Configuration (and bugs)

KISS Principle

Simple approaches are more robust than complex ones

- You can fully understand the solution
- ... which automatically leads to fewer bugs

Bugs grow with the square of the lines of code

- Thus, modular designs feature fewer bugs than monolitic ones
- It's always a trade-off between modular design and speed

Every "if" statement doubles the possible code paths

- Does it mean that bugs grow exponentially with conditionals?
- Beware of conditionals. They bite back. Always.
- Every code path must be tested

Sometimes complex wins over simple

A single complex solution is better than many simple ones

- If everybody uses the same code, everybody wins
- Even if there may be some performance cost
- Sometimes the compiler can optimize away complexity
  - __builtin_constant_p() often helps
  - or structures that downgrade to identity assignments

Example: Linux virtual memory

- The various CPU types organized their pate tables in 1, 2, 3 or 4 levels.
- Linux code is working with 4 levels, always.
  - The same code is used for all CPU architectures
  - Code is laid out so to optimize away some levels

Design a generic API that covers all cases

- This avoid conditionals in the calling code
- Then, some back-end may miss some features and error out

Build time configuration

Every more-than-trivial project needs configuration

- The target environment may be X or Y
- The host environment may be A or B
- And some user choices may apply
  - You need to include or remove some features
  - You need to enable or disable some diagnostics

Build-time configuration is the source of countless bugs

- Some configuration is rarely built
  - Some configuration is never built, actually
  - Un-polished code rusts, and then fails if you run it
- Some combination is not even allowed

Please limit the number of options

- Better, arrange for all options to be build-tested every time

This is why in fsmos I try to build all source files

- The linker then discards what is not used
- If your build takes minutes, or hours, please reconsider

The auto-tools way (a quick rant)

```
./configure --with-this --without-that --option=foo
```

A zillion languages

- configure.ac is written in M4
- configure is unadorned sh (= binary file)
- Makefile.am is written in god-knows-what
- ...

If it breaks, you are lost

No way to ship a configuration to the user

- You can ship a over-long command line
- Actually, your developer rarely offers serious built-time config
- Which may be a plus: fewer options means fewer bugs

The Kconfig way

Kconfig/Kbuild is a complex beast

- It is a set of Makefile rules and dependencies
- Somebody complained that instead of augmenting make, we should replace it
  - Those developers eventually surrendered and used Kconfig

But the Kconfig language is simple

- It is documented in the kernel sources
  - Documentation/kbuild/kconfig-language.txt
- But it's so easy that for most things you can just copy other Kconfig files

You can configure with any out of a range of tools

- config, menuconfig, nconfig, qconfig, gconfig, ...

The output is a text file

- You can ship configuration examples
- And you easily apply a predefined configuration

Then, Kconfig shapes the Makefile

Using the "y" trick you turn 3-lines conditionals into 1 line

- obj-$(CONFIG_FOO) += foo.o
- cflags-$(CONFIG_BAR) += -DBAR

Kconfig appears to suggest use of #ifdef/#endif

- Please avoid #ifdef, it's the source of most bugs in C

With proper Kconfig values (int, not bool) you can do better

- Remember that no code is emitted for "if (0)" blocks

If any, can turn #ifdef into a constant in your own file

Possibly, use the compiler's help

- __builtin_constant_p() for example

Kconfig useful rules

make config

make menuconfig

- **Asks questions your should reply to**

make defconfig (*_defconfig)

- **Apply a predefined configuration**

make oldconfig

- **Applies current .config, asking only unanswered questions**

More options, not in the current code base:

- **make randconfig**
  - **Randomize, to build-test in a loop**
- **make allyesconfig**
  - **All yes, where possible**
- **make allnoconfig**
  - **All no, where possible**

Emitting errors

In a project, you can fail in several ways:

- Fail at build time (compilation error)
- Fail at init time (early run-time)
- Fail at run-time
- Misbehave with no hard failure

A build failure is to be preferred, whenever possible

- You save a lot of time if you error out early

How to fail at build time

- Complain with #warning/#error (on condition)
- Overflow ram/stack at build time (on mishap)
- Refer to an undefined symbol (on condition)
- Create an array of negative size (on condition)

```
#define BUILD_BUG_ON(condition)
        ((void)sizeof(char[1 - 2*!!(condition)]))
```

Failing at run time

A run-time failure is worse than a build-time failure

- You need to install and run the binary to just fail...

Failing at init time is preferred

- ... if you are unable to fail al build time
- Init time is not performance-critical *at all*
- If RAM/flash allows, please be verbose in your messages

Sometimes you can trust your caller and avoid checks

- For example, gpio_set/gpio_get, after checking at init time
- Or you can offer the double API:
  - __some_function() doesn't check
  - some_function() checks arguments, and is slower

Any "assert" or "panic" in a critical path is a cost

- Assertions can be a build-time option
- Or you can think about a faster way to do BUG()

Diagnostics

Never remove diagnostics

- You can't rebuild and reflash just to debug
- Especially if the problem happens rarely

Diagnostics can be disabled, but should not be removed

- Reporting to the user is the most time-consuming task
- You can enable diagnostics on demand
- Still, having two different performance figures is bad
  - Sometimes, you can leave everything on all the time
  - Which save a scaring "if" as well
- Collecting information is useful anyways
  - You may save diags to a hidden log file

And never say never

- If RAM/flash is an issue, you can choose to build without diagnostics
- A diag-less build may fit a smaller CPU model
- A bugless system requires no diagnostics at all

Testing

You should test all of your lines of code

- Especially error paths
  - Really: fake all errors and check the outcome
  - Whe you have a bug, you can't debug your error management too
- Any "if" is one test-run more in your way to delivery

Which doesn't mean I support "test-driven programming"

- To be honest, I see the rationale behind TDD
- But the true gospel of TDD kills creativity

And please remember to test corner cases

- Run-time is the best testing environment
- If come scary code is unlikely to run, make it frequent
  - Scared about overflows? Start at 0xffff
  - Scared about 64-bit ops? Shift your data up
- "I hope it won't happen" leads to failure