energyAware Drivers

Requirement Specification

Introduction

This document describes the rationale and background for the energyAware Driver project. In addition, it describes the goals and purpose of the project.

1. Revision Records

Revision Number	Effective Date	Change Description	Author	Approved
0.1	25.06.2012	Initial revision	JNO	
0.2	14.09.2012	Updated with new requirements. Added information from the wiki	MG	
0.3	19.09.2012	Updated requirement 4.1.2 to clarify which use-cases are covered. Added new requirement 4.1.9 runtimes. Added figure 1 to clarify the relationship between different components.	MG	
0.4	24.09.2012	Updated with feedback from version 0.3. Clarified section 4.1.13 on the pointer mechanism. Added a Scope section.	MG	
0.5	24.09.2012	Added some requirements, clarified some requirements and improved language.	JNO	

Contents

2. Background

Today we have emlib, which is a small Hardware Abstraction Layer (HAL). Today, emlib covers all peripherals for the microcontroller side with good test coverage and a mature code-base. Emlib was developed with a few ground rules:

- As few as possible dependencies across modules
- No memory allocation
- No interrupt handlers

These limitations lets the developer just include all the emlib files without increasing the size of the binary. By not including interrupt handlers we increase flexibility, since there are many potential ways to use the interrupt handler for different functionality. By disallowing dependencies across modules (where possible), we make sure that we don't consume resources that the developer didn't expect. Because emlib is a HAL, we also look at this from a module perspective, rather than a functionality perspective. As an example, the TIMER module can be used to both time external events and as a PWM source. Since both of these functionalities are covered by emlib, emlib needs to take care of both cases.

However, these limitations taken together, severely limits the usability of emlib. In practice, getting data out of the USART for instance, requires many lines of code, typically encompassing clock setup, usart setup, gpio setup, route registers. In addition, if DMA or interrupts are used, then more code is needed.

3. New energyAware Drivers

First, let's look at why we want to create drivers/higher level abstractions for the Energy Micro ecosystem. There are two main goals in this regard:

- Simplicity
- Energy Friendliness

3.1. Simplicity

We want it to be as easy for the developer to use our drivers as possible. Ideally, the less configuration that needs to be done, the better. This comes at the expense of flexibility, which means that we will need to cover the most common use-case instead of supporting every possible combination. Thus, some customers will need to either use emlib directly, or modify our driver. In addition, some of our peripherals (especially LESENSE) requires a lot of in-depth knowledge to set up correctly. Minimizing this effort would be a huge boost.

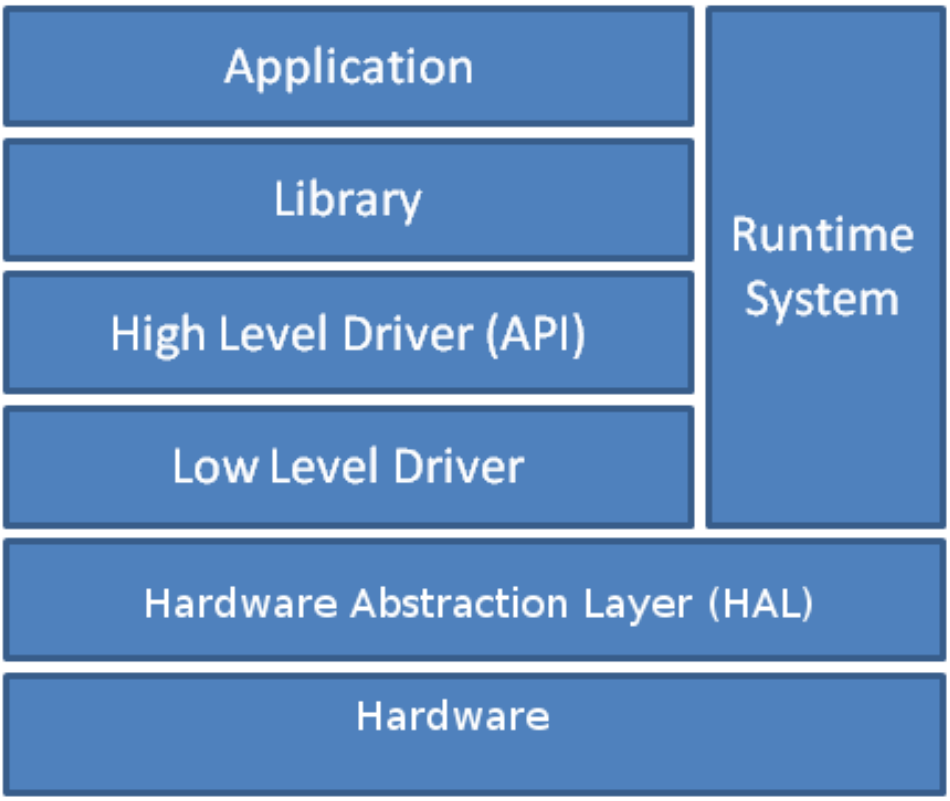
3.2. Energy Friendliness

The other objective is to optimize for energy-friendliness. Support has reported several cases where they have been given optimized code by customers where they have been able to reduce the power consumption by 100x. This happen often in the benchmarking / evaluation phase of our products. Of course, we have no idea about how many people have failed to use our energy-friendly peripherals correctly and ended up with a system consuming more power than necessary. By providing energy-optimized drivers we can reduce this gap.

3.3. Relation to emlib

Where emlib focuses on the modules themselves (for instance the USART module), the drivers should focus on functionality. Therefore there should be separate drivers for UART, SPI, IrDA functionality. This is especially true for complex modules such as LESENSE. This module is very difficult to use, simply because of its complexity. Therefore, providing drivers for capacitive touch, inductive sensing would provide a huge productivity boost.

The energyAware Drivers will typically use emlib where that makes sense (See figure 1). If functionality is missing from emlib, then adding that functionality in emlib falls within the scope of this project.



Figur 1 Software Architecture

4. Scope of the project

We wish to provide energy friendly drivers for as many use-cases as possible. Because there is a huge variation in different applications, there is also a huge amount of potential driver candidates. We are not aiming at solving every use-case out there, but rather provide drivers for the most common use-cases.

However, as Energy Micro expands in terms of market share, application space and number of devices, the number of use-cases, and thus applicable drivers will also increase. Thus, this project, like emlib, will never be truly finished, but is rather a living project that constantly changes according to the needs of our customers.

However, it is important to prioritize which drivers should be developed and in what order. In this regard, the drivers that provide the most customer value is to be prioritized over drivers which provide less customer value.

5. General Requirements

5.1. Design

5.1.1. Programming language

The drivers must be written in ANSI C. Additionally the drivers shall be written for use in a C++ environment, and as such have the necessary extern “C” declarations added where appropriate.

5.1.2. Focus on functionality

Each driver should attempt to focus on one single use-case. E.g., even though the USART module itself supports UART, SPI, I2S, IrDA, the driver itself should focus on one these functionalities. Other examples include capacitive touch and inductive sensing for the LESENSE module. However, each driver should not cover every use-case, but focus on the most common ones.

To cover special needs, a user can either use emlib directly to tweak or modify the operation of the driver, or write a new driver entirely.

5.1.3. Focus on low energy

Each driver should focus on minimizing the energy requirements of the driver, and ideally represent the lowest possible energy consumption the hardware can offer.

Each driver should also be able to report its own resource requirements (energy mode etc).

5.1.4. Focus on simplicity

It should be easy to use the driver. Getting started with a driver should not take more than about 10 lines of C code.

The drivers need to be well written, have a clean and consistent API, and needs to be well documented.

5.1.5. Same API for different implementations

If it is possible to implement some functionality using two different modules (e.g capacitive sensing could be implemented using ACMP or LESENSE), then the API for both should be the same.

In the case that there are different solutions to an energy optimizing driver as a function of operational parameters, these should be runtime configurable, and ideally hidden from the end user. The ADC driver is a good example of this, since the choice of the sample rate will dictate different solutions. For low sampling rates, a driver would typically stay in EM2 most of the time, wheras for high sample rates this is impossible, yielding EM1 as the lowest energy mode.

5.1.6. API should be technology agnostic

We expect that our peripherals will change over time, so it is important that the API itself is not tied to specific EFM32 functionality.

For some of the drivers, an STM32 version will be created to help with the kit porting effort.

5.1.7. All drivers should be multiple-instance (where applicable)

All drivers should be possible to instantiate multiple times. It does not make sense to limit ourselves to one single SPI driver for instance. However, for peripherals where this does not make sense, this

can be ignored (examples are WDOG, DMA and LCD). If you are in doubt, make the driver multi-instance.

5.1.8. Object oriented design paradigm

C is an imperative language, so to clarify what we mean by object oriented design paradigm. We want each function in the driver to use a handle as the first argument. The handle is a pointer to an instance object where all instance state variables etc. are stored. Ideally, the driver code itself shall contain no global state variables. Only when this is strictly necessary shall global state variables be present in the driver.

5.1.9. Runtime Environments

The drivers must be able to co-exist or cooperate with different runtimes, including a minimal runtime provided by Energy Micro that provides a minimum set of functionality for proper driver operation (e.g timers, IRQ dispatchers etc).

5.1.10. All drivers need to be preemptive and thread-safe

To facilitate the use of the drivers under different runtime environments (RTOS, main-loop, minimalist etc), we need to make sure that all drivers are preemptable and thread-safe.

A piece of code is ***thread-safe*** if it only manipulates shared data structures in a manner that guarantees safe execution by multiple threads at the same time. [Wikipedia]

Also, the code needs to be written to be preemptable, meaning that every driver can be preempted by a higher priority task at any time.

5.1.11. Low memory footprint

Each driver should strive for a low memory footprint, both in terms of SRAM and flash. This is important because some of our customers are very price-sensitive (especially in Asia) and will not use a driver they deem to be excessive in terms of memory usage.

5.1.12. Future-"proof" API

Predicting the future is hard. Changing function signatures should be avoided. Keep this in mind when designing an API for a driver, so that possible expansions are made impossible.

5.1.13. Status reporting and handling.

Each function should return a status code. Even if this is just hardcoded to be OK. Return values are returned as pointers. In emlib, there is little opportunity for a call to fail, but a driver has many failure modes, so we need to be consistent in the way we report status and failures. Example:

```
EMSTATUS MODULE_ReadData(int numBytes, void *destBuffer);
```

In this case, the function will return an error code if there is an error ant destBuffer will not be updated. For pure get methods, returning the value is OK. Example:

```
uint32_t UART_BaudRateGet(void);
```

5.1.14. Consistent API

The API should be consistent across different drivers. The idea is that if you know one driver, you already know how to use the next. This means that each driver should have the same format or

mindset. Where applicable, use the same function names. A good example is that the different serial communication devices such as USART, UART, LEUART and SPI have the same API names for writing and reading data.

5.2. Coding Style

5.2.1. General

The coding style of the energyAware drivers should follow the general guidelines that are documented on the wiki .

5.2.2. Naming convention

Special care should be taken to avoid naming conflicts with emlib, making sure that it is clear from the naming of functions if it resides in emlib or in the driver project.

5.2.3. Consistent wording

API needs to be designed to be highly consistent across all drivers. Verbs used in APIs should be **last**;

```
ADC_SampleRateSet()
ADC_SampleRateGet()
```

5.3. Documentation

5.3.1. Doxygen

Doxygen code should be used wherever possible.

5.3.2. User guide

After the completion of each module a small user guide should be written. This user guide should contain a basic overview of how to use the driver. This can be written as an application note in cooperation with the support team.

Energy Micro AS
Sandakerveien 118
P.O. Box 4633 Nydalen
N-0405 Oslo
Norway

www.energymicro.com