Software Development Processes
For Embedded Development
A checklist for efficient development using open-source tools

Gian-Carlo Pascutto
Senior Embedded Software Architect

Embedded evolution

- More processing power available for lower power and price
  - Evolution from 8-bit and 16-bit to 32-bit CPUs
  - Bigger flash memories for same price
  - High resolution LCD screens

- Increased customer expectations
  - Colorfull touchscreen GUIs
  - USB & SD support
  - Networking support

- A need for large software stacks as foundation for the actual application or appliance

- Building on FOSS is a good way to deal with this need

mind
embedded development
an ESSENSIUM division

Bigger stacks in embedded

- When extending your software stack (with FOSS), the requirements on your development process will increase
- Much more code to deal with
  - Including code which didn't originate in your organisation
- More developers to deal with
  - Including external ones from the community
- Software becomes more defining for your product
  - You'll become as much a software as a hardware company
  - Product is judged on the software features
- **Need to establish good processes for software development or improve existing ones to cope with this!**

mind
embedded development
an ESSENSIUM division

What's the goal today?

- Present a checklist for the development process
  - You probably have some areas already covered
  - You probably **don't** have **all** areas covered
- Share good practises established and observed at customers
- Focus on **free** tools
  - Cost
  - Extendability
  - Avoiding vendor lock-in
- Focus on embedded development
  - Will point out typical pitfalls
  - If regular software development is already your main business, you'd better not need this presentation ☺

mind
embedded development
an ESSENSIUM division

Outline

1. **Working with FOSS code**
2. Hardware and drivers
3. Version control systems
4. Tracking requirements and bugs
5. Documenting the software
6. The build process
7. Release management
8. Regression testing and validation
9. Code optimization & debugging

mind
embedded development
an ESSENSIUM division

1. Working with FOSS code

- FOSS advantages
  - Don't need to reinvent the wheel
  - No large investements or royalities needed
  - Ability modify all parts of the software to own need
  - No vendor lock in
- FOSS commoditizes the software foundation
  - Portability can help commoditize the hardware too
- Must think about where you add value
  - Hardware?
  - Customization?
  - Own IP?

1. Working with FOSS code

- License compliance as a policy
  - Need policies in place for
    - Feeding back code upstream
    - What licenses are acceptable where
  - Without them, developers will be too cautious, and you'll miss the advantages
  - Without them, developers can be too enterprising, and the product might end up depending on a GPL library where you really didn't want it
    - Remember where you add value

- License compliance as part of the process
  - You should make tarballs of your open source components as part of the release process, NOT as an afterthought
  - If you don't, you're guaranteed to end up violating a license sooner or later
    - Results can be costly: just ask Cisco

1. Working with FOSS code

- Working with FOSS = working with the community
- Even reporting bugs is giving back
  - Bug report from seasoned developer is aprox. 10 times as useful as one from a user
  - People using your software = positive feedback
- When you patch or extend things, send back upstream
  - If it gets accepted, rebasing will be easier for you
    - Keeping a lot of patches for yourself may make upgrading painful
  - If it doesn't get accepted, you'll be told why not
    - Bugs or design mistakes you'd rather know about
  - Gives information to upstream about what the people using his/her software want
    - Can make your job easier, too

mind
embedded development
an ESSENSIUM division

Outline

1. Working with FOSS code
2. **Hardware and drivers**
3. Version control systems
4. Tracking requirements and bugs
5. Documenting the software
6. The build process
7. Release management
8. Regression testing and validation
9. Code optimization & debugging

mind
embedded development
an ESSENSIUM division

2. Hardware and drivers

- Selecting the right hardware is critical
  - Proper driver support in your stack can make or break your embedded project
- So FOSS people should have a say!
  - Not just hardware guys
  - ....or management
- Remember working with the community?
  - How many people are using it?
  - “Worse” solutions with better support or larger communities can be easier to get working
  - This also applies to everything else you select

2. Hardware and drivers

- The levels of hardware support
  - No driver
    - Is there sufficiently complete documentation?

  - Binary driver
    - Basically as good as not having a driver at all
    - Can break on kernel config changes
    - Tends to be badly documented

  - Out of tree driver (unmaintained)
    - Quality can be bad, design mistakes, not up to date

  - Public git
    - Can be immature, more difficult upgrade path

  - Mainline
    - Frequency of updates?
    - Lots → Actively developed or still full of bugs
    - Few → Abandoned or working

2. Hardware and drivers

- When rolling your own kernel drivers...
  - Contentious area regarding licensing
    - Derived work of the kernel or not?
    - Binary kernel modules are controversial
  - What if a driver exposes significant trade secrets?
    - Or embarassing hardware bugs
  - Put as little as possible in the kernel
    - Kernel driver acts as a conduit
    - Have the actual driver work from userspace
    - Less licensing worries **and** better design

mind
embedded development
an ESSENSIUM division

Outline

1. Working with FOSS code
2. Hardware and drivers
3. **Version control systems**
4. Tracking requirements and bugs
5. Documenting the software
6. The build process
7. Release management
8. Regression testing and validation
9. Code optimization & debugging

3. Version control systems

- **Why version control?**
- Distributed vs. Centralized
- Nontrivial features
- Componentizing
- Integration

3. Version control systems

- Why?
  - Trace the history of code and features
    - Go back in time and reproduce an old version
    - Find out who wrote a piece of code – and why
    - See changes between versions
    - See removed things (no more commenting out)
  - Allow parallel development
    - Developers can work seperately and *merge* their work
    - Can work on new features for new versions and fix bugs for old releases
  - Place where developers can get the latest version

3. Version control systems

- Why version control?
- **Distributed vs. Centralized**
- Nontrivial features
- Componentizing
- Integration

3. Version control systems

- Centralized version control
  - RCS, CVS, Subversion, Perforce, ClearCase
  - Central repository to which everyone commits
- Distributed version control
  - Git, Mercurial, Monotone, Bazaar, Darcs, BitKeeper
  - Every developer has his/her own local respository
  - Synchronizes local repository with others

3. Version control systems

- Advantages of distributed version control
  - Allows working off-line
  - Load on server is split (faster)
  - No network required for most operations (faster)
  - Developers can experiment and commit locally
    - ...and clean up before committing final versions
    - Threshold to commit early is lower, lower risk of losing work
  - Automatic backups
- Disadvantages
  - Slightly harder to use
  - No monotonic versions (uses cryptographic tags)
  - Cannot revise history once a tree has been synchronized with others

3. Version control systems

- Speed matters more than you think
  - Interruptions of workflow and loss of concentration
  - Scaling in code size
    - It will for sure get worse as your projects progress
  - Scaling in number of developers
    - Let's hope it gets worse ☺
- Distributed systems are faster and scale better
- Embedded projects are often very big in terms of versionable source size
  - Toolchains, kernel, bootloader, libraries, applications, ...

3. Version control systems

- The only reason not to use a distributed system is misconceptions!
  - Misconception 1: *A distributed system gives less control because there is no central repository.*
  - Truth: You can designate central repositories with a distributed system, and force developers to synchronize with them.
    - Easy to build hierarchies for subsystems and code review, too.
    - Field proven in enormous project: Linux kernel
  - Misconception 2: *A distributed system might encourage developers to keep their work local and not publish or rebase it often.*
  - Truth: They can do this with centralized systems too, and it's easier to lose work if they don't commit **at all**.

3. Version control systems

- Why version control?
- Distributed vs. Centralized
- **Nontrivial features**
- Componentizing
- Integration

3. Version control systems

- Nontrivial features (that make your life easier)
  - Interactive add
    - Select chunk by chunk what to commit
    - Allows splitting up a big patch in features
  - Stashing
    - Kind of a temporary commit
    - Saves the state of tree, restores clean state
  - Cherry-picking
    - Pick individual commits from a branch onto another branch
  - Bisecting
    - Allows finding the exact commit that introduced a problem

3. Version control systems

- Why version control?
- Distributed vs. Centralized
- Nontrivial features
- **Componentizing**
- **Integration**

3. Version control systems

- Componentizing
  - Don't put the entire source in one repository
  - Analyze which parts you want to *freeze* individually
  - Can then branch / tag individual components
  - If upstream uses versioning, clone and track them (git!)

- Integration
  - Develop features in branches, merge to the trunk
  - The trunk must **always** compile
  - Integrating (merging) should be done by the developer who knows the component best
  - Consider doing code reviews

3. Version control systems

- We recommend
  - git
    - Linux kernel, GNOME, Perl, Qt, Samba, X Windows, …
  - Mercurial (Hg)
    - Mozilla, Java, Solaris, OpenOffice, Symbian, …
- Windows support for git is so-so, but rapidly improving
  Mercurial works fine
- Unix based: choose git

mind
embedded development
an ESSENSIUM division

Outline

1. Working with FOSS code
2. Hardware and drivers
3. Version control systems
4. **Tracking requirements and bugs**
5. Documenting the software
6. The build process
7. Release management
8. Regression testing and validation
9. Code optimization & debugging

4. Tracking requirements and bugs

- Bug tracking systems
  - Keep list of known issues and how to reproduce them
  - Missing features are bugs too!

- Project management tool
  - Can assign bugs to milestones and releases
  - Can assign bugs to developers

- Getting most out of them
  - Understand dependencies
  - Can adapt bug states to your organisation
  - Keep the link between commits and bugs

trac
Integrated SCM & Project Management

- Recommendation: Bugzilla (complex, powerful), Trac (simpler)

mind
embedded development
an ESSENSIUM division

Outline

1. Working with FOSS code
2. Hardware and drivers
3. Version control systems
4. Tracking requirements and bugs
5. **Documenting the software**
6. The build process
7. Release management
8. Regression testing and validation
9. Code optimization & debugging

5. Documenting the software

- Code documentation
  - Low level, implementation details can be documented inline /* */
  - Don't document the obvious, but document hidden assumptions and side-effects (don't document what you can see – document what you can't see)
  - Assert() is an underestimated documentation tool
  - Functions, API's and class hierarchies can be documented inline with tags and then processed into full-blown API docs
  - Recommendation: Doxygen
  - Even if you don't use it, some editors understand the format!

- Design documentation
  - High level, coding, organisation or structuring guidelines
  - Should be easily accessible, easily editable, and versioned
  - Recommendation: use wikis like MediaWiki, or Trac

mind
embedded development
an ESSENSIUM division

Outline

1. Working with FOSS code
2. Hardware and drivers
3. Version control systems
4. Tracking requirements and bugs
5. Documenting the software
6. **The build process**
7. Release management
8. Regression testing and validation
9. Code optimization & debugging

6. The build process

- Quick starting
  - New developers arrive, to help make a critical deadline
  - How long does it take for a new developer to make a working build?
    - How long to set up a development PC?
    - How long to get an environment similar to other developers?
      - Do they even remember how?
      - Are the same sources/versions still available?
    - How long to install all dependencies?
    - How long to figure out how to set the build environment?
      - Need to install all crosscompilers for all supported devices
  - This time is 100% lost!
  - ...and it happens more than you would like
    - What if a laptop is stolen?
    - What if a harddisk crashes?

6. The build process

- Quick starting
  - Streamline the process
    - Virtual machines with preconfigured build enviroments
      - Recommendation: VirtualBox
    - Shellscripts that automate most of the setup
    - Make sure everything is documented

6. The build process

- Making the software
  - Don't rely on IDE's for the build process
    - Will run into automation problems sooner rather than later
  - Wide choice of buildsystems
    - Makefiles (manually constructed)
      - *Relatively* easy to understand and learn
      - No support for advanced configuration (porting, libraries, …)
    - automake / autotools
      - Very common in open source software
      - Very powerful configuration
      - Nobody really understands how they work
      - Not so well supported on Windows

6. The build process

- SCons
  - Python based
  - Well-featured, but fairly slow
  - Widely used in industry

- CMake
  - Own scripting language
  - Well-featured
  - Outputs native buildsystem (Makefiles, Visual Studio projects)
  - Used by some major open source projects
  - Commercially supported

- Recommendations
  - The jury is still out, but CMake appears to be winning
  - You will want to write a wrapper
    - To interact with your version control system
    - To handle whatever further automation you need
  - Write it in something most of your developers will be familiar with
    - Probably shellscript, Perl or Python
    - **Not:** Fashionable scripting language of the day

6. The build process

- Crosscompiling
  - Distinction between host CC and target CC
  - Add to, don't replace CFLAGS, LDFLAGS, …
  - Export usefull stuff: endianness, kernel version, architecture, libc style
- Autotools crosscompiling
  - Override CC, CXX, AR, LD, etc, set --host and --target vars
  - Might need to disable faulty tests
  - Importance of DESTDIR for staging
- CMake crosscompiling
  - Good support via toolchain configuration files
    - Variables set at top level to find system, toolchain and libs
- Staging areas
  - Prevent headers from cluttering the flash filesystem

mind
embedded development
an ESSENSIUM division

Outline

1. Working with FOSS code
2. Hardware and drivers
3. Version control systems
4. Tracking requirements and bugs
5. Documenting the software
6. The build process
7. **Release management**
8. Regression testing and validation
9. Code optimization & debugging

7. Release management

- Reproducibility
  - Make sure the builds can be 100% reproduced
  - Tag all components in the VCS
  - Make sure the build is from a clean state
    - Virtual machines and dedicated buildservers can help
- Store *everything*
  - Store all builds
  - Store the debug information to get backtraces
  - Diskspace is cheaper than developer time
- Automated nightly builds
  - Build as much configurations, variations and platforms as you can
  - Detect failures on exotic combinations early

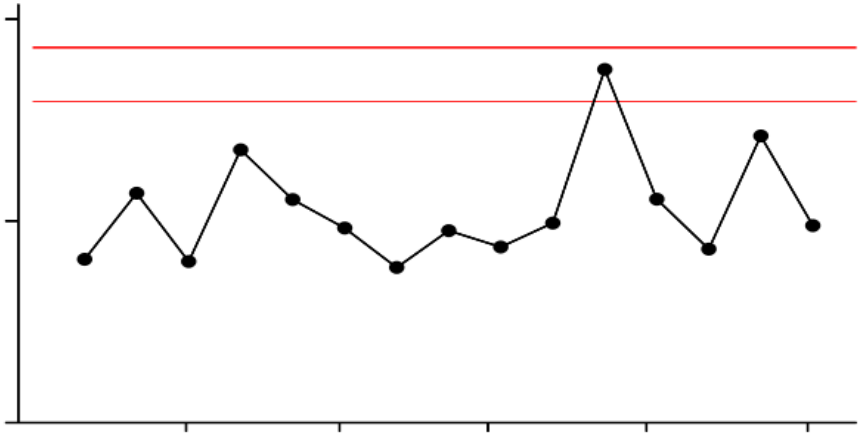
mind
embedded development
an ESSENSIUM division

Outline

1. Working with FOSS code
2. Hardware and drivers
3. Version control systems
4. Tracking requirements and bugs
5. Documenting the software
6. The build process
7. Release management
8. **Regression testing and validation**
9. Code optimization & debugging

8. Regression testing and validation

- Start testing immediately, should not be an afterthought
- Automated testing
  - Couple with nightly builds
  - Alert on regressions



- Keep statistics
  - Code size
    - Make sure it fits into flash
  - Performance
    - Boot time can be critical in embedded!
  - Memory usage
    - Leaks are deadly for long running devices

mind
embedded development
an ESSENSIUM division

Outline

1. Working with FOSS code
2. Hardware and drivers
3. Version control systems
4. Tracking requirements and bugs
5. Documenting the software
6. The build process
7. Release management
8. Regression testing and validation
9. **Code optimization & debugging**

9. Code optimization & debugging

- Simulation enviroments
  - Make sure you can crosscompile to & run on x86
  - Need to stub your hardware devices (and maybe the OS)
  - Enables you to run the software on the development system
  - Vastly increases the number of debugging and analysis tools you can use

- Debugging, tracing & logging
  - GDB on target
    - Can be tricky, gdb is nontrivial to crosscompile
  - Remote debugging with gdbserver
    - If you have networking...
    - Keep nonstripped versions of the binaries and libs
  - Valgrind
    - On host

9. Code optimization & debugging

- Code size optimizations
  - -Os, strip and sstrip
  - Platform specific switches
  - For C++
    - Know the compiler options (-fno-rtti, -fno-exceptions, arch specifics)
    - Beware of templates (see Qt)
  - Avoid needless dependencies (./configure –disable-foo)
  - Can be worthwhile to strip packages manually

- Profiling
  - **First** profile, **then** optimize
  - Profiling tools
    - OProfile
    - Performance counters
    - Valgrind, (K)Cachegrind, Callgrind
  - Availability depends on architecture. If you can run on x86...

mind
embedded development
an ESSENSIUM division

Conclusions

- FOSS is a great opportunity for embedded development...
  - Allow focus on the application, not the stack
  - Allow fast and low cost prototyping
  - Freedom allows you to stay in control
- ...which presents some challenges...
  - Much more code to deal with
  - From various external sources
  - That will define look, feel and quality of the product
- ...that can be efficiently dealt with!
  - By using proper software development processes

mind
embedded development
an ESSENSIUM division

www.mind.be

www.essensium.com

http://www.mind.be/content/100206_SW_Development_Best_Practices.pdf

Essensium NV
Mind - Embedded Software Division
email : info@essensium.com