

# Softwareentwicklung für eingebettete Systeme

Bachelorstudiengang Technische Informatik  
Hochschule Pforzheim

Vorlesung 5  
Sommersemester 2014

Dipl.-Ing.(FH) Marc Jüttner


# Beispiele...

- Beispiele stammen aus Beej's Guide to Network Programming
  - Datagram client and server
  - Stream server for use with Telnet

# Standard-OS

- Lowlevel-Dienste (Start, E/A, ...)
- Multitasking
- Speicherverwaltung
- Hardwareabstraktion
- Dateisysteme
- Kommunikations-Stacks
  - USB, Netzwerk, ...

# RTOS

- Sehr kurze Antwortzeiten
  - Minimale Latenz für Interrupts und Taskwechsel
- Deterministisches Antwortverhalten
  - Sehr kleiner Jitter
- Präemption
- Optimierte Speicherverwaltung 
  - Einfach, minimale Fragmentierung

# HLOS vs. RTOS

	<b>HLOS: Linux</b>	<b>RTOS: SYS/BIOS</b>
<b>Einsatzgebiet</b>	Allgemein	Sehr spezifisch
<b>Größe</b>	Groß: 5-50MB	Klein: 5-50kB
<b>Antwortzeit</b>	1ms – 0.1ms	100ns - 10ns
<b>Dynamischer Speicher</b>	Üblich	Möglich
<b>Threads</b>	Prozesse, pthreads, Interrupts	HWI, SWI, Tasks, Idle
<b>Scheduler</b>	Zeitscheibenbasiert	Präemption
<b>Kernel</b>	Monolith	Mikrokern
<b>Hardware/Treiber</b>	Integriert, Module	Eingelinkte Bibliotheken
<b>Echtzeitfähigkeit</b>	Sehr begrenzt	Vorhanden

# Beispiel: TI SYS/BIOS

- SYS/BIOS ist Nachfolger des DSP/BIOS
  - DSP-Echtzeitbetriebssystem
- Umbenennung nach Erweiterung auf andere Architekturen
  - Mikrocontroller, ARM, ...
- Mittlerweile „Open Source Software“

# Konfiguration

- Codebasis von SYS/BIOS ist plattformunabhängig
- Aufteilung in Module
- Konkretisierung durch Konfiguration für eine Plattform
  - .cfg-Datei zur Konfiguration
  - Subset von Javascript
  - Grafische oder textuelle Konfiguration

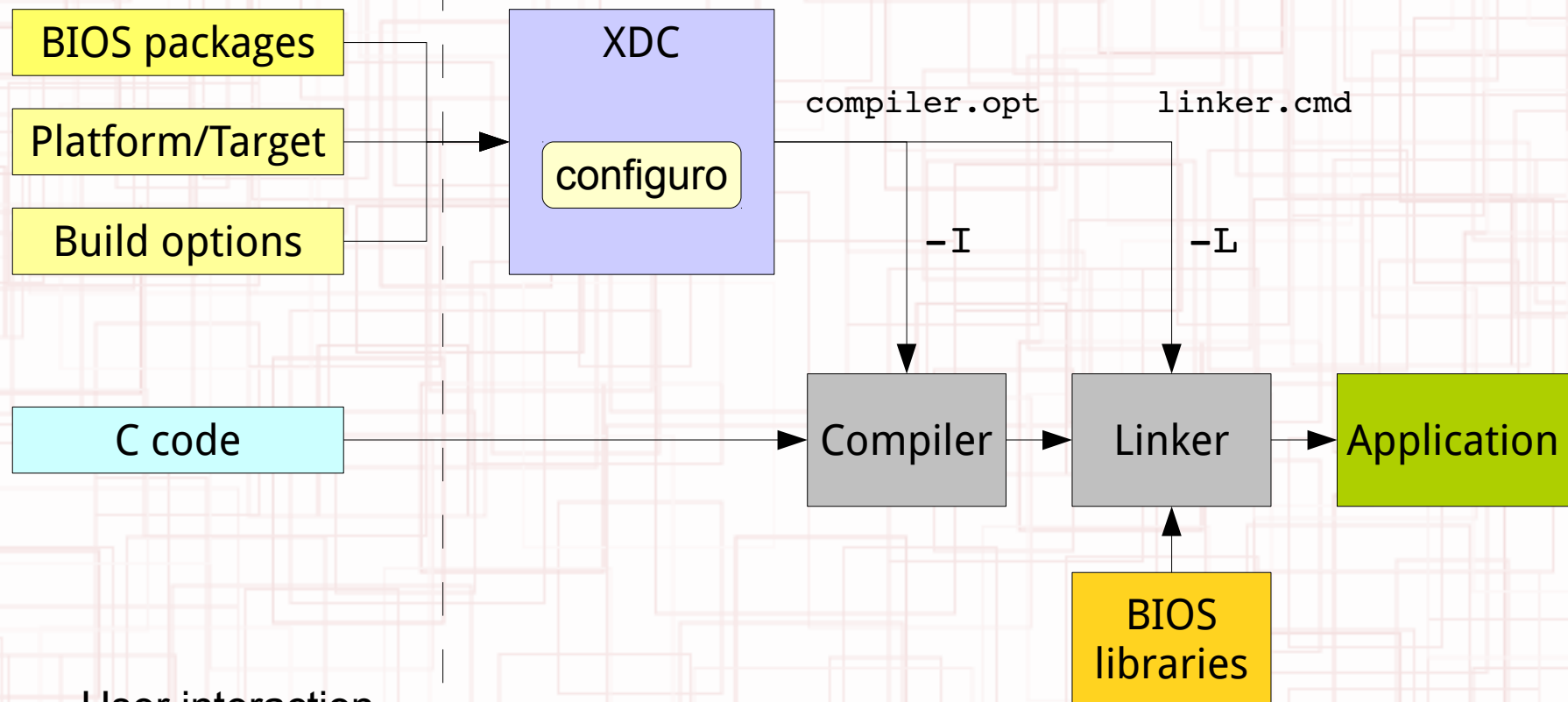
# SYS/BIOS-Bibliothek

- Durch die Konfiguration wird eine **statische Bibliothek** definiert
- Bibliothek wird vom Buildsystem erzeugt
  - Linker verbindet eigenen Code mit der Bibliothek
- Konfigurationswerkzeug erforderlich
  - XDCTools



# Konfiguration

Hidden by tooling



# Kurzbeispiel

```
#include <xdc/std.h>
#include <xdc/runtime/System.h>
#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Task.h>

Void taskFxn(UArg a0, UArg a1) {
    System_printf("enter taskFxn()\n");
    Task_sleep(10);
    System_printf("exit taskFxn()\n");
}

Int main() {
    System_printf("enter main()\n");
    BIOS_start();    /* does not return */
    return(0);
}
```

```
var BIOS = xdc.useModule('ti.sysbios.BIOS');
var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');
var Task = xdc.useModule('ti.sysbios.knl.Task');


...

var task0Params = new Task.Params();
var task0 = Task.create("&taskFxn", task0Params);
```

# Threads

- „Ausführungsfaden“
  - Ausführbarer Maschinencode
  - Befehlszeiger, Stack, Registerwerte
- Mehrere Threads können gleichzeitig ausführbereit sein
  - Scheduler bestimmt Ausführung
  - Kriterien müssen bekannt sein!

# Threadtypen in SYS/BIOS

- SYS/BIOS kennt
  - Hardware Interrupts
  - Software Interrupts
  - Tasks
  - Idle Task 
- Threads haben eine Priorität
  - Implizit durch Threadtyp
  - Explizit durch Programmierung

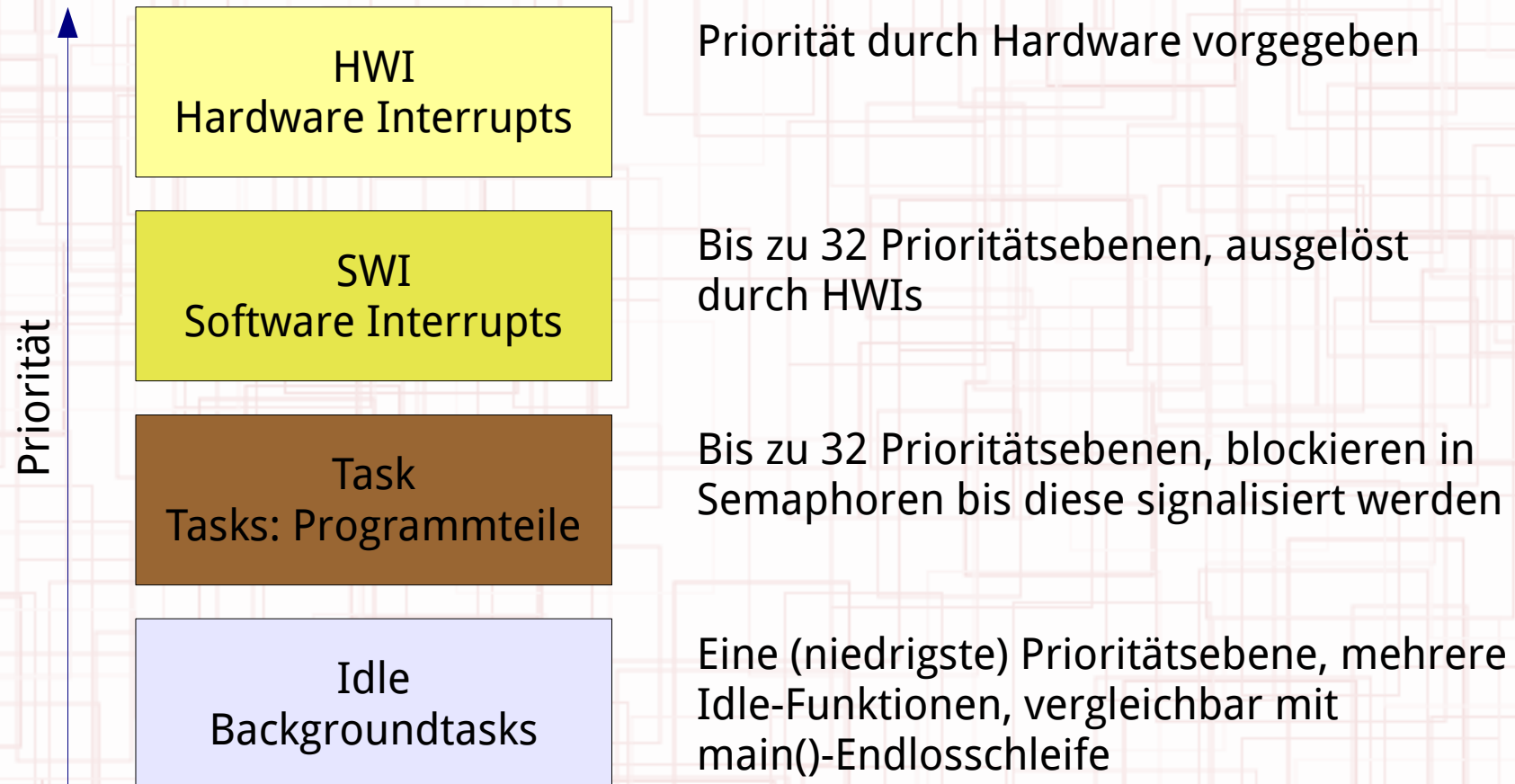
# Threadtypen in SYS/BIOS

- Hardware Interrupt
  - Interrupt Service Routine
  - Ausgelöst durch Interrupts
  - Präemptiv oder Nichtpräemptiv
- Software Interrupts
  - Flexibler Ausführungszeitpunkt
  - *Deferred Procedure Call*
  - Präemptiv

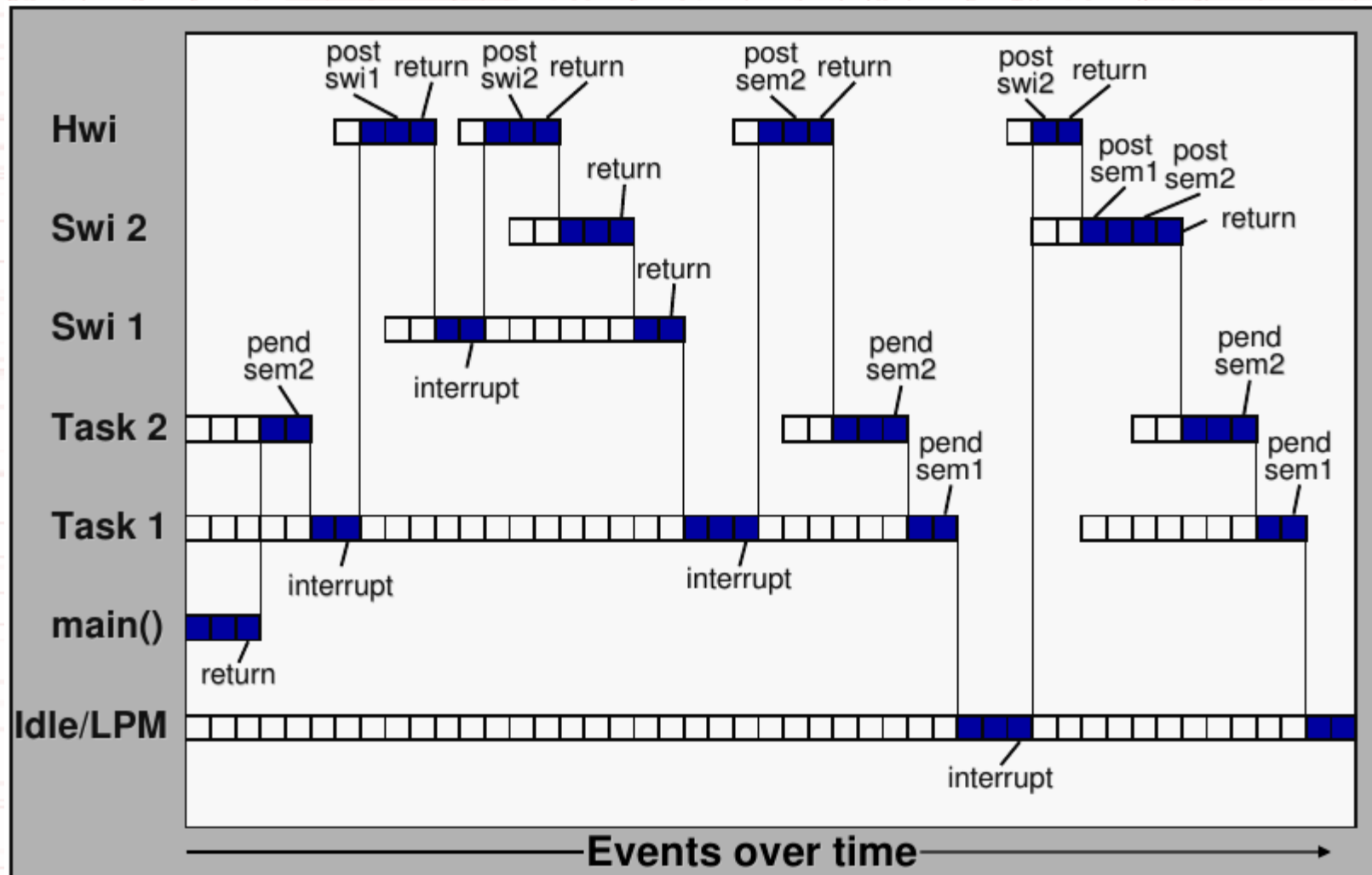
# Threadtypen in SYS/BIOS

- Task
  - „Normaler“ Thread
  - Ausführbar oder blockiert
- Idle
  - Niedrigste Priorität
  - Läuft nur, wenn kein anderer Thread lauffähig ist

# Threadtypen in SYS/BIOS



# Threadpräemption



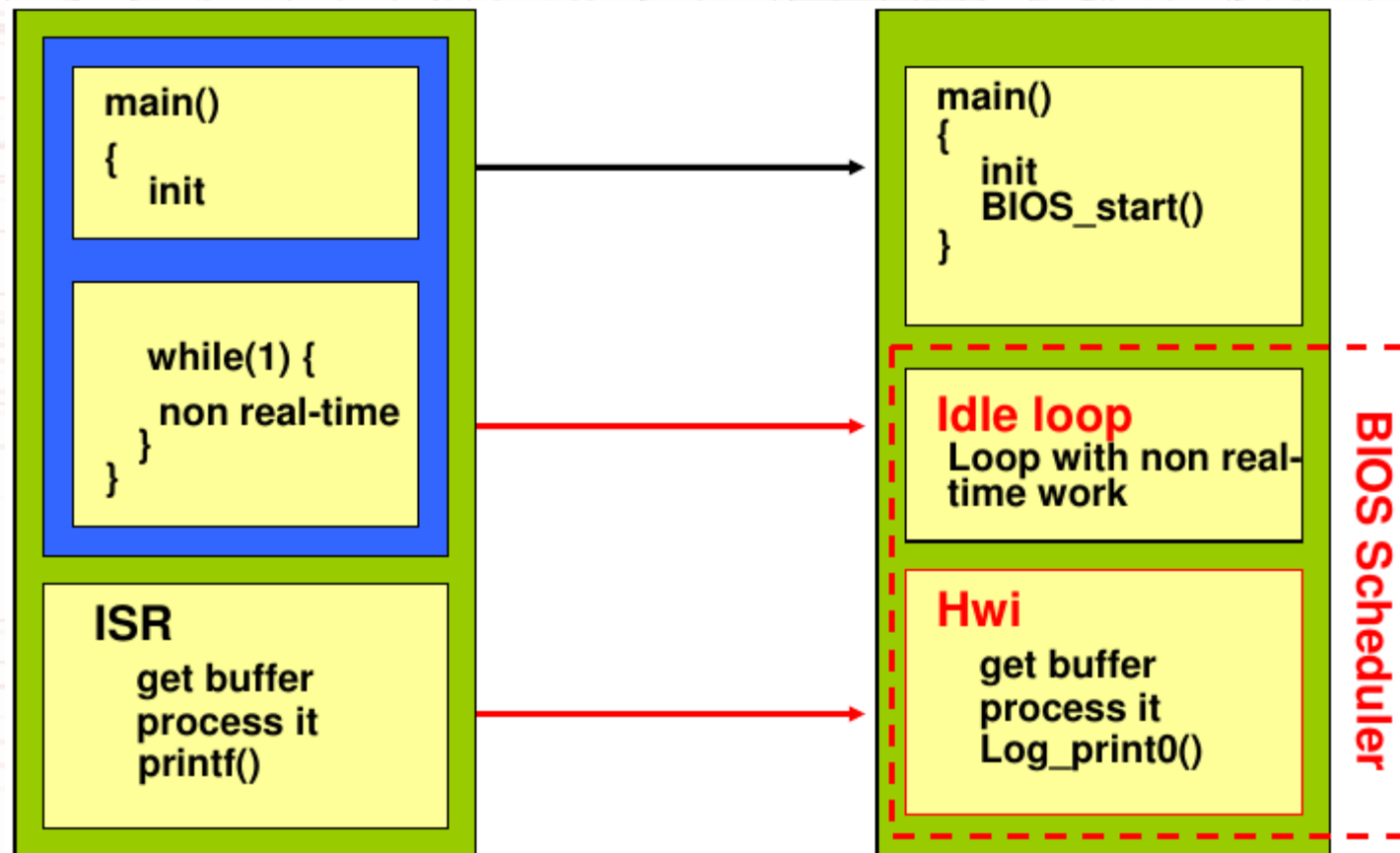
Quelle: TI SYS/BIOS online training



# Scheduling

- Idle-Task entspricht „main loop“ in hardwarenahen Implementierungen
  - Hintergrundfunktionen
  - Unterbrechung durch Interrupts/ISRs
  - Mehrere Idle-Funktionen sind konfigurierbar
- ISRs laufen als HWI threads

# HWIs

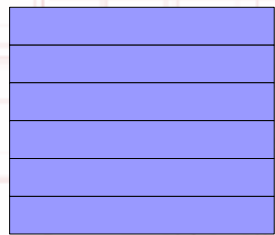


Quelle: TI SYS/BIOS online training

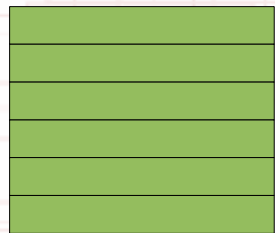
# Interruptbehandlung



Interruptvektoren



Interrupt-Stack



Dispatch-Tabelle

## Interrupt-Dispatcher

- Task-Scheduler deaktivieren
- Auf ISR-Stack umschalten
- Rücksprungadresse speichern
- SWI-Scheduler deaktivieren
- Auto-Nesting:
  - Interrupts maskieren und reaktivieren
- ISR aufrufen
- Interruptmasken wiederherstellen
- SWI-Scheduler reaktivieren
- Auf Task-Stack umschalten
- Task-Scheduler reaktivieren

# Besonderheiten

- Schnelle Reaktion auf Interrupts
- Minimales *context switching*
- Priorität an Hardware-Interrupt gebunden
  - Nesting möglich
- Kann SWIs triggern
  - Swi\_post()
- Id: Interruptnummer, systemabhängig

# Konfigurationsbeispiel

```
Void hwiFunc(UArg arg)
{
    System_printf("Entering myTimerHwi\n");
}
...

Hwi_Handle hwi0;
Hwi_Params hwiParams;
Error_Block eb;

Error_init(&eb);
Hwi_Params_init(&hwiParams);
hwiParams.arg = 5;
hwi0 = Hwi_create(id, hwiFunc, &hwiParams, &eb);
if (hwi0 == NULL) {
    System_abort("Hwi create failed");
}
...
```



```
var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');
var hwiParams = new Hwi.Params;
hwiParams.arg = 5;
Program.global.hwi0 = Hwi.create(id, '&hwiFunc', hwiParams);
```

# SWIs

- Nicht-zeitkritischer Teil einer ISR
- Präemptiv, Priorität von 0 – 15 (31)
  - Innerhalb einer Ebene werden SWIs entsprechend ihrer Aktivierung ausgeführt
- Einfaches Stackmodell
  - Ein Stack je Priorität
  - Mehr Prioritäten → mehr Stackspeicher!

# Besonderheiten

- Latenz im Antwortverhalten
  - **Schedulereinfluss**
- Kontextwechsel werden durchgeführt
- Kann andere SWIs triggern

# Konfigurationsbeispiel

```
Void swiFunc(UArg arg0, UArg arg1)
{
    System_printf("Entering mySwi\n");
}
...

Swi_Handle swi0;
Swi_Params swiParams;
Error_Block eb;

Error_init(&eb);
Swi_Params_init(&swiParams);

swi0 = Swi_create(swiFunc, &swiParams, &eb);
if (swi0 == NULL) {
    System_abort("Swi create failed");
}
```



...

```
var Swi = xdc.useModule('ti.sysbios.knl.Swi');
var swiParams = new Swi.Params();
program.global.swi0 = Swi.create(swiParams);
```

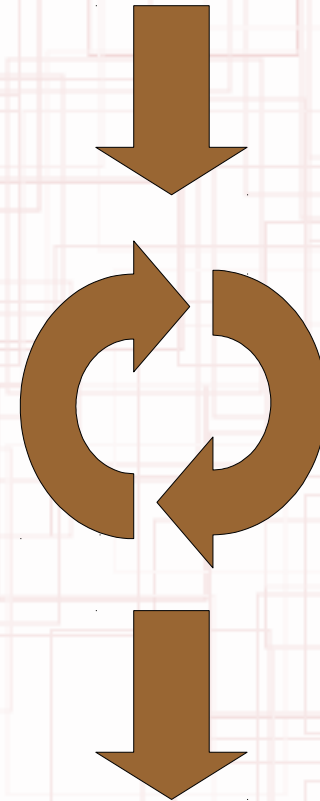


# Tasks

- „Klassischer“ Thread
  - Spezielle Programmfunktion
- Einfache Struktur
  - Initialisierung
  - Warten auf Ressourcen, Algorithmen
  - Deinitialisierung

# Tasks

```
...  
Void taskFxn(args) {  
    /* Initialization */  
    ...  
    while (condition) {  
        ...  
        Semaphore_pend();  
        ...  
    }  
    /* Deinitialization */  
    ...  
}  
...
```



# Konfigurationsbeispiel

```
Void hiPriTask(UArg arg0, UArg arg1) {  
    System_printf("Entering myTask\n");  
}  
...
```

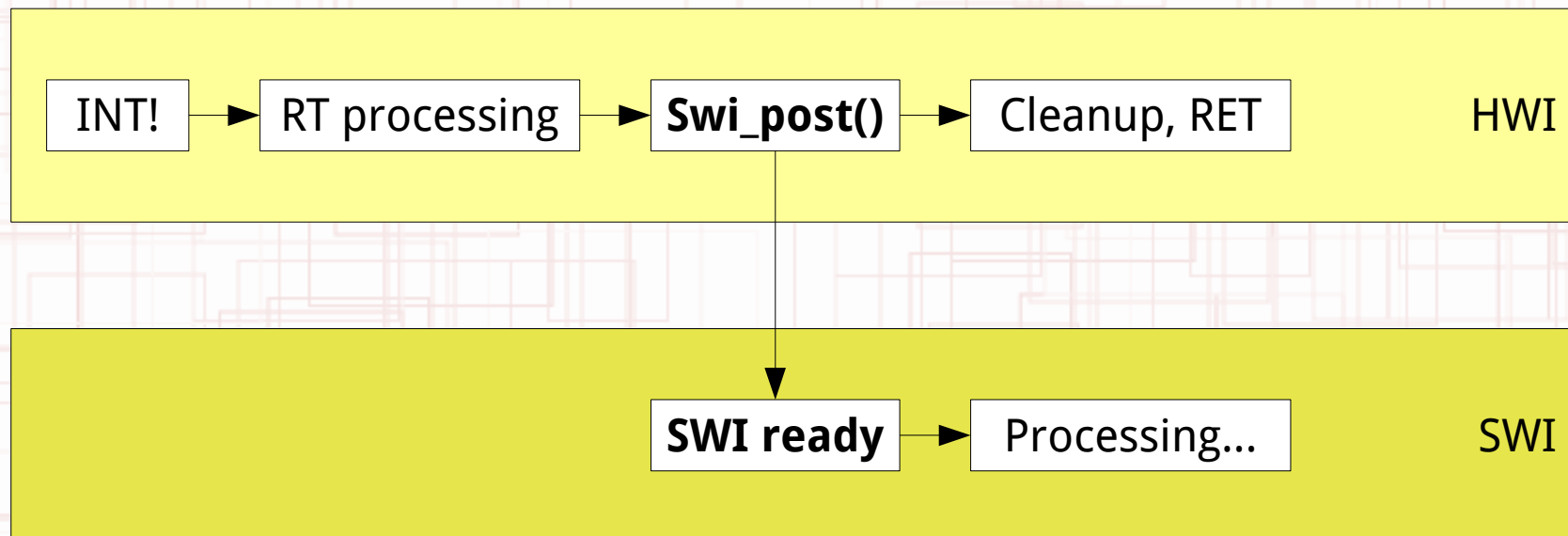
```
Task_Params taskParams;  
Task_Handle task0;  
Error_Block eb;
```

```
Error_init(&eb);
```

```
/* Create 1 task with priority 15 */  
Task_Params_init(&taskParams);  
taskParams.stackSize = 512;  
taskParams.priority = 15;  
task0 = Task_create((Task_FuncPtr)hiPriTask, &taskParams, &eb);  
if (task0 == NULL) {  
    System_abort("Task create failed");  
}  
...
```

```
var Task = xdc.useModule('ti.sysbios.knl.Task');  
...  
var task0Params = new Task.Params();  
var task0 = Task.create("&taskFxn", task0Params);
```

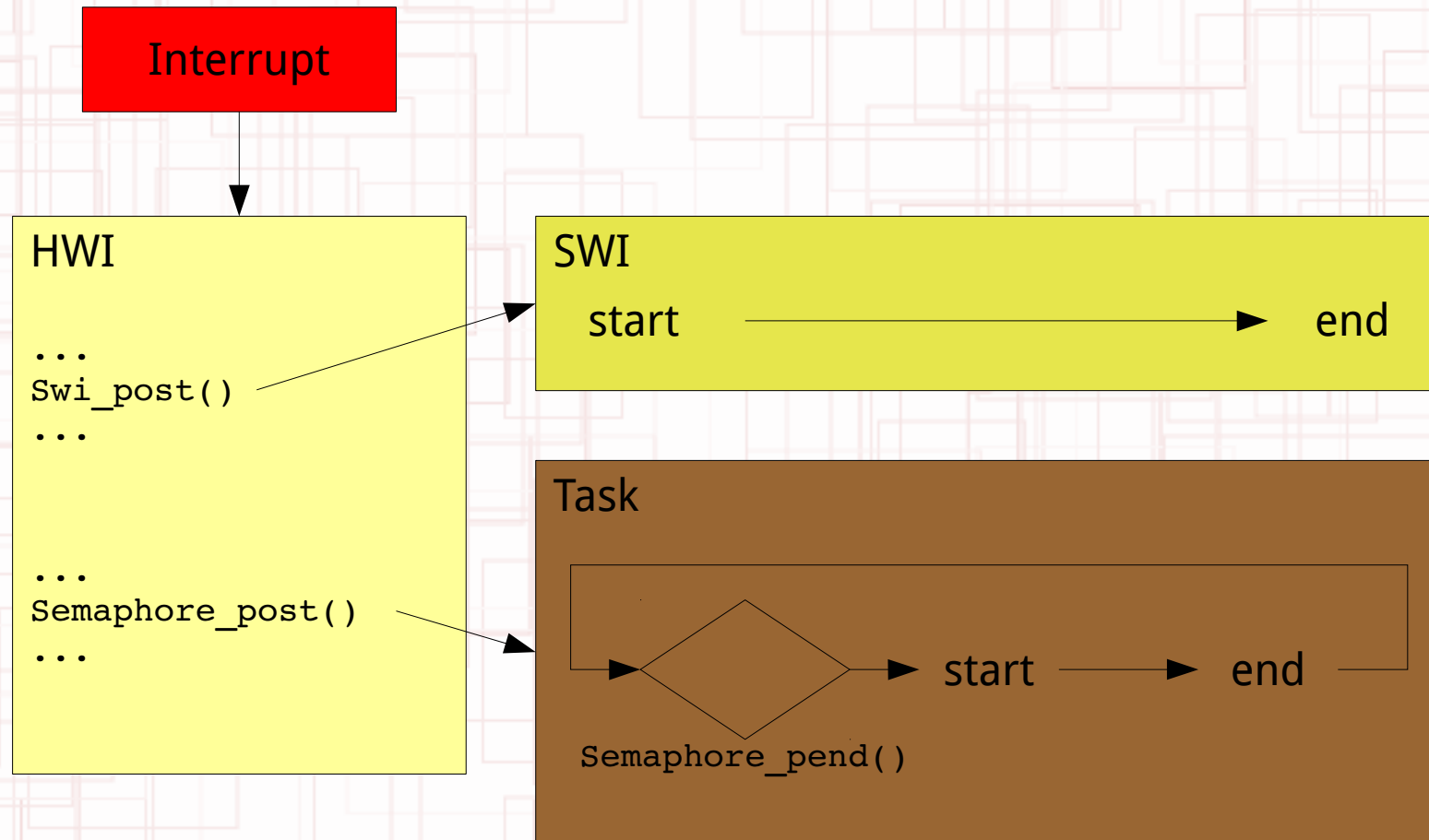
# HWI/SWI



# SWI vs. Tasks

- „Ready“ nach Swi\_post()
  - Reinitialisierung vor jedem Durchlauf
  - Kann nicht blockieren
  - HWI-Stack
- „Ready“ nach Erzeugung
  - Initialisierung ist persistent
  - Kann ggfs. Blockieren
  - Eigener Stack

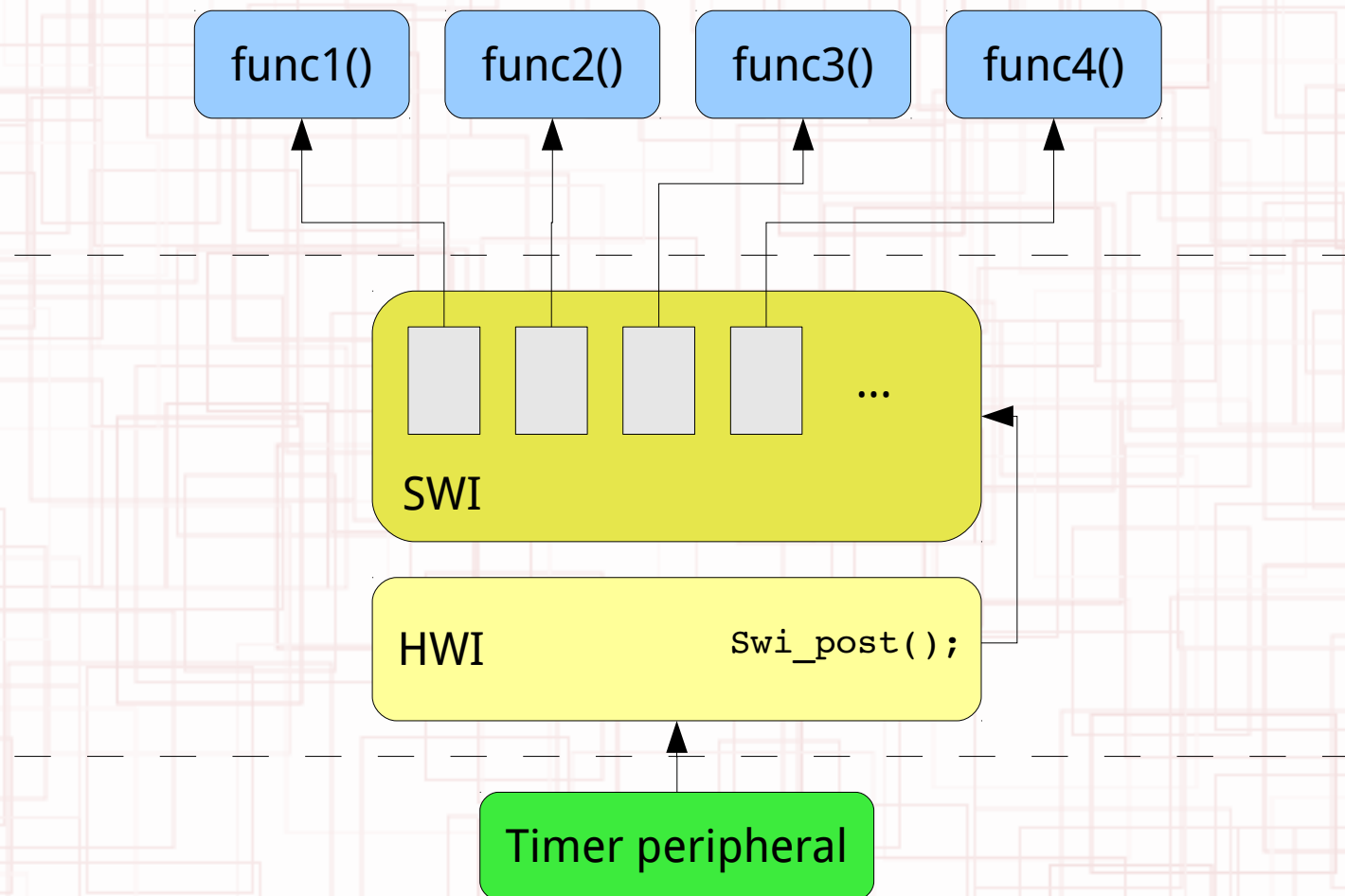
# Signalisierung



# Beispiel

- Clockmodul in SYS/BIOS
  - Aufruf einer Funktion nach einem konfigurierbaren Timeout
- HWI, SWI und Task
- Keine Laufzeitkompensation
  - Handhabung mit SWI verletzt Timerbehandlung nicht
  -

# Aufbau





# Laufzeitkompensation?

