technische universität
dortmund

LavA OS: Ein Betriebssystem für konfigurierbare MPSoCs

Diplomarbeit Abschlussvortrag

Stephan Vogt

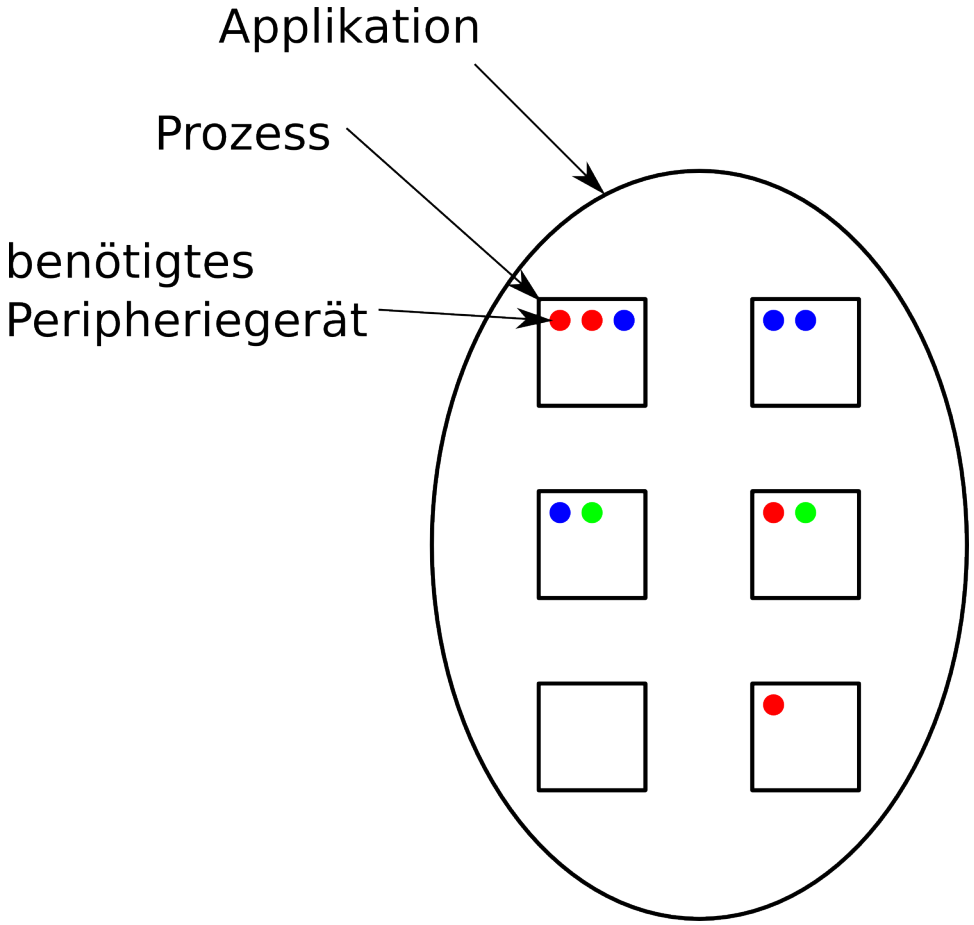
stephan.vogt@cs.uni-dortmund.de

Inhalt

- **Einleitung**
- Wiederverwendung von BS
- Arbeiten an CiAO
- Kommunikation
- Evaluation
- Fazit

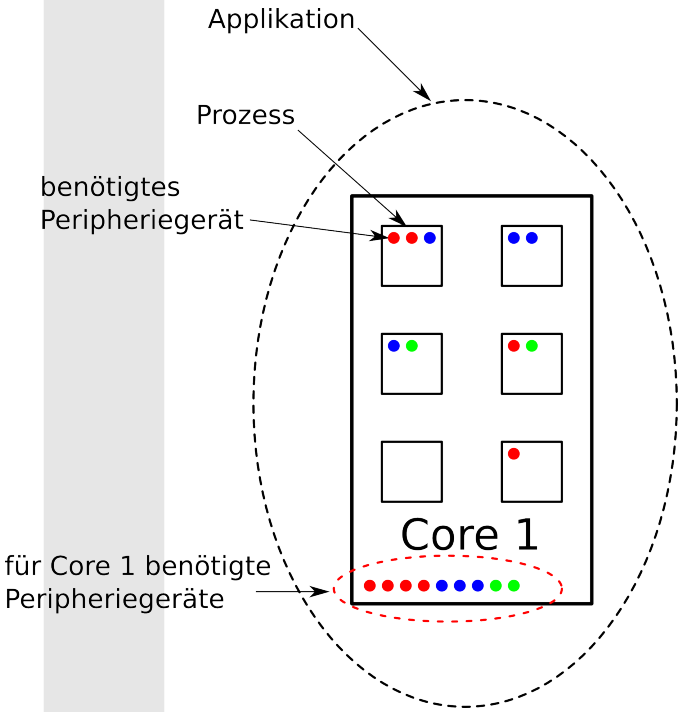
Struktur der Anwendung

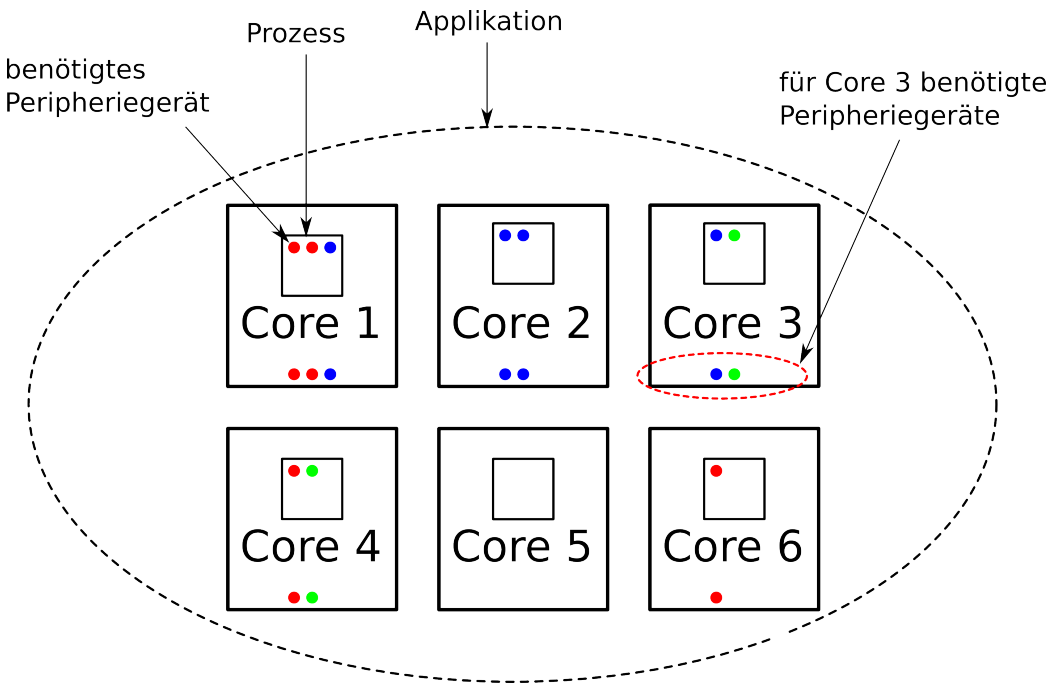
- Anwendung besteht aus mehreren Prozessen
- Jeder Prozess benötigt bestimmte Geräte (wie UART)
- Prozesse werden auf Kerne verteilt



Mögliche Aufteilungen

- Simple Aufteilung
  - Alle Prozesse auf einem Kern
  - Jeder Prozess auf eigenem Kern





Inhalt

- Einleitung
- **Wiederverwendung von BS**
- Arbeiten an CiAO
- Kommunikation
- Evaluation
- Fazit

Wiederverwendbarkeit OS

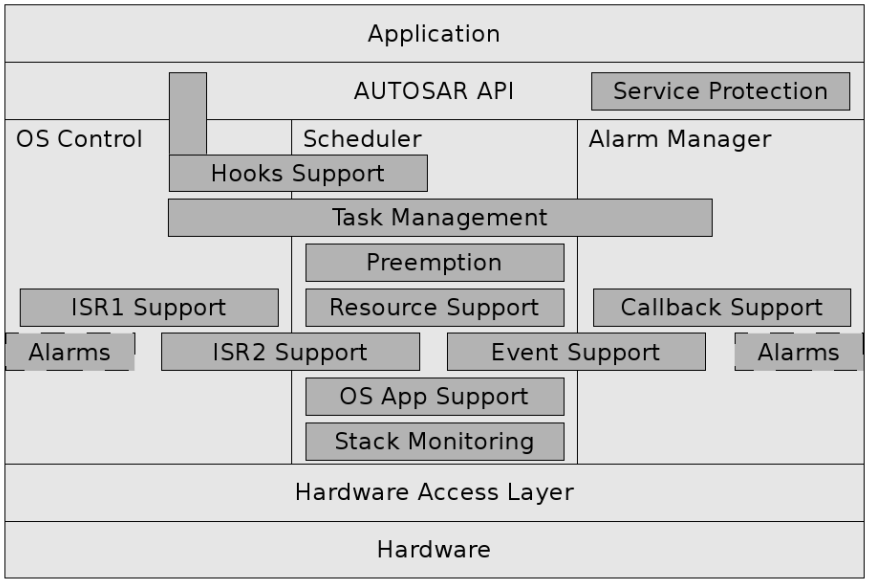
- Verwendung/Anpassung eines vorhandenen OS (bzw. dessen API) oder Neuentwicklung?
- Vorhandene Programme/bewährte API gegen stark konfigurierbare Strukturen im LavA Projekt
- Anforderungen: Konfigurierbarkeit, statische Konfigurierbarkeit, Analysierbarkeit, Kommunikationsfähigkeit, kein shared-memory System
- OS aus den Bereichen Mikrokerne, Verteilte Systeme und Embedded Systems wurden untersucht
- Resultat: CiAO als Basis für LavA OS

Inhalt

- Einleitung
- Wiederverwendung von BS
- **Arbeiten an CiAO**
- Kommunikation
- Evaluation
- Fazit

CiAO

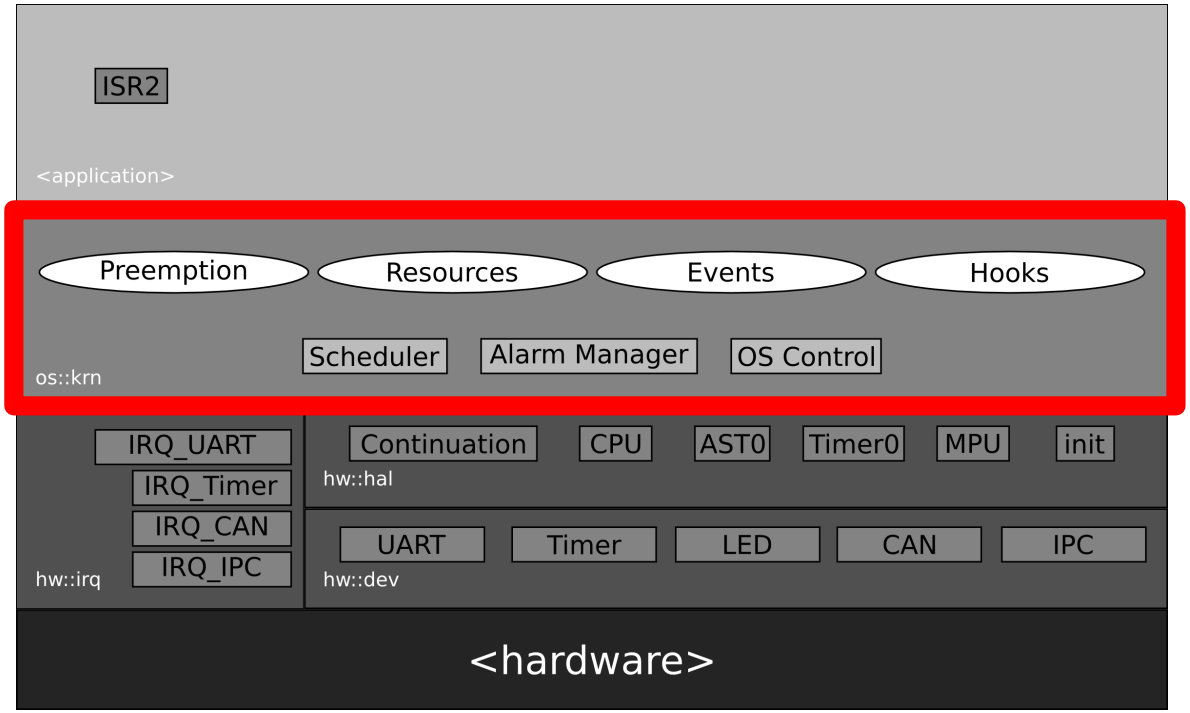
- CiAO (CiAO is aspect oriented)
  - implementiert die AUTOSAR OS Funktionalität
  - ist hochgradig Konfigurierbar
    - mittels graphischem Tool pure::variants
    - Merkmalmodell
  - aspektgewahrer Systementwurf
    - Extension Aspekte
    - Policy Aspekte
    - Binding Aspekte
  - original verfügbar für den TriCore Prozessor
  - muss fürs LavA Projekt angepasst werden



Quelle: Wanja Hofer - Aspect-Oriented Design and Implementation of an AUTOSAR-Like Operating System Kernel

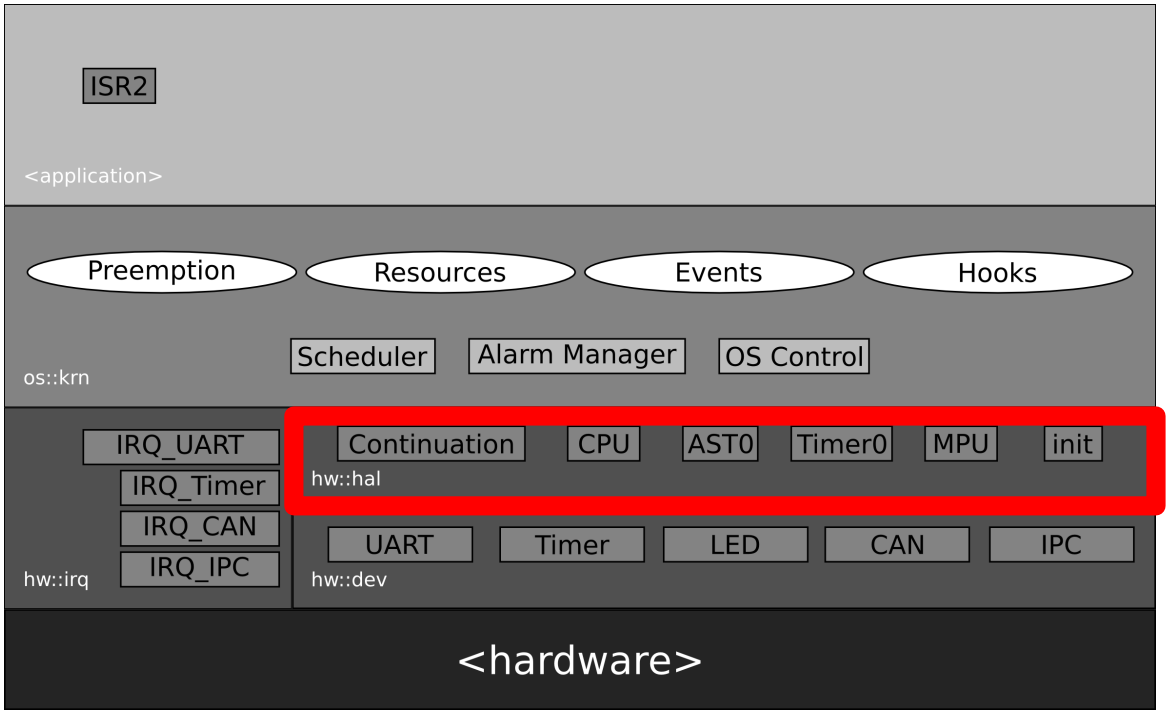
Die Schichten in CiAO

- Highlevel Schicht
- Wird nicht weiter modifiziert
- Ausnahme: TriCore spezifische Codestücke



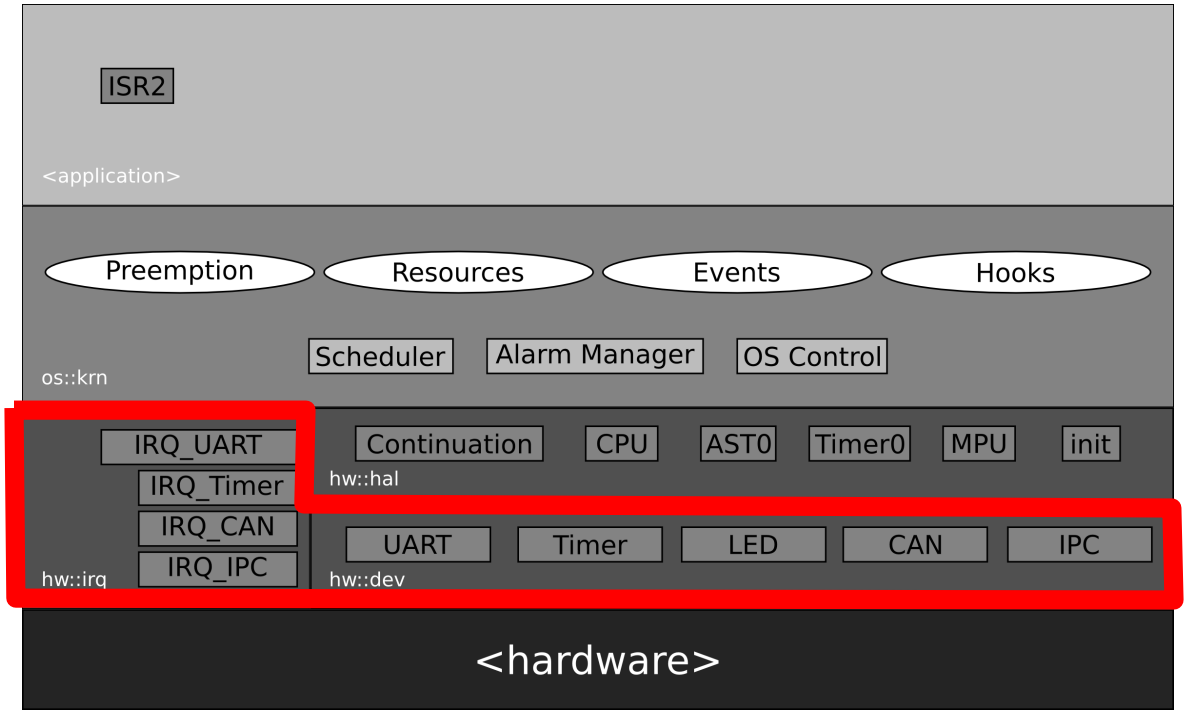
Die Schichten in CiAO

- HAL Schicht
- Muss für den MicroBlaze portiert werden
- Das Interface nach oben muss beibehalten werden



Die Schichten in CiAO

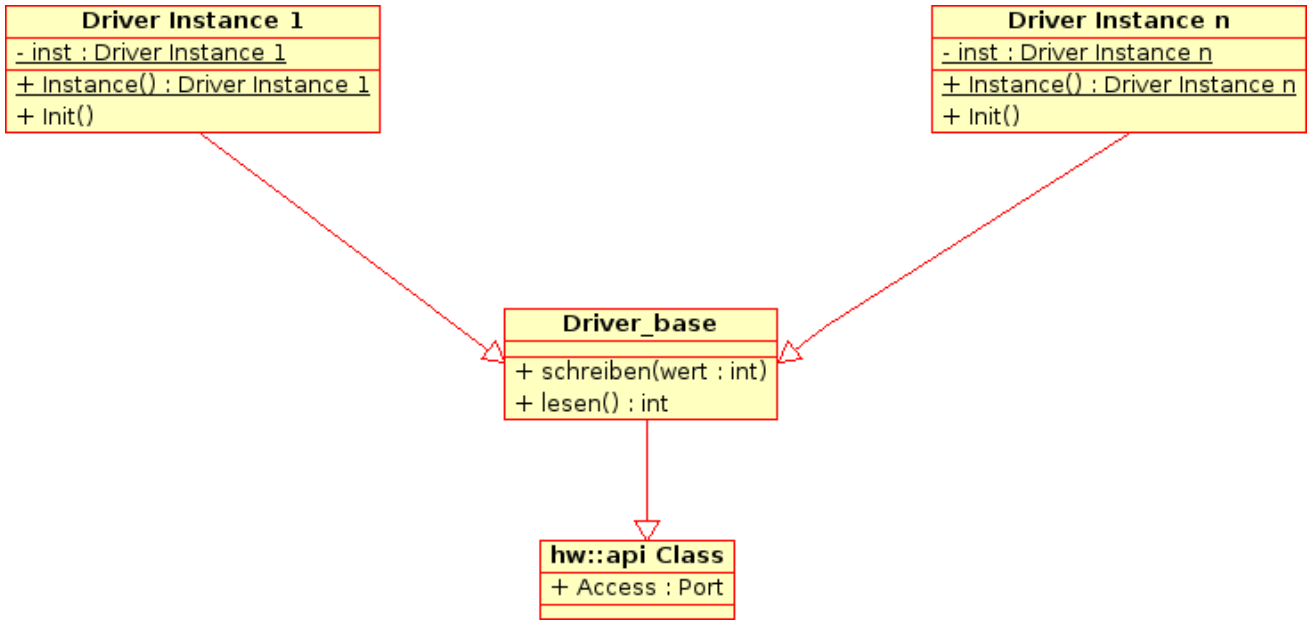
- Treiber und Interruptsystem
- wurde komplett umgebaut
- Grund: Anforderungen von LavA gerecht werden



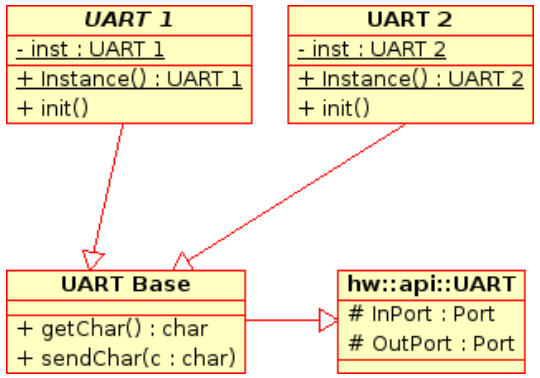
Treiber

- Die Schicht der Treiber musste komplett neu gestaltet werden
- CiAO Design Prinzipien (z.B. Singleton Design Pattern und init() Funktionen) beibehalten
- Verfügbare Geräte:
  - UART
  - Timer
  - IPC
  - CAN
  - Outport
- Aufteilung in Basis- und Instanzklassen

Treiber



- Beispiel: Zwei UART Instanzen



Treiber

- Beispiel: UART
  - Basisklasse:

```
class UART_base : public hw::api::UART<> {
        public:
                unsigned char getc();
                void putc(char c);
};
```

  - Instanzklasse:

```
class UART0 : public UART_base {
        private:
                static UART0 _theUART0;
        public:
                void init() {}
                static UART0& Inst()
                {
                        return _theUART0;
                }
};
```

Interruptsystem

- Musste ebenfalls komplett umgestaltet werden
- zentraler Bestandteil: IRQ Objekte

```
template <typename Device>
struct IRQ
{
    enum { NUM = -1 } ;
    int num() { return NUM; }
};

template <>
struct IRQ<hw::dev::Timer1>
{
    enum { NUM = 3 } ;
    int num() { return NUM; }
};
```

Interruptsystem

- Ein-/Ausschalten von Interruptnummern im IntC.
- Handlertabelle in der CPU

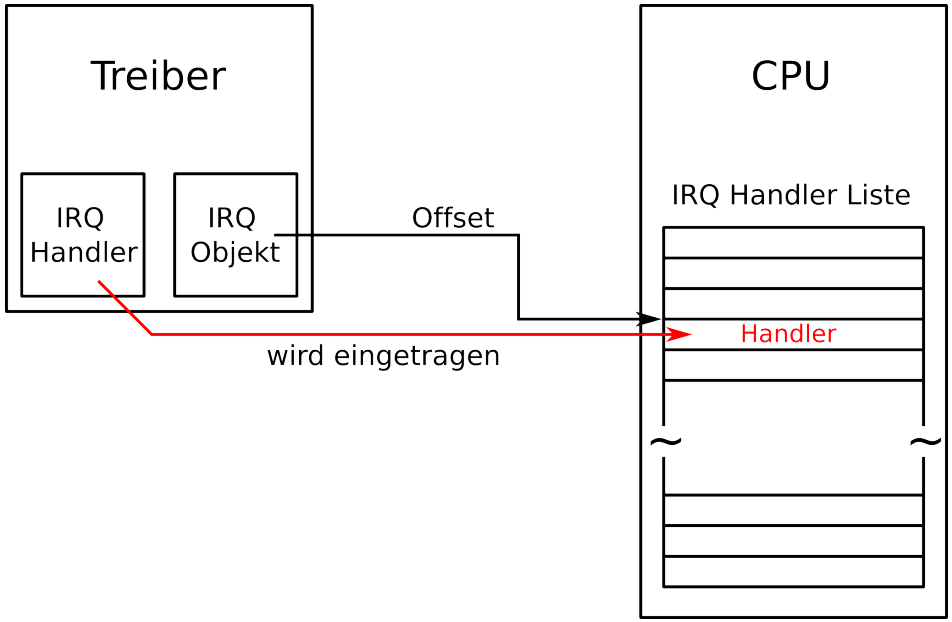
```
class CPU
{
    bool (*handler_table[32])();

    void irq(int num)
    {
        handler_table[num]();
    }

    template <typename Device>
    void registerHandler(IRQ<Device> &irq, bool (*handler)())
    {
        handler_table[irq.num()] = handler;
    }
    [...]
}
```

Interruptsystem

- Erweitern eines Treibers um einen Interrupt
- Drei Schritte:
  - Erweitern des Treibers um die Handler Funktion und das Interrupt Objekt
  - Eintragen der Handlerfunktion in der Klasse CPU
  - Aktivieren des Interrupt im Interrupt-controller



Interruptsystem

- Beispiel: Interrupt für den UART

```
aspect hw_dev_UART0_irqbind
{
    advice "UART0" : slice class
    {
        public:
            static bool Handler() { return true; }
            static bool HWHandler() { return Handler(); }
            IRQ<UART0> _irq;
    };

    advice execution( "% hw::dev::UART0::init()" ) : after()
    {
        CPU::registerHandler(tjp->that()->_irq, UART0::HWHandler);
        intc::Inst().enableIRQ(tjp->that()−>_irq);
    }
};
```

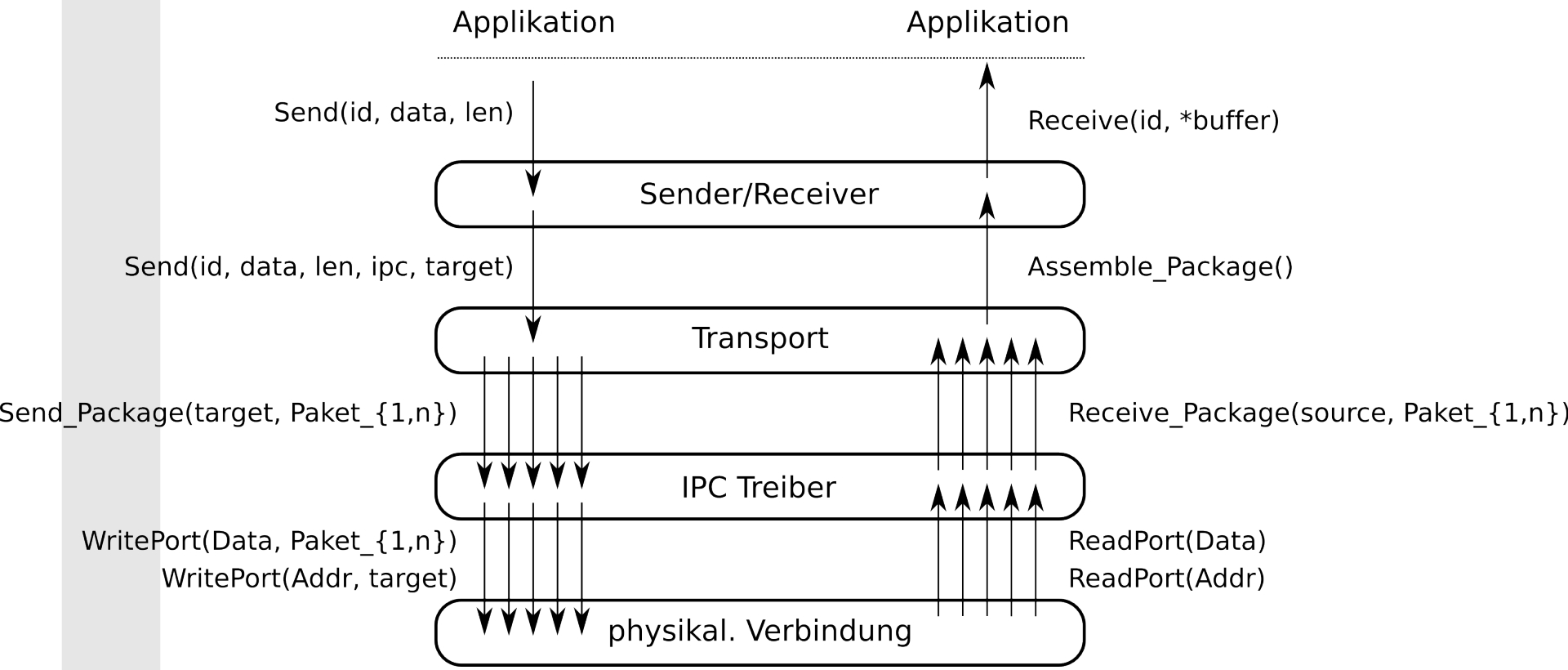
Inhalt

- Einleitung
- Wiederverwendung von BS
- Arbeiten an CiAO
- **Kommunikation**
- Evaluation
- Fazit

Kommunikation

- Momentan ist eine recht dünne Kommunikationsschicht implementiert
- Diese unterstützt:
  - Mehrere Nachrichtentypen pro Kern
  - Identifikation der Nachrichten durch unique ID
  - interne und externe Kommunikation
  - Setzen von Events beim Eintreffen von Nachrichten
  - Puffern von mehreren Nachrichten beim Empfänger
  - Segmentierung
  - Details (wie IPC Controller) nicht in der API vorhanden
- Benutzt wird dafür der IPC

Kommunikation



Kommunikation

- Kommunikation ist ausbaufähig

- Betriebssystem API (unterschiedliche Funktionen benötigen u.U. unterschiedliche APIs)
- Synchronisierung (Barrieresynchronisierung, Token)
- Adressierungsmöglichkeit (möglich: Core ID, Task ID, unique ID, muss eindeutig sein)
- Segmentierung
- Kommunikationstopologien (Bus, Stern, Mesh, mehrere Netzwerke)
- Bussysteme (nicht echtzeitfähig, echtzeitfähig)
- Fehlertoleranz?

Inhalt

- Einleitung
- Wiederverwendung von BS
- Arbeiten an CiAO
- Kommunikation
- **Evaluation**
- Fazit

Evaluation

- Auswirkung des Hinzufügens von Treibern auf die Codegröße
- Testprogramm für einen UART

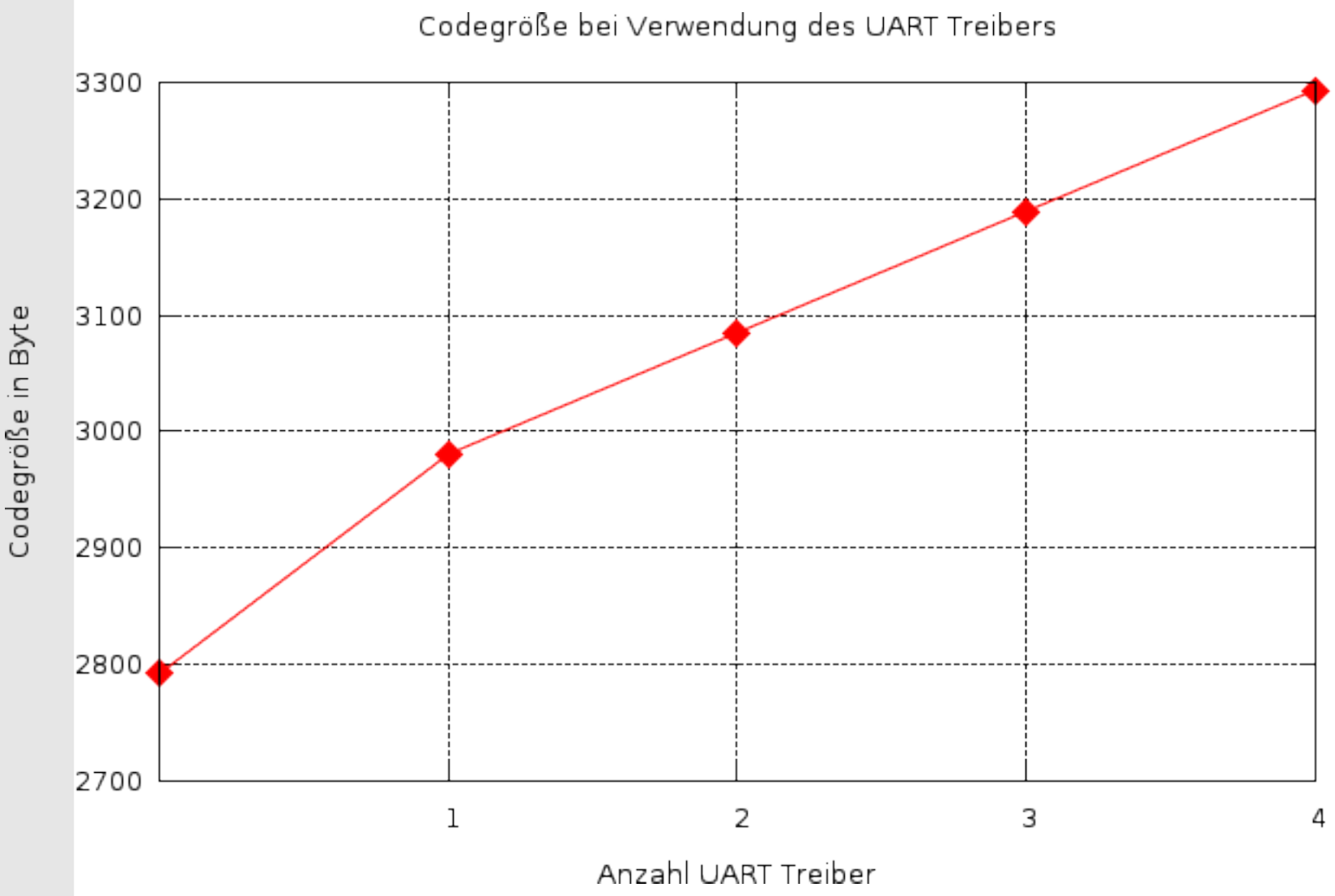
```
void Alpha::functionTaskTask0()
{
    // UART Instanz
    hw::dev::UART0& uart0 = hw::dev::UART0::Inst();
    // einzelnes Zeichen ausgeben
    uart0.putc('x') ;
    // einzelnes Zeichenlesen
    char dummy = uart0.getc();
    // String ausgeben
    uart0.print(" LavA ");
    for(;;);
    AS::TerminateTask();
}
```

Evaluation

- Programm mit einem, zwei, drei und vier UARTs getestet

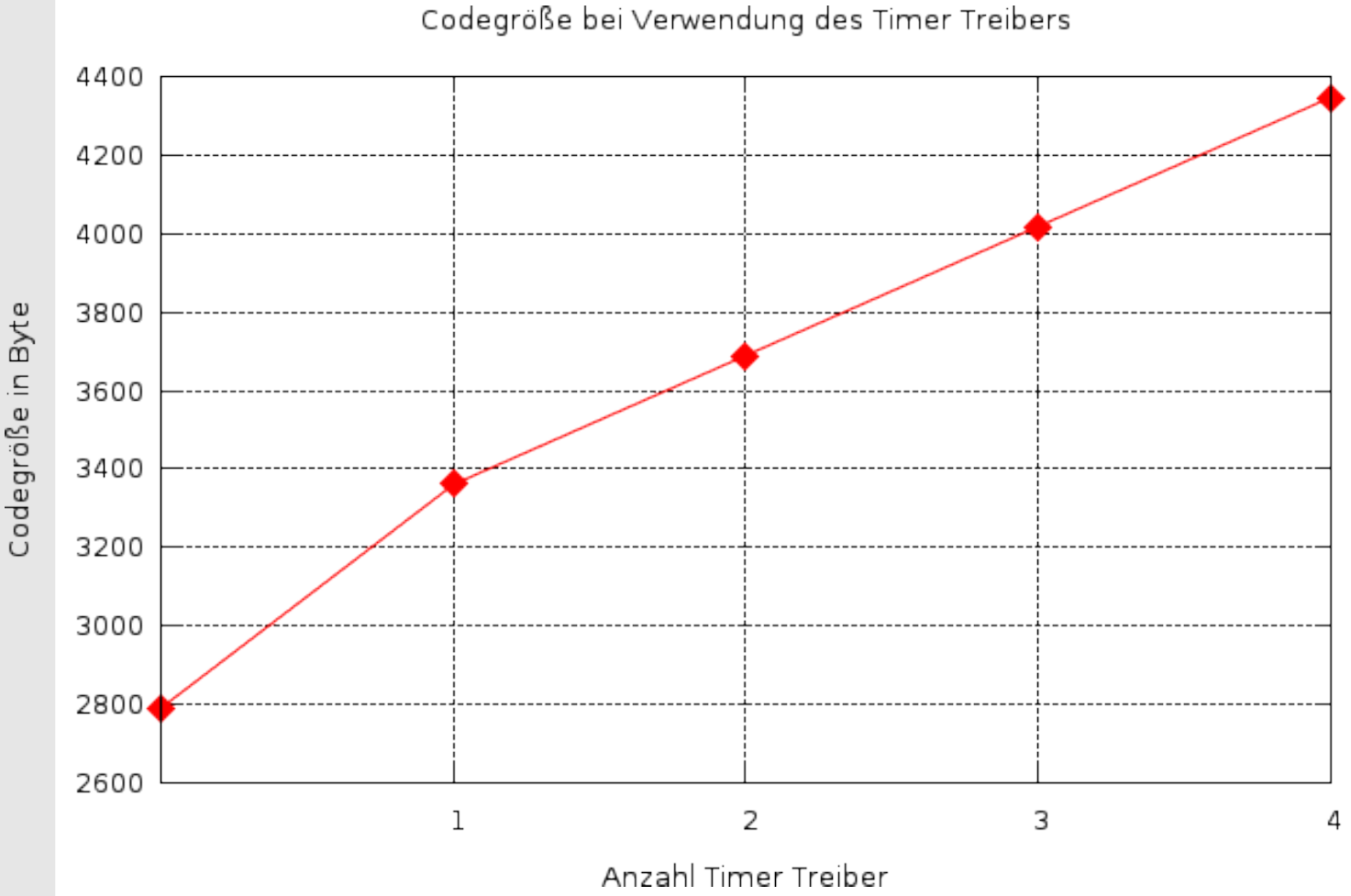
	Größe .text	Δ.text	Größe .bss	Δ.bss	Größe .data	Δ.data
Basissystem	2793		25		2832	
Ein UART	2981	188	25	0	2832	0
Zwei UARTs	3085	104	25	0	2840	8
Drei UARTs	3189	104	25	0	2840	0
Vier UARTs	3293	104	25	0	2848	8

Evaluation


Codegröße bei Verwendung des UART Treibers
Codegröße in Byte
3300
3200
3100
3000
2900
2800
2700
1
2
3
4
Anzahl UART Treiber

Evaluation

- Selbe Vorgehensweise bei Tests für Outport und Timer



Inhalt

- Einleitung
- Wiederverwendung von BS
- Arbeiten an CiAO
- Kommunikation
- Evaluation
- **Fazit**

Fazit

- Entwicklung von Betriebssystemkomponenten für LavA
- Basierend auf CiAO
- Unterstützung von mehreren Geräten pro Kern
- grundlegende Kommunikation

Fragen?

¿