# avr-halib
## Eine Hardwareabstraktion für AVR Mikrocontroller in C++

Karl Fessel Philipp Werner

EOS.IVS.FIN.OvGU

15. März 2010

# avr-halib

# avr-halib: AVR hardware abstraction library

## AVR

- für AVR Mikrocontroller
- für avr-gcc toolchain

## hardware abstraction

- Maskierung von Heterogenität
- einfache Schnittstelle

## library

- Bibliothek
- C++

# Designziele

- leichte Portierbarkeit (Anwendung und Bibliothek)
  - andere AVR Mikrocontroller
  - andere Hardwarekonfigurationen
- einfache Verwendung
- geringer Overhead (RAM, Programmspeicher und Laufzeit)

# Template-kofigurierbare Klassen (TKK)

Trennung von allgemeiner Schnittstelle und spezifischer Konfiguration

```
Led<Led0> statusLed;
```

- Led (Templateklasse): allgemeine Schnittstelle und Algorithmus (z. B. Setzen, Toggeln, Statusabfrage)
- Led0 (Templateparameter): Hardwareschnittstelle via Portmap
- statusLed (Objektbezeichner)

# Template-kofigurierbare Klassen (TKK)

## Vorteile

- große Flexibilität (z. B. Parametrisierung zur Compilezeit)
- Komplexität des Quellcodes zur Compilezeit aufgelöst
- nur verwendete Funktionen werden generiert
- ohne oder geringer Overhead

## Nachteile

- bei komplexen Klassen, die mehrfach mit verschiedenen Parametern verwendet werden, wird mehr Code generiert

# TKK: AVR-Klassen und Registermaps

## AVR-Klassen

- abstrahieren AVR-Komponenten (ADC, Timer, Uart, Ports...)
- Konfiguration und prozessorspezifische Implementierung über Registermaps (Template-Parameter)

## Beispiel:

```
Uart<Uart1> uart;
uart.put("A");
```

# TKK: Device-Klassen und Portmaps

## Device-Klassen

- abstrahieren externe Hardware, die über Ports angesprochen wird (LEDs, Buttons, LCD...)
- Konfiguration über Portmaps (Template-Parameter)

## Portmaps

- Hardware-Pins werden funktionelle Bezeichner zugeordnet
- Definition in spezieller Syntax
- automatische Generierung des C++-Codes

# TKK: Schnittstellenvereinfachung

Zusatzfunktionen und/oder vereinfachte Anwendung für AVR- oder Device-Klassen

## Beispiel: CDevice

```
CDevice< Uart<Uart1> > uart;
int32_t a, b;
uart.readInt(a);
uart.readInt(b);
uart << a << "+" << b << "=" << (a+b) << "\n";
```

# Delegates

Verwendung von Delegates für Interrupts

- C-Funktionen und Objektmethoden als ISR
- ISR zur Laufzeit austauschbar (mit AVR-libc nicht möglich!)
- skaliert gut

## Beispiel:

```
class foo {
public:    void dot() { /* ISR */ }
};

int main() {
  foo hallo;
  redirectISRM(SIG_INTERRUPT3, &foo::dot, hallo);
}
```

# Ordnerstruktur

- **build**: zu linkende Bibliothek
- **docs**: Dokumentation
- **examples**: Beispiel-Programme
- **include**: der größte Teil der Bibliothek befindet sich hier
  - **avr**: Abstraktion der Mikrocontrollerkomponenten (z. B. Port, ADC) und Interrupt-Mechanismus
  - **ext**: Abstraktion von externen Komponenten (z. B. LED, Sensor)
  - **share**: gemeinsam genutzte Klassen, Klassen zur vereinfachten Verwendung von Klassentypen (z. B. Sensoren, zeichenorientierte Kommunikation)
  - **portmaps**: Häufig verwendete Portmaps (z. B. RobbyBoard)
- **src**
- **tools**: Portmap-Generator

# Erste Schritte

1. `avr-halib$ make`
2. Beispiele unter `avr-halib/examples/application` anschauen
3. Ordner für Projekt und cc/cpp-Datei anlegen und Makefile schreiben (oder von Examples kopieren und anpassen)
4. Includes für benötigte Komponenten und eigenen Quellcode einfügen, ggf. nötige Portmaps erstellen

# Beispielprogramme

# Status und Planung

## Status

- Ports, Interrupts, Uart, ADC, einfacher Timer, Buttons, Leds, Sensoren, SHT-Sensoren, Motor, LCD
- Dokumentation für Portmap-Konzept und einige Klassen

## Planung

- Code aufräumen und dokumentieren
- I2C-Bus
- Timer-Implementierung

# Fragen

???