Hardware abstraction layer for microcontrollers

Lukasz Krzak
Department of 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 prove 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

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

embedded software

- simple
- feature-rich
reusable == portable
Reusable == portable

2013 Embedded Market Study

Did you use the same processor as in your previous embedded project?

<table>
<thead>
<tr>
<th>Yes, used the same processor as in previous embedded project</th>
<th>No, did not use the same processor as in previous project</th>
</tr>
</thead>
<tbody>
<tr>
<td>45% 45% 44% 48% 48%</td>
<td>55% 55% 56% 52% 52%</td>
</tr>
</tbody>
</table>

2013 (N = 2,047) 2012 (N = 1,654) 2011 (N = 1,859) 2010 (N = 1,516) 2009 (N = 1,533)

Copyright © 2013 by UBM. All rights reserved.
Reusable == portable

What were your reasons for switching processors?

- New processor had better features: 46% 46%
- Previous processor too slow: 31% 31%
- New processor had better future growth...: 24% 27%
- Not my choice/processor chosen for me: 15% 15%
- New processor had better SW/dev tools: 14% 18%
- Previous processor no longer available: 13% 13%
- Needed a lower power processor: 12%
- Previous processor too expensive: 10% 9%
- To change operating system: 8% 9%
- Unhappy with previous processor's supplier: 3% 2%
- Other: 9% 11%

Base = Those who did not use the same processor as in previous project

source: ubmdesign.com / http://www.slideshare.net/MTKDMI/2013-embedded-market-study-final
Reusable <=> portable

2013 Embedded Market Study

Why did you use the same processor?

- Happy with current processor/supplier: 62% (2013), 59% (2012)
- To maintain software compatibility: 50% (2013), 57% (2012)
- To maintain the same tools or software: 49% (2013), 47% (2012)
- To make use of expertise/familiarity: 43% (2013), 41% (2012)
- To use same operating system: 29% (2013), 28% (2012)
- Switching is too expensive/time-consuming: 23% (2013), 27% (2012)
- Not my choice/processor chosen for me: 9% (2013), 8% (2012)
- No other suitable processors available: 4% (2013), 3% (2012)
- Other: 2% (2013), 3% (2012)

Base = Those who are currently using the same processor as in previous project

because changes sucks!
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 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!

```c
#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

- User Interface
- Data processing
- Graphical User Interface (GUI) Library
- Communication stack
- LCD driver
- Communication port driver
- Hardware
How to design reusable/portable code?

My Precious Code
Important design choices

Applications

My Precious Code

What is the potential range of applications?
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

- Applications
- Other libs
- My Precious Code
- Hardware
- RTOS (Really Tricky Operating System)

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

Applications

Toolchain (Complainer etc)

Other libs

My Precious Code

Standards

Hardware

RTOS (Really Tricky Operating System)

What **standards** must be obeyed?
How to design reusable software?

- Application code
- Component code

- Defines usability
- Defines portability

- Hardware
- RTOS
How to design reusable software?

Application code

Component code

HW glue

OS glue

Hardware

RTOS

“egocentric” approach
How to design reusable software?

- Application code

  - Component code
    - HW glue
    - OS glue

  - Component code
    - HW glue
    - OS glue

- Hardware

- RTOS

“egocentric” approach multiplied
Towards abstraction...

- Application code
  - Component code
    - HW interface
      - Hardware
    - RTOS interface
      - RTOS
  - Component code
    - HW/OS abstraction layer
Hardware Abstraction Layer (HAL)
Bad reputation: HAL 9000

“Open the pod bay doors, HAL.”

I’m sorry, Dave. I’m afraid I can’t do that.
Consequences of having HAL

Application-dependent code
(changes with application)

Independent code = resusable
(changes with hardware or OS or application changes)

HW/OS-dependent code
(changes with hardware or operating system used -in ex. peripheral drivers)
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

Effort ~ number of errors

- requirement analysis
- architecture
- coding
- testing

deadline!

time

Effort ~ number of errors

- requirement analysis
- architecture
- coding
- testing

deadline!

time
Advantages of HAL: cross platform testing

We can reuse the software across platforms 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.
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 chosen 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 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 microcontroller representatives
- Analyze architecture (core, peripherals, memory etc.)
- Design abstractions (UML)
- Generate interfaces
- Write test cases
- Implement code
- Test
- Repeat :)
STEP1: Analyze representative microcontrollers

STM32F1
STEP1: Analyze representative microcontrollers

EFM32LG
STEP2: Identify abstract model

output state register (0/1)

peripherals (USART, SPI etc.)

input state register (0/1)

interrupts

configuration register

OD/PP

PORT.PIN
STEP3: Design and generate interfaces
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

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

hal_config.h
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
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 standardize 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 ambiguous 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**.
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)