A View from the Trenches: Embedded Functionality and its Impacts on multi-arch Kernel Maintenance

Bruce Ashfield – Principal Technologist – Wind River

ELC

February 2012

Introduction

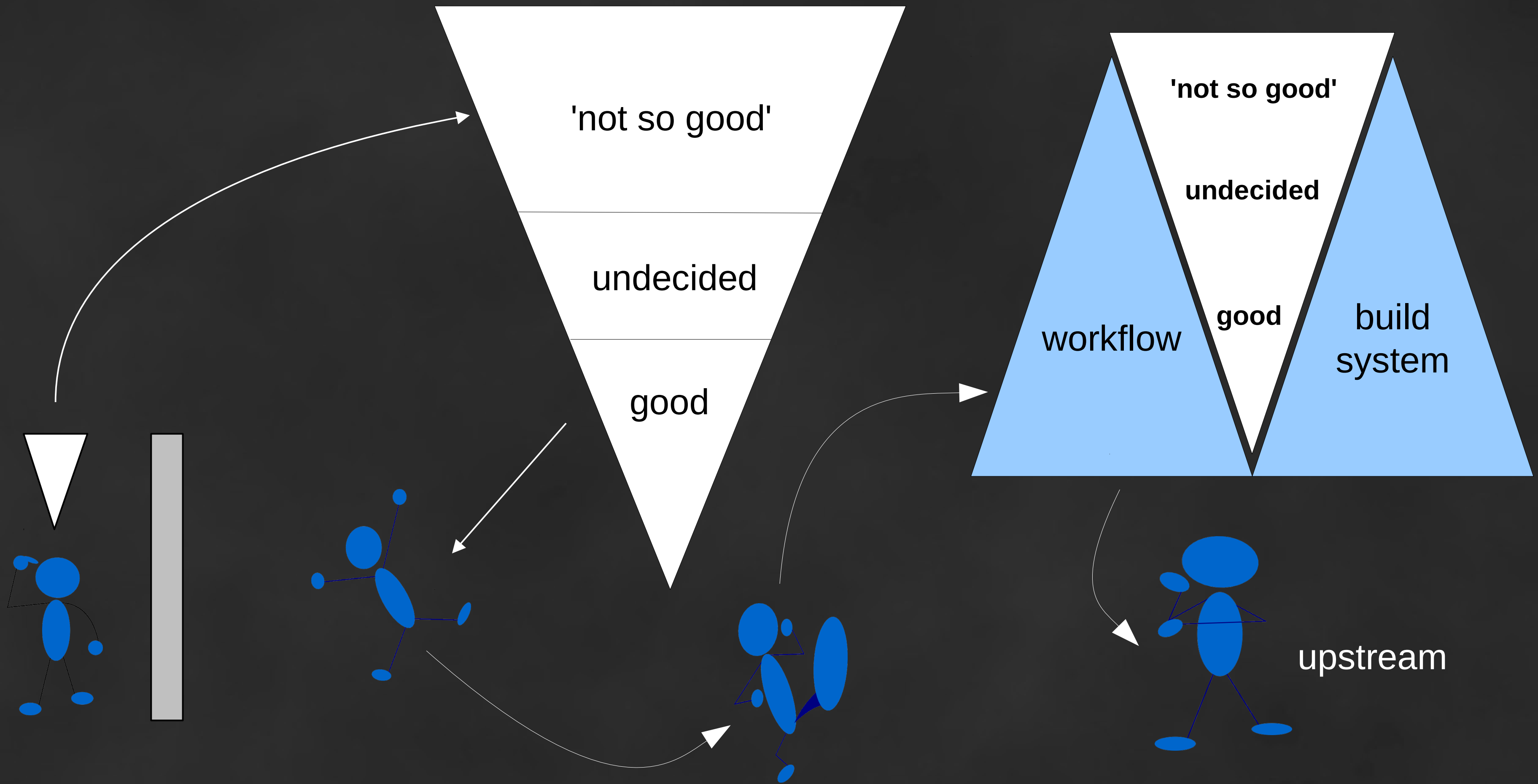
- (Embedded) maintenance is challenging .. and sometimes just 'different'
  - No single dominant reason
  - Code is code and a good change is a good change
  - No silver bullet, flexibility is key
    - Have a toolkit of tricks
- Experience can make a difference
  - 5+ years, 6 arches and 100's of BSPs
  - Many maintenance techniques later ..

Properties of an Embedded Changeset

- Vendor / expert driven
  - Low level, written by those that know the hardware
- Specific
  - Focussed development
  - Specific board, specific problem, specific kernel
- Potential for conflicts
- Not always (rarely?) developed with upstream in mind
  - Quality is typically 'good enough'
  - Reuse, maintainability and conformance suffer

- Given (tossed) to others to support and clean up
  - Developer and maintainer can have different priorities
  - Intersection is key

Anatomy of an Embedded Changeset



Change Lifecycle: High Level

1. Arrival

   triage and assess (@#$#@)

2. Merge

   Where? How ?

   Refactor and recycle

3. Maintain

   build, boot, regression test

4. Upstream

   not always possible

5. Carry forward / uprev
6. Repeat (goto #2)

Understand the Subject

- mechanics
  - manipulating and merging
- understanding
  - the goal
  - the change
- Look at the patches and learn the basics
  - consult as required
  - tune in: follow mainline and arch development

It's Merged .. Now What ?

- Does it work ?
  - Build coverage
  - Boot coverage
  - Self / feature tests
- Carry forward plan
  - Carry for as little time as possible
  - Upstream merge strategy
- Look for refactoring opportunities
  - Keep up to date with mainline evolution

Management Techniques: evolution

- Directories full of patches
  - ~20-100 patches, largely single variant
- Patch lists + tools
  - ~200-400 patches, a few variants
- Patch lists with intelligence
  - ~400+ patches, several variants
- Revision control + tracking
  - ~2000 patches, many variants
- Revision control + tracking + change control
  - ~20000+ changes, many variants and flexibility
- Ordering and stacking is important
  - Protect the 'hard' parts of the system
  - Allow the portable / Easy part to flex

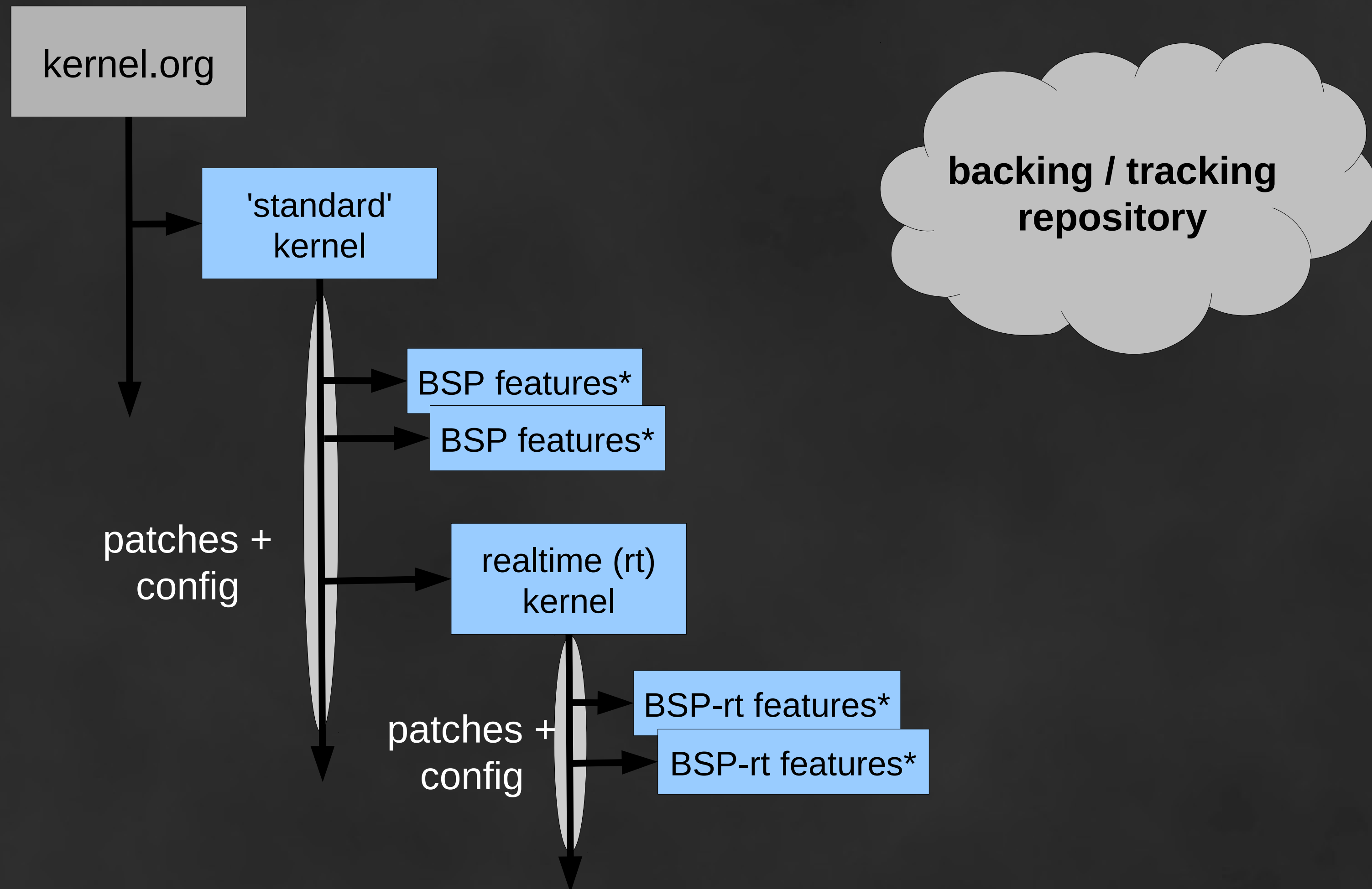
Tools & Techniques

- Goal: produce a clean and obvious change history
  - reproducible, extractable, maintainable and 'upstreamable'
- Contentious topic
- Techniques and workflow are as important as tools
- Use a SCM
  - git .. or something else
- Add some tools
  - git, quilt, guilt, stgit, topgit ...
- Resolve and merge conflicts
  - git, wiggle, merge tools ...
- Develop, build and test
  - Same environment and techniques as maintenance

Yocto Kernel Model

- Revision Control Based
  - hybrid model
    - patches backed by a SCM or a SCM backed by patches
    - fast forward and/or rebased
  - code and config are coupled
- Separate repository can track patches
  - tree can be rebuilt from scratch at any time
  - clear and obvious history
- Branches track incompatible / conflicting changes
  - isolation and control
- Manipulated using the tools of your choice
- Maintenance, development and build are integrated
- Has a complexity cost

Yocto Kernel Overview



Examples / War Stories

- Schedulers
  - EDF, BFS, CFS and O(1)
- Size versus flexibility
  - Linux tiny
- Extensive, but optional, functionality
  - preempt-rt
  - lttng
- Extensive and specific functionality
  - SDKs
  - cramfs linear XIP
  - grsecurity
- “Don't change that”
  - 8250.c

Directions & Solutions

- Tools are important, but not the answer
- Evolution and following of best practices
- More “upstream first”
- Collaboration
  - community kernels and consolidation
  - Sharing of tools and techniques
- Less work for everyone

Q & A