Initcalls

The ugly "setup.c" file

Our current setup.c is as ugly as vendor ones

- It refers to several other drivers (timer, uart, pll, ...)

It must be augmented any time a new driver is added

Worse, such additions may depend on what we configure

- Worse, it may need to depend on what main() does

What we really need, is a simpler approach

- The init function should be part of the driver
- If the driver is not built, its initialization would not be built either
- If the driver is not used, its initialization should not be run

Initcalls, the mechanism

The simple solution is defining a special ELF section

- Every driver defines a structure in that ELF section
- The linker builds an array for us
- The linker script provides the usual names for begin and end

Considering the linking mechanism, this is what happens

- Each object files has code and data section
- It may also have an init section

Then, if the linker finds an unresolved symbol in this object file

- It picks the whole file for the final link
- Eventually, it may discard unused sections (--gc-sections)

Two fine points to consider

The compiler discards any unused code and data

- This includes all static functions you don't call
- And also data structures you don't refer to.
- In the blatant case, a warning is reported
  - "function F defined but not used"
  - "unused variable V"

Constants known at build time help in this decision

- This is why "if (0)" is good trick
- No warning about "unused code/data" is reported
- But still the function/data is discarded

Additionally, the linker eventually discards unused sections

- This only if "--gc-sections" is requested

Thus, you need to protect precious data

- Use "__attribute__((used))" in source files
- Use KEEP(section) in the liker script

A quick look at the code

To use inicalls in the hsw2020 repository you need to

- **include <init.h>**
- **declare your init function as device_initcall() or otherwise**

```
typedef int (*initcall_t)(void);

#define __initcall(level,fn) static initcall_t __initcall_ ## fn
      __attribute__((used, __section__(".init" level))) = fn

#define core_initcall(fn)       __initcall("1", fn)
/* ... */
#define late_initcall(fn)       __initcall("7", fn)
```

The new setup.c

The role of setup.c, now, is just calling them, in order

```
for (f = __initcall_begin; f < __initcall_end; f++) {
        errors += (*f)();
```

And, obviously, we panic if any fails

As a side effect, we alway link in usleep and its init code

- **Panic calls usleep**

How could we solve the problem?

- **If the target CPU is very small, we want to avoid panic()**
- **... while keeping the same code structure overall**

A problem with init-only drivers

The library-based approach looks best for hsw2020

- We build-test everything, so nothing rusts away
- We may change API (e.g. gpio_setup()) and fix it all

We pick a driver in the final binary only if we need it

- e.g., the SPI driver is built not not used but for a demo

Initcalls are collected and executed only when needed

- usleep_init is only there if we actually run usleep
- We don't turn-on the GPIO block unless we use it

However, we have a problem with init-only drivers

- We are not pulling-in them by actual use
- One of them is the PLL initialization
- The other is TMR32B1 that runs our jiffies variable

Hooks

The simple solution to the above problem is hooks

- Every driver that would not be pulled-on by the linker has a hook symbol
- Some relevant code refers to the hook (main, or setup.c)
- We waste at least one byte per driver, but it may be acceptable

By referring to the hook, you claim your need for the code

```
if (CONFIG_HAS_USB || CONFIG_PLL_CPU > 1)
        pll_hook = 1;
```

A more structured operating system may be hook-based

- If drivers have standard operations and you open them by name
- You would implement an "import" or "use" mechanism like this

This is what BertOS was doing, for example

- It is a library-based OS building multiple applications
- Each application requests its own feature-set
- Then, the match at run-time is name-based

A look back at the linker

The role of the linker

The linker resolves symbols to addresses, nothing more

- Symbols are in the input object files
- And unresolved symbols are in there, too

It uses (or discards) input object files as a whole

It then reassembles ELF sections

- It does so according to the linker script
- Each ELF section is an atomic item
  - The same-name sections of different input files are merged
  - Originally, this was only .text, .code and .bss

Eventually, it can discard any sections not referenced to

- This is what happens with "--gc-sections"
- This is usually after "-ffunction-sections" in the compiler

Discarding unused sections is useful in the uC world

- Similar functions live in the same file, like strcpy() and strcmp()

The three ways to build a binary image

Link it all into a big object file

- This is what we were doing at the beginning
- It allows for a platform-independent bigobj.lds intermediate script
- But it shows its limits when the code base increases
  - I had to abandon this to implement BUG()

Build it all and create a library file (.a)

- This is the current approach
- We build-test all of the code
  - Well, excluding alternatives for other architectures
- Object files that are not picked are just ignored
  - If a weak symbol exists, no strong symbol is looked for
- This is useful in a multi-application environment
  - In our code base, we build several .bin files

Only build what you need to run

- You lack build-test for all code, but you save compiler time
- Useful in big projects that create a single binary
- Examples: Linux, u-boot, barebox, buildroot

BUG()

The need for BUG()

During code development we really need assertions

- We need to verify values are as expected
- Mishaps happens, and they need to be notified

Unfortunately, the assert code takes quite some space

- Besides the conditional, you need to pass __FILE__ etc
- You call printf and panic, with arguments (or __assert)

This can be an issue in hot code paths

- Assert can be conditional (e.g.: CONFIG_ASSERT)
- But sometimes you want to keep the check in production too

So we can use the undefined instruction handler

- One instruction (two bytes in thumb code)
- And an handler that looks up which asertion it was

Implementation in our code base: bug.h

Again, the idea and code hints are from the kernel

```
asm volatile(
  "1: " __BUG_INSTR "\n"
  ".section .rodata.str, \"aMS\", %progbits, 1\n"
  "2:\t.asciz " #__file "\n"
".previous\n"
".section .bug_table, \"a\"\n"
".align 2\n"
".word 1b, 2b\n"
".hword " #__line ", " #__id "\n"
".previous\n");
```

Implementation: irq-lpc-debug.c:

```
printf("BUG @ 0x%08lx\n", (long)pc);

for (b = __bug_first; b < __bug_last; b++)
        if (b->pc == pc)
                break;

if (b == __bug_last) {
        printf("-- not found in table\n");
        return 0;
}

printf("ID %i %s:%i\n", b->id, b->file, b->line);
```