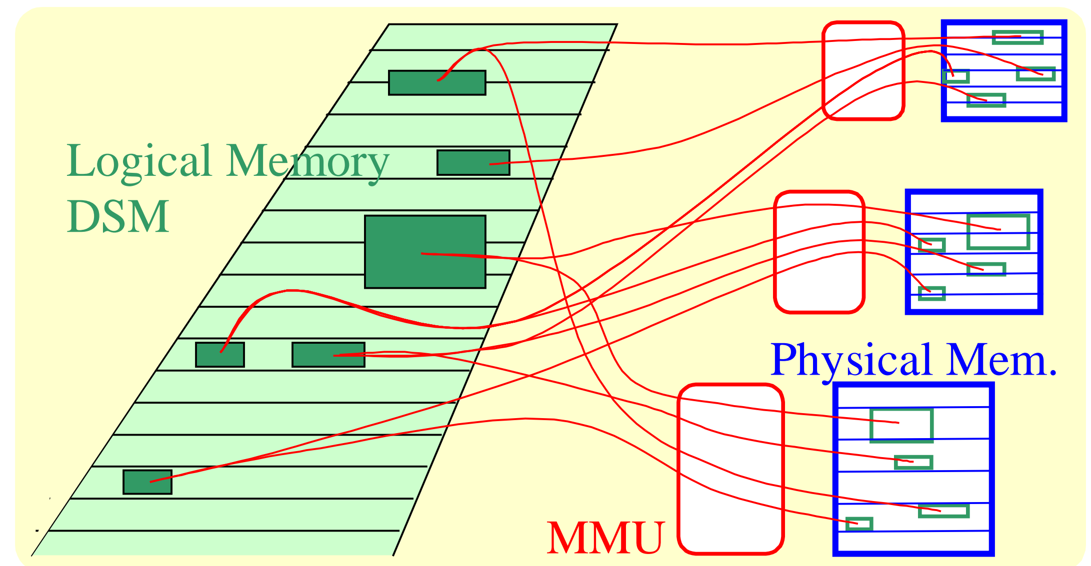
6. Plurix-Treiber

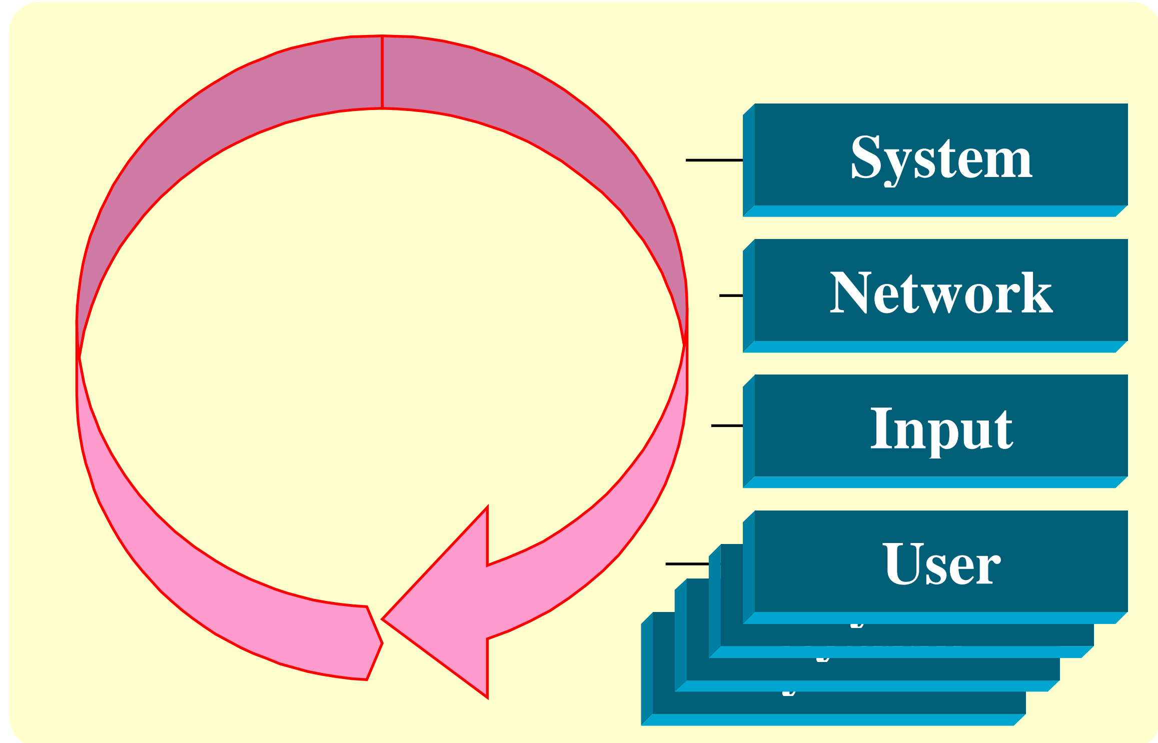
6.1. System Architektur

- DSM=**D**istributed **S**hared **M**emory, VVS=**V**erteilter **V**irtueller **S**peicher.
- Granularität: Viele Objekte pro Seite.
- Seitenorientierter DSM in Plurix:
  - Logischer Speicherzugriff mit einer Byteadresse,
  - Übersetzung in phys. Adresse durch MMU,
  - Falls erforderlich werden 4K übertragen.
- Implizite Kommunikation:
  - Über gemeinsamen Speicher,
  - Kein RMI/RPC/Corba,
  - TCP/FTP extern,
  - Keine Pipes,
  - Kein MPI ...
- Info: www.plurix.de



Transaktionsschleife

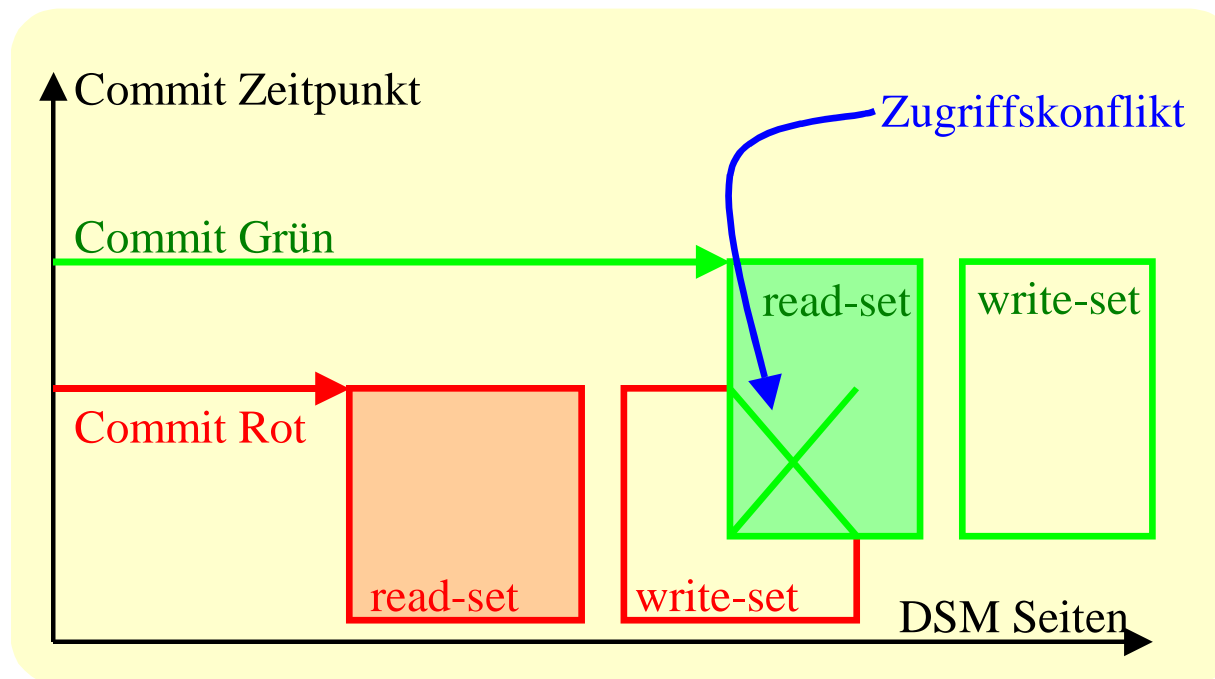
- Kooperatives Multitasking (Oberon-style):
  - **Zentrale Event-Schleife =>**
  - System Transaktion (GC ..),
  - Legacy Netzwerkprotokolle,
  - Tastatur & Mauslistener,
  - Installierbare Transaktionen.



- Kernel Funktionen:
  - Initialisierung,
  - Interrupt dispatching,
  - Memory management,
  - Stations-Objekt als Kontext,
  - Begin- & End-of-Transaction.
- Besonderheiten:
  - Kurze Transaktionen erwünscht,
  - Pro Befehl implizites Begin- & End-Of-Transaction,
  - Automatischer Wiederanlauf nach Zugriffskonflikten im DSM ...

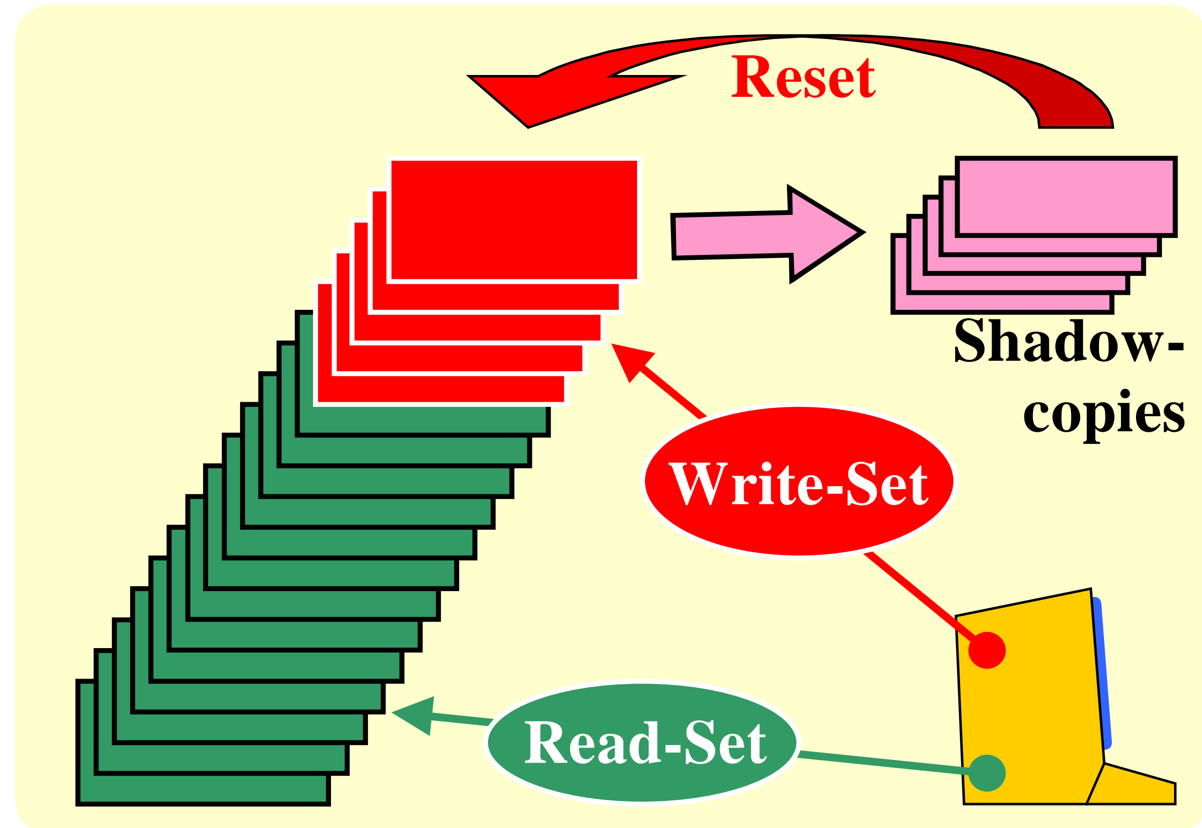
Transaktionale Konsistenz

- Sequentielle Konsistenz zwischen den verschiedenen Transaktionen.
- Die Reihenfolge der Transaktionen ist vorerst unbestimmt.
- Kollidierende Transaktionen werden erneut gestartet.
- Erkennen der Zugriffskonflikte mithilfe der Read- & Write-Sets:



Schattenkopien und Rücksetzung

- Zwischen Transaktionen in verschiedenen Stationen:
  - Im Lauf einer Transaktion wird ein Read- & ein Write Set aufgebaut,
  - Schattenkopie erstellen, bevor eine Seite beschreiben wird,
  - Commit-Request am Ende einer Transaktion,
  - **Im Kollisionsfall Reset ➔**
- Kollisionsauflösung:
  - Serialisierung über Token,
  - Derzeit mit „First wins“,
  - Fairness nicht garantiert.
- Kurze Transaktionen:
  - Niedrige Kollisionsrate,
  - Oberon-artige Befehle,
  - Tastatur und Maus,
  - Klasse compilieren,
  - Ein Video Frame …

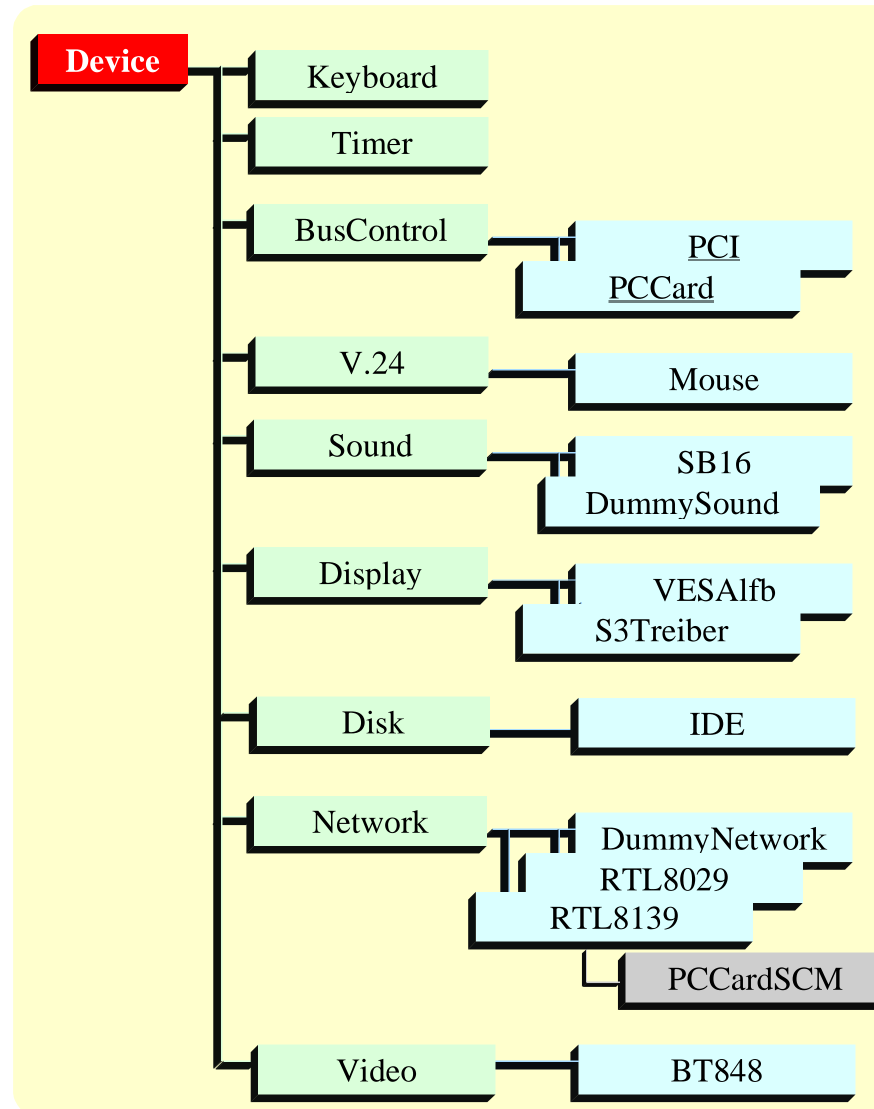


6.2. Treiber-Architektur

- Objekt-orientierte Treiberarchitektur:
  - Definition von Schnittstellen durch abstrakte Klassen.
  - Instanzen für Treiber, logische und physikalische Geräte.
  - Anpassen von Standard-Funktionen durch Überschreiben von Methoden.
- Treiber-Entwicklung in einer typsicheren Sprache (hier Java):
  - Einfachere Fehlersuche,
  - Mehr Disziplin erzwungen,
  - Punktuell Maschinenbefehle notwendig.
- Code von Treiber-Klassen über Verteilten Virtuellen Speicher gemeinsam nutzbar → Zustand von einzelnen Treibern in Instanzen!
- Bus-Treiber:
  - Eintragen gefunden Geräte im cluster-weiten Namensdienst.
  - Automatische Zuordnung von Treibern zu Geräten.
  - Für PCI-Bus implementiert.
- Plug&Play vorgesehen, Power-Management derzeit noch nicht.

Objekthierarchie für Geräte

Ausgehend von der abstrakten Klasse „**Device**“:

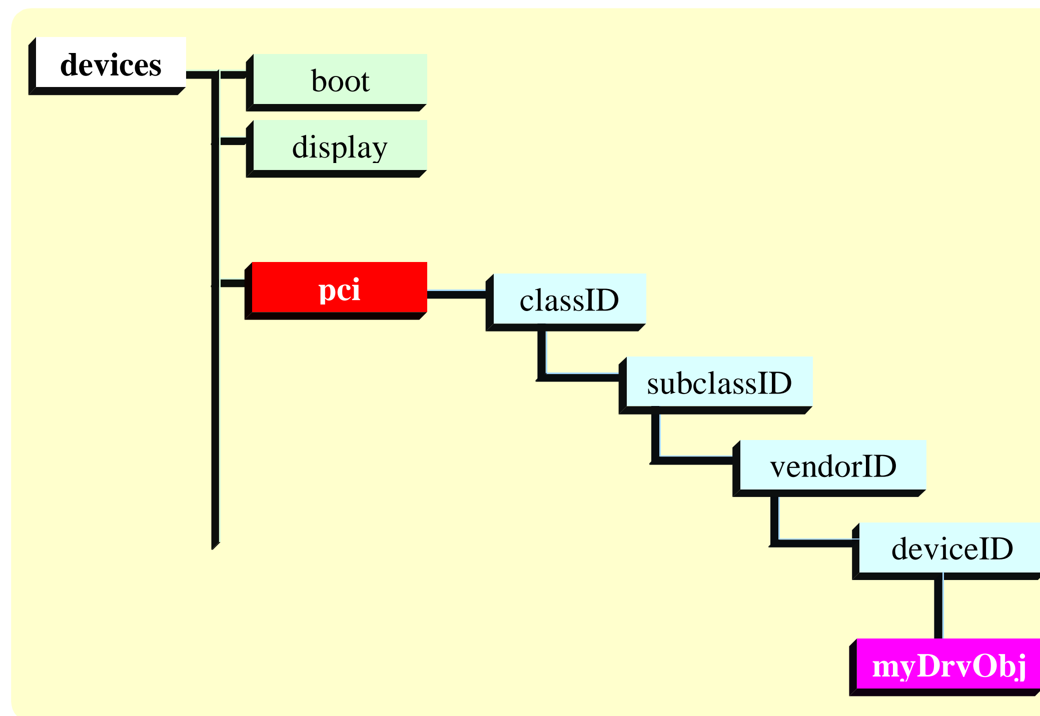


Abstrakte Klasse „Device“

```
package  devices;

import kernel.*;

abstract public attr_sys class Device
{
    final static int STOPPED = 0, STARTED = 1;

    int             state = STOPPED;
    Device          next;                                   // chain shared interrupts

    abstract PObject newDeviceInst(int bus, int dev, int func); // create a device object
    abstract boolean detectDevice();                            // for ISA & friends
    abstract int reinitialize();                            // for restart in case of a fault
    abstract void register();                               // register driver in naming service
    abstract int initialize();
    abstract int unload();                                  // unregister driver
    abstract int start();
    abstract int stop();
    abstract int ISR();

    public void registerDriver(int classID,int subclassID,int vendorID,int deviceID) {…}
}
```

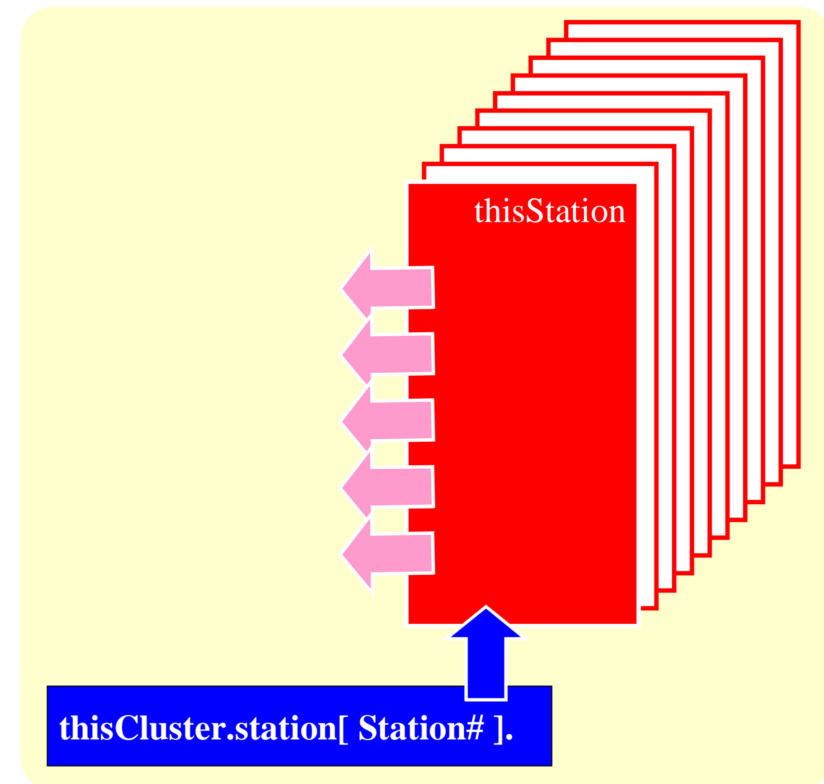
Automatische Treiber-Zuordnung

- Vorerst nur für PCI-Bus.
- Treiber-Klassen befinden sich im Namensdienst im Package */devices*.
- Installationsphase (Aufbau des Heaps):
  - Für jeden Treiber eine **Instanz** angelegt (mithilfe des Reflection-API).
  - Diese **Treiber-Instanz** wird mit *pDrv.register()* registriert.



- Bootphase:
  - PCI-Bus wird nach Geräten durchsucht.
  - Passender Treiber im Namensdienst unter */devices/pci* identifizieren.
  - Mit gefundenem Treiber-Objekt für jedes Gerät eine (private) Instanz erzeugen.
    → *pDrvObj.newDeviceInst* rufen
    → Referenz wird im Stations-Objekt vermerkt

- **Clusterobjekt:**
  - Einmal pro Cluster vorhanden,
  - Enthält unter anderem den Stationsarray.
- **Stationsobjekt:**
  - Liefert den lokalen Kontext,
  - Für private Instanzen,
  - Lokale Treiber,
  - IP-Adresse, ...



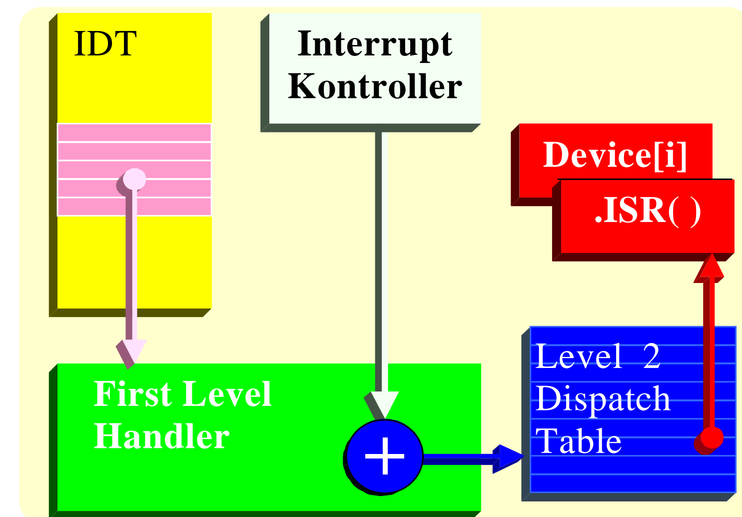
6.3. Interrupt Organisation

- Treiber melden sich an mit:

  *Interrupts.InsertHandler(**this, irq**)*

- First Level Handler:
  - Unterstützung von Shared Interrupts,
  - Programmierer muß sich nicht um EOI kümmern,
  - Lädt den gewünschten Klassendeskriptor & Instanzreferenz.

- Interrupt Steuerung:
  - Interrupt Deskriptor Tabelle (IDT),
  - First Level Interrupt Handler im Kern,
  - IRQ-Nummer aus Status Register des 8359,
  - Treiber Instanz aus Level 2 Dispatch-Tabelle,
  - Interrupt Service Routine (ISR) des Treibers.



6.4. I/O Verarbeitung

- I/O aus Sicht von Anwendungen:
  - Callback: Listener-Konzept von Java (z.B. für Mouse, Tastatur).
  - Pollen: TA in Loop registrieren, die periodisch gerufen wird.

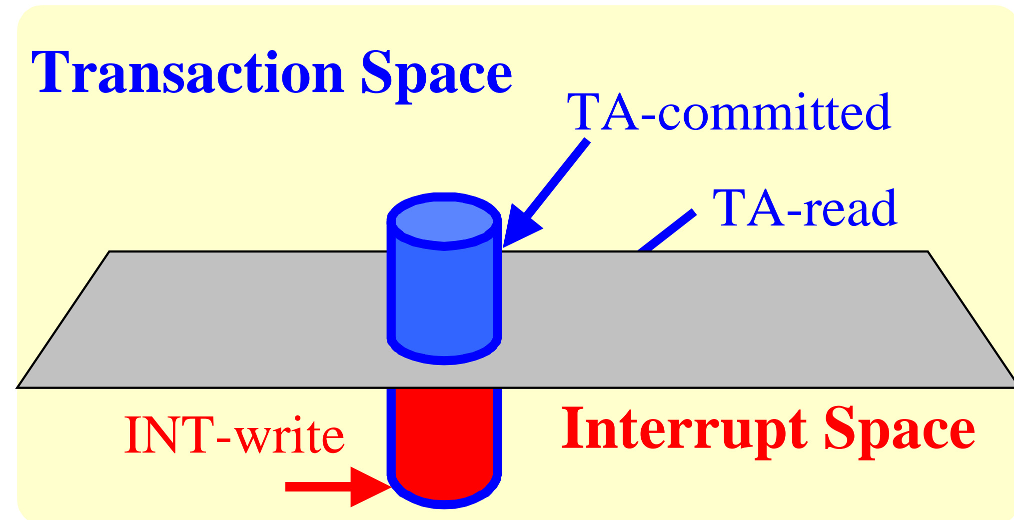
```
public class TextViewer implements MouseListener, KeyListener {
    void mousePressed (MouseEvent mouseEvent) { … }
}

public interface MouseListener {
    void mousePressed (MouseEvent mouseEvent);
    …
}
```

- Kernaufrufe benötigen kein Umkopieren:
  - Keine Unterscheidung zwischen User- und Kernel-Mode.
  - Keine getrennten Adressräume zwischen Anwendungen und Kern.
- Problem: Transaktionen sind rücksetzbar, nicht aber die Hardware:
  - Ausgaben von TAs nur im Commit freigeben (sonst evt. redundant).
  - Eingaben von Hardware dürfen bei einem Abort nicht verloren gehen.

→ **Lösung: SmartBuffers**

Rücksetzbare Puffer: Smart Buffers

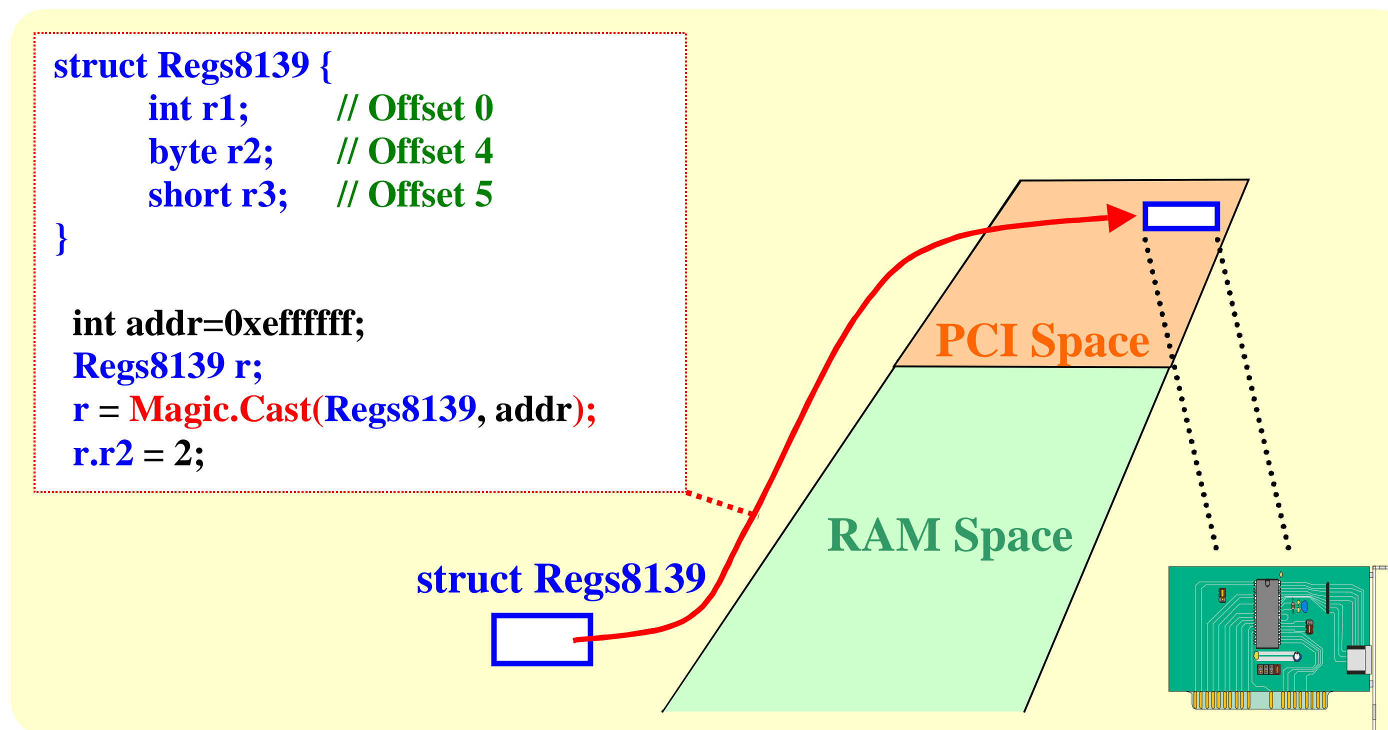
- Für wiederholbare **Eingabe**:
  - **Schreibzeiger** für die Interrupt-Routine ist definitiv.
  - **Commitzeiger** ist wirksam nach einem erfolgreichem Commit.
  - **Lesezeiger** für die Transaktion ist rücksetzbar.



- Smart Buffers liegen zwischen Interrupt- & Transaktionsraum:
  - Die Interrupt-Routine legt Eingabe-Daten in den Smart Buffer,
  - Eingaben in einer erneut gestarteten Transaktion können erneut gelesen werden.
- Die Pufferinhalte sind als Klassen mit Attribut „NonTransactional" angelegt und nehmen damit nicht am Rücksetzungsalgorithmus teil.
- **Ausgabepuffer** können die Ausgaben einer Transaktion verwerfen.

6.5. Systemprogrammierung mit Java

- Spracherweiterungen:
  - Strukturierte Geräteprogrammierung,
  - Einbettung v. Hex-Maschinencode,
  - Direkter Speicher- & Portzugriff,
  - Beliebige Typumwandlungen,
  - Interrupt-Stackframe.



6.6. Treiberbeispiel: UART

- Hier nur Auszüge bzgl. Einbindung in das System.

```
public class UART extends Device {
    final static int  BPSMax=115200, BufferSize=512;

    // a lot of registers should be declared here

    int             bps;
    short           port,irq;
    SmartBuffer     Send,  Receive;
    byte            dataBits, parity, stopBits;

    int initialize() {
            Send     =  new SmartBuffer(BufferSize, false);
            Receive = new SmartBuffer(BufferSize, true);
            return  reinitialize();
    }

    int reinitialize() {
            // config hardware using Magic.Out8();
            return  0;
    }
```

```
    int  start( )                  {  Interrupts.InsertHandler(this, irq); return 0; }
    int stop( )                    {  Interrupts.RemoveHandler(this, irq ); return 0; }
    int ISR()                      {  return 0; }
    void register()                { // nothing to do here }
    int unload()                   { return 0; }
    boolean detectDevice()         { return true; }

    public PObject newDeviceInst(int bus, int dev, int func) {
            return new UART();
    }

    byte ReadByte() {
    while( Receive.ElementsInBuffer( ) ==0 ) ;
            return Receive.GetElement( );
    }

    void  WriteByte(byte b ) {
            Send.AddElement( b );
            DirectWrite( Send.GetElement( ) );
    }

    private void DirectWrite(byte b) { // write byte to port }
    private byte DirectRead()        { // read byte from port }
}
```