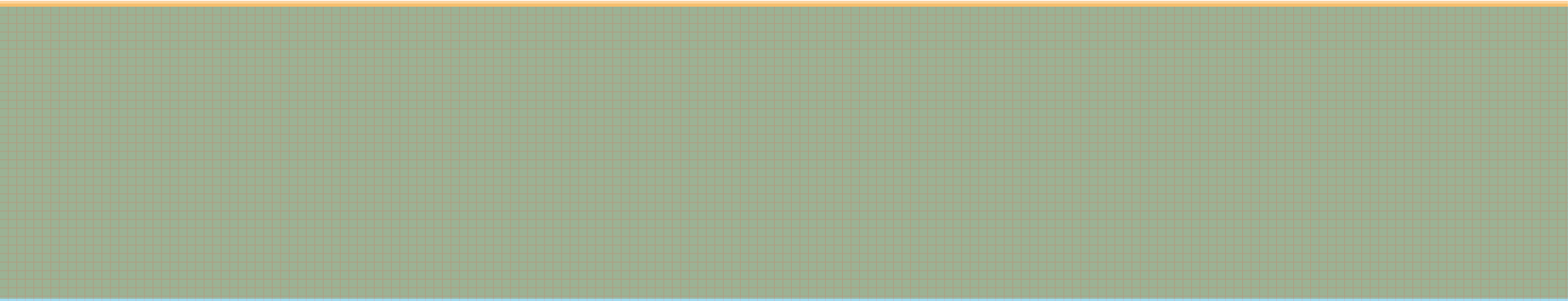


Why add capabilities  
and move Flash slowly  
to generic MTD class.



# 0. Why add new capability to Flash?

- We are starting to support new kind of devices that replace Flash as code/storage solution in SoC devices and are becoming more common as external devices. These are RRAM, MRAM, FRAM and other devices that have similar characteristics of Flash in use, but do not have erase requirement or sometimes even lack endurance problem.
- This device are now connected to Flash Driver API, because code is basically tied to Flash at this point in various ways.

# 0. Why add new capability to Flash?

- In Zephyr flash is kinda generic subsystem that allows to access Code/Storage within SoC device and external devices. There is a lot of Kconfig, Linker scripts, DTS definitions around flash that usually mean „CODE STORAGE“, for example `CONFIG_FLASH_BASE_ADDRESS`, etc.:

```
Name: FLASH_BASE_ADDRESS  
Prompt: Flash Base Address  
Type: hex  
Value: 0x0
```

Help:

```
This option specifies the base address of the flash on the board. It is normally set by the board's defconfig file and the user should generally avoid modifying it via the menu configuration.
```

# 0. Why add new capability to Flash?

- A lot of our subsystem uses Flash Map or Fixed Partitions, which are positioned around Flash and moving them to other subsystems, like EEPROM, would require significant rework and ability to demultiplex type of device API is invoked for, rather than what API could do with the device.etc.
- A lot of code directly accesses devices via Flash API, this means that if we would move the SoC non-Flash devices to EEPROM we would generate two paths in every instance when storage is accessed to allow choosing between EEPROM and Flash API.
- All devices use Flash API, EEPROM is optional. Moving these devices to EEPROM makes EEPROM API mandatory for them, even though they „run on Flash“.

# 1. We need new subsystem to handle that

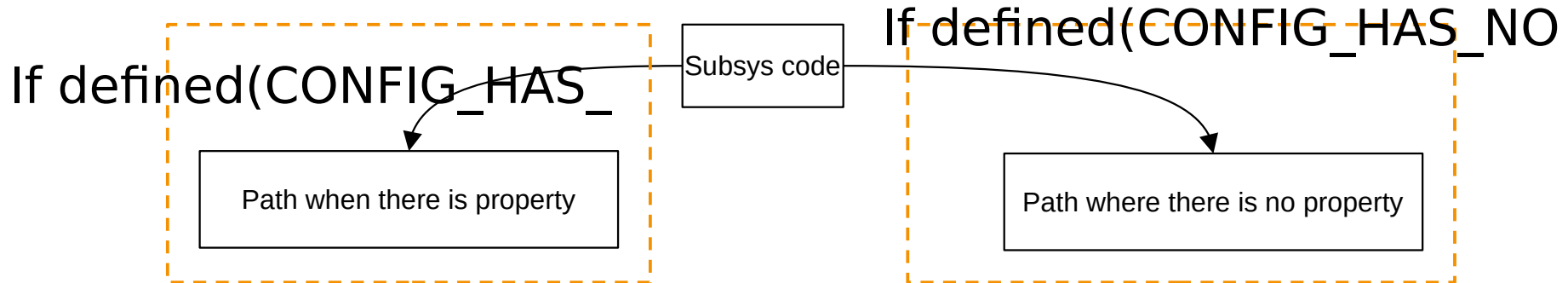
- We need transition from what we have to more generic storage system. Moving to a new subsystem will require a lot of work, which has been already discussed while this ZSAI (Zephyr Storage Access Interface, do not stick to a name) <https://github.com/zephyrproject-rtos/zephyr/issues/64732>

## 2. What is your proposal then?

- Slowly adapt Flash API as a generic storage API, and then renaming it into some „common storage“ API.
- This I am approaching with addition to capabilities to struct `flash_parameters`, where first capability is `explicit_erase`, a capability that is now set by each Flash driver to indicate that it requires explicit call to erase in comparison to no-erase device like RAM, or auto-erase/erase-on-write devices like EEPROM.
- I am also adding `CONFIG_HAS_EXPLICIT_ERASE` and `CONFIG_HAS_NO_EXPLICIT_ERASE`, for a device driver to indicate what kind of devices are within a system.
- There are additions to Flash API in form of `flash_flatten` and `flash_fill` functions, where the first one is for applications that use have been using erase for purpose other than mandated by hardware, for example to remove data or set specific pattern (`erase_value`) accross some storage area.

### 3. Why CONFIG\_HAS\_ and CONFIG\_HAS\_NOT\_? Wouldn't CONFIG\_HAS\_ suffice?

- No it would not. The CONFIG\_ indicates for subsystems what capabilities various drivers provide.
- Subsystems can use both to optimize out alternative paths in code when there is no device with specific capability or all devices have capability a subsystem will not work at all.



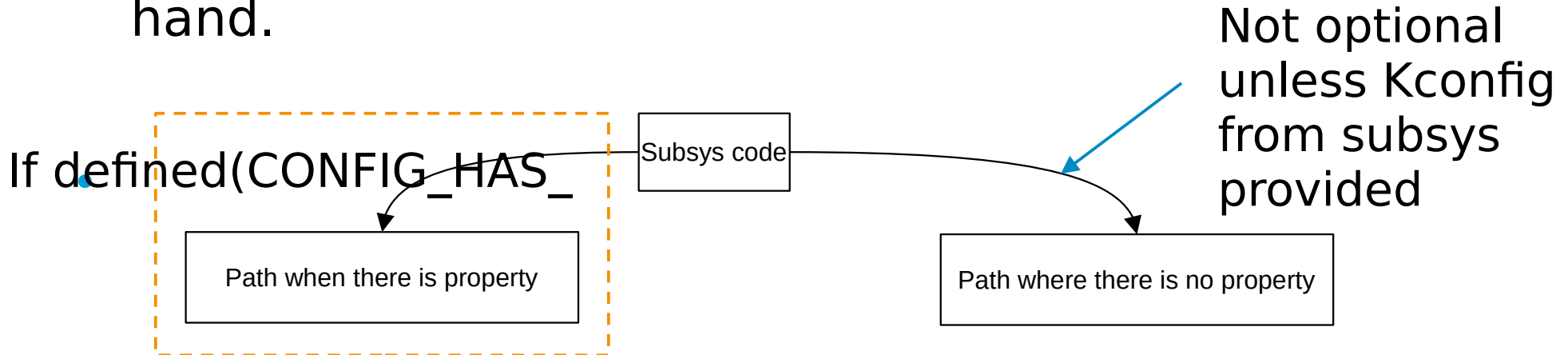
## 4. Why CONFIG\_HAS\_ and CONFIG\_HAS\_NOT\_? Wouldn't CONFIG\_HAS\_ suffice?

- This also allows at the level of Menuconfig show only subsystems that support hardware with given capabilities.
- Additionally the subsystems can decide to provide sub-menus that are dependent on declared capabilities.
-



## 5. No, the CONFIG\_HAS\_ would suffice!

- Yes, kinda but in clumsy way. You have no way to inform code that there are no devices that have no capability and so the paths that support such case could be removed. Now you have to add Kconfig to every subsystem to disable the path and do that by hand.



## 6. So just control it from subsys.

- Yeah, that is fine, but you have no feedback from drivers what is supported or not, so from careful configuration this changes to trail-and-error approach.



7. This should be property in DTS to allow compile time optimization by picking it from DTS directly

- This only works if you have subsystem targetting single instance of a device.
- Kconfi works here as well and you are able to set if from multiple drivers.

## 8. User should be able to set it for a device in DTS.

- Capability is not configurable at level of device – device has it or not; driver may support it or not, which can be indicated by Kconfig and turned off at the level of driver, if it is required.
- Properties, applied to a device, should modify how a common driver code works; if they do not do that, they are pointless.

9. No, DTS. Since subsystems can check that as property/capability this this belongs to a device node.

- First this makes it harder to actually make the compile time optimizations (yeah, so it breaks 7. anyway, and is worse than 3.)
- For the mental experiment let assume we have

```
some_dev0 {  
    prop_a = ?;  
    prop_b = ?;  
};
```

DTS definition

```
class {  
private:  
    int prop_a;  
    int prop_b;  
public:  
    set_prop_a()  
    set_prop_b()  
    get_prop_a()  
    get_prop_b()  
    int the_only_fun() { if (prop_a) then something; };  
};
```

Generated Object

- Note that no code in object uses the prop\_b, it is only passed further. So the property does nothing within class of the object and should not have interface to be modified.

# 10. You are inconsistent: you want subsystems to check for the capability but do not allow to set it

- No I am consistent. Storing property that does nothing for the object makes no sense. Capability is property of a class not a specific instance of it.
- If system check for that capability then in this scenario calling `set_prop_b()`, within object, changes behaviour of that subsystem, without changing anything in the object or behaviour of common class code. This means that `prop_b` is property of subsystem not a class defining the object that is used by the subsystem.
-

# 11. We can automatically pick the property from DTS, rather than having Kconfig

- It only works within single driver. (DT\_INST and so on)
- You still need to export it as Kconfig finally.
- Now there is need to maintain binding.
- It is hard to have some logic around it in linker scripts and so on (should we pick the highest value, lowest? Does one true make everything true or one false make everything false?)

# 12. You are inconsistent write\_block\_size works in Flash, why new property would not.

- write\_block\_size is broken design. There is nearly no chance to change the property of write\_block\_size within drivers.
- This property is usable only if you target directly specific device (by node).
- You can not figure out write\_block\_size common for entire system if there are multiple device.
- It is actually used by subsystems only, because device can only write by write\_block\_size hardcoded in factory
- If you set a write\_block\_size it has effect on everything that uses device, except for driver that will still write by what design of chip tells it.
- It allows user to break subsystems in bunch, and should be rather property set for instances of subsystems that work on a device.



# 13. we need another layer to gather all devices underneath

- We have a layer we can extend it. Result will be the same.
- No layer fixes problem with too many layers.
- No layer fixes problems with layer underneath.
- The layer still needs to figure out what devices it works with to pass that info to next layer.

# 14. New layer should focus on types of devices rather than capabilities

- Devices of various types share the same capabilities. There is no point to distinguish devices by type, as in the end we have to dismantle them to capabilities anyway.
- No layer fixes problem with too many layers.
- No layer fixes problems with layer underneath.

```
1 if (device_with_erase) {  
2     // code for device with erase  
3 }  
4  
5  
6 if (!(EEPROM || MRAM || OTHER_THING) && !(THAT_OTHER_EPROM)) {  
7     // code for device that is not EEPROM and NOT MRAM ...  
8 }  
9 }
```

# 15. No. People know Flash and that breaks assumptions

- Things change. We had ROM→PROM→EPROM→EEPROM→Flash evolution and nobody remembers that Flash is EEPROM anymore.

# 15. Software should erase when it wants to

- Software can just erase if it wants, but that may not be provided by a device.
- Erase is not convenience for software, there is no software that is happy that device forces it to erase entire area. Software would preferably change single bit.

16. OK, why add flag that device requires erase rather than the one that states otherwise.

- Because you do not want to do extra steps when not need.
- Software is the same, and it is more convenient to check „do I need to do erase“ and call operation than, then have negative check „!do I need to do erase“ and call erase.