newcircle

PSCG

# BYOD Revisited: **BUILD** Your Own Device

Ron Munitz, The PSCG
@ronubo

# about://Ron_Munitz

- **Distributed Fault Tolerant Avionic Systems**
  - Linux, VxWorks, very esoteric libraries, 0's and 1's

- **Highly distributed video routers**
  - Linux

- **Real Time, Embedded, Server bringups**
  - Linux, Android , VxWorks, Windows, devices, BSPs, DSPs,...

- **Distributed Android**
  - Rdroid? Cloudroid? Too busy working to get over the legal naming, so no name is officially claimed for my open source

# about://Ron_Munitz

**What currently keeps me busy:**

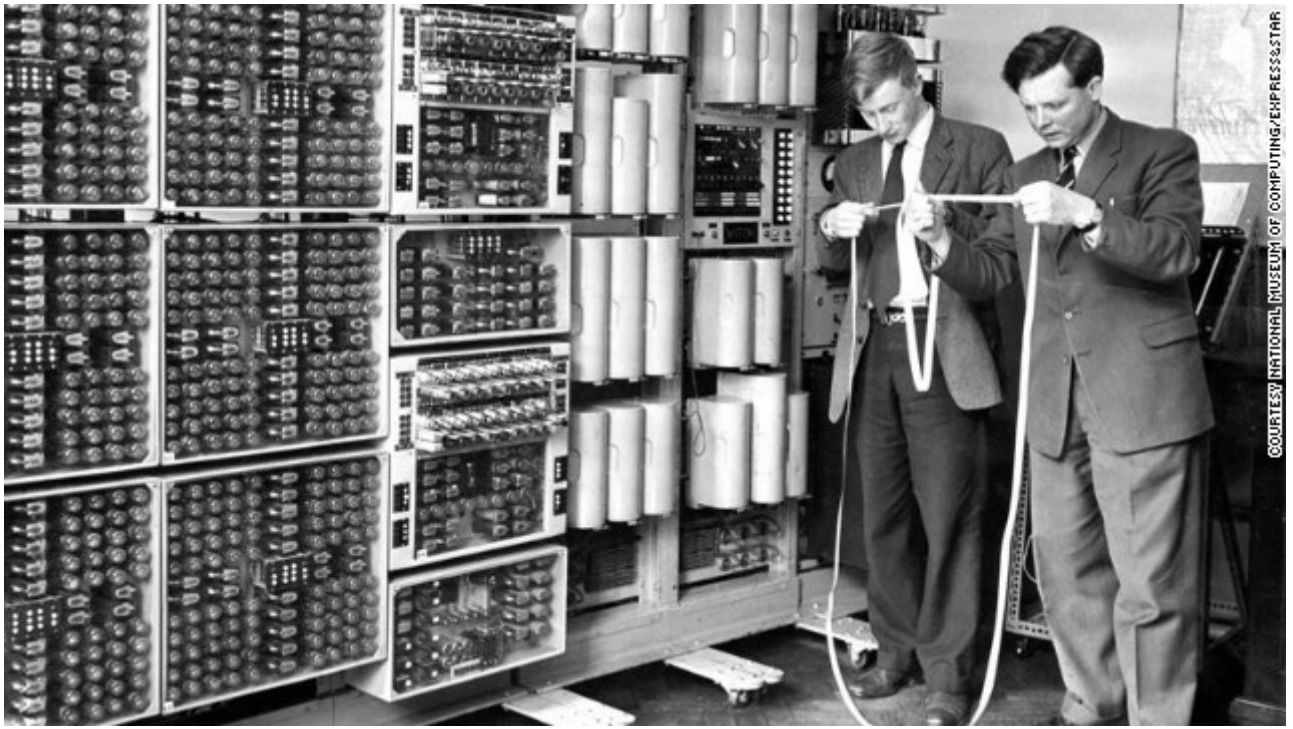
- **Running The PSCG**, an Embedded/Android consulting and Training
- **Managing R&D at Nubo** and advising on Remote Display Protocols
- Promoting open source  with **The New Circle expert network**
- Lecturing, Researching and Project Advising an **Afeka's college of Engineering**
- **Amazing present, endless opportunities.** (Wish flying took less time)

# Agenda

- History 101: Evolution of embedded systems
- Software Product 101: Past, Present, Future.
- Software Product 201: The Cloud Era.
- Hardware Product 101: Building Devices
- Hardware Product 201: The IoT Era

# History 101:

## 20th Century, 21st Century, Computers, Embedded Devices and Operating Systems




http://i2.cdn.turner.com/cnn/dam/assets/121121034453-witch-computer-restoration-uk-story-top.jpg

# Selected keypoints in the evolution of Embedded systems: The 20th century

- Pre 40’s: Mechanical Computers, Turing Machine
- The 40’s: Mechanical Computers, Embedded Mechanical Computers(V2...), Digital Computers (Z3...)
- The 50’s: Digital Computers, Integrated Circuit, BESYS
- The 60’s: MULTICS, Modem, Moore’s Law, PC
- The 70’s: Apple First Computer, Unix, B becomes C,
- The 80’s: DOS, MacOS, Windows, System-V, GNU
- The 90’s: GSM, World Wide Web, Windows 3.0/9*, 36/56Kbps Modems, ISDN, Linux

# Selected keypoints in the evolution of Embedded systems: The 21st century

Windows, Linux, OS X, Symbian, Moblin, Meego, Cellular Phones, PDA's, IEEE802.11, Wireless, Mobile Data, 2G, 3G, Bluetooth, NFC, GPS, Cameras iPod, iPhone, Android, iPad, Laptops, Tablets, Ultrabooks, ChromeOS, Google, Apple, Amazon, Facebook, Twitter, Pebble, Smart Phones, Smart Watches,Smart *, Smart Home, Sensors, Location, Data, Big Data, 4G, LTE, DSL, Cable, Broadband, IPTV, Streaming TV, Crowdsourcing, Social Media, Autonomous Cars, Robots, iRobot, The Internet of Things….

# Embedded Systems: Generation I

- Mostly Analog devices.
- Some examples include:
  - Laundry Machines
  - Refrigerators
  - “Ancient” Missiles
  - Cars
  - Thermostats

# Embedded Systems: Generation II

- Dedicated Hardware, dedicated software.
- Introducing the  Micro-Processor.
- No Operating Systems
- Heavy use of product dependent Assembly
- When lucky enough - also C.
- When extremely lucky - team is large enough to have some libraries.
- Still very common with DSPs and other performance critical components.

# Embedded Systems: Generation III

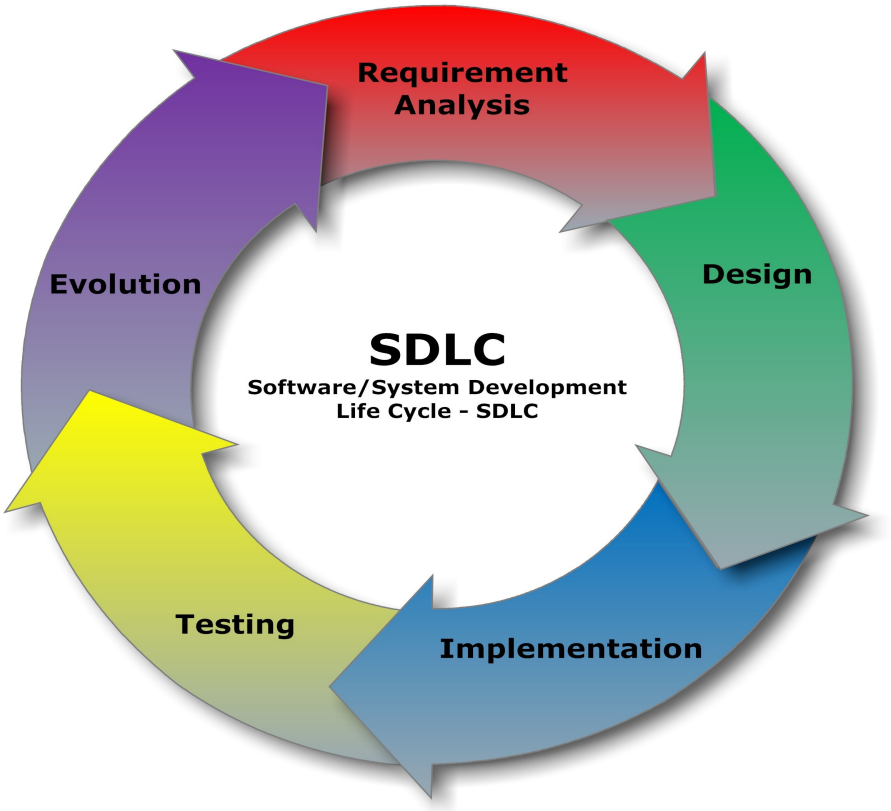
- Using Operating Systems:
  - Core/Real time embedded OS such as VxWorks, Embedded Linux variants, Windows, Integrity, and a lot lot more
  - General Purpose Operating Systems such as some Embedded Linux Variants
- Also run servers
  - You really use them anywhere and everyday:
  - Set Top Boxes, Televisions, Entertainment Systems, **ROUTERS**, Phones, Smart watches...

# Embedded Systems: Generation IV

- Using Android,iOS, Tizen, FirefoxOS or other “mobile” Operating Systems.
- It is somewhere between:
  - Core/Real time embedded Opsrating Systems
  - General Purpose Operating Systems
  - Also run servers (@ronubo et. al , android-x86.org)
- You CAN really use them anywhere and everyday:
  - Set Top Boxes, Smart TV’s, Entertainment Systems, Cellular Phones, Tablets, smart watches...

# Generation IV - Example

# Software Product 101: Past, Present/Future



# What we won't cover:

- Software Development Methodologies “Blah-Blah”
  - Waterfall/Spiral/Rapid/Agile/XP,...
- Antique technology which doesn't interest anyone (almost)
  - Card Punching
  - Mainframes

- If the latter interests you join my OS course next year. They always start with a blast from the past...

# What we will cover

- Practical Software Development tools and practices
  - From the not so far past
  - From the present

# How to build software: Prior to the cloud

- Get struck by an idea
- Think whether it's worth spending the time on implementing it.
- Define requirements (at least partially)
- Choose technology
  - Hardware for which it is based/on which it will run.
    Can be a PC, can be a server, can be some embedded device (e.g. cashier)
  - Choose Operating system on which it will run.
  - Choose programming languages/libraries
  - Choose other Infrastructure, middleware etc. (e.g. DB, etc.)
- Design/Code/Test (and usually skip two...)

# Simple Example: Portable Text Processor

- Idea: A word processor to be written (coded) once and used everywhere
- Is it worth it? Probably not, but it's just an example
- Requirements: Some sort of GUI, run across all PC platforms.
  - Assume supplemented libraries (e.g. QT, GTK+, maybe assume a JVM etc.)
- Technology: C/C++ with QT. SVN for source control.
- Design/Code/Test.
  - Testing: Open a file,  read a file, close a file, upload to sourceforge ;-)

# Simple Example: Analysis

- There is only one component, which runs at the Client side.
- Costs:
  - Development
  - Maybe marketing (but I won't address it here)
  - Maintenance (Bugs, feature requests)
- There is a single point of failure, and failure will affect only the very same user.

# Building More Complex Software: Using Servers

- Scenario: Clients, Servers. Real Software.
  - Multiple Clients, User Centric.
  - Multiple Servers, Business Logic Centric
- Choose Protocols
- Choose Technology
  - Backend (C,C++, Java, .NET, Scala, Python, Go,...)
  - Frontend (C, C++, .NET, HTML/JavaScript/CSS,...)
- Choose Data Center. **Operate it**

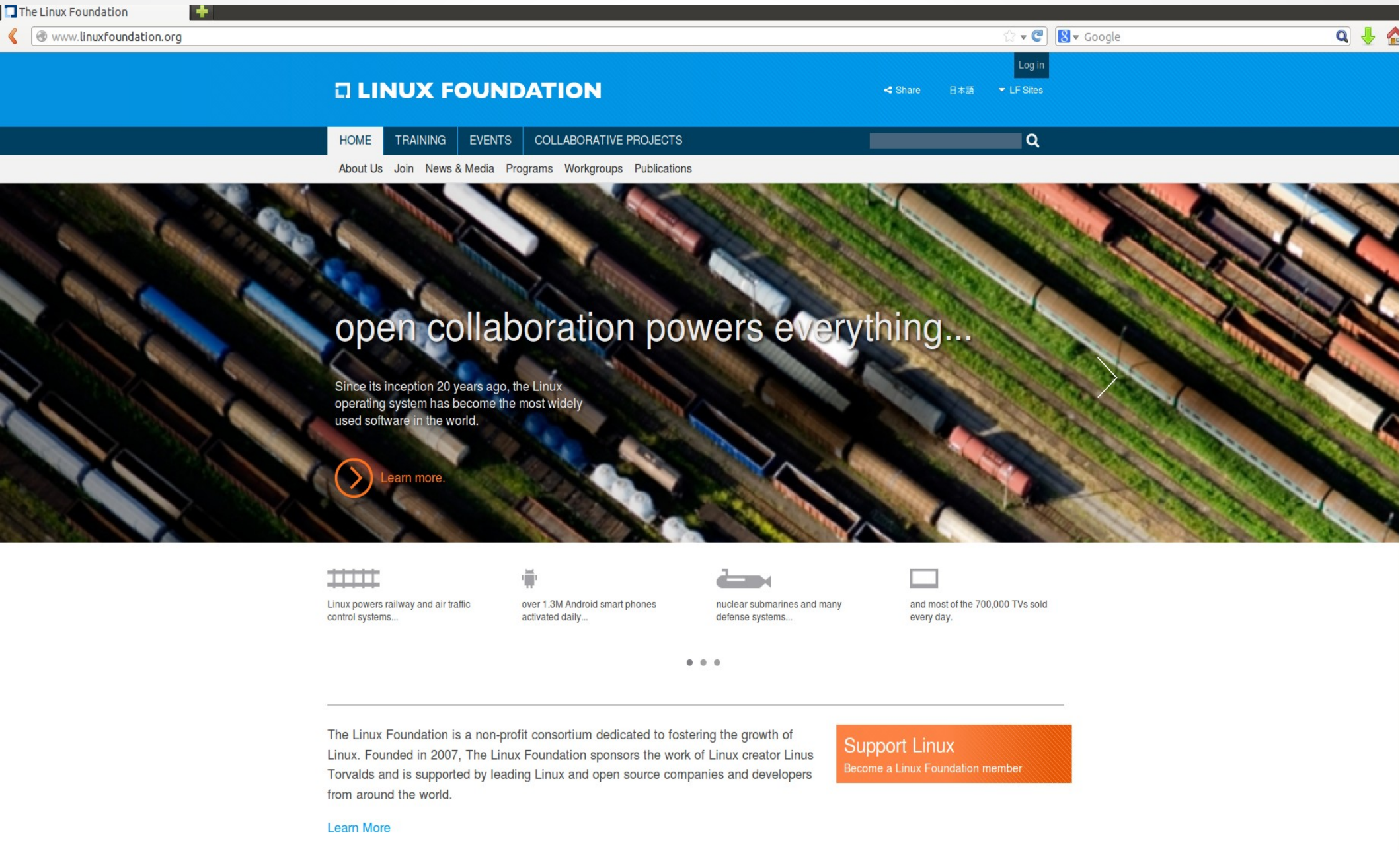
# Building More Complex Software: Analysis

- Multiple components.
- Multiple users on a single or multiple servers → Any server failure can affect all clients
- High Availability and fault tolerance
- Need to save state → Use Databases.
  - That's another design/development/testing phase.
  - Most companies did they're on DB frameworks, even on top of existing ones as Oracle/GoAhead,SAP etc.
- Scalability
  - Today we work with M users :-)
  - Tomorrow comes the M+1 user :-(

# Building More Complex software: Analysis

- Additional Costs (cf. The Simple example):
  - Equipment. Servers cost.
  - Data Center operation costs
    - Rent, electricity, cooling, technicians...
    - Apply whether you run your own Data Center, or use someone else's
  - High Availability engineering costs (design/development/testing)
  - More costs if you build your own hardware (i.e. Network processors etc.)
  - And repeat
    - Hardware malfunctions, grows old, need to purchase new equipment, etc.
- More operating costs **– running a data center is a never ending maintenance effort**

# Example: The Web






# Software Product 201: The Cloud Era.



# Enter the cloud

- I don't believe this subject requires any introduction these days.
- And that is for the following reasons:
  - Relieves the pain of Operating a data center
    - Much of the maintenance becomes someone else's problem
    - Including Hight Availability, scaling, storage and DB handling etc.
  - Eases deployment at diverse locations
  - Enables seamless models of IaaS, PaaS
  - **Significantly eases rapid prototyping**
  - **And <u>Deployment</u>**

# The importance of rapid prototyping

- The tech world is evolving quickly.
- And so does competition
- Many times you don't know what you've got until you start  “human trials”
  - That's legal in most software industries
- **And you want to know what you got.**
  - Fast.
  - **NOW!**

# How to build software: Using the cloud

- Get struck by an idea
- Think whether it's worth spending the time on implementing it.
- Define requirements (at least partially)
- Choose technology. For both ***Frontend*** and ***Backend***
  - Hardware for which it is based/on which it will run.
    Can be a PC, can be a server, can be some embedded device (e.g. cashier)
    - **Can select hardware type at your cloud vendor (Google Compute Engine, AWS,...)**
  - Choose Operating system on which it will run.
  - Choose programming languages/libraries
  - Choose other Infrastructure, middleware etc. (e.g. DB, etc.)
- Design/Code/Test (and usually skip two...)

# Modern Software Development Process - Frontend

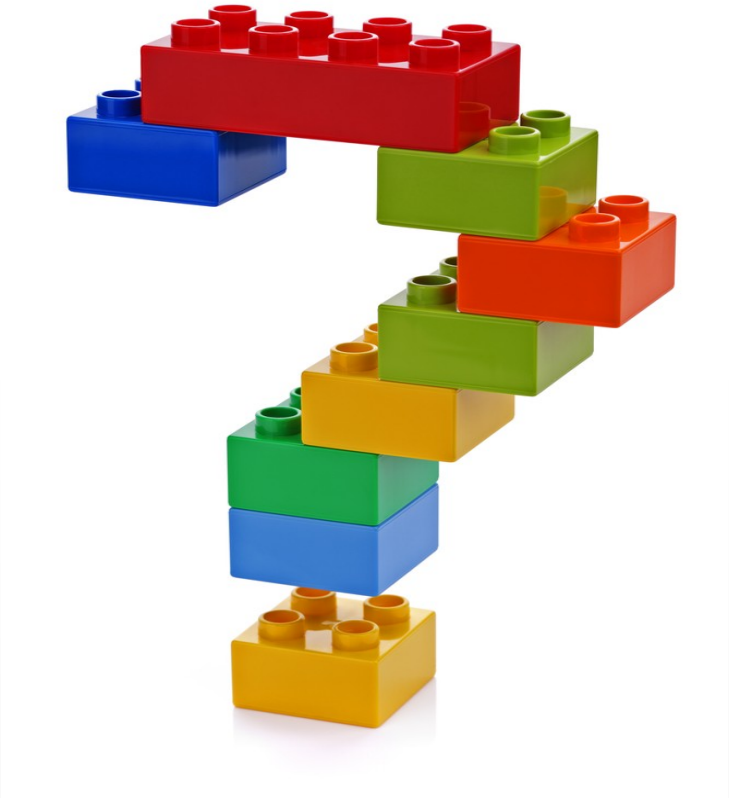
- Choose compatible OS/devices: Android, iOS, Win8, HTML5, Web
- Choose corresponding languages and frameworks.
  - i.e. Objective C+Cocoa,  Android+OpenGL+Volley, Win8+.NET,...
- **Integrate Open Source (@see github et. al)**
  - When you want to do something – most of it might have already be done
- Adhere to client/server protocol
  - Usually some REST-ful API, JSON, XML these days
  - But also binary streams, and more.

# Modern Software Development Process - Backend

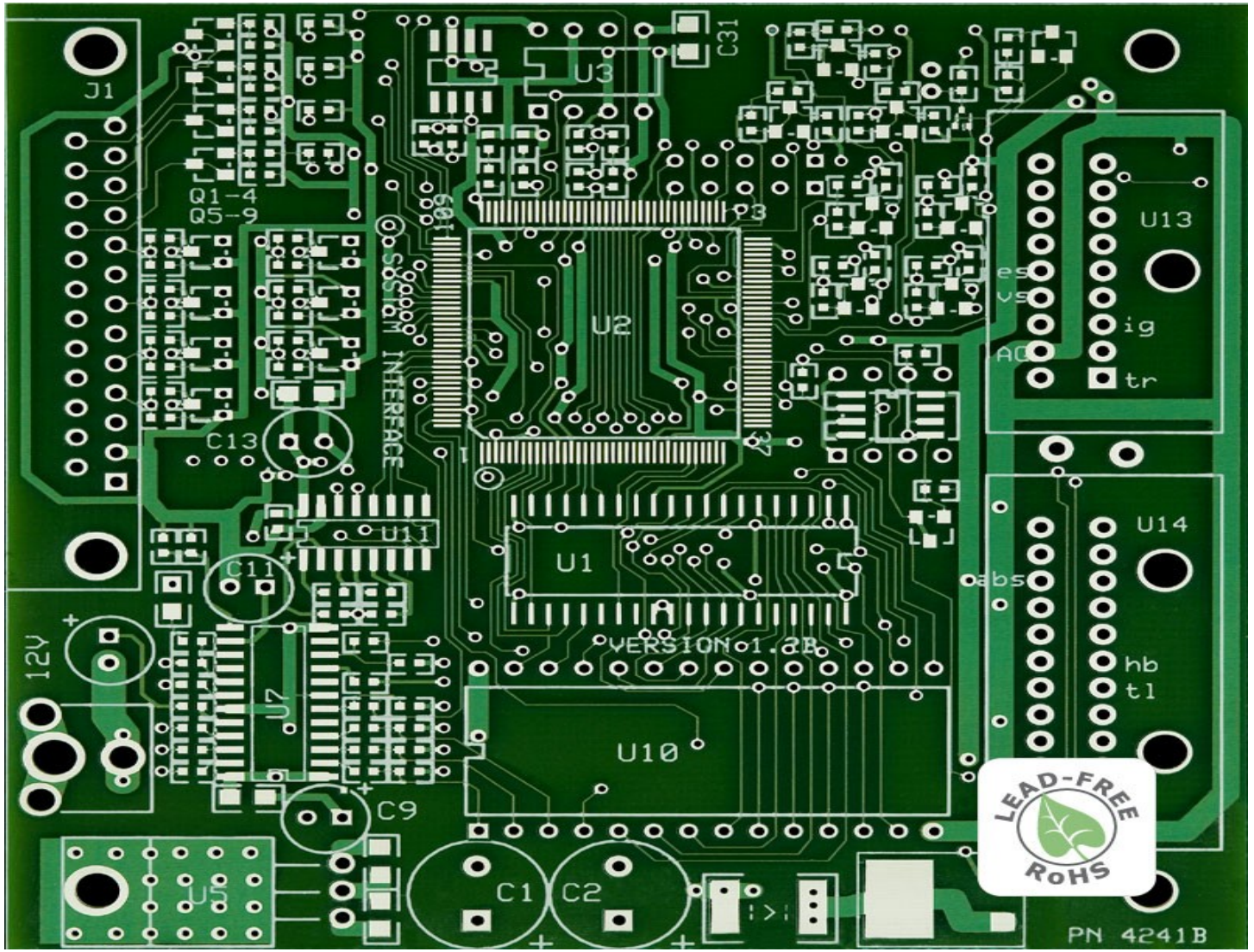
- Choose Framework
  - Ruby on Rails, Django, Node.JS, Java, C/C++, scala, etc.
- Choose cloud provider and model:
  - Amazon, Google, Rackspace, Azure...
- Choose operation mode
  - PaaS, IaaS, e.g. Google's App Engine vs. Google Compute Engine
  - Choose pricing model (fixed, flexible)
- Choose DB frameworks
  - Desired replication, SQL/NoSQL, etc.
    - MongoDB, DynamoDB, Cassandra, Oracle, SQL, Google Big Query, Redis, ...
- **Integrate Open Source (@see github et. Al , Ruby Gems, node modules etc.)**
  - When you want to do something – most of it might have already be done
- Adhere to client/server protocol
  - Usually some REST-ful API, JSON, XML these days
  - But also binary streams, and more.

# Modern Software Development Recap

- It's pretty much like building something out of LEGO:

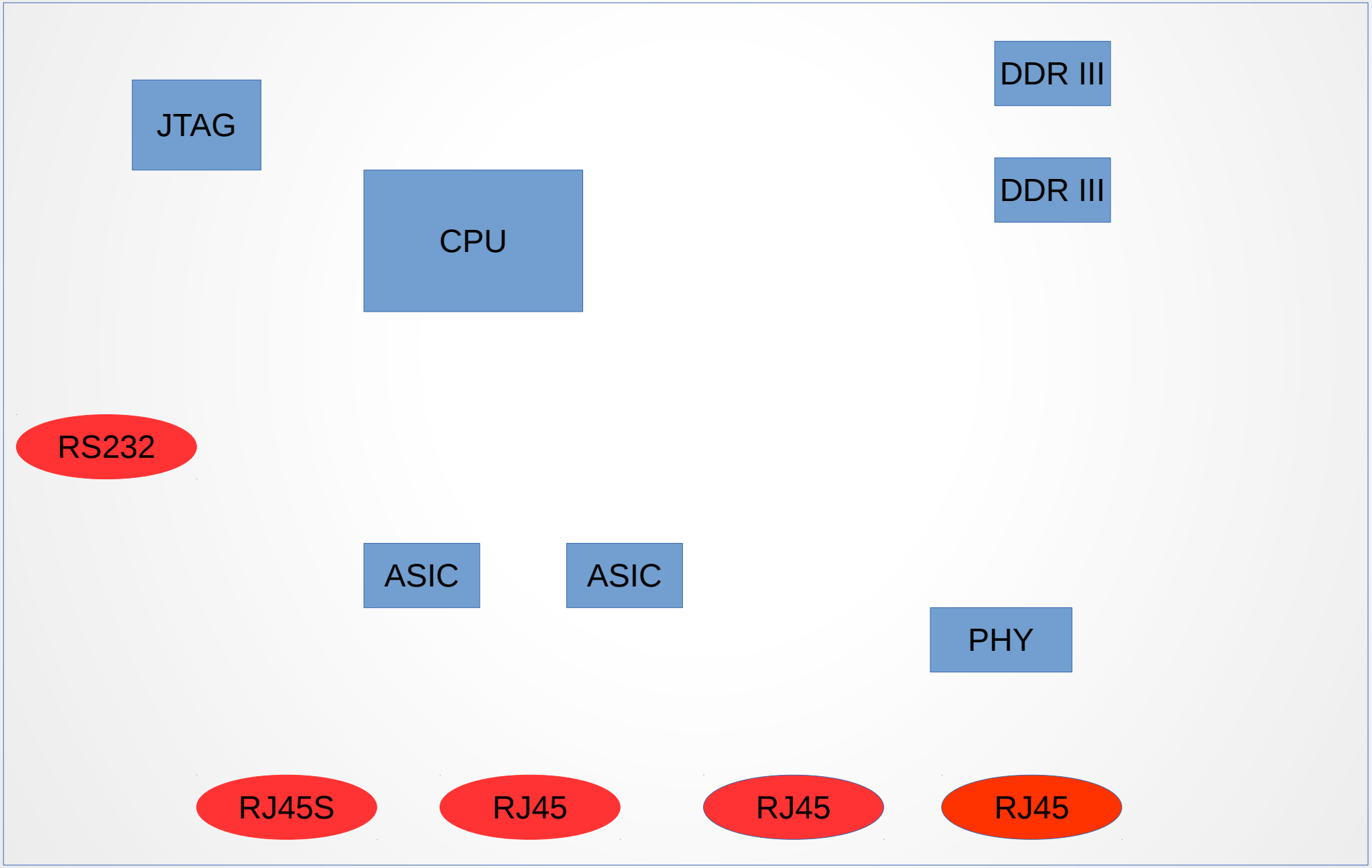


# Hardware Product 101: Building Hardware




http://www.expresspcb.com/GSpecs/PhotoProductionPCB_LE_800.jpg

# High level Requirements/Block Diagram



# The Manufacturing Process: A 50000 feet high overview

- Get Struck by an Idea and see if it's worth doing it...
  - Just like the SW phases.
- Deciding needed components and connectivity (netlist)
  - **To be bought:** e.g. A CPU. Memory. Components that can do some features
  - **To be manufactured**: e.g. a chip, a mezzanine board, an ASIC, and FPGA, or maybe even a core or two.
- **BOM**: Bill of Materials
  - Comparing specs, prices, alternatives of the required components.
  - **This is a lot like choosing the platform components in PaaS**

# The Manufacturing Process: A 50000 feet high overview

- PCB design, Place and Route
  - Decide the layout of the board
  - Decide the number of layers (i.e. noise isolation)
  - Decide wiring
  - Resolve constraints
- Send PCB design (a couple of files) to manufacturing (**Assembly)**
  - PCB in → Board out.
  - Assembly = PCB design + PCB (FR4) = BOARD
- Wait for boards to come back from manufacturing
- Integrate and Verify
- Pack it nicely into the desired form factor
  - Sell a product, not a board!

# The Manufacturing Process: A 50000 feet high overview

- The output of such a complex process is Hardware
  - Which may take time to manufacture
  - Which may take time to ship
  - Which may suffer natural or unnatural causes
  - Which may suffer... Bugs.
- This is usually **EXTREMELY** expensive, and hardly ever cost-effective before mass manufacturing.
- That is also a critical time, in which you don't really have something to work with
  - cf. to the immediate startup of a component in the Software Cloud case.

# The critical cost problem

- This actually applies to both SW and HW.
- You cannot determine scale ahead of time.
- If you overestimate – you will over pay.
- If you underestimate – you will over pay (You may pay with your business)
- Software industry is permitting:
  - Flexible plans. Start another Virtual Machine when you're close to limits
  - You can start with optimistic scale – and then downgrade
- Hardware is not
  - Once you manufactured – there is no way back (Talking about boards, not FPGA of course)

# Quick Comparison

- Making **soft**ware is (arguably, sometimes)
  **easy**

- Making **hard**ware is
  **HARD**
  - And risky.
  - And depends on too many parties
  - And takes time to ship

# But what if I need to manufacture a device?

- Assuming you understand the risks.
- And yet “I AGREE”.
- You're going to make the most amazing phone or wearable device ever made, and are confident you will make **billions**.
- You already told all your friends, colleagues, and some bloggers from the other part of the world, and they all worship you.
- And you hired (The PSCG) for consulting on the manufacturing process, and begging them to start working on a prototype.
- In the meantime, you dream of **getting funded.**
- And once you wake up you got 100M$ in 48 hours on KickStarter (Chapeau!!!)

- **How can product dreams come true, in such a hard, hardware world?**

# Hardware Product 201: The IoT Era

# Building A Modern IoT Product

- Assuming you were struck by an idea
- And want to “go for it”.
- You are aware of the Software Development Paradigms and
  - Know what you can get on the client side
  - Know what you can get on the server side
  - Know you can utilize the Cloud for IaaS/PaaS et. al.
- **You also know you are going to have to build your own hardware.**

# Building a Modern IoT product

- Hardware manufacturing takes time
- But in the meantime, you can decide your **Software** requirements, for that hardware:
  - Choose application suites (if relevant)
  - Choose interfaces and design SW/HW interaction
  - Choose an **Operating System**

# Building a Modern IoT product

- Modern products, are hardly limited by amount of memory, number of processors, etc.
  - It takes a “bad” programmer to abuse them.
  - But there are a lot of such programmers now. Because programming is **no longer a niche**.
- So they can run fully fledged, powerful GPOS.
  - Mobile OS are definitely GPOS
- The #1 constraint, in terms of operation is **power**.

# Building a Modern IoT product

- The important thing about the OS is how it interacts with your Hardware.
- And that is mostly the responsibility of the **Kernel**
- Moder Operating Systems running an open source, Linux Kernel:
  - Linux
  - Android
  - Tizen
  - FirefoxOS
  - WebOS
  - ...

# Rapid Prototyping

- As we previously mentioned, Creating Hardware takes time.
- In the mean time, you want to progress with the rest of the system.
- 80% of your hardware requirements (Wi-Fi, Cores, BT, Serial interfaces, memory, flash,...) are probably already implemented in modern, COTS boards
  - PandaBoard, BeagleBoard, Raspberry Pi, Banana Pi, DragonBoard, Freescale WARP, Intel developer reference platforms and so on
- And if something is not available – you can probably find something similar with matching pinouts on the web (digikey, avnet etc.)

# Rapid Prototyping

Which means you can actually do a significant part of the **board bringup**, **prototype** and show a **PoC** before you really started manufacturing the hardware!

# Rapid Prototyping Friends



# Hardware Manufacturing and Shipping

- PoC – Checked.
- Hardware functionality – Checked
- Packaging – Checked
- Now what's left to do is minor adjustments:
  - From your reference board
  - To your final form factor

# Rapid Prototyping Cycle

- Select the desired match:
  - HW components on reference/eval boards
  - Bootloader (available with those boards probably)
  - Operating System (available with those boards, you can reflash another one as per your choice)
  - Build System, Source Control, etc.
  - Application sets  and techonology
- Have all your teams working – before, or in parallel to HW manufacturing

# Shipping and Integration

http://www.collectorsquest.com/blog/wp-content/uploads/lego_ironman.jpg

# Thank You!

(Also for tolerating the ugly slides!)

**@ronubo**