagram for an individual GPIO pin is given below :




Page 89

# Set register

| GPIO Pin Output Set Registers (GPSETn) | |
|---|---|
| **SYNOPSIS** | The output set registers are used to set a GPIO pin. The SET{n} field defines the respective GPIO pin to set, writing a “0” to the field has no effect. If the GPIO pin is being used as in input (by default) then the value in the SET{n} field is ignored. However, if the pin is subsequently defined as an output then the bit will be set according to the last set/clear operation. Separating the set and clear functions removes the need for read-modify-write operations |

| Bit(s) | Field Name | Description | Type | Reset |
|---|---|---|---|---|
| 31-0 | SETn (n=0..31) | 0 = No effect<br>1 = Set GPIO pin *n* | R/W | 0 |

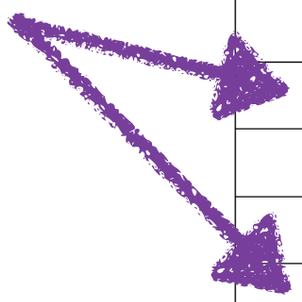
**Table 6-8 – GPIO Output Set Register 0**

| Bit(s) | Field Name | Description | Type | Reset |
|---|---|---|---|---|
| 31-22 | - | Reserved | R | 0 |
| 21-0 | SETn<br>(n=32..53) | 0 = No effect<br>1 = Set GPIO pin *n.* | R/W | 0 |

**Table 6-9 – GPIO Output Set Register 1**

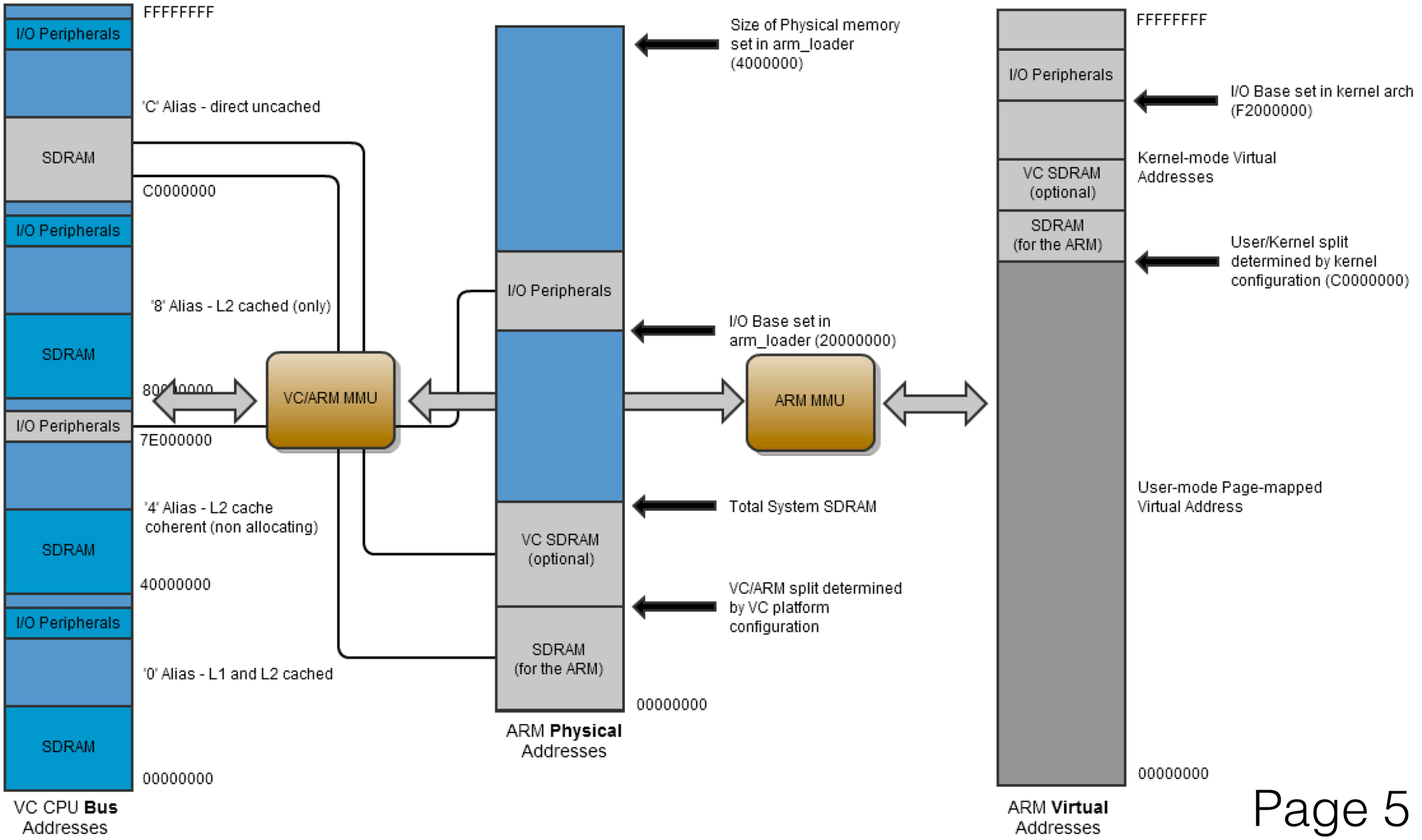
# How to get to the register

| Address | Field Name | Description | Size | Read/ Write |
|---|---|---|---|---|
| 0x 7E20 0000 | GPFSEL0 | GPIO Function Select 0 | 32 | R/W |
| 0x 7E20 0000 | GPFSEL0 | GPIO Function Select 0 | 32 | R/W |
| 0x 7E20 0004 | GPFSEL1 | GPIO Function Select 1 | 32 | R/W |
| 0x 7E20 0008 | GPFSEL2 | GPIO Function Select 2 | 32 | R/W |
| 0x 7E20 000C | GPFSEL3 | GPIO Function Select 3 | 32 | R/W |
| 0x 7E20 0010 | GPFSEL4 | GPIO Function Select 4 | 32 | R/W |
| 0x 7E20 0014 | GPFSEL5 | GPIO Function Select 5 | 32 | R/W |
| 0x 7E20 0018 | - | Reserved | - | - |
| 0x 7E20 001C | GPSET0 | GPIO Pin Output Set 0 | 32 | W |
| 0x 7E20 0020 | GPSET1 | GPIO Pin Output Set 1 | 32 | W |
| 0x 7E20 0024 | - | Reserved | - | - |
| 0x 7E20 0028 | GPCLR0 | GPIO Pin Output Clear 0 | 32 | W |
| 0x 7E20 002C | GPCLR1 | GPIO Pin Output Clear 1 | 32 | W |
| 0x 7E20 0030 | - | Reserved | - | - |
| 0x 7E20 0034 | GPLEV0 | GPIO Pin Level 0 | 32 | R |
| 0x 7E20 0038 | GPLEV1 | GPIO Pin Level 1 | 32 | R |



# Memory addresses

- Physical addresses
  - The real ones that connect to DRAM and memory-mapped peripherals
    - Rarely used anymore except on small microcontrollers
- Virtual addresses
  - Layer of indirection to provide the illusion of contiguous and "infinite" memory to processes
  - CPU's MMU translates virtual addresses to physical ones






GPSET1 is at 0x7e200020 -> Physical address 0x20200020

# In summary

- The Status LED is connected to GPIO 47
- To set GPIO 47
  - Write bit 15 to register GPSET1 at 0x7e200020
  - Set physical address 0x20200020 to 0x8000
- To clear GPIO 47
  - Write bit 15 to register GPCLR1 at 0x7e20002c
  - Set physical address 0x2020002c to 0x8000

# Try it out

```
Disable Linux's LED driver for now
iex> File.write!("/sys/class/leds/led0/trigger", "none")
:ok

GPIO 47 on (GPSET1)
iex> cmd("devmem2 0x20200020 w 0x8000")
:ok

GPIO 47 off (GPCLR1)
iex> cmd("devmem2 0x2020002c w 0x8000")
:ok

Read the state of GPIOs 32-63 (GPLEV1)
iex> cmd("devmem2 0x20200038 w")
Value at address 0x20200038 (0xb6fb8038): 0x3EEAFF
```

This may seem complicated, but it cuts out a lot of software that could be buggy and is very meaningful to an electrical engineer.

ELIXIRCONF™ 2017

# Nerves Training
# Part 4: Systems