Hardware abstraction layer for microcontrollers

Lukasz Krzak
Department od Electronics,
Faculty of Computer Science, Electronics and Telecommunications
AGH University of Science and Technology in Kraków
lukasz.krzak@agh.edu.pl
www.wsn.agh.edu.pl

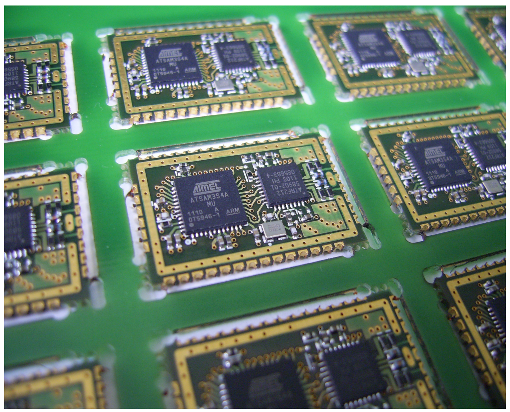
Outline

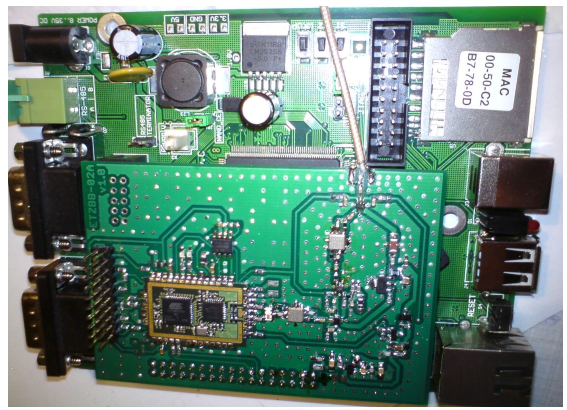
1. How the story began.
2. Important qualities of embedded software and how to reach them.
3. How others are doing embedded software.
4. What we did and what is already done.
5. What are the results.
6. What we still want to do.
7. How **YOU** can participate.

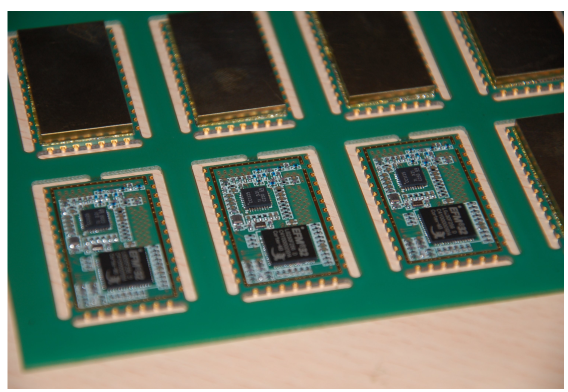
How the story began...

Wireless Sensor and Control Networks Laboratory

- Wireless communications (hardware and software)
- Embedded systems
- Resonant power supply and energy harvesting
- Electromagnetic compatibility

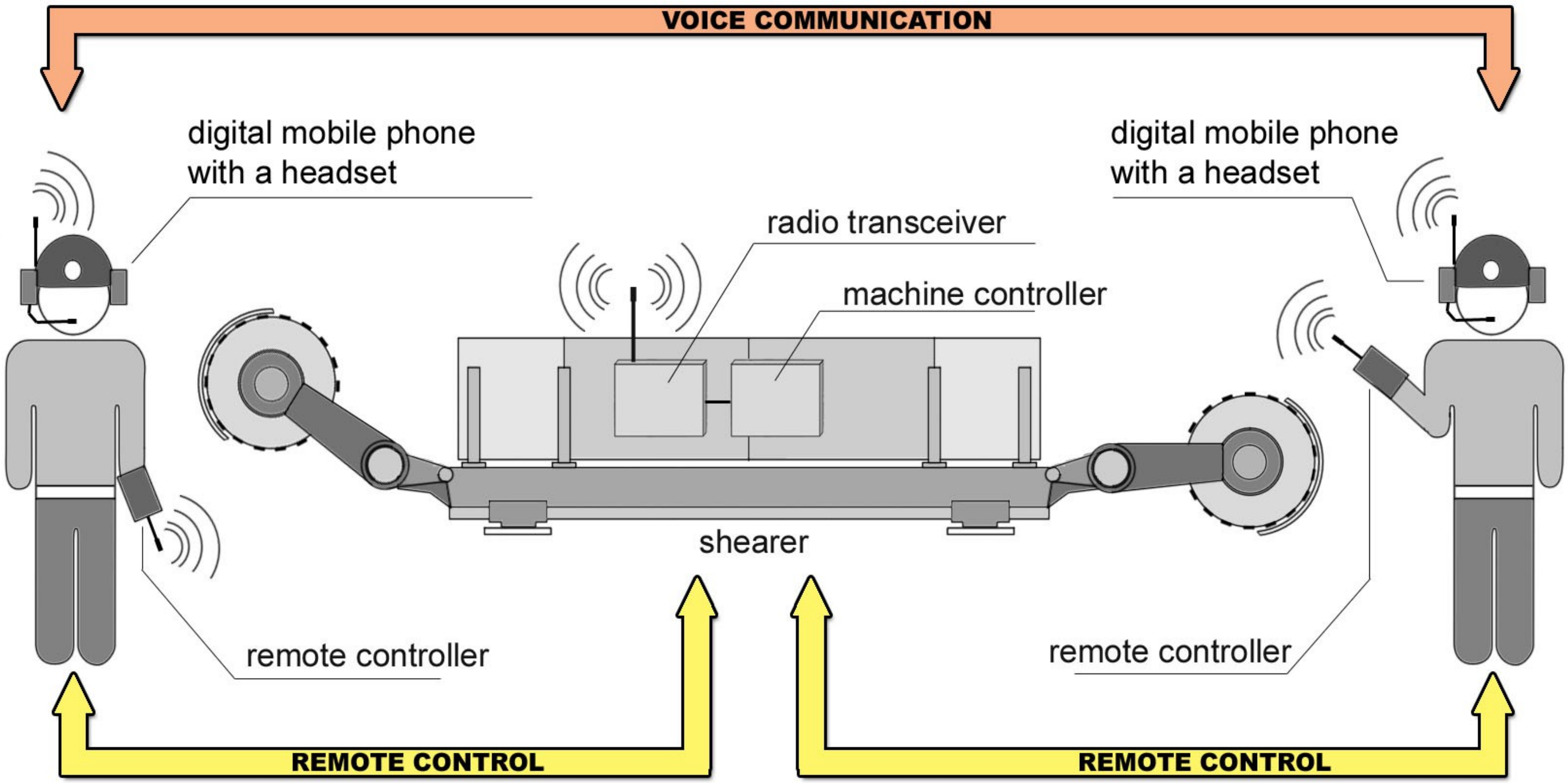






Many of these projects were commercially deployed in industrial applications.
The developed hardware and software solutions are licensed by AGH.

Our initial motivation (2007)



Question: What hardware/software platform to choose ?

Important qualities of embedded software

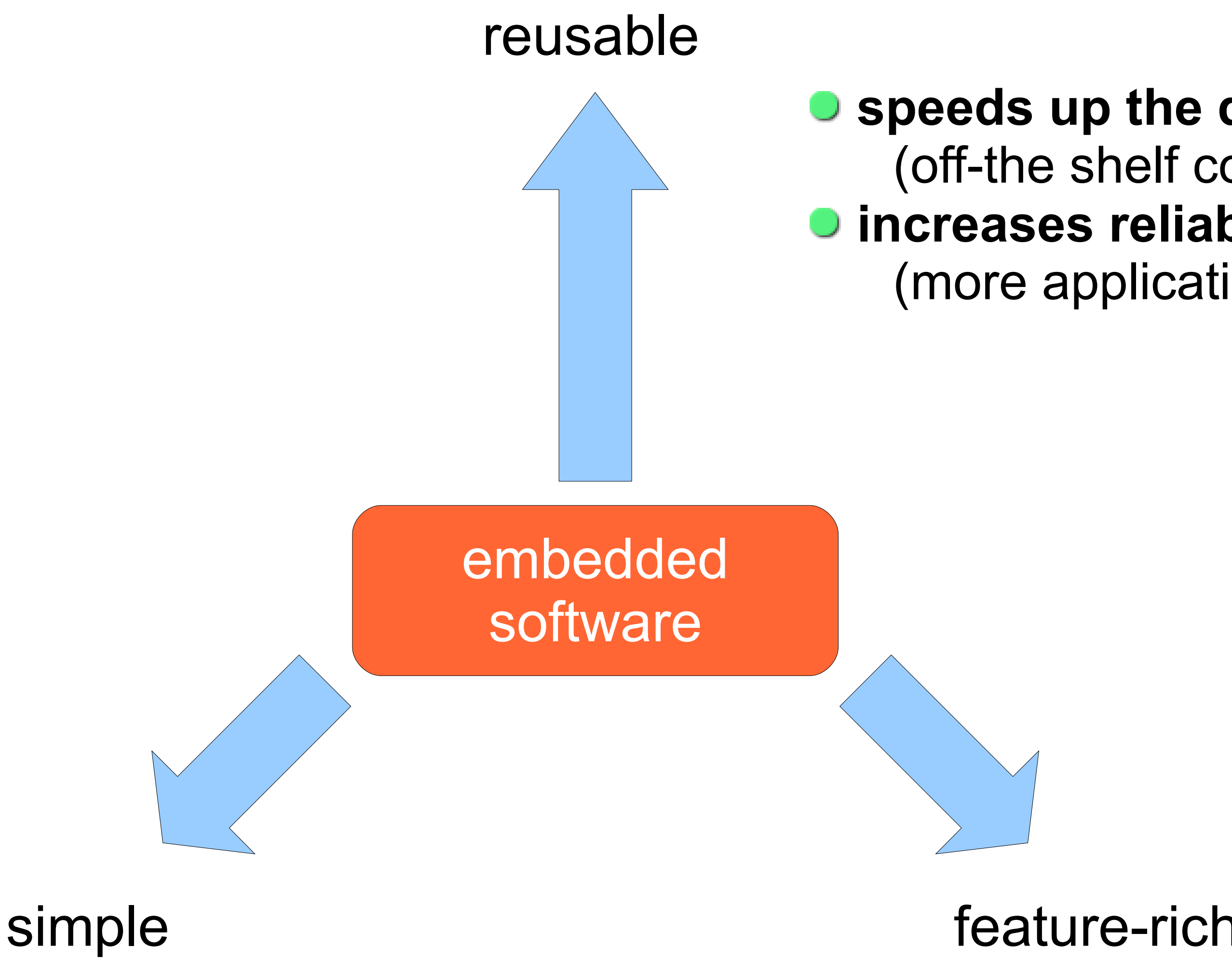
Important qualities of embedded software

AKA: How to distinguish good code from bad code

A good code:

- **works!** (but that's obvious)
- is **reliable** (works every time)
- is **testable** (we can easily proove that it works)
- is **portable** (to different hardware and build tools)
- is **reusable** (we can use it many times)
- is **simple**, **user-friendly**, and **easy to maintain**
- is **feature-rich**

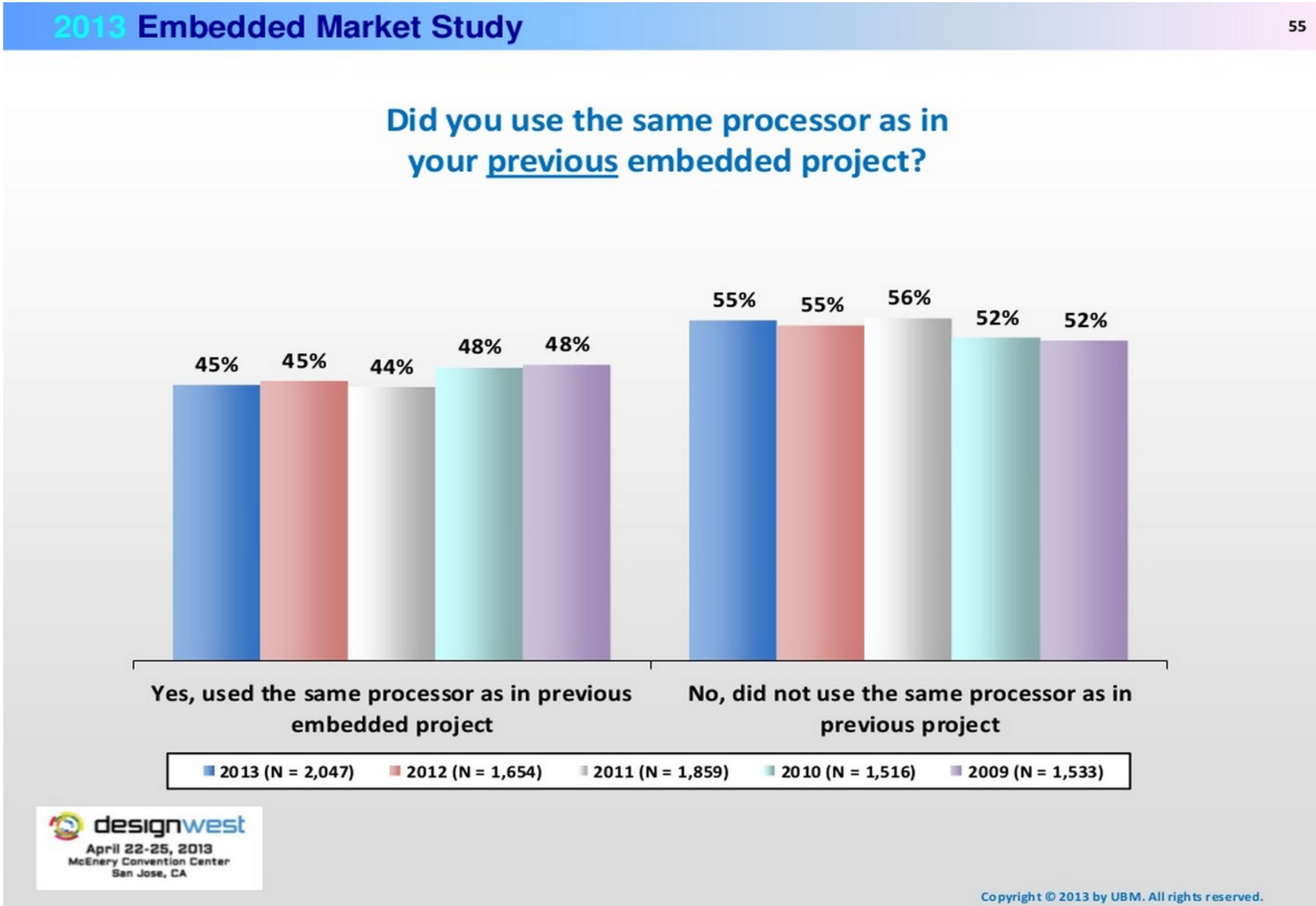
Important qualities of embedded software



- **speeds up the development**
  (off-the shelf components)
- **increases reliability**
  (more applications, more confidence)

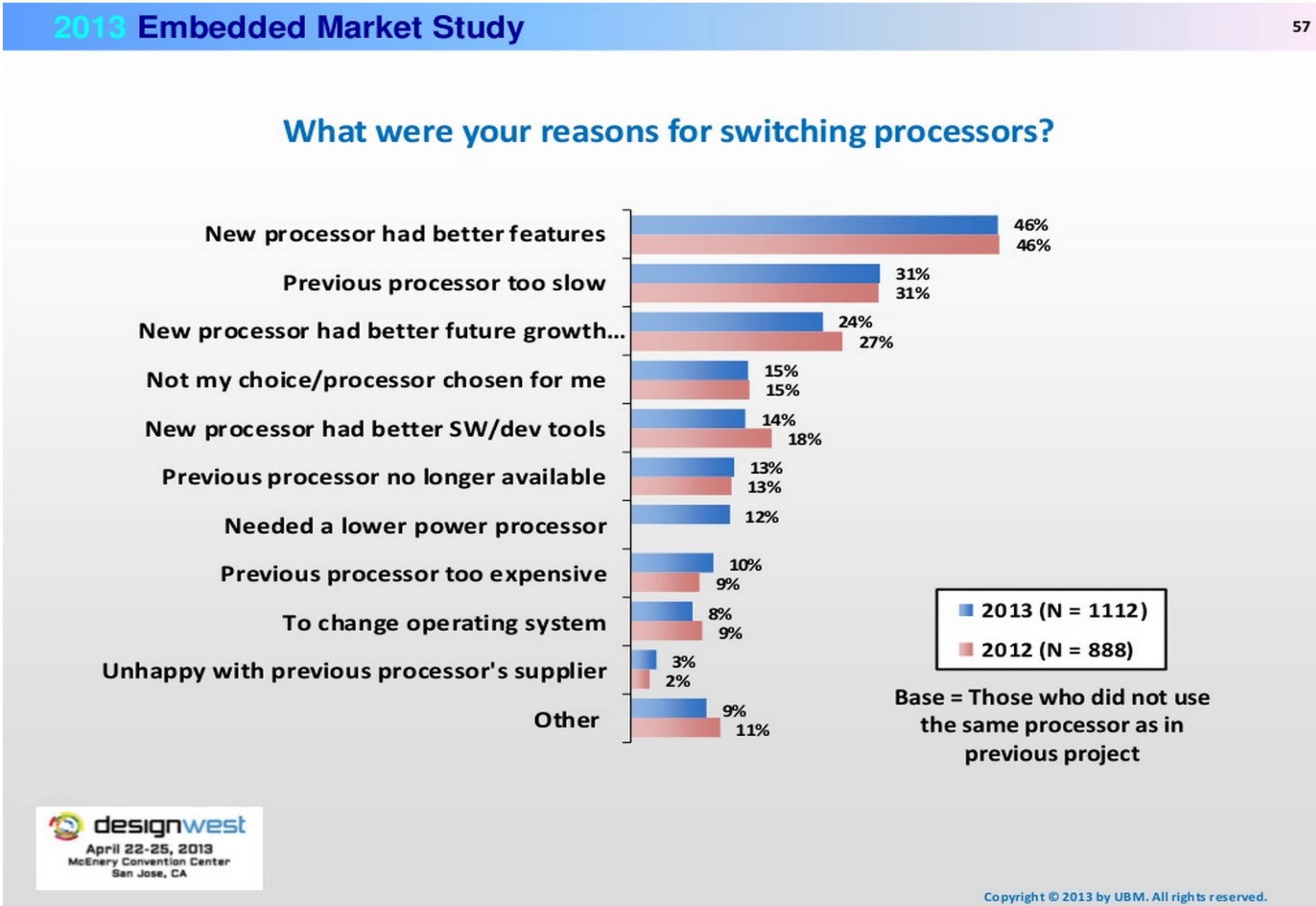
reusable == portable

Reusable == portable



source: ubmdesign.com / http://www.slideshare.net/MTKDMI/2013-embedded-market-study-final

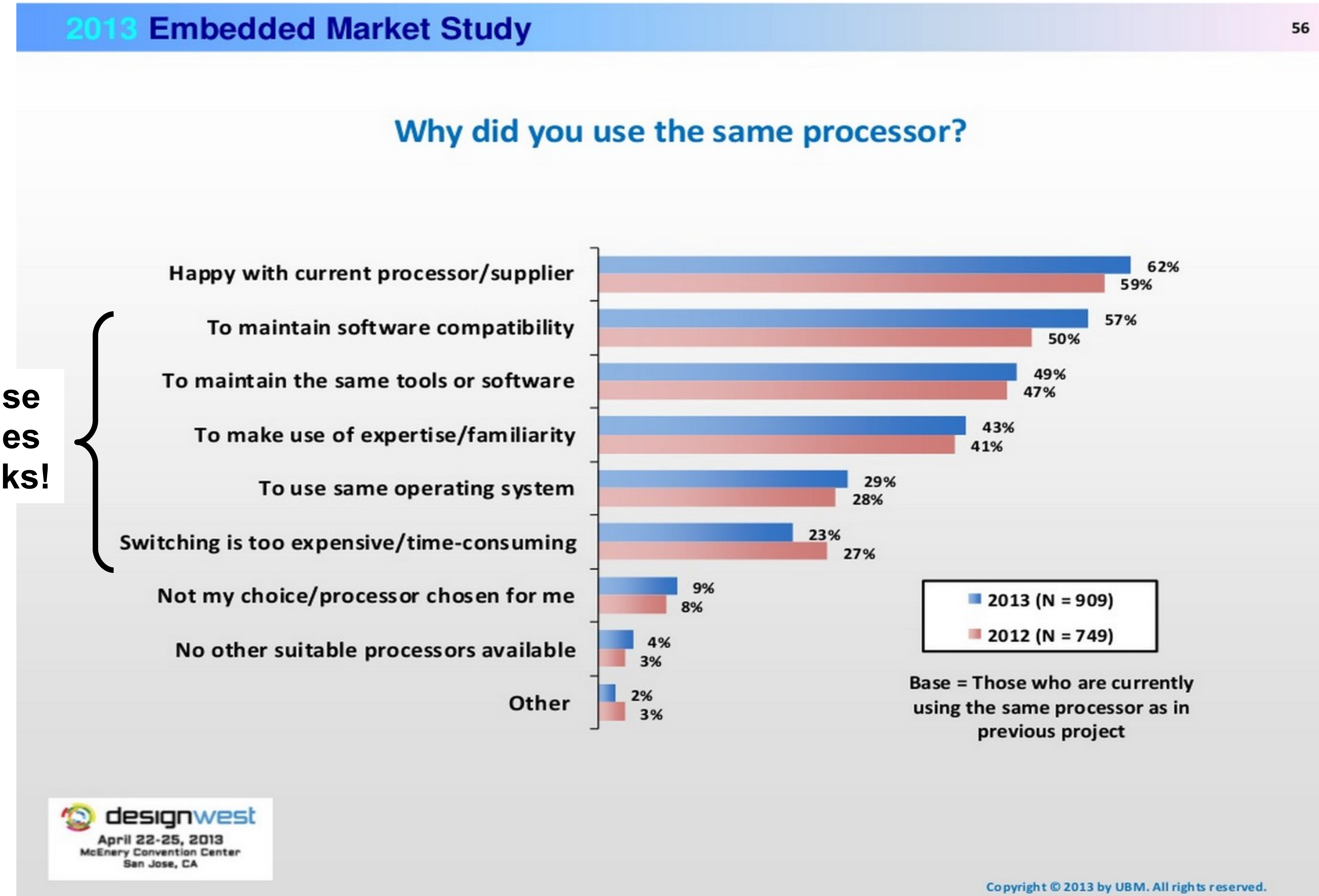
Reusable == portable



source: ubmdesign.com / http://www.slideshare.net/MTKDMI/2013-embedded-market-study-final

Reusable <=> portable

because changes sucks!

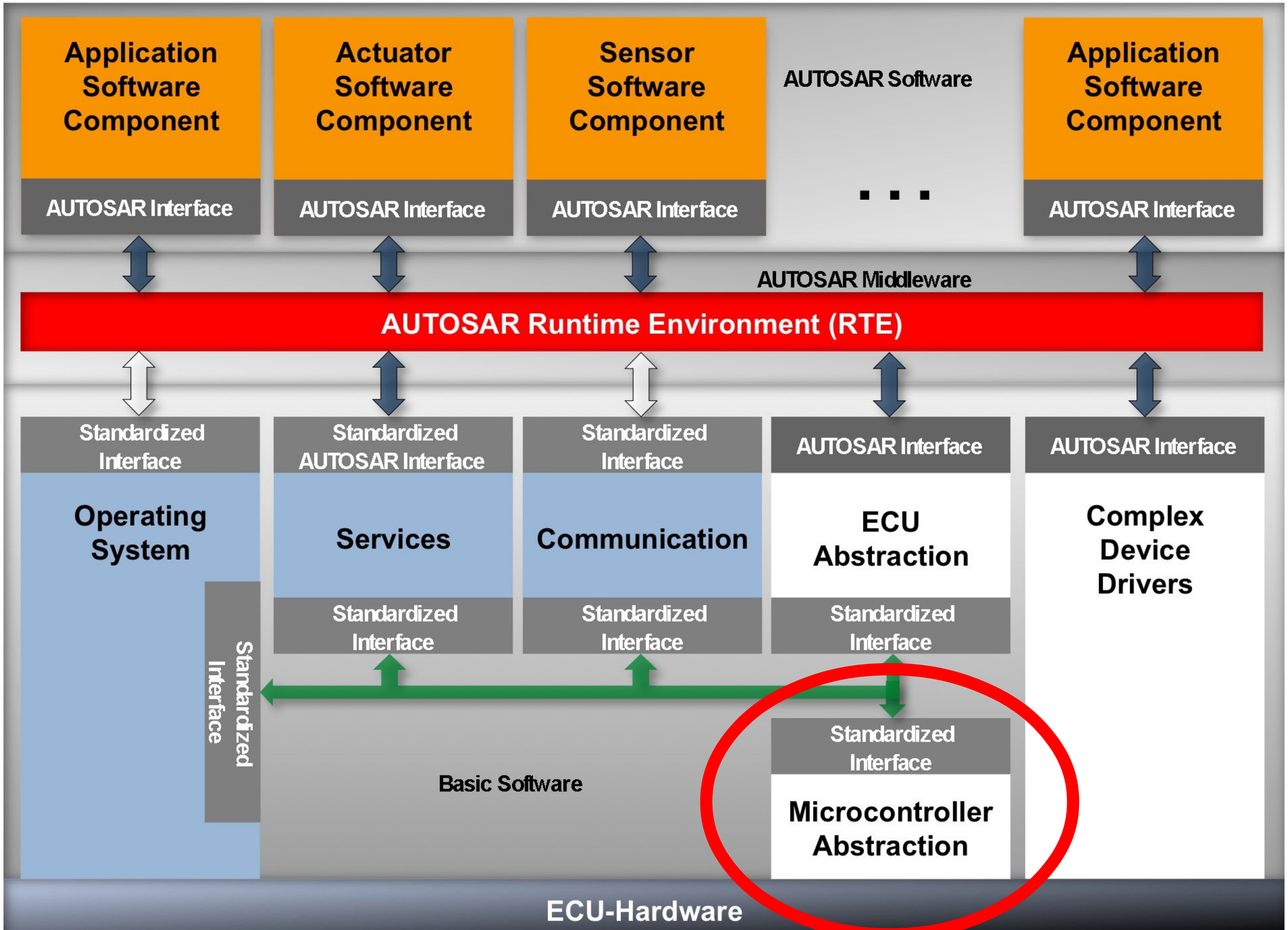


source: ubmdesign.com / http://www.slideshare.net/MTKDMI/2013-embedded-market-study-final

How others are doing it?

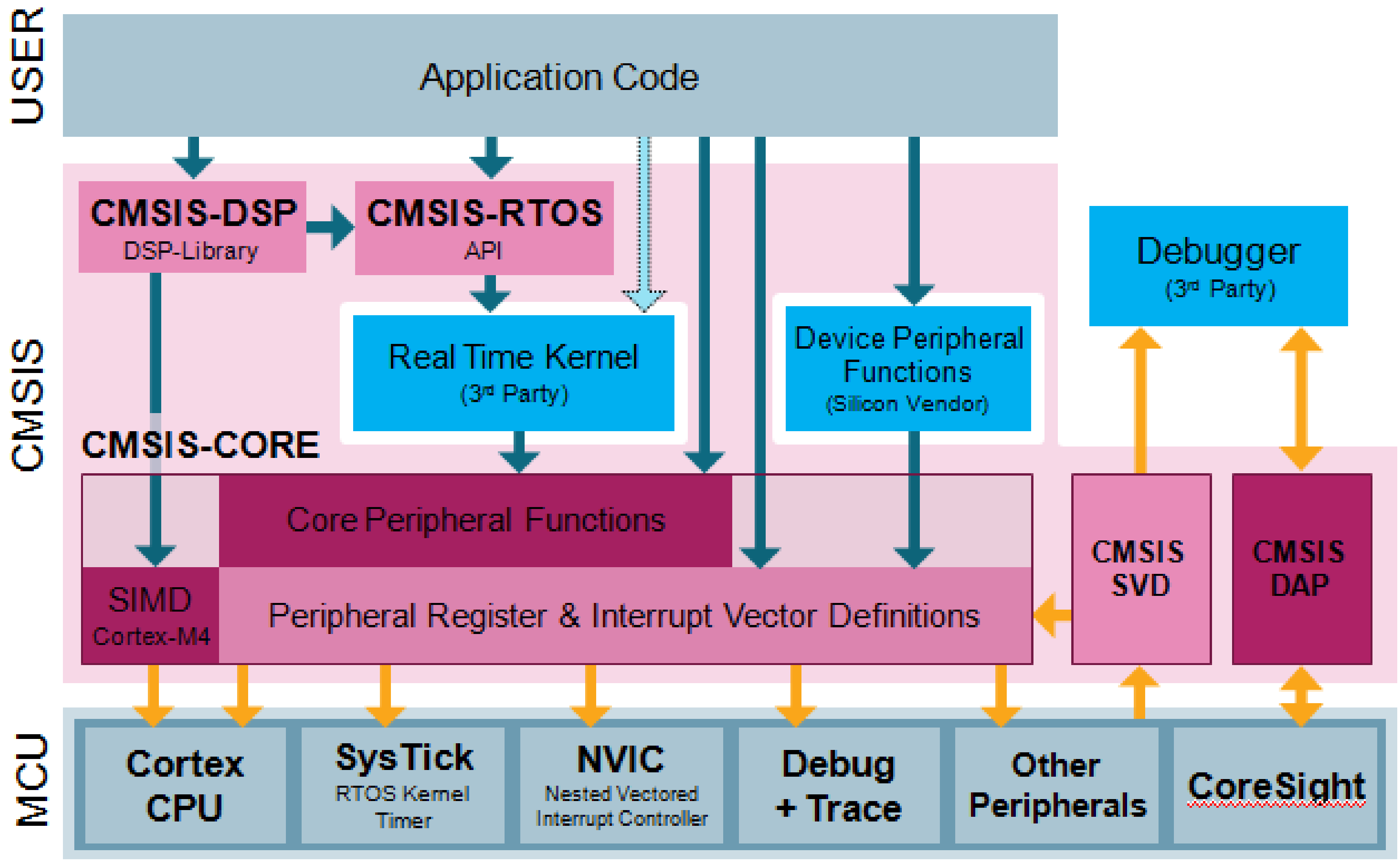
AUTOSAR (AUTomotive Open System Architecture)

"Cooperate on standards,compete on implementation"



source: www.autosar.org

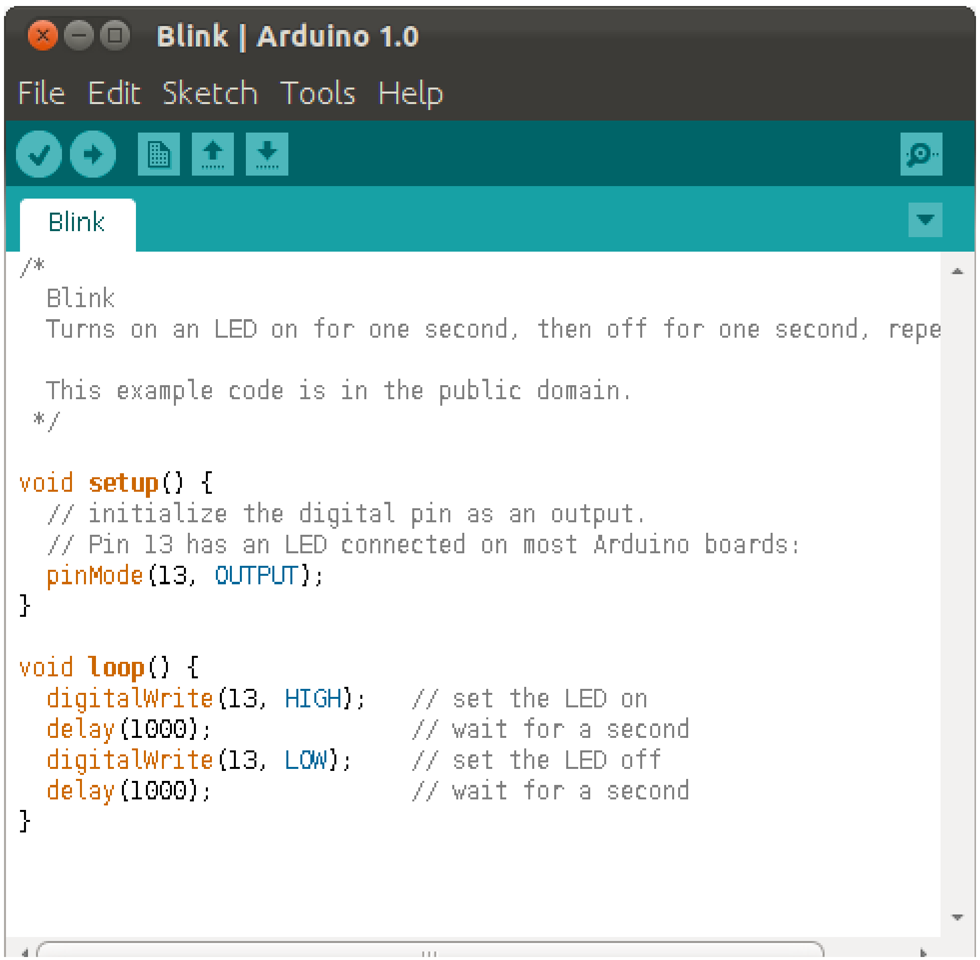
CMSIS (Cortex Microcontroller Software Interface Standard)



source: www.arm.com

Arduino

Arduino is an open-source electronics prototyping platform, based on flexible, easy-to-use hardware and software. It's intended for artists, designers, hobbyists and anyone interested in creating interactive objects or environments.



Arduino programs are written in C or C++. The Arduino IDE comes with a software library called "Wiring" from the original Wiring project, which makes many common input/output operations much easier.

source: http://arduino.cc/

Why reinvent the wheel? We have C stdlib!

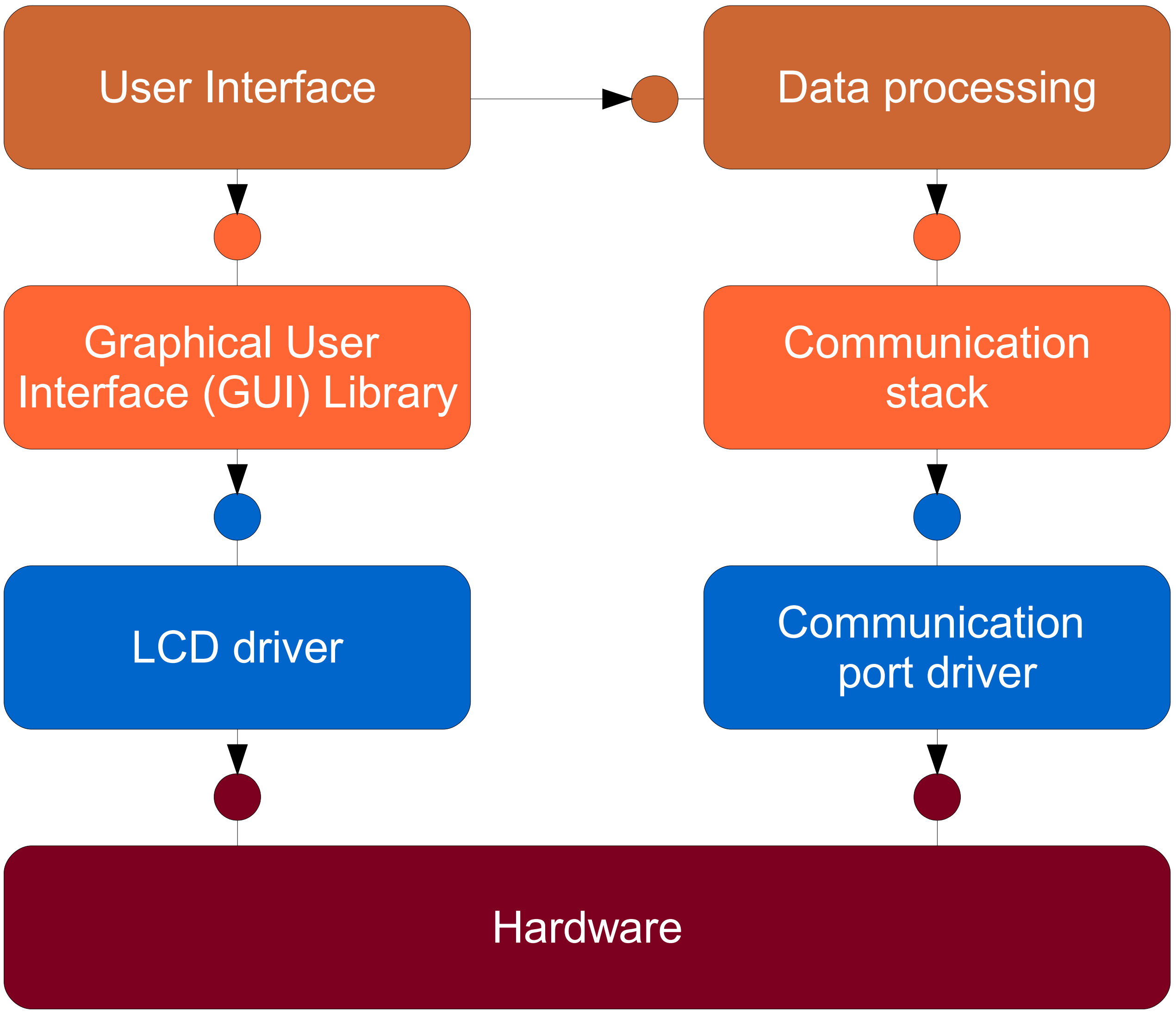
```
#include <stdio.h>

int main(void)
{
    printf("Hello World!");
    return 0;
}
```

- Standard C library is already portable (same with C++/STL)
- It supports I/O operations
- Works in embedded world too!

- Focused on batch processing and text communication
- Lack of support for multithreaded applications
- Lack of support for real-time
- Usually leads to large code
- MISRA says: no!

Component based software architecture

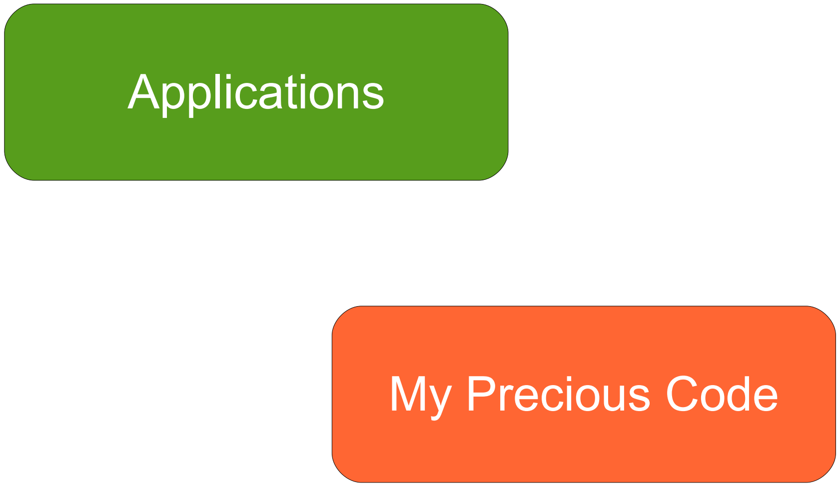
Component-based architecture



How to design reusable/portable code?

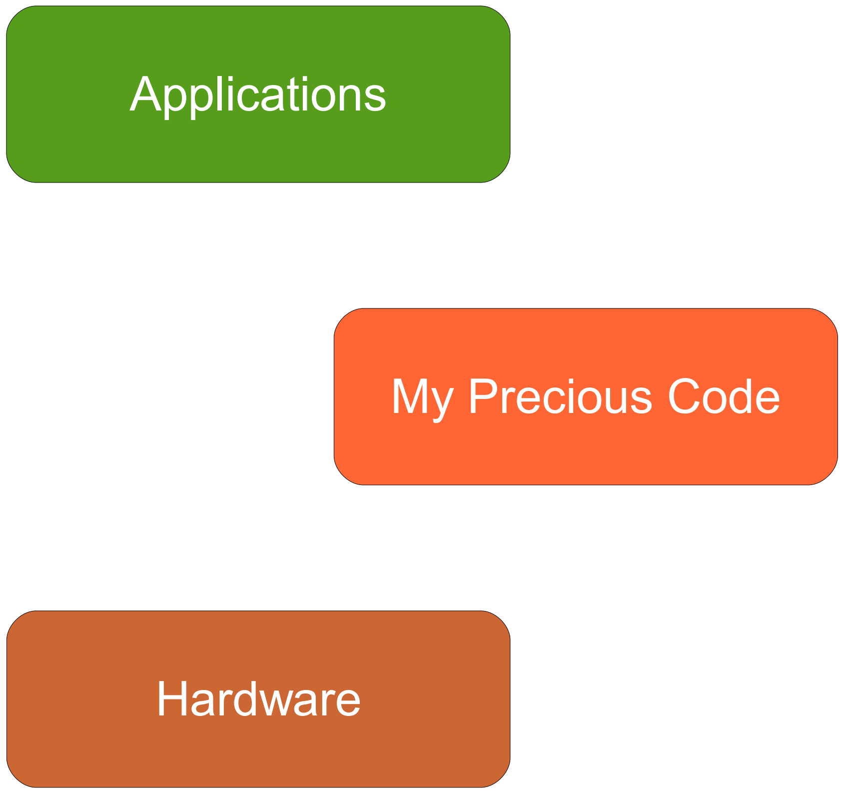
My Precious Code

Important design choices



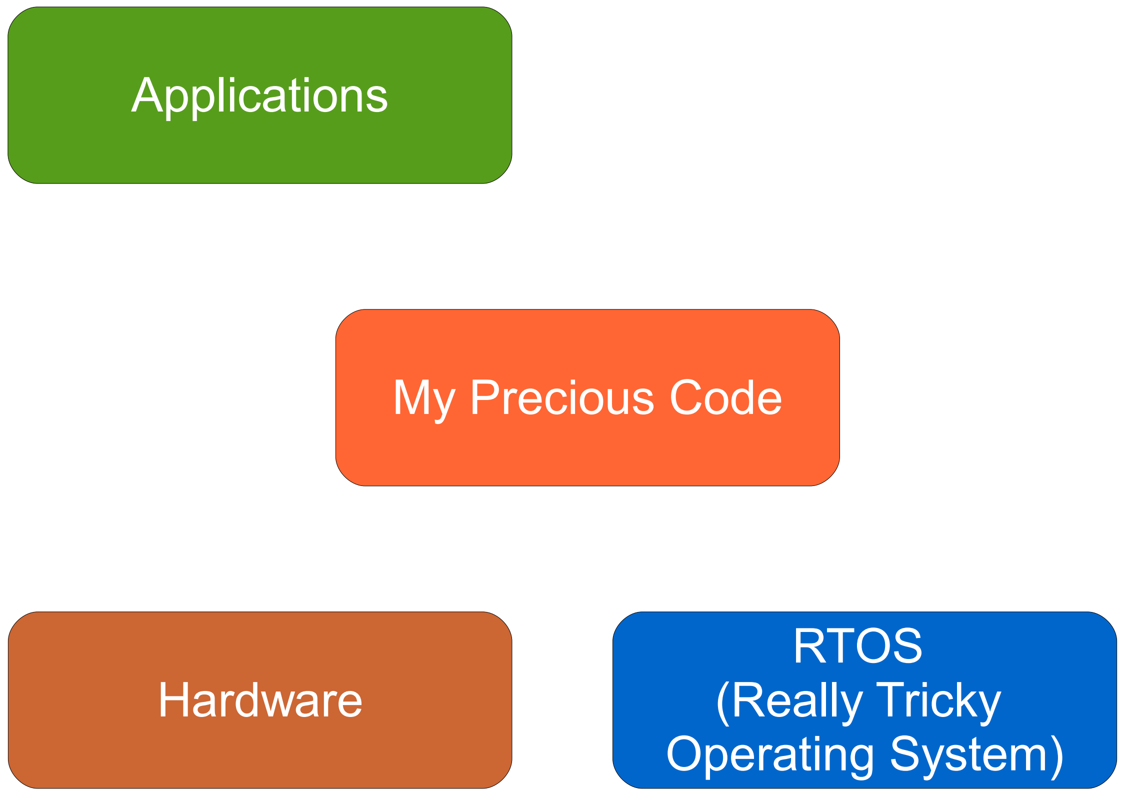
What is the potential range of **applications**?

Important design choices



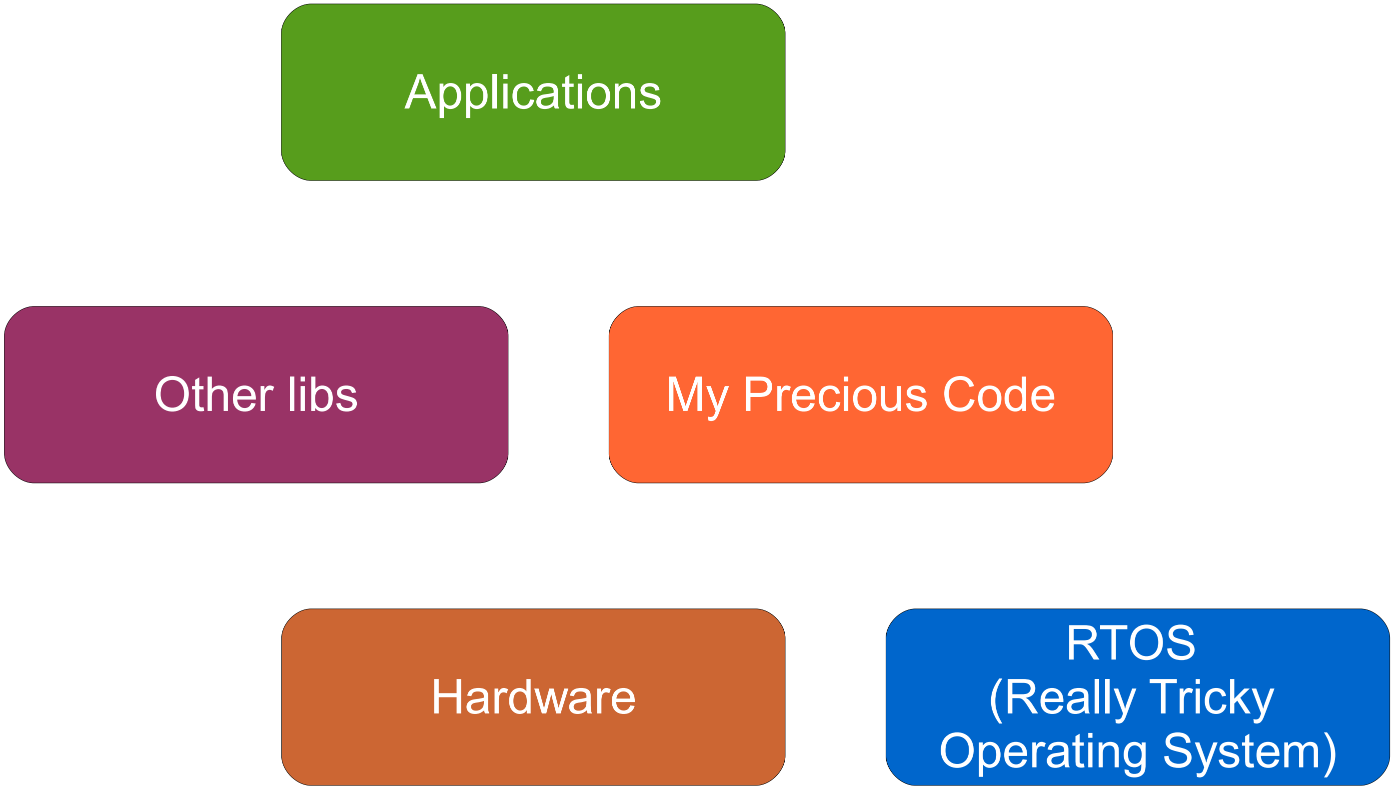
What is the potential range of **hardware** we need to run on?

Important design choices



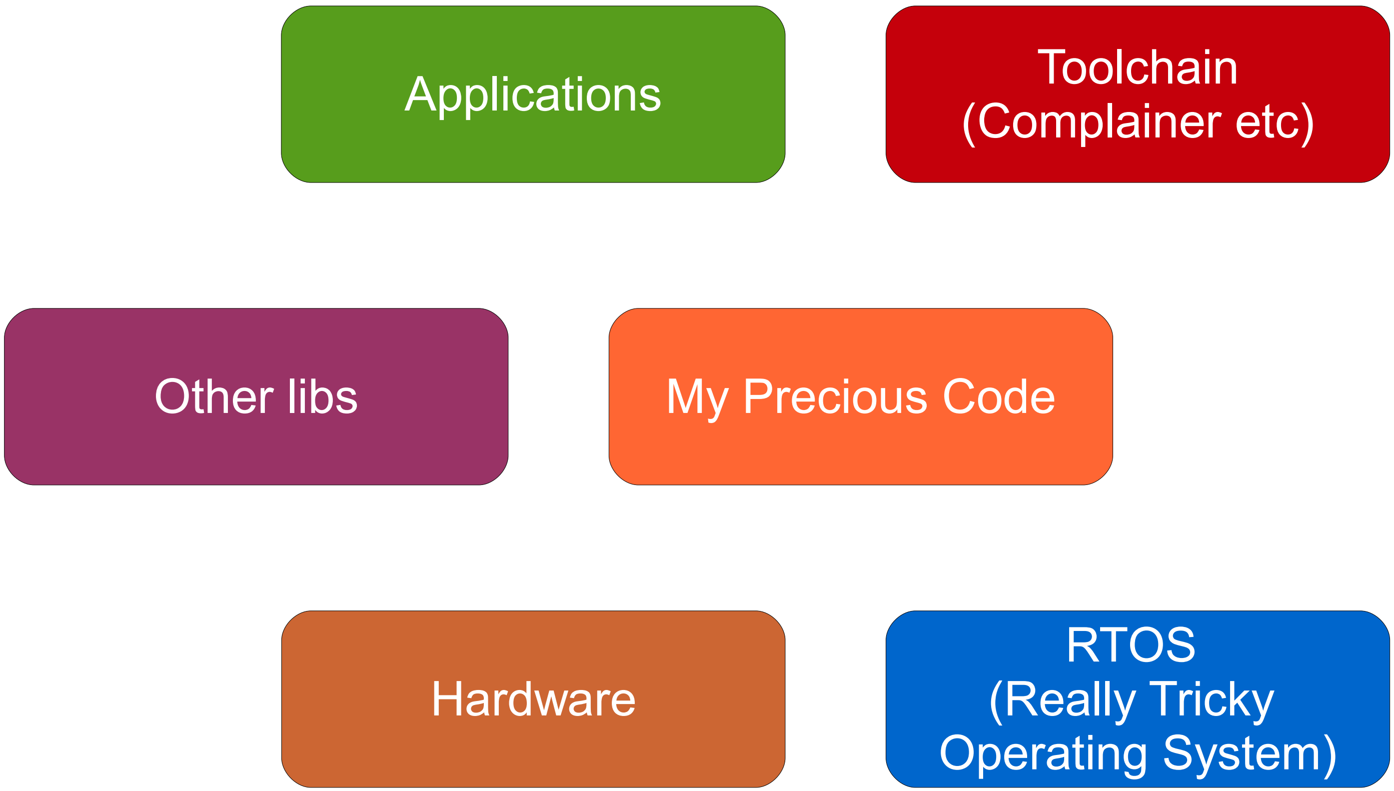
Will an **OS** be used? Which one(s)?

Important design choices



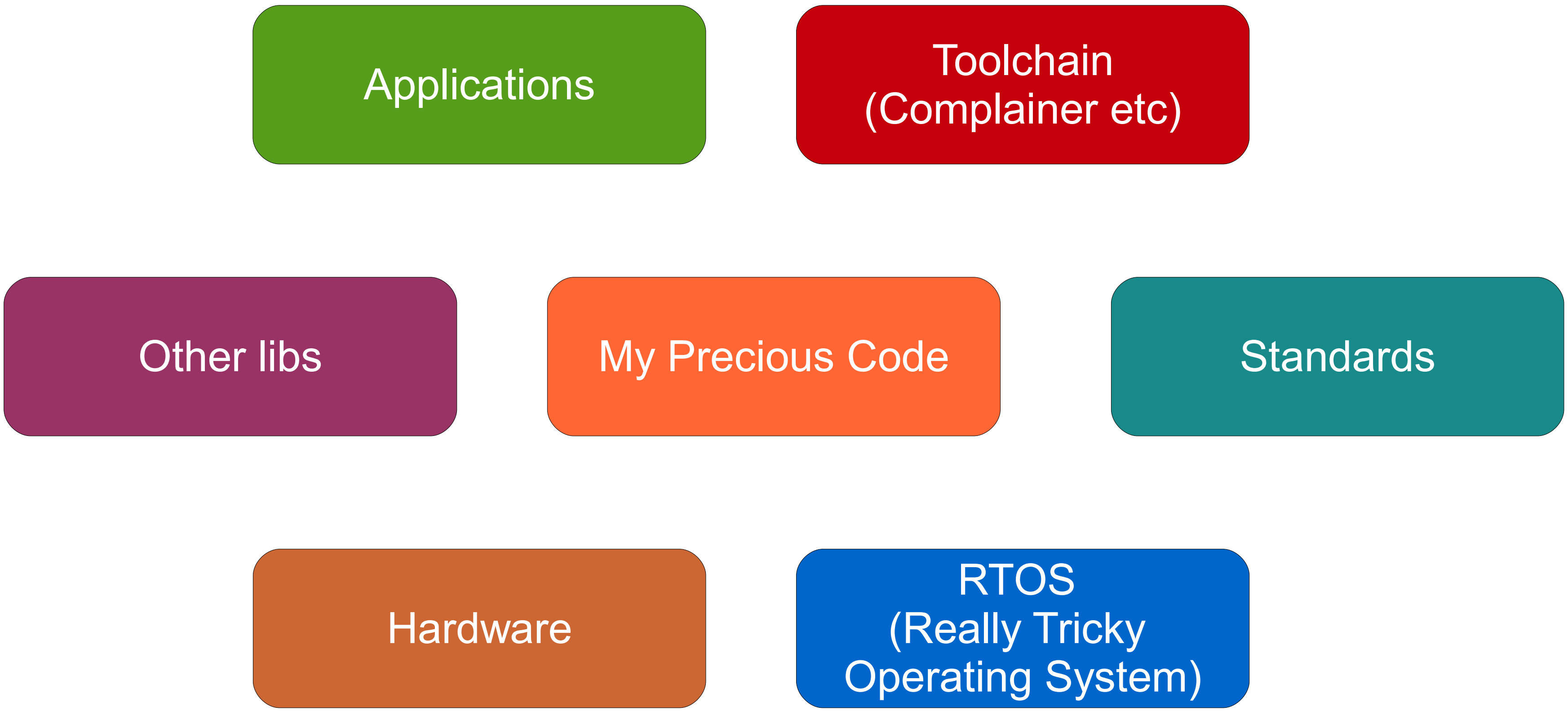
What is the level of integration with **other** software components?

Important design choices



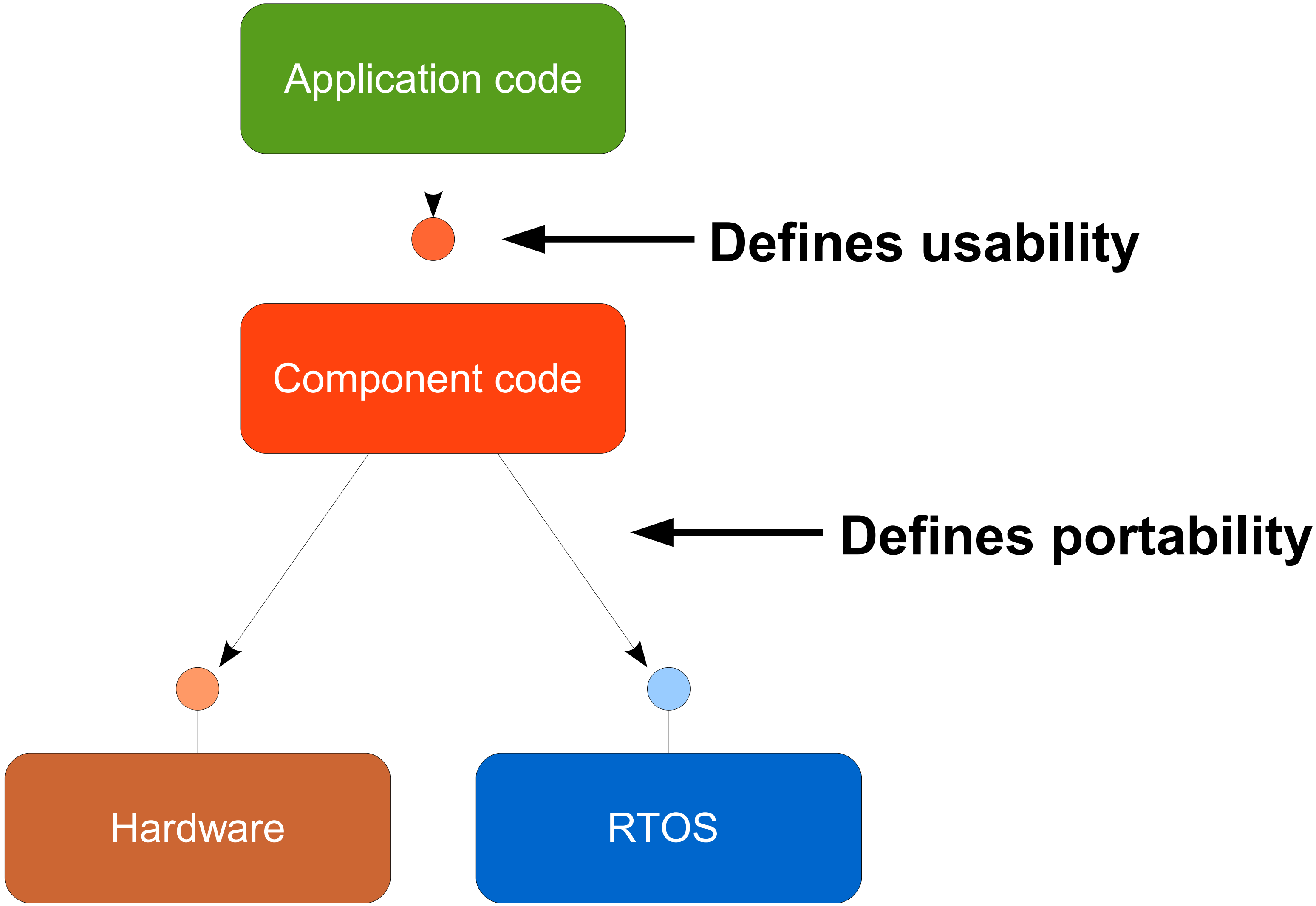
What programming **language**? What **toolchains** need to be supported?

Important design choices

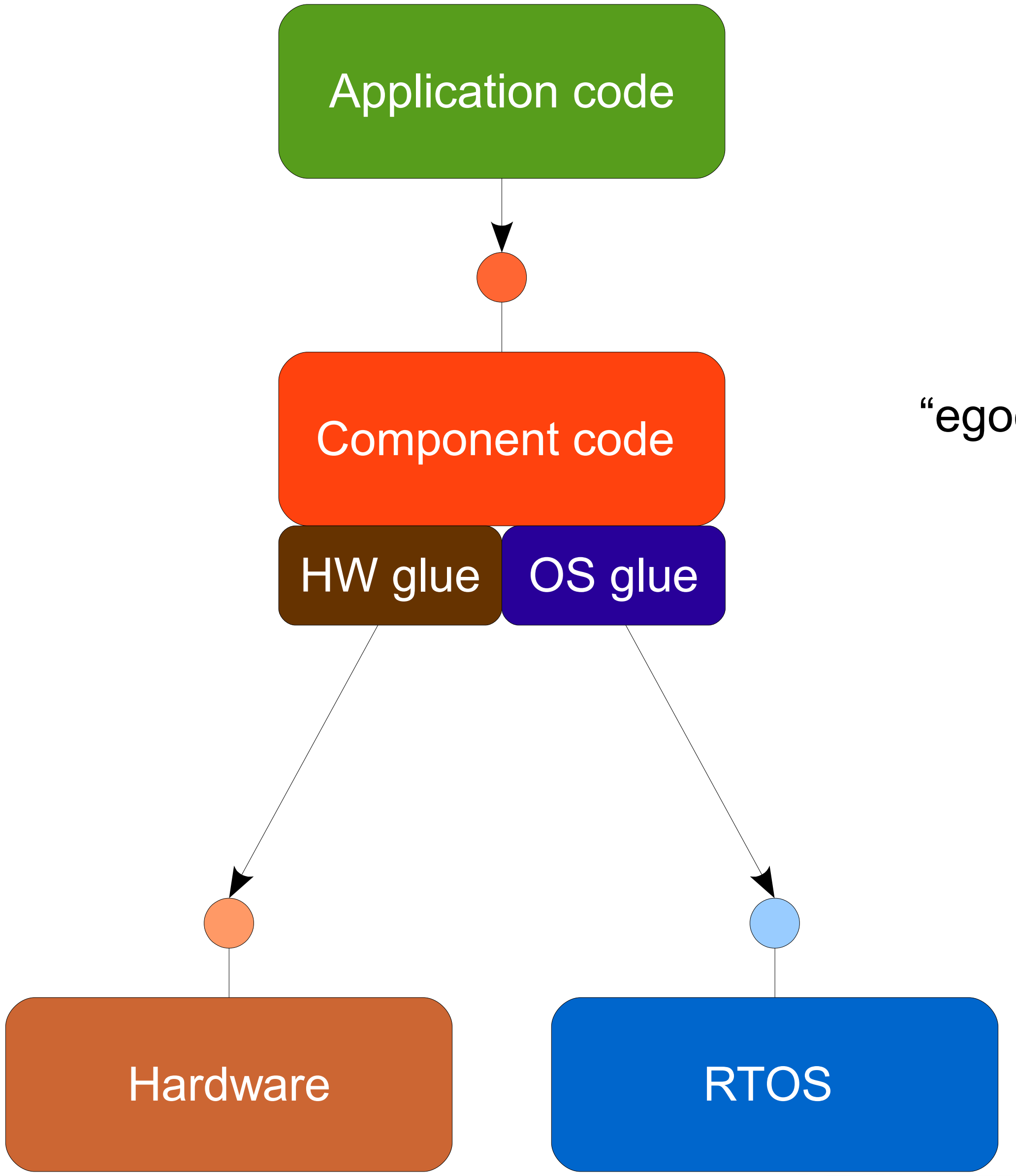


What **standards** must be obeyed?

How to design reusable software?

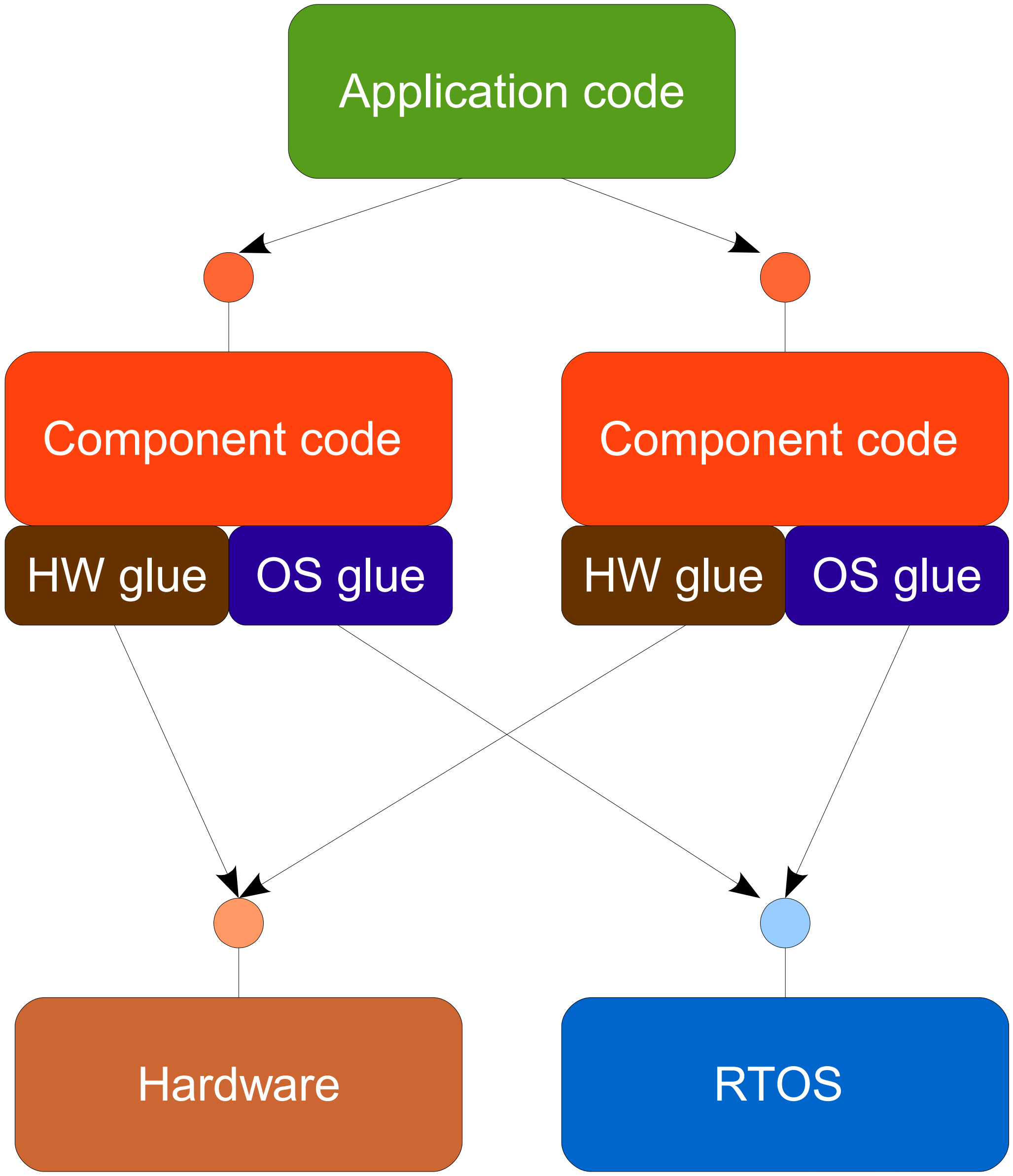


How to design reusable software?



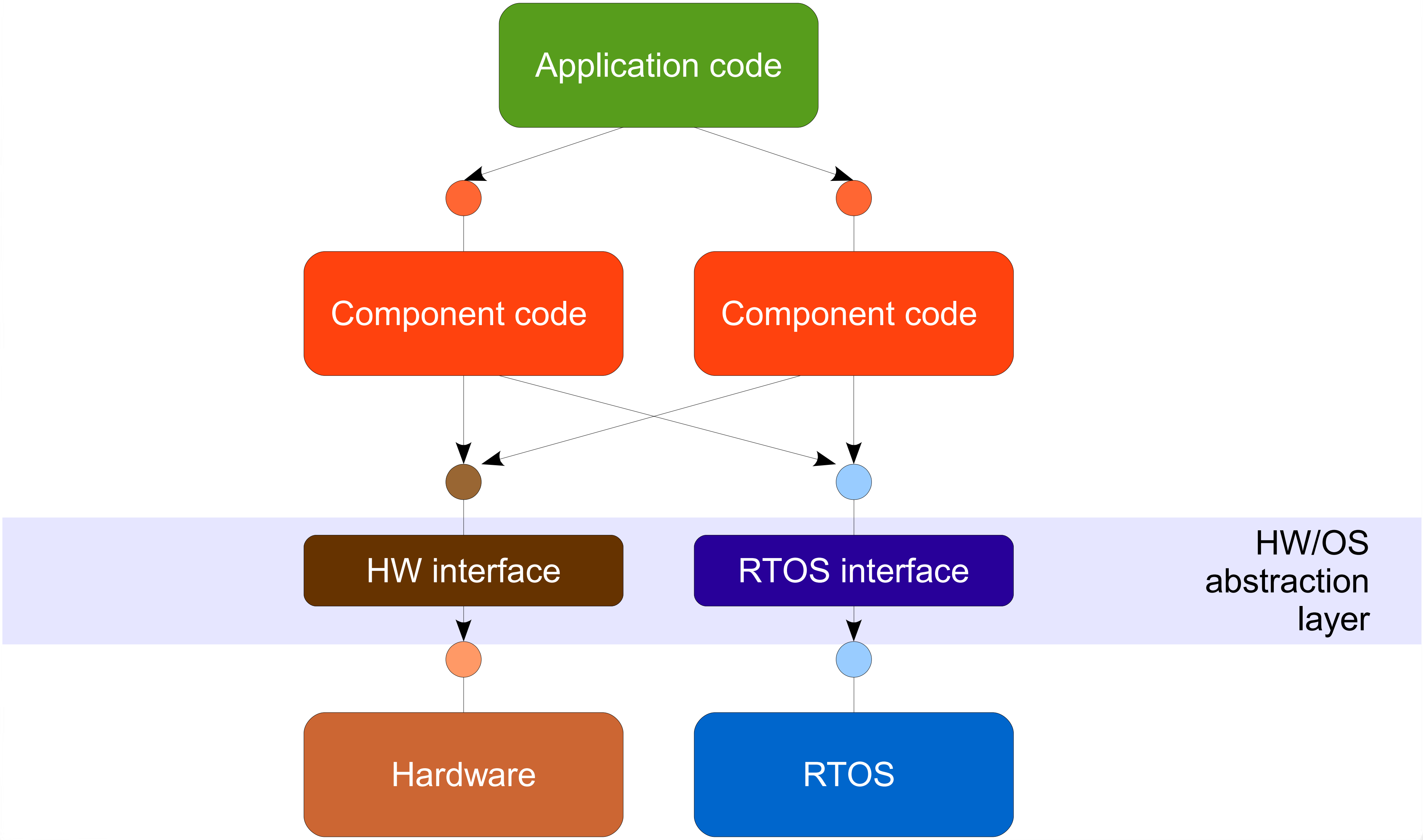
"egocentric" approach

How to design reusable software?



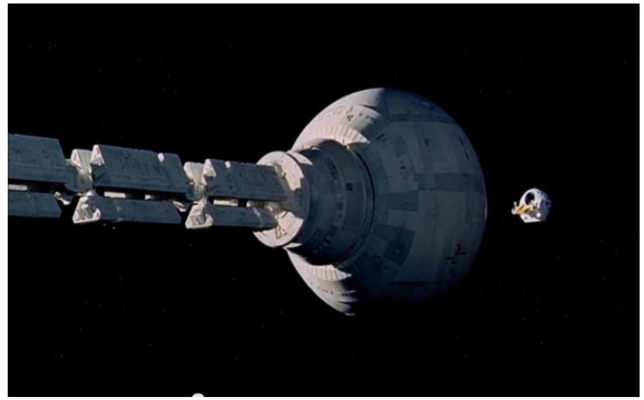
"egocentric" approach
multiplied

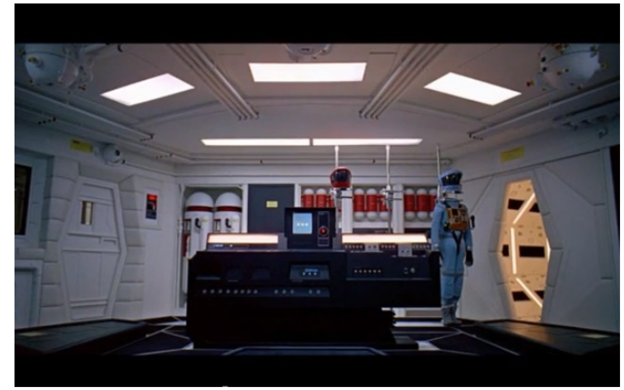
Towards abstraction...



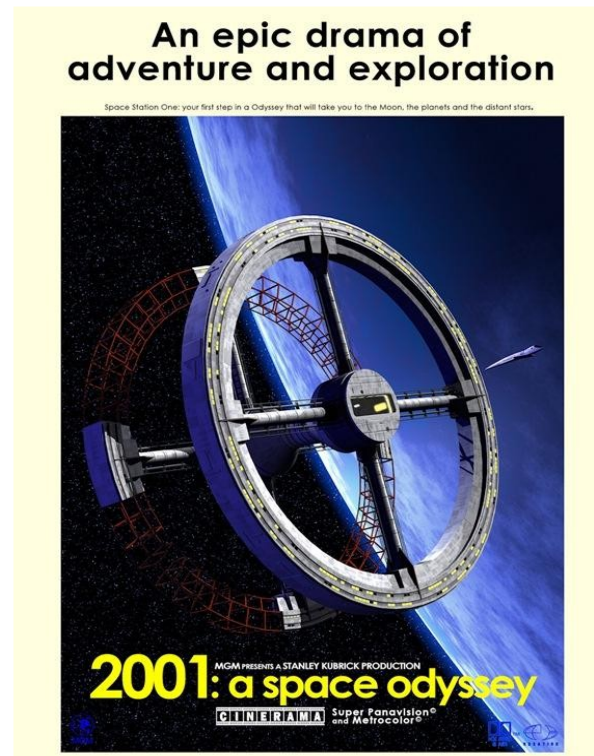
Hardware Abstraction Layer (HAL)

Bad reputation: HAL 9000

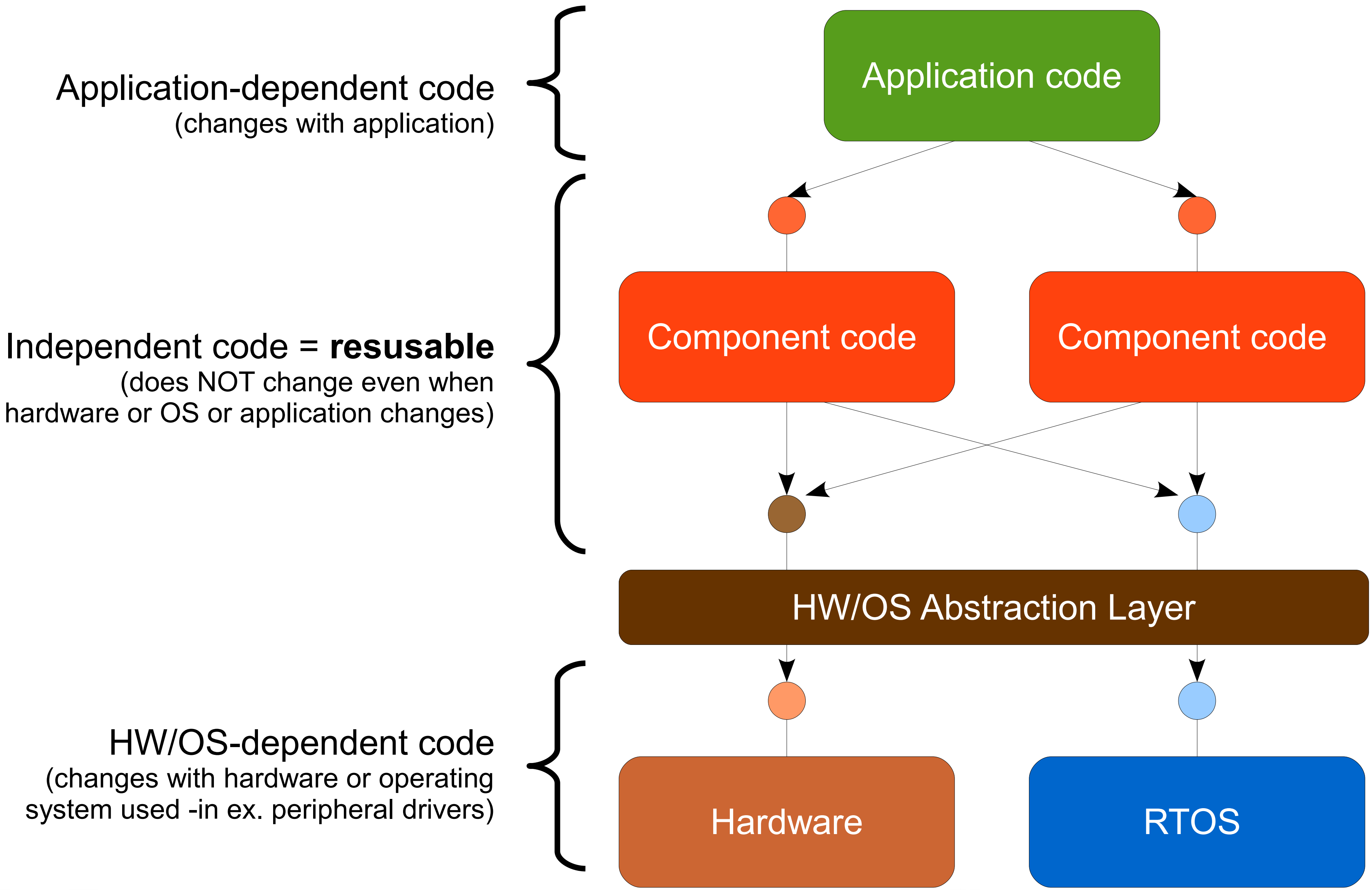




“Open the pod bay doors, HAL.”



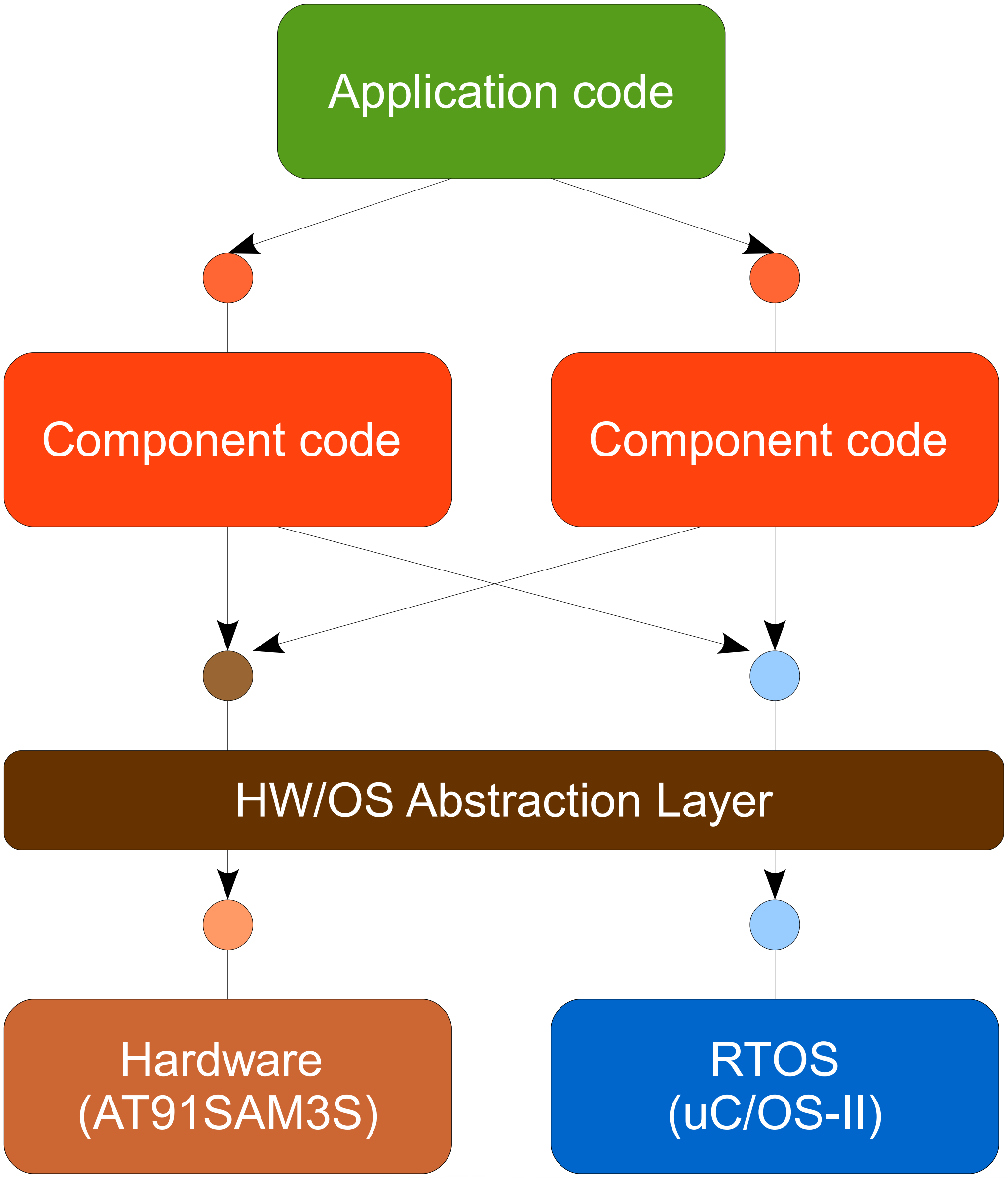
Consequences of having HAL



Advantages of HAL: switching HW/OS

It is possible to more easily **switch** to other microcontroller or other operating system during development:

- less risk in picking up wrong tools

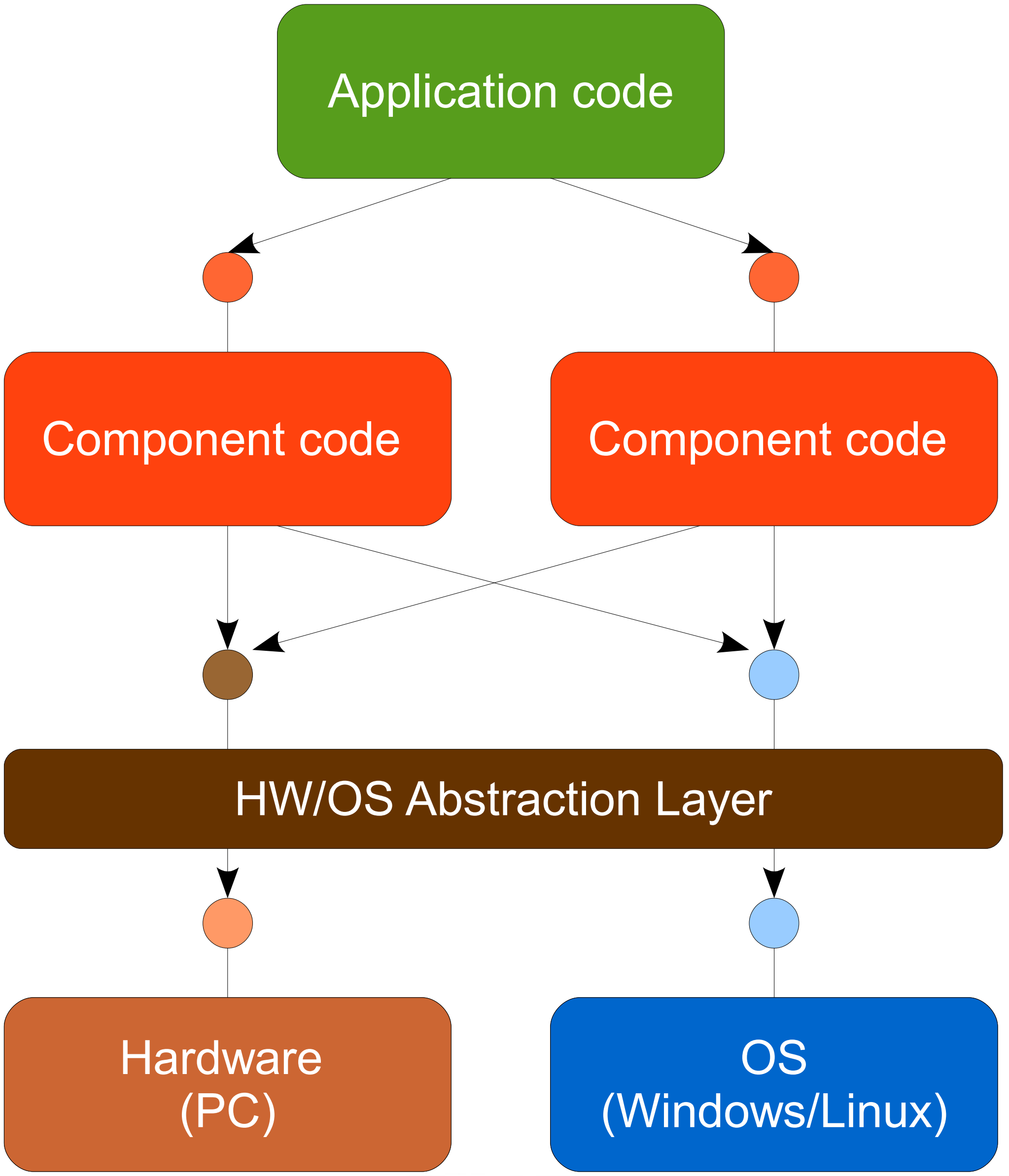


Advantages of HAL: cross-development

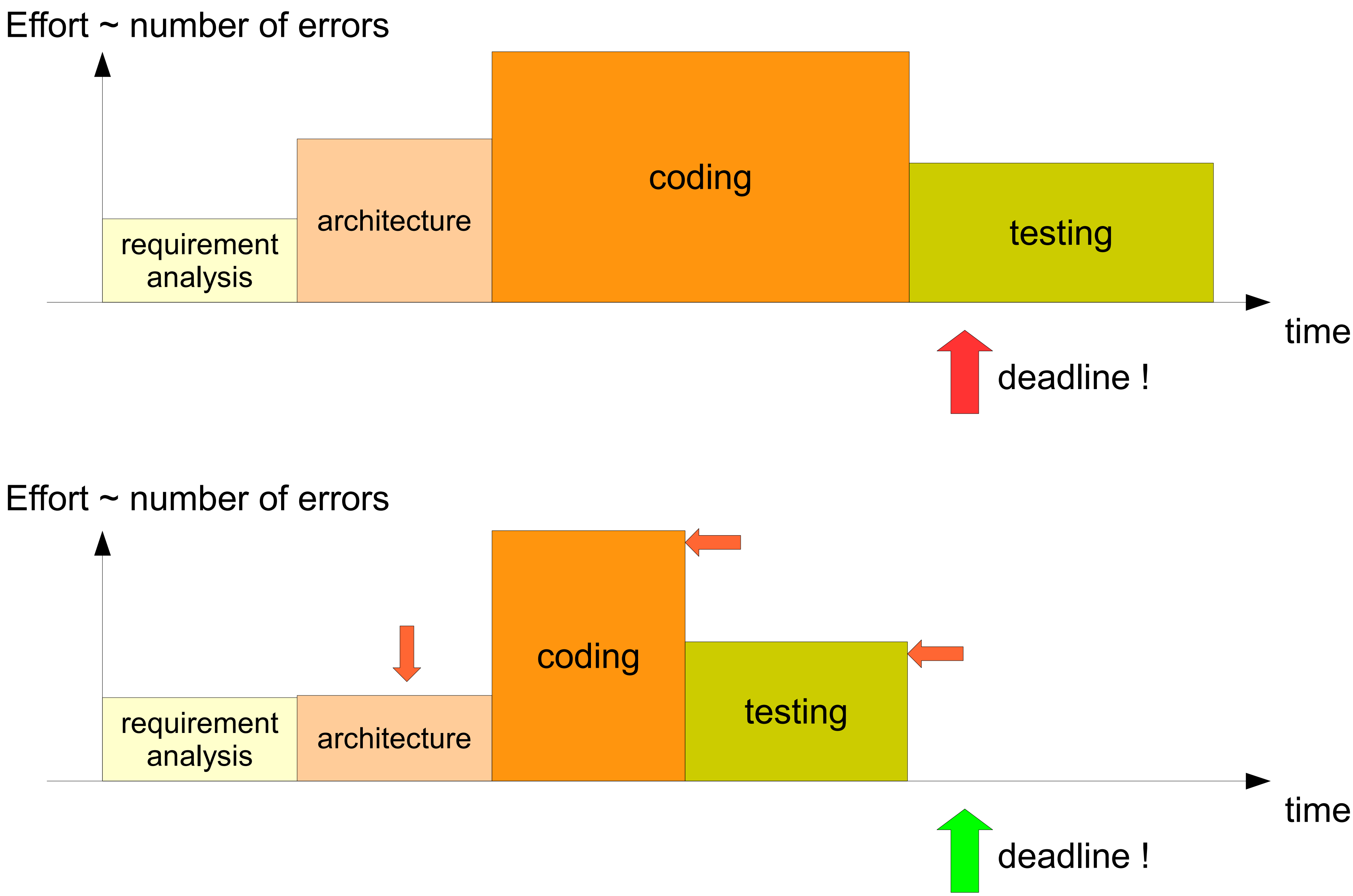
It is possible to develop component and application code in a more convenient environment on a PC:

- speeds up the development
- allows easier unit and integration testing of components
- allows to build large scale simulation environments

- stress tests not possible

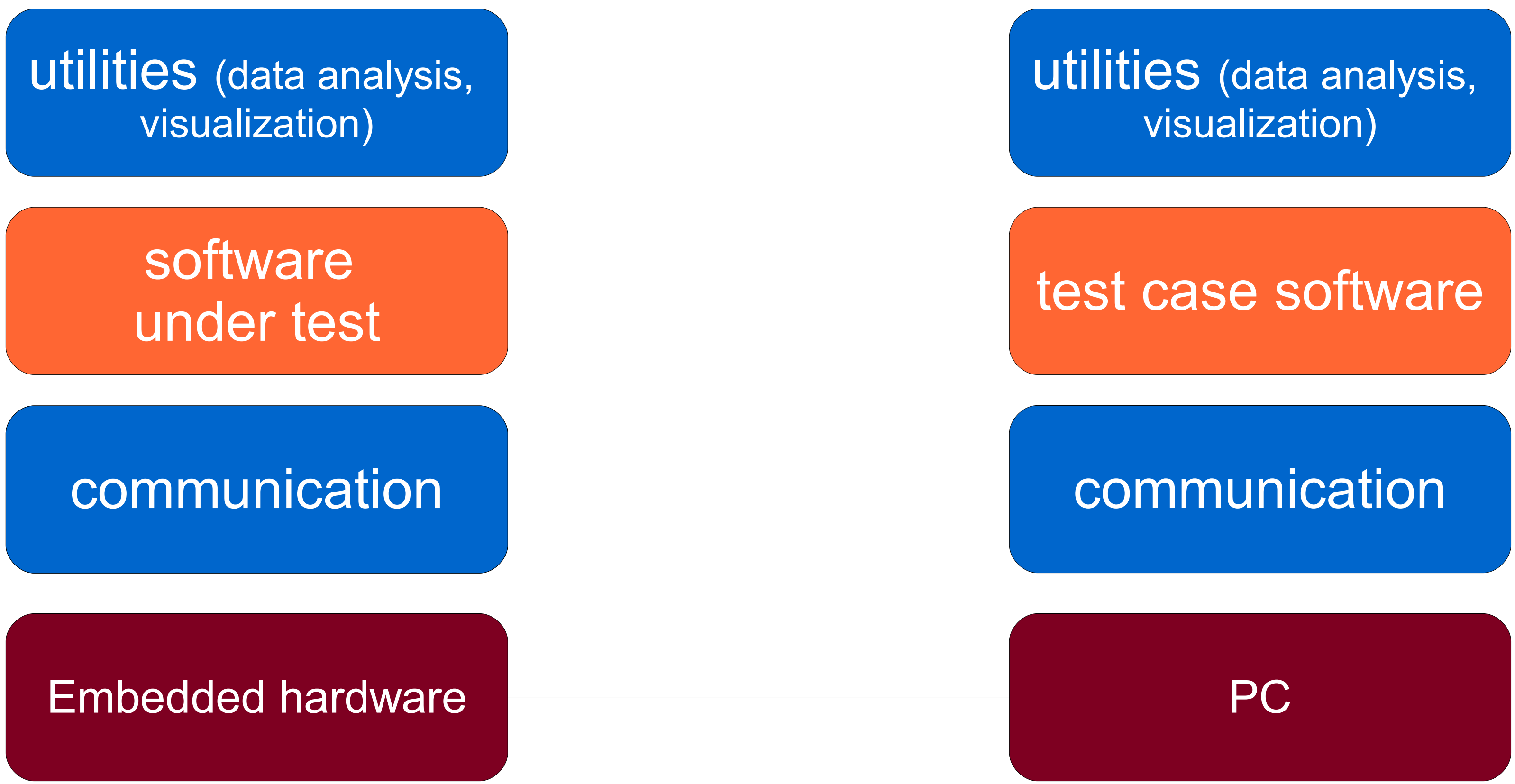


Advantages of HAL: less effort == less bugs



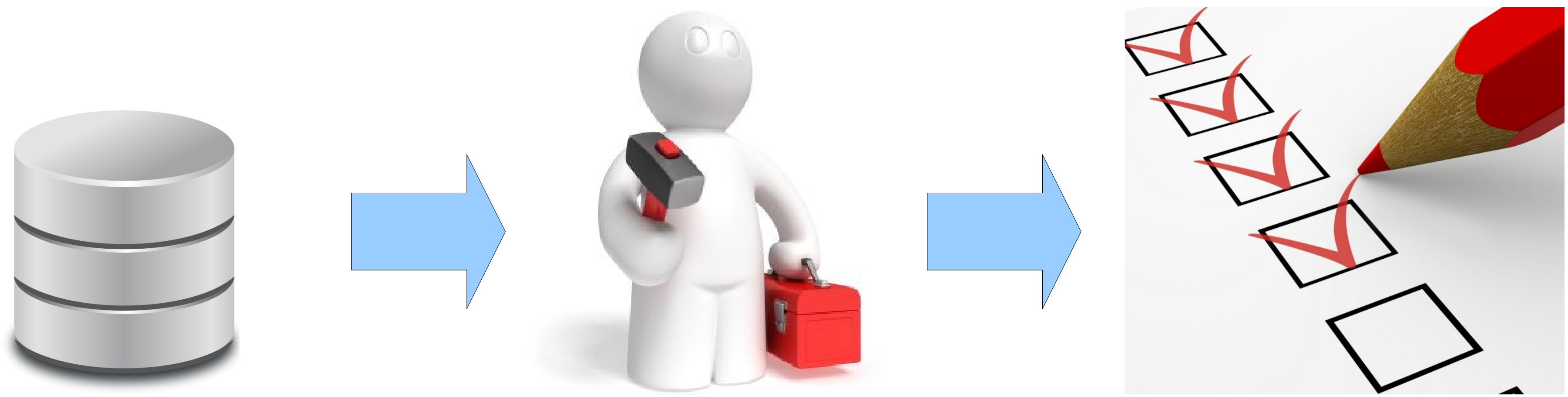
Advantages of HAL: cross platform testing

We can reuse the software across platfroms to enable communication between them. This is useful for both the application development and testing.



Advantages of HAL: automated unit testing

We can run the embedded software on a PC platform, extending the concept of continuous integration with automated unit tests.



version control repository

automated build

automated test

continuous integration system

Disadvantages of HAL – major concerns

Efficiency.

Our experiments show that handling HAL abstraction can have little or no overhead compared to chip vendor libraries. We try to follow the “only pay for what you use” paradigm. The efficiency depends heavily on the actual realization of HAL interfaces on the target platform.

Limitation of functionality due to choosen abstraction.

Although HAL cannot cover 100% of all available functionality of a microcontroller, it shall not limit the potential usage of additional functionality.

Conflicts with other frameworks / libraries / components.

Modular HAL design shall help with the integration of different libraries.

HALFRED

Hardware Abstraction Layer For Real-time Embedded Designs

www.wsn.agh.edu.pl/halfred

HALFRED wishlist

- Universal layer acting as a bridge between hardware and reusable software components
- Unified interfaces covering as much microcontroller functionality as possible
- Clear line between hardware dependent and independent code, maximizing the second one
- No assumptions about the application style
- Built-in support for multithreaded applications
- Good support for real-time applications
- Included support for in-application diagnostics
- Modular, tunable architecture
- Compatible between modern compilers
- Good documentation
- Test driven development
- Written in C (C99)

HALFRED current modules

The up-to-date documentation can be found on the project webpage:

www.wsn.agh.edu.pl/halfred

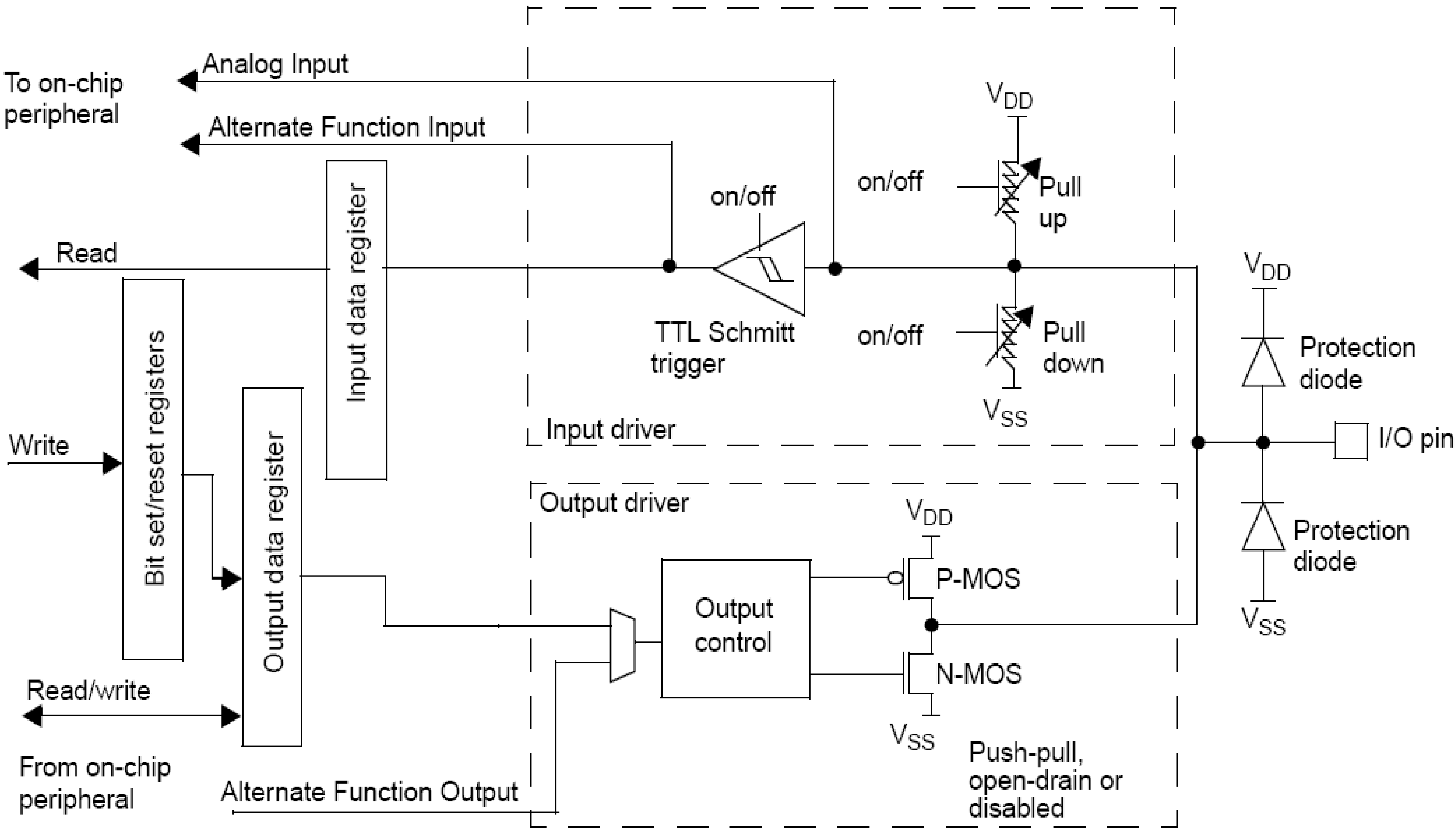
Design process example (GPIO module)

HALFRED design process

- Choose microcontroler representatives
- Analyze architecture (core, peripherals, memory etc.)
- Design abstractions (UML)
- Generate interfaces
- Write test cases
- Implement code
- Test
- Repeat :)

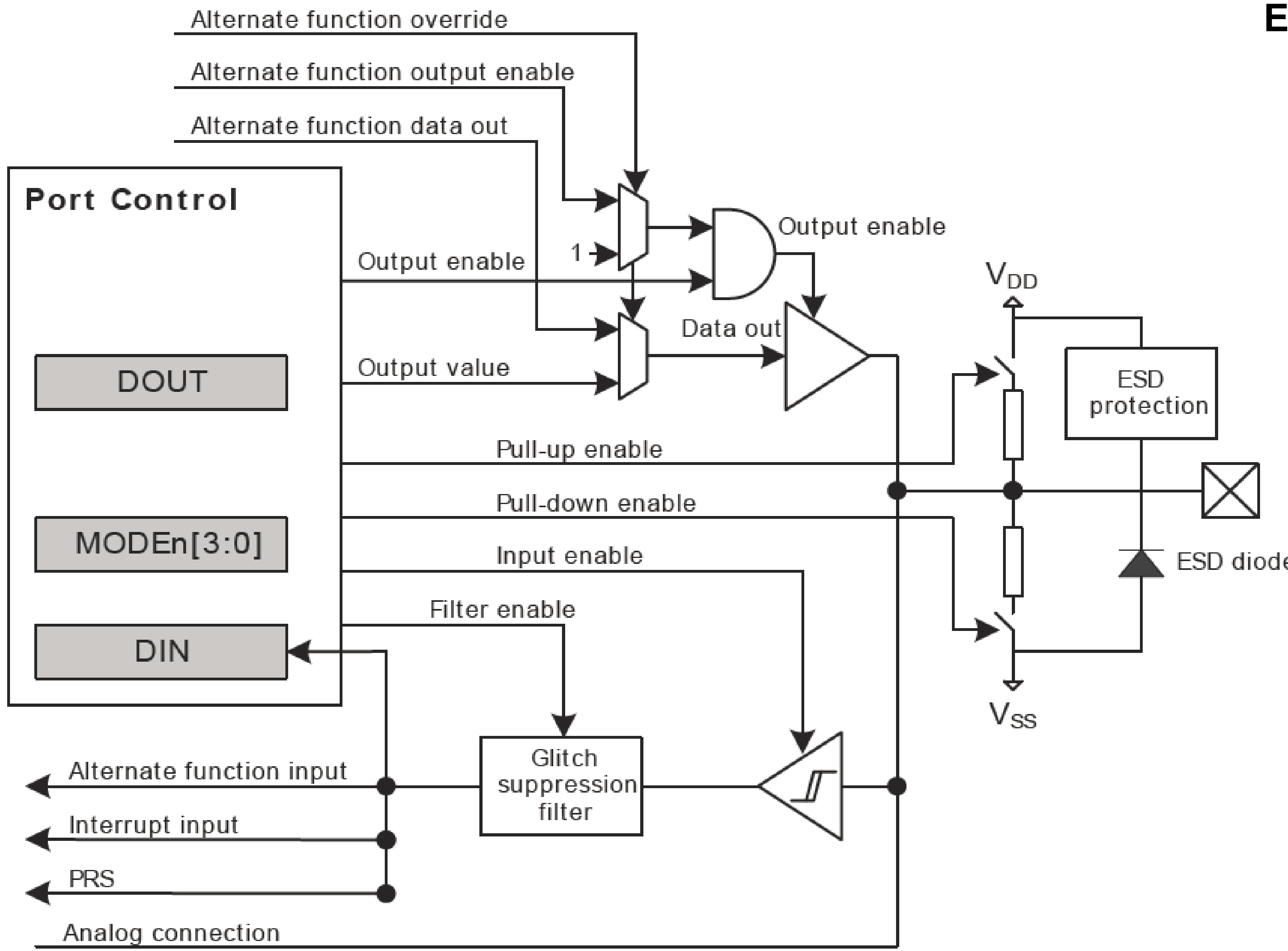
STEP1: Analyze representative microcontrolers

STM32F1

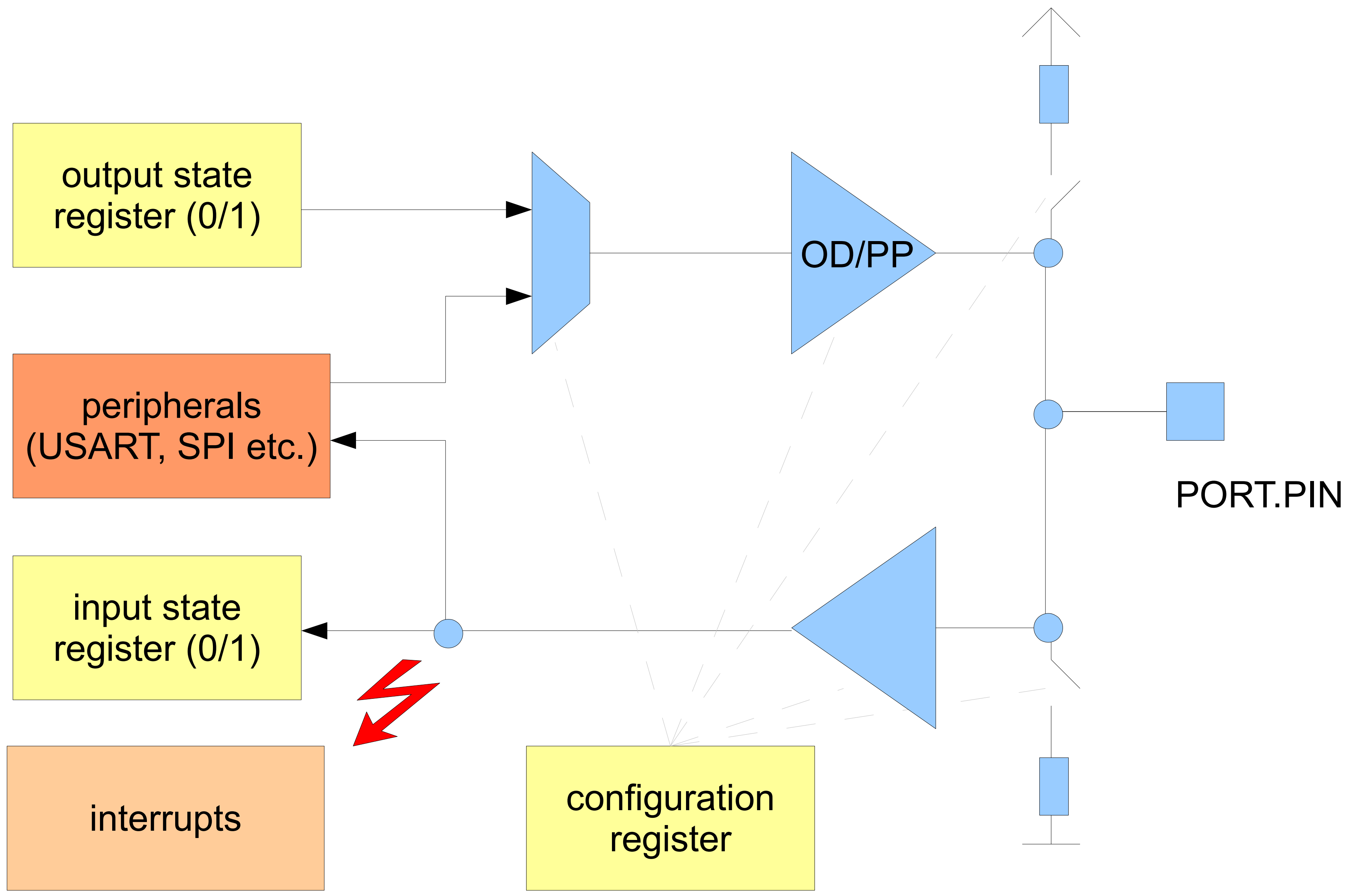


STEP1: Analyze representative microcontrolers

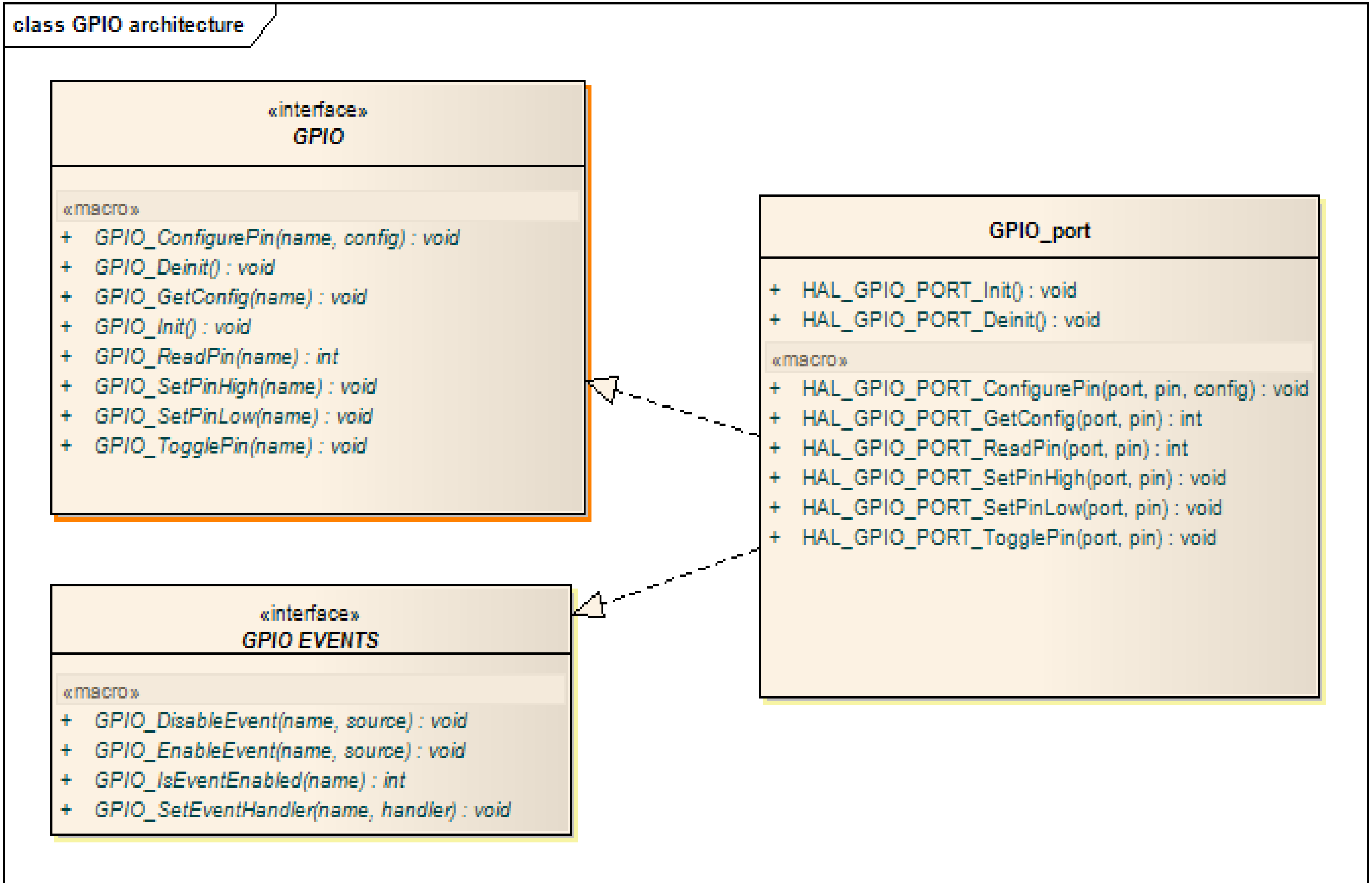
EFM32LG



STEP2: Identify abstract model



STEP3: Design and generate interfaces



STEP4: Write test case

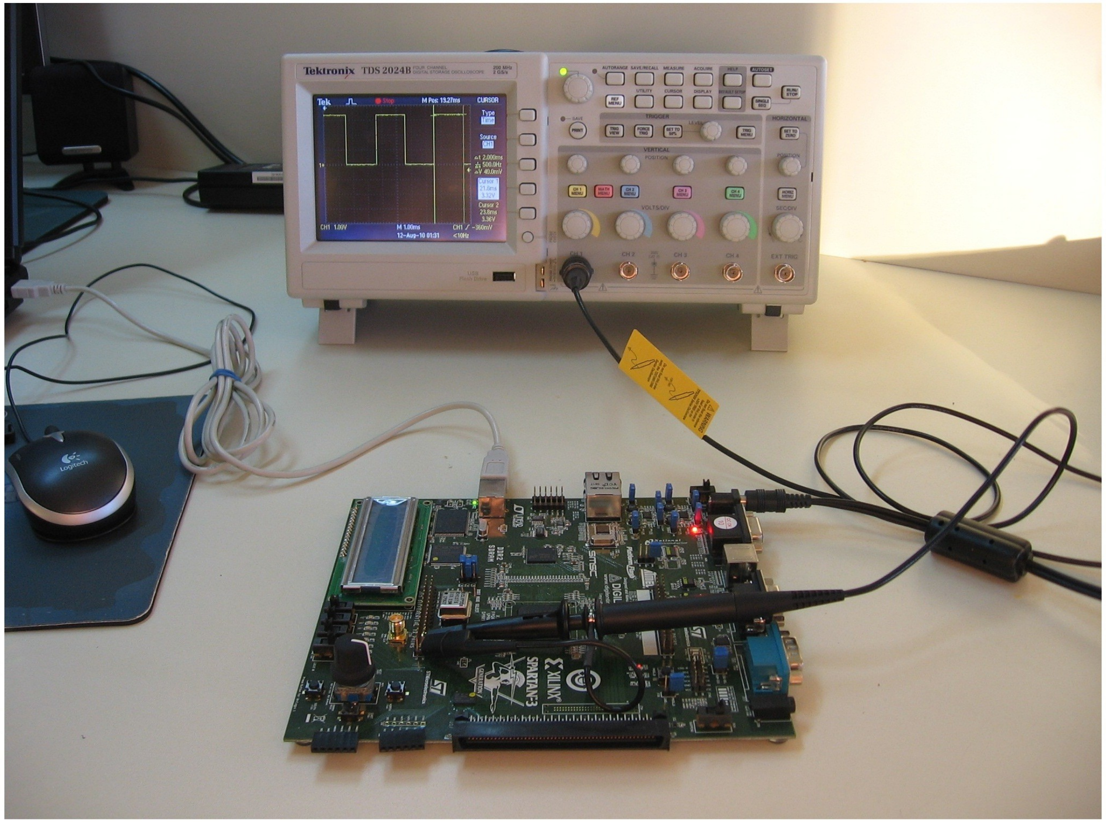
main.c

```
void testGPIO(void)
{
    int i;

    // initialize GPIO module
    GPIO_Init();

    // configure test port
    GPIO_ConfigurePin(TEST_PIN, DEFAULT_CONFIG);

    // do some GPIO stuff
    for (i=0; i < 100; i++) {
        GPIO_TogglePin(TEST_PIN);
    }

    // deinitialize GPIO module
    GPIO_Deinit();
}
```

STEP5: Implement and test

hal_config.h

```
#define TEST_PIN_PORT                GPIOC
#define TEST_PIN_PIN                 13
#define TEST_PIN_DEFAULT_CONFIG      GPIO_Mode_Out_PP
```



Results

What we've done so far

- Identified the level of abstraction needed, identified key modules
- Designed a modular architecture (UML)
- Made first implementation supporting various microcontrollers
- Documented it.
- Prepared simple examples.
- Used it in several complex real-world projects
- Gathered test results, performance metrics and user remarks
- Updated architecture and implementation based on user reviews

UNIFIED
MODELING
LANGUAGE

Supported hardware / OS / toolchain

- STM32F1, STM32F4 from STMicroelectronics
- ATSAM3S from Atmel
- EFM32LG, EFM32GG from Silicon Labs (formerly Energy Micro)
- ATmega from Atmel
- PCs

- FreeRTOS
- uC/OS-II
- Linux (posix)
- Windows (win32 api)

- GNU Compiler Collection
- MS Visual Studio

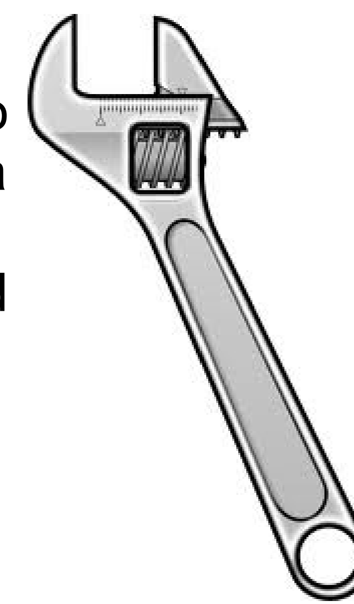
Feedback from our initial project

- STM32 turned out to be fine, we didn't have to make the switch
- Designing/implementing HAL took more time that I thought it will :)
- Having HAL positively influenced the architecture of other components
- It was easy to standarize components on HAL
- Components tested on STM32 worked out-of-the-box on AVR
- The project was deployed successfully in an industrial application

(not so) Unexpected outcomes

• Thanks to the PC port quite a lot of embedded software was **developed** (coded/debugged) in a convenient PC environment, and then just tested on the target hardware platform. It was possible to run **unit tests** on a PC.

• Having ports for linux (posix) and Windows (WIN32 API) allowed for easy writing of **cross-platform** utilities (for testing purposes).

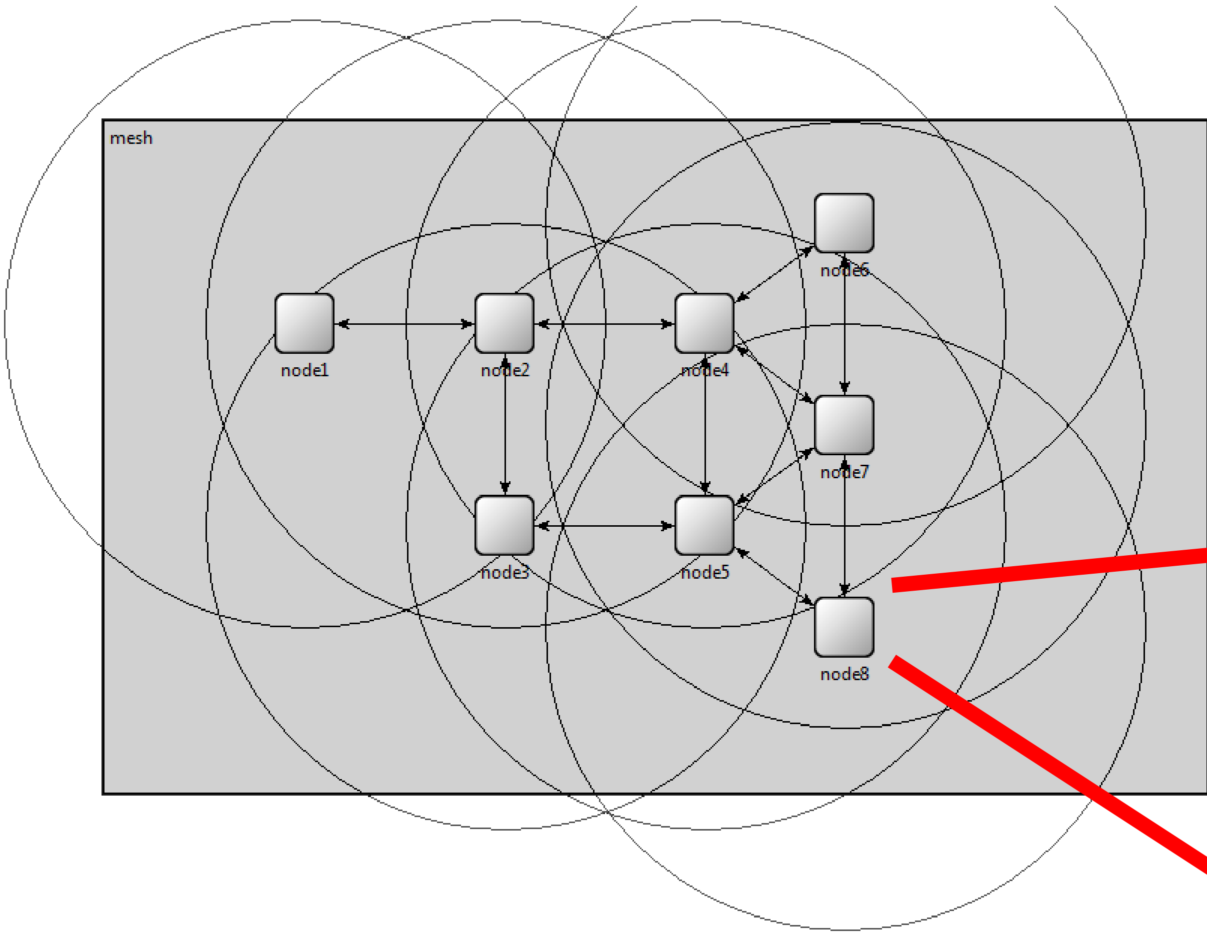


• It was easy to insert other **general-purpose** components into HAL (buffer pools, heap managers, logging/diagnostic tools, data structures)

• It was natural to incorporate **build tools** into HAL, which shifted a lot of makefile horror away from application code.

HAL became more like a **framework** than just a bunch of drivers.

Emulating and developing distributed systems



OMNeT++
discrete event simulator

Application code

Component code

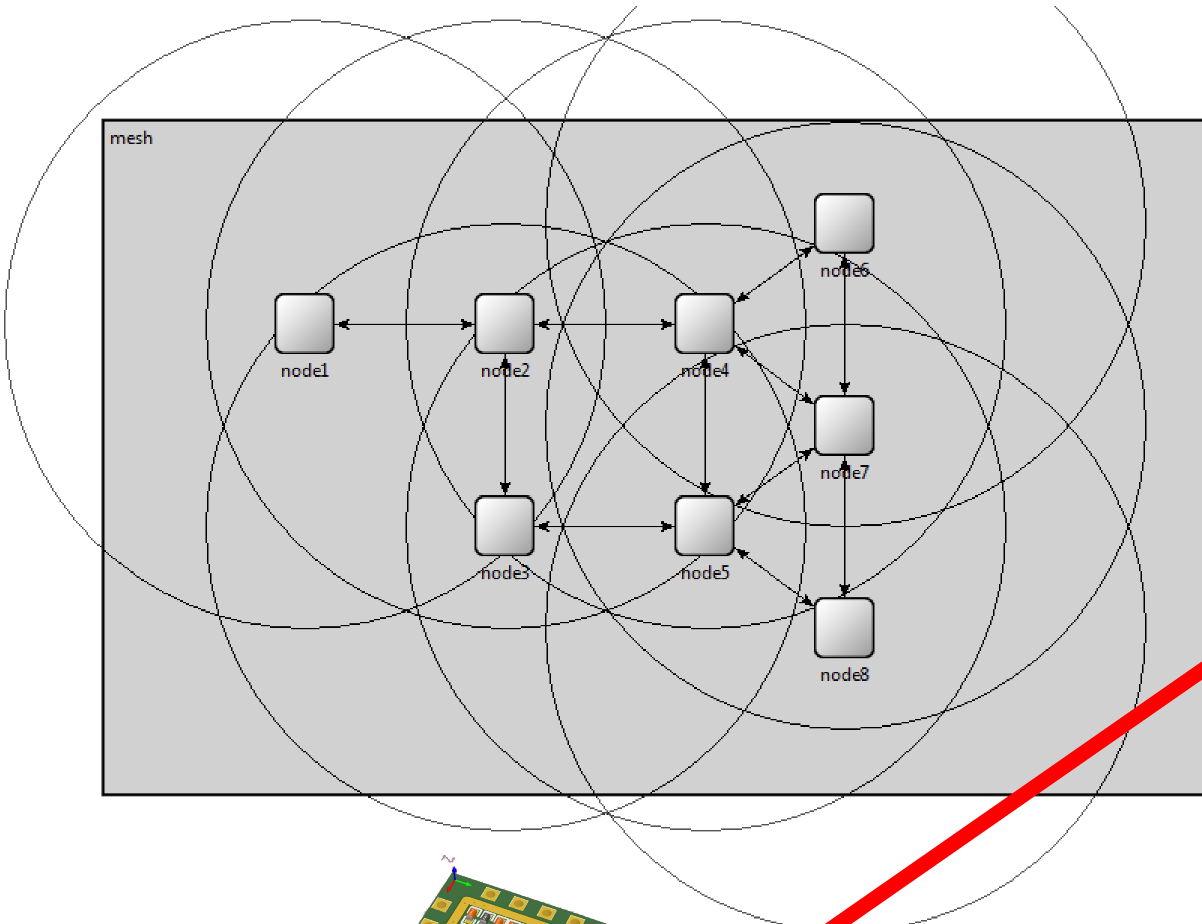
HALFRED

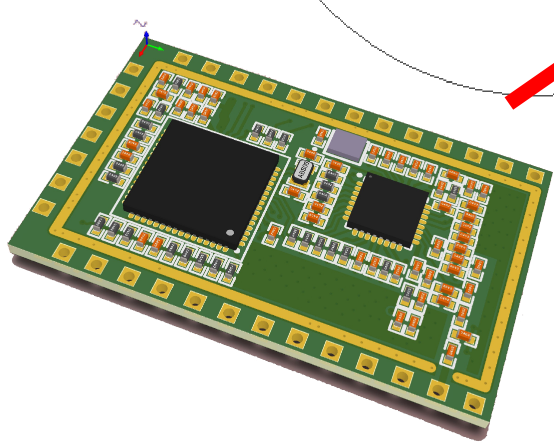
OMNeT++

Emulating and developing distributed systems

OMNeT++

discrete event simulator





- Application code
- Component code
- HALFRED
- EFM32

The future

Current design choices

Switch to **C++11**.

Depend on **GNU tools** with options to support **other toolchains**.

Do **not depend** on chip vendor libraries (efficiency, co-existence)

Make full usage of **language and toolchain features** (optimization, removal of unused code sections, detection of ambigous constructs, etc.)

Gracefully degrade when there's **no OS**.

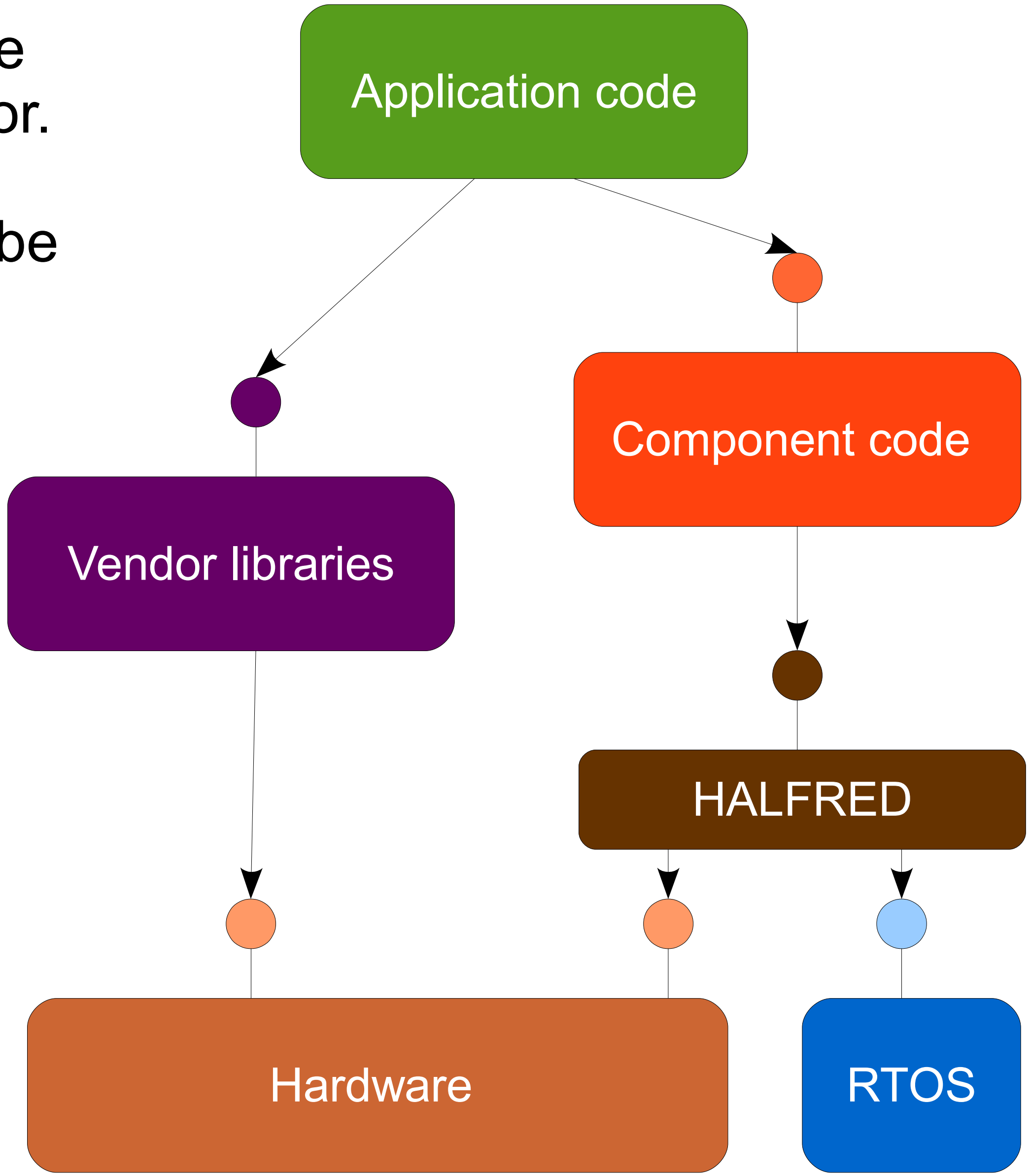
Follow reasonable safety guidelines, such as **MISRA C++**.

Consider certification options, such as **IEC 61508 SIL**.

Vendor libraries independency

HALFRED now tends not to use libraries provided by chip vendor.

This is why these libraries can be easily used by the application!
(no version conflicts)



We need your help!

Check out how can you participate on the project webpage
(link will be available soon)