Lessons learned from CE embedded Linux

9 trivial tips given from early Linux adaptor

Hisao Munakata

Linux Foundation Consumer Electronics working group

September 19th 2012, ALS2012

Agenda

Who am I ?

- From embedded SoC provider company Renesas
- Linux Foundation CE[1] working Gr. Steering committee member, LF/CEWG Architecture Gr. co-chair
- One of LF/CEWG LTSI[2] project initial proposer
- ALS2012 (and others) steering committee
- At my company, I had been encouraging my team developers to send a patches upstream
- Also I have supported various CE customers who develop digital-TV, Blu-ray recorder and Smart-phone

[1] CE = consumer electronics
[2] LTSI = Long Term Stable kernel Initiative

What was your original intention to adopt Linux ?

New technology is not always better than previous

- Analog TV channel selection was much faster
- Feature phone battery lasts longer than Smartphone
- Traditional IVI device could be designed to boot-up faster

If you simply want to run the same application in the same way, adoption of Linux does not make any sense. Clarification of your goal with Linux is essential.

Linux might be mandatory option if you want to run. . .

- Social service application like Twitter, Facebook
- Play various open contents/library like Youtube
- Connect IVI device as a mobile node of cloud computing

History of struggles

We, Consumer Electronics (=CE) industries, heavily depend on the power of Linux now. However, we experienced various troubles at the early stage of Linux adoption.
So it would be a lovely idea to hear previous struggles not to repeat the same experiences in the automotive industry.

CE industry problems with Linux adoption

- Management thought Linux is low-cost replacement
- Developer thought they are consumer (=user) of Linux
- They defined `CE-Linux spec` and asked the community to develop it, but it never happened. The community requested CE industry to submit patch proposal instead.
- Also they wanted to reuse existing RTOS asset on Linux
- Finally they start creating unique kernel for embedded

Important findings after severe struggle

Recommended approach to utilize Linux

- **Not to create your own fork** because it would create enormous burden at next kernel migration
- **Work with the community** developer because the code you want to develop might be already written by them
- **Adopt** proven development procedure like **git**
- **Refer** public code repository **(git-hub, ML and Wiki)**
- **Upstream** developed code as much as possible

One practical idea is having opensource team in your company who can work with community developers.
They can be a bridge between business and opensource.

Linux vs RTOS, how technically differs

Technical essence of Linux design

Essence of Linux is **automatic coordination by heuristics,** so you should write code to fully utilize to this mechanism. If you try to override this, it will break Linux built-in system coordination.

	Linux	RTOS
scheduling	**heuristics**	deterministic
resource allocation	flexible	fixed
execution order	runtime coordination	predictable
MMU (virtual address)	support by default	not support
multi-processor	support by default	not support
virtualization	yes (KVM, others)	not support
address description	abstracted	absolute value

Statistics of the latest Linux (is huge ! asset)

Table : Statistics of Linux kernel 3.5.3 (released 2012.8.26)

number of code lines	**15,598,058**
number of files	39,097
number of registered maintainer	1,255
number of newly added patches	9,534
update cycle	every 75 - 80 days
configuration items	5,386
history of development	over 20 years

No single company can do the same scale of development

Smartphone, DTV, Enterprise server and Super Computer runs Linux kernel image generated from **exact single source code** with different configuration setting. So handling Linux kernel source is not a trivial thing. Be careful not to break system coordination.

1. kernel space and user space isolation

Porting fully debugged RTOS code to Linux

Keep original code as possible

- access h/w register via **mmap** from user space
- inherit original RTOS code structure as much as possible
- not fully utilize Linux kernel service

Adopt POSIX API for isolation

- rewrite device driver to access real hardware
- isolate user/kernel space by **standard POSIX**
- design for multi threaded program execution

1. kernel space and user space isolation

Porting fully debugged RTOS code to Linux

Keep original code as possible

- access h/w register via **mmap** from user space
- inherit original RTOS code structure as much as possible
- not fully utilize Linux kernel service

Adopt POSIX API for isolation

- rewrite device driver to access real hardware
- isolate user/kernel space by **standard POSIX**
- design for multi threaded program execution

Recommended choice is

Clean POSIX space isolation is essentially important

2. device driver API design

Writing a device driver for on-chip/on-board device

Full functional driver

- **designed to support all device capabilities** at initial release
- adopt **original ioctl** to control original function
- kept as in-house code (no mainlining attempt)

Limited functional driver

- **compatible with Linux standard framework**
- to add support for IP unique function, framework code is enhanced
- code is mainlined

2. device driver API design

Writing a device driver for on-chip/on-board device

Full functional driver

- **designed to support all device capabilities** at initial release
- adopt **original ioctl** to control original function
- kept as in-house code (no mainlining attempt)

Limited functional driver

- **compatible with Linux standard framework**
- to add support for IP unique function, framework code is enhanced
- code is mainlined

Recommended choice is

Framework compatibility is preferable than coverage

3. task execution order (priority) control

Tune kernel scheduler setting to improve response

Entrust Linux scheduler

- rely on kernel scheduler to minimize latency
- not attempt to elaborate tune of priority setting
- release CPU resource while waiting ready status

Fine tune scheduling setting

- fine tune process priority by **nice** setting
- adopt **rt-scheduling policy** to secure execution
- utilize busy wait loop (**udelay**) for fine tune

3. task execution order (priority) control

Tune kernel scheduler setting to improve response

Entrust Linux scheduler

- rely on kernel scheduler to minimize latency
- not attempt to elaborate tune of priority setting
- release CPU resource while waiting ready status

Fine tune scheduling setting

- fine tune process priority by **nice** setting
- adopt **rt-scheduling policy** to secure execution
- utilize busy wait loop (**udelay**) for fine tune

Recommended choice is

Do not block (hold) CPU by excessive timing manipulation

4. resource (memory) allocation

Allocating large contiguous memory for video buffer

Static allocation

- reserve memory statically at out of side kernel management
- program want to reserve local buffer space
- random alloc/dealloc

Dynamic allocation

- kernel memory allocation, low-memory detection
- **memory pool** to reduce page fragmentation
- adopt modern memory allocator like **CMA**

4. resource (memory) allocation

Allocating large contiguous memory for video buffer

Static allocation

- reserve memory statically at out of side kernel management
- program want to reserve local buffer space
- random alloc/dealloc

Dynamic allocation

- kernel memory allocation, low-memory detection
- **memory pool** to reduce page fragmentation
- adopt modern memory allocator like **CMA**

Recommended choice is

Speculative resource allocation works remarkably well

5. binary components (binary blobs)

Handling binary component (library) with Linux code

Avoid binary code

- **kernel code must be GPL** (source available)
- should have source of userland library as well
- source is mandatory if you want to generate **symbol for debug**

Accept binary code

- IP/middleware vendor ask to use binary code
- When environment (kernel, toolchain) update is needed, **it may hit serious difficulty of whole system rebuilt**

5. binary components (binary blobs)

Handling binary component (library) with Linux code

Avoid binary code

- **kernel code must be GPL** (source available)
- should have source of userland library as well
- source is mandatory if you want to generate **symbol for debug**

Accept binary code

- IP/middleware vendor ask to use binary code
- When environment (kernel, toolchain) update is needed, **it may hit serious difficulty of whole system rebuilt**

Recommended choice is

Avoid binary code as possible to eliminate problems

6. code control process

patch review / integration / tracking / debug flow

Sustain in-house method

- keep use traditional way
- **merge without maintainer review** then test and debug
- improper use of git (**too large commit, no log**)

Innovate open source way

- allocate **in-house tree maintainer** who review all codes before merge
- **use git for code control**
- review public repo, BTS, ML discussion information

6. code control process

patch review / integration / tracking / debug flow

Sustain in-house method

- keep use traditional way
- **merge without maintainer review** then test and debug
- improper use of git (**too large commit, no log**)

Innovate open source way

- allocate **in-house tree maintainer** who review all codes before merge
- **use git for code control**
- review public repo, BTS, ML discussion information

Recommended choice is

Adopt and utilize OSS development methodology

7. kernel version selection (LTS / LTSI)

Security maintenance period of Linux kernel vary

Simply adopt given kernel

- simply use SoC vendor's kernel adopted in BSP
- distribution / platform defined (Android, Genivi)
- **stick on the old version**

Conscious of version

- adopt latest version
- chose community LTS
- chose **LF/CEWG LTSI (Long-Term Stable kernel for Industry)**

7. kernel version selection (LTS / LTSI)

Security maintenance period of Linux kernel vary

Simply adopt given kernel

- simply use SoC vendor's kernel adopted in BSP
- distribution / platform defined (Android, Genivi)
- **stick on the old version**

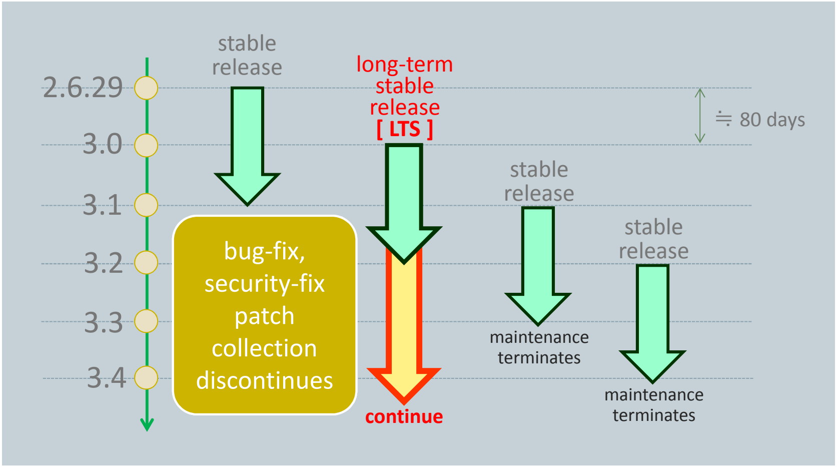
Conscious of version

- adopt latest version
- chose community LTS
- chose **LF/CEWG LTSI (Long-Term Stable kernel for Industry)**

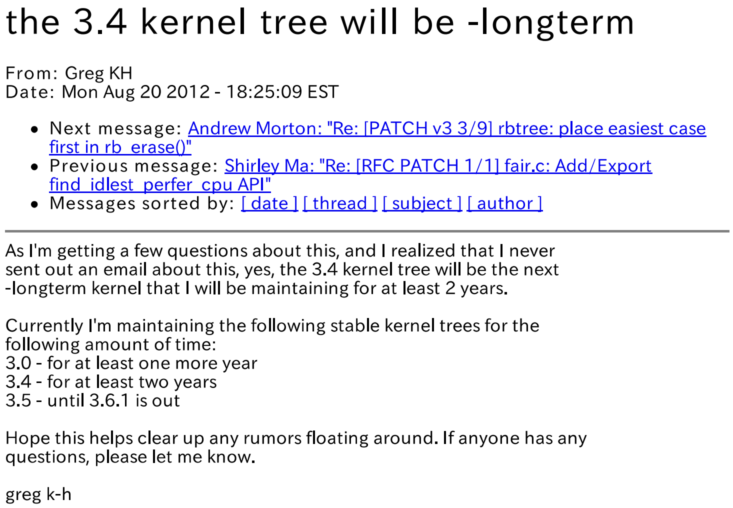
Recommended choice is

Ask SoC/distro/integrator to adopt LTSI in their BSP

Community kernel maintenance scheme



3.4 is next LTS (and LTSI) version



the 3.4 kernel tree will be -longterm

From: Greg KH
Date: Mon Aug 20 2012 - 18:25:09 EST

- Next message: Andrew Morton: "Re: [PATCH v3 3/9] rbtree: place easiest case first in rb_erase()"
- Previous message: Shirley Ma: "Re: [RFC PATCH 1/1] fair.c: Add/Export find_idlest_perfer_cpu API"
- Messages sorted by: [date] [thread] [subject] [author]

As I'm getting a few questions about this, and I realized that I never
sent out an email about this, yes, the 3.4 kernel tree will be the next
-longterm kernel that I will be maintaining for at least 2 years.

Currently I'm maintaining the following stable kernel trees for the
following amount of time:
3.0 - for at least one more year
3.4 - for at least two years
3.5 - until 3.6.1 is out

Hope this helps clear up any rumors floating around. If anyone has any
questions, please let me know.

greg k-h

8. code upstreaming

Is there any value / demands to do upstream ?

No needs for industry user

- simply passive Linux user
- no chance/intention to join upstream development
- can improve kernel inside a company if needed
- **hold in-house patches**

Sure, it certainly helps

- actively feedback bug info to distro/integrator
- ask SoC to upstream device support code
- want to **minimize delta against upstream code**

8. code upstreaming

Is there any value / demands to do upstream ?

No needs for industry user

- simply passive Linux user
- no chance/intention to join upstream development
- can improve kernel inside a company if needed
- **hold in-house patches**

Sure, it certainly helps

- actively feedback bug info to distro/integrator
- ask SoC to upstream device support code
- want to **minimize delta against upstream code**

Recommended choice is

Many companies start contributing upstream. Why ?

9. work with distribution

Maintain binary compatibility of Linux application

It is a part of SIer's job

- **simple headless system**
- no OSS middleware
- no OSS library
- no OSS application
- have full source code

SI integrate code from distro

- want to utilize binary distro like PC Linux
- **keep system-wide binary compatibility**
- require security update for userland program

9. work with distribution

Maintain binary compatibility of Linux application

It is a part of SIer's job

- **simple headless system**
- no OSS middleware
- no OSS library
- no OSS application
- have full source code

SI integrate code from distro

- want to utilize binary distro like PC Linux
- **keep system-wide binary compatibility**
- require security update for userland program

Recommended choice is

Distro manages interdependency of OSS packages

Conclusion

- Essence of Linux is automatic coordination by heuristics. It is crucial to write the correct code that can work with such automation mechanism. Code originally developed for RTOS need to be rewritten to comply with Linux environment.
- CE industry had struggled to find the right approach to utilize Linux environment. Here, introduced **9 tips** for your reference of early CE Linux. Please do not reproduce the same mistakes again and become successful Linux adaptor.
- Linux is a massive asset contains over 15 million lines of code. It is pretty sure that migration of kernel version needed to use new function. It is essential to care for **future version upgradability and code reusability** when editing the code.