Advanced Embedded Systems Zusammenfassung

Joel von Rotz & Andreas Ming 2023-04-25

Quelldateien

Inhaltsverzeichnis

1 Einführung **4**

2 Systeme **4**
2.1 Transformierende Systeme . . . 4
2.2 Reaktive Systeme . . . 4
2.3 Interaktive Systeme . . . 4
2.4 Kombiniertes System . . . 5

3 Architektur **5**
3.1 Systemaufbau . . . 5
3.2 Systemwahl . . . 5
3.3 Rechnerarchitektur: von Neumann . . . 5
3.4 Rechnerarchitektur: Hardvard . . . 5

4 Entwicklung **5**
4.1 Prozess . . . 5
4.1.1 Produktzyklus . . . 5
4.1.2 Wasserfall Modell . . . 6
4.1.3 V-Modell . . . 7
4.1.4 Agile Modell . . . 8
4.2 Werkzeuge . . . 8

5 Firmware **8**
5.1 Architektur . . . 8
5.1.1 Super Loop . . . 8
5.1.2 Event Loop . . . 10
5.1.3 Embedded OS . . . 11
5.2 Modularisierung . . . 12

6 Module **12**
6.1 Anforderungen . . . 12

7 Echtzeit **12**
7.1 Harte und Weiche Echtzeit . . . 12

8 FreeRTOS **13**
8.1 Kernel API . . . 13
8.1.1 Scheduler starten . . . 13
8.1.2 Scheduler beenden . . . 13
8.1.3 Kernel/Scheduler anhalten . . . 13
8.1.4 Kernel/Scheduler fortsetzen . . . 13
8.1.5 Kontext Switch forcieren . . . 13
8.2 Queues . . . 13

9 Kernel **13**
9.1 FreeRTOS Interrupts . . . 13
9.2 Hardware Stack . . . 14
9.3 Critical Sections . . . 14
9.4 `PRIMASK` & `BASEPRI` . . . 14

10 Synchronisation **15**

11 Nachrichten **15**
11.1 Queues . . . 15
11.2 Timer . . . 15

12 Parallelität **15**
12.1 Reentrancy . . . 15
12.1.1 Dis-/Enable Interrupts . . . 15
12.1.2 Critical Sections . . . 15

13 Benutzer **16**

14 Grafik **16**

15 MCUXpresso **16**
15.1 Semihosting . . . 16
15.1.1 Unterschied *none*, *nohost* and *semihost* . . . 17
15.2 DWT . . . 17

16 Fragen **17**
16.1 SW01 Einführung . . . 17
16.1.1 Administratives . . . 17
16.1.2 Software & Tools . . . 18
16.2 SW02 Architektur . . . 19
16.2.1 System . . . 19
16.2.2 Rechner . . . 20
16.2.3 Cortex . . . 21
16.3 SW03 Entwicklung . . . 21
16.3.1 Prozess . . . 21
16.3.2 Werkzeuge . . . 22
16.4 SW04 Firmware . . . 23
16.4.1 Architektur . . . 23
16.4.2 Module . . . 24
16.4.3 Bibliotheken . . . 24
16.5 SW05 RTOS . . . 26
16.5.1 Echtzeit . . . 26
16.5.2 FreeRTOS . . . 26
16.5.3 Archtiektur . . . 26
16.5.4 Kernel API . . . 26
16.5.5 Tasks . . . 26
16.6 SW06 Kernel . . . 26
16.6.1 Interrupts . . . 26
16.6.2 Visualisierung . . . 26
16.7 SW07 Synchronisation . . . 26
16.7.1 Synchronisierung . . . 26
16.7.2 FreeRTOS & Interrupts . . . 26
16.8 SW08 Nachrichten . . . 26
16.8.1 Queues . . . 26
16.8.2 Timer . . . 26
16.9 SW09 Parallelität . . . 26

16.9.1 Reentrancy 26
16.9.2 Sema 26
16.10SW10 Benutzer 26
16.10.1 Benutzerschnittstellen 26
16.11SW11 Grafik 26
16.11.1 Graphical User Interface 26

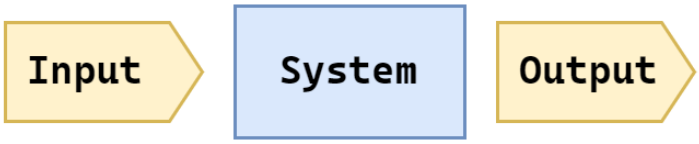
1. Einführung

2. Systeme

2.1 Transformierende Systeme

Verarbeitet ein Eingabesignal (*Input*) und gibt ein Ausgabesignal (*Output*) aus. Wichtige Charakteristiken sind **Verarbeitungsqualität** (effiziente Datenverarbeitung), **Durchsatz** (kleine Latenz zwischen In- und Output), **optimierte Systemlast** (ein System, welches für die Aufgabe ausgelegt ist ; nicht überdimensioniert) und **optimierter Speicherverbrauch** (wenig Speicher bedeutet meistens auch langsames System, daher muss Speicher effizient gebraucht werden).

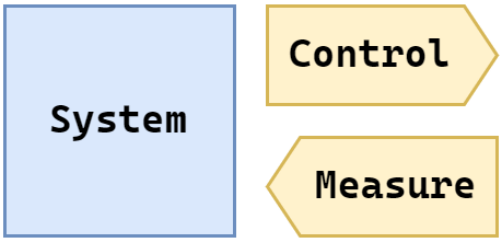
Beispiele: Verschlüsselung, Router, Noise Canceling, MP3/MPEG En-/Decoder



2.2 Reaktive Systeme

Ein Reaktives System reagiert auf gemessene Werte, also von externen Events. Das System muss eine **kurze Reaktionszeit** garantieren, da meistens solche Systeme für Notfallsituationen verwendet werden. Ebenfalls werden diese für **Regelkreise** verwendet und sind typisch **Echtzeitsysteme**.

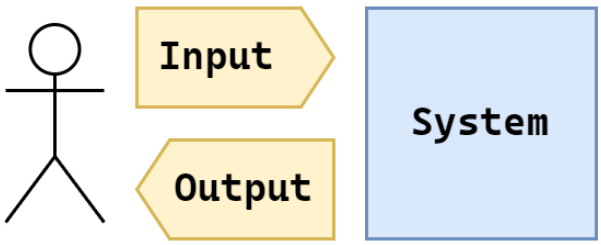
Beispiele: Airbag, Roll-Over Detection, ABS, Brake Assistance, Engine Control, Motorsteuerung



2.3 Interaktive Systeme

Interaktive Systeme werden von Benutzer interagiert. Sie haben eine **hohe Systemlast**, da zum Beispiel die Interaktion einer Benutzeroberfläche ausgewertet werden muss. Damit ein Benutzer mit dem System interagiert, muss es ein **optimiertes HMI** (Human-Machine-Interface) sein und eine **'kurze' Antwortzeit**.

Beispiele: Ticket-Automat, Taschenrechner, Smart-Phone, Fernsehbedienung



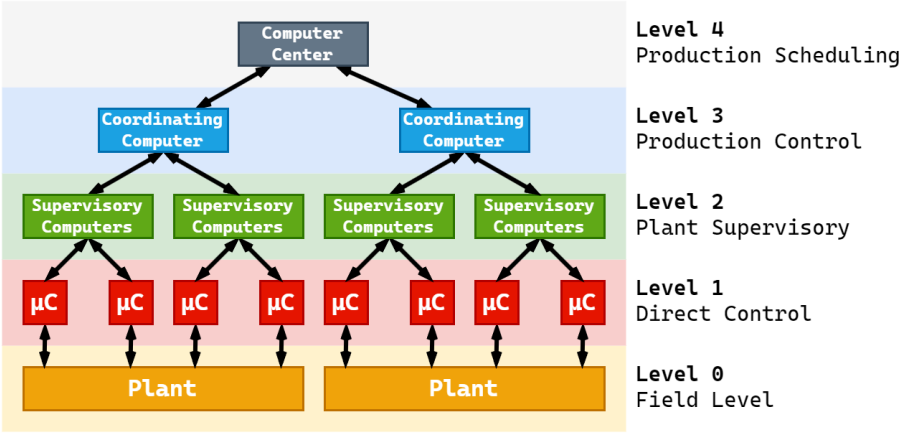
2.4 Kombiniertes System

Ein kombiniertes System ist, wär hätte es gedacht, eine Kombination von den erwähnten Systemen und anderen. Zum Beispiel kann ein Smartphone ein kombiniertes System ist, da es aus einem interaktiven Teilsystem für Homescreen- & App-Interaktionen, transformierendes Teilsystem für Audio-Decodierung für Musikhören und weiteren kleineren Teilsystemen.

3. Architektur

3.1 Systemaufbau

Der Systemaufbau beschreibt die stufenweise Interaktion mit Systemen. Solche Systeme können in Produktionslinien aufgefunden werden, wo die einzelnen Produktionstationen Teil eines ganzen Produktionssystems sind.



0. **Plant** – Produktionsperipherien wie Roboterarm oder Pressanlage
1. μC – Ansteuerung der Peripherien
2. **Supervisory Computers** – Computer Teil der Station (Anwendung der Produktionssoftware)
3. **Coordinating Computers** –
4. **Computer Center** –

3.2 Systemwahl

3.3 Rechnerarchitektur: von Neumann

3.4 Rechnerarchitektur: Hardvard

4. Entwicklung

4.1 Prozess

4.1.1 Produktzyklus

Der Produktzyklus zeigt den idealen Verlauf einer Produktentstehung. Ideal, weil es in der Realität anders ist. Es werden gewisse Schritte wiederholt, da zum Beispiel die Idee zu früh ist, also die Konsumenten noch nicht bereit sind umzusteigen oder das Produkt zu kaufen.

- **Idea** – Wie kann Problem (z.B. verlorene Gegenstände wieder finden) gelöst werden? → Ideen sammeln
- **Concept** – Wie sollte das Produkt ungefähr aussehen und wie könnte es man bedienen? → Fokus auf Form und vereinzelte Teilfunktionen
- **Analysis** – Was für bestehene Lösungen gibts es und wie könnte man es mit anderen Systemen kombinieren?
- **Design** – Wie sollte das Produkt funktionieren? Wie sieht das Produkt aus? → Genauere Beschreibungen

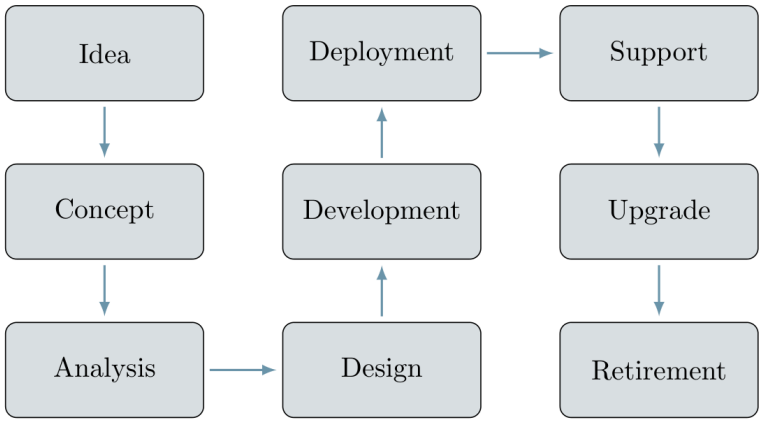


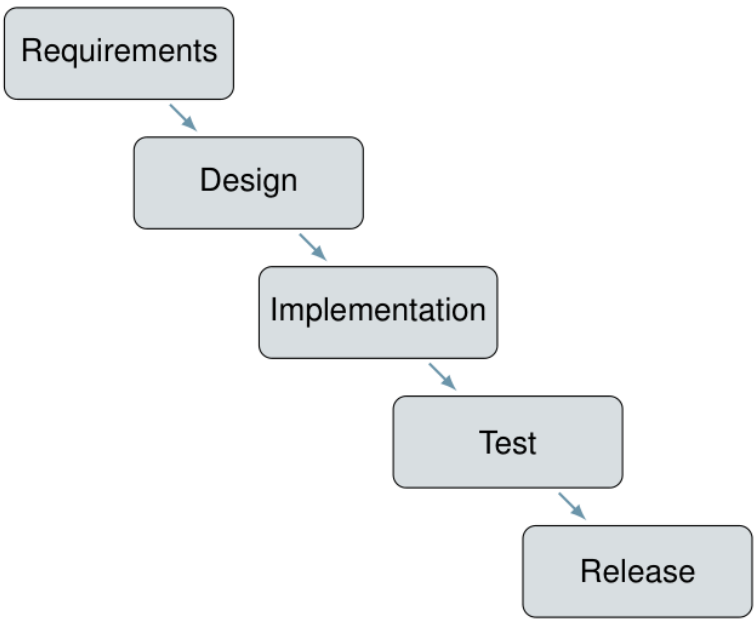
Abbildung 1: Produktzyklus

- **Development** – Entwicklung und Testen einer Hardware & Firmware mit der Funktionalität.
- **Deployment** – Verteilung in die Verkaufsstellen/-läden, etc.
- **Support** – Rücknahme, Reparatur, Austausch Komponenten (z.B. Batterie), Software Updates, etc.
- **Upgrade** – Verbesserungen bestehender Hardware
- **Retirement** – Abkündigung des Produktes, Gratis Rücknahme und Recycling von zurückgegeben Geräten.

4.1.2 Wasserfall Modell

Das *Wasserfall*-Modell fokusiert sich auf die Entwicklung des Produktes. In diesem Modell werden die fünf Hauptschritte nacheinander abgearbeitet und sind daher so aufgebaut, der nächste Schritt auf den Informationen des vorherigen basiert.

> ⚠ Warnung
>
> Das Modell ist problematisch, wenn die Anforderungen nicht klar sind oder man ein Neulands-Projekt macht. Es wird daher meistens in Kombination mit anderen Modellen/Systemen verwendet.



- **Requirements** – die Anforderungen werden möglichst komplett erfasst und dokumentiert
  - Resultat: *Product Requirement Document*

- **Design** – Ein Lösungsvorschlag wird erstellt, um die Anforderungen zu erfüllen, inklusive einer Zeitplannung.
  - Resultat: Modelle, Block Diagramme, Schemas, Klassen, Zeitplan
- **Implementation** – Die Umsetung des Designs (Hardware, Firmware, etc.)
- **Test** – Die Anforderungen werden mit systematischen Tests am Produkt überprüft und allenfalls korrigiert.
- **Release** – Produkt wird installiert (*Operations* oder Betrieb), gewartet und nötigenfalls repariert.

> Tipp
>
> Da die zukünftigen Schritte immer von Informationen von ihren vorherigen Schritten abhängen, ist man gezwungen, eine gute Dokumentation zu führen.

Vorteil:

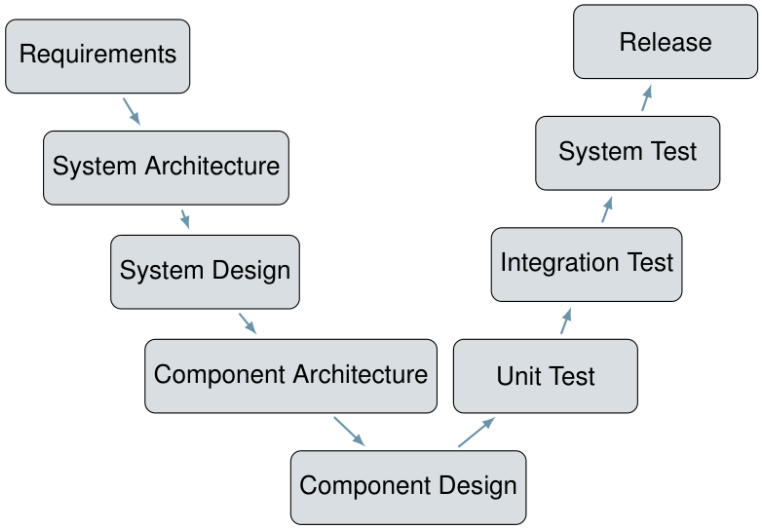
- Dokumentation wird genügend gut geführt
- einfach & skalierbar

'**Nachteile**':

- Kunde erhält das Produkt erst am Ende

4.1.3 V-Modell

Das V-Modell versucht das Wasserfall-Modell zu reparieren und verläuft vom **Grossen zum Kleinen** und wieder zurück. Die rechte Seite ist jedem Prozess auf der linken Seite eine Qualitätssicherung gegenübergestellt (*Unit Test*, *Integration Test*, *System Test*).



Der linke Teil wird *Realisierungsphase* genannt, im unteren Teil ist die Entwicklung (Source Code, Layout,...) und der rechte Teil ist die *Testphase*. Insbesondere ist wichtig, dass jede Verifikation auf der linken Seite mit einer Validierung auf der rechten Seite gegenübergestellt wird.

Anwendungsgebiet: Fokus auf 'Sicherheit' und 'Zuverlässigkeit', wie etwa Raumfahrt oder Automobilbau.

Vorteil:

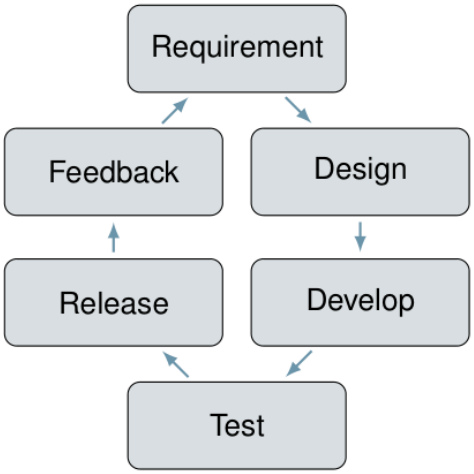
- Jedem Prozess wird eine Qualitätssicherung gegenübergestellt

'**Nachteile**':

- Kunde erhält das Produkt erst am Ende

4.1.4 Agile Modell

Der Fokus bei diesem Modell ist das Feedback. Da in den anderen Modellen der Kunde erst am Schluss das Produkt erhält, kann es sein, dass der Kunde während der Entwicklung kleinere Anpassungen möchte.



Die Phasen des Prozess werden mehrfach für ein Projekt durchschritten, bis es entweder das Budget aufgebraucht hat oder der Kunde mit dem Produkt zufrieden ist.

- **Requirement** – Anforderungen werden gesammelt und priorisiert für die nächste Phase.
- **Design** – Erstellung der Lösungsmöglichkeiten für die Anforderungen (Skizzen oder bereits Design-Dokumente). Meistens werden **einzelne Anforderungen** schrittweise geprüft.
- **Develop** – Realisierung des Designs mit einem normalen Entwicklungsprozess.
- **Test** – Mittels Unit- und System-Tests werden die Anforderungen verifiziert. Es sagt aber nicht aus, ob der Kunde es brauchbar findet.
- **Release** – Das Produkt wird internen oder externen Benutzer zur Benutzung ausgestellt.
- **Feedback** – Rückmeldungen zum Produkt: was funktioniert, was nicht, was soll geändert werden, was verbessert, etc.

Vorteil:

- Kommunikation mit Kunde wird gefördert
- Missverständnisse können früher geklärt werden
- viele schnelle Iterationen

Nachteile:

- keine klare Dokumentationsschritte, da man nicht weiss ob etwas in das Endprodukt schlussendlich integriert wird.

4.2 Werkzeuge

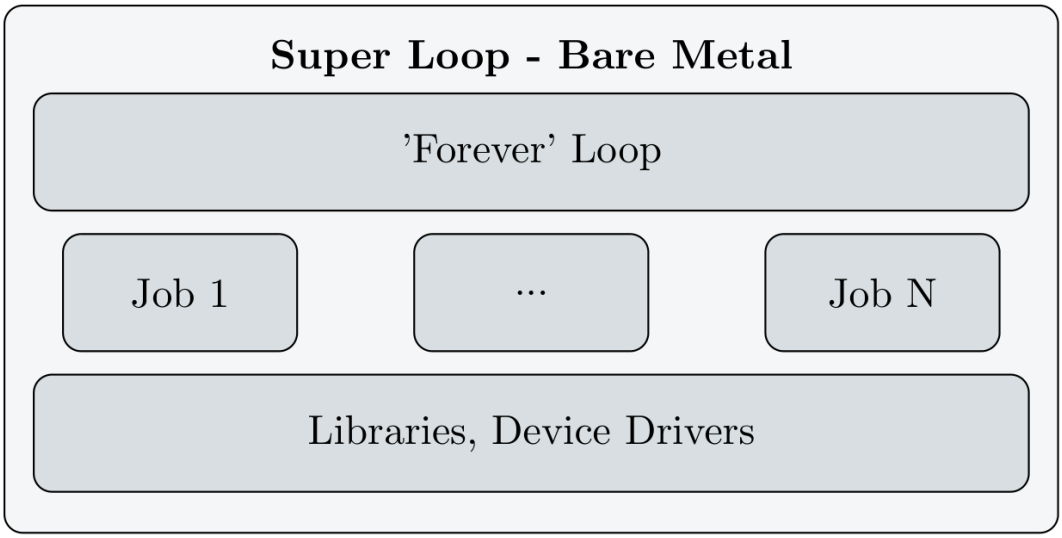
5. Firmware

5.1 Architektur

Firmware benötigt je nach Anwendung selbst eine Code-Architektur, welche die Art, wie Jobs ausgeführt werden, definiert. Welcher Loop verwendet wird, ist von verschiedenen Kriterien wie Grösse, Komplexität, Funktionalität, Anzahl Aufgaben, Anforderungen, Wartbarkeit, Erweiterbarkeit, wenige Ressourcen, usw. abhängig.

5.1.1 Super Loop

Beim *Super Loop* werden alle Arbeiten nacheinander erledigt und gehört zu den einfachen Ansätzen. Diese Art ist ebenfalls gut wartbar.



1. Initialisierung Hardware & Gerätetreiber
2. Unendlicher Loop welcher die Arbeiten nacheinander ausführt.

Beispiel

```
void main(void) {
  InitHardware ();
  InitDrivers ();
  for (;;) {
    DoJob1 ();
    DoJob2 ();
    /* ... */
    DoJobN ();
  } /* forever */
}
```

Herausforderung

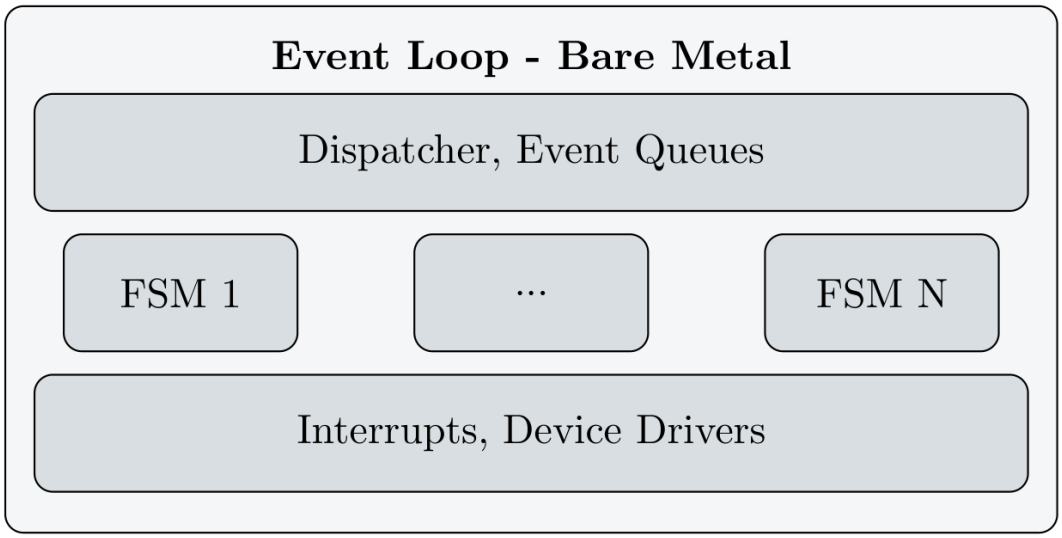
Eine Problematik, welche dieser Ansatz bringt, ist, wenn Arbeiten oder Jobs länger brauchen und somit andere verzögern. Besonders ist es ein **Problem**, wenn die **Verzögerungen variable** sind und sich **je nach Situation unterscheiden**.

Finite State Machine (FSM)

Eine Lösung zu diesem Problem ist, dass eine FSM verwendet wird, welche die Schritte der einzelnen Jobs aufteilet und somit die Latenz zwischen den Jobs verkürzt. Dies macht aber die Implementation der Zustände und Abarbeitung komplexer.

```
for (;;) {
  DoJob1_part1 (); /* split Job1 into smaller parts */
  DoJob2 ();
  DoJob1_part2 (); /* split Job1 into smaller parts */
  DoJob3 ();
  DoJob1_part3 (); /* split Job1 into smaller parts */
  /* ... */
  DoJobN ();
} /* forever */
```

5.1.2 Event Loop



Bei einem *Event Loop* werden Ereignisse/(Hardware) Interrupts verwendet, um Arbeiten zu veranlassen. Diese Art macht den Ablauf **ereignisgesteuert** und reduziert somit die Latenz der Jobs, da diese nur ausgeführt werden, wenn es nötig ist.

Der grösste Vorteil von diesem Ansatz ist es, dass der Main Loop in einem stromsparenden Modus gehen kann und später durch einen Event oder Interrupt wieder aufgeweckt werden kann.

Beispiel

```
void ButtonInterrupt (void) {
  QueueEvent( Button_Pressed );
}

void main(void) {
  InitHardware();
  InitDriversAndInterrupts();
  for (;;) { /* smaller loop */
    GoToSleep(); /* wait for event */
    ProcessQueues();
  } /* forever */
}
```

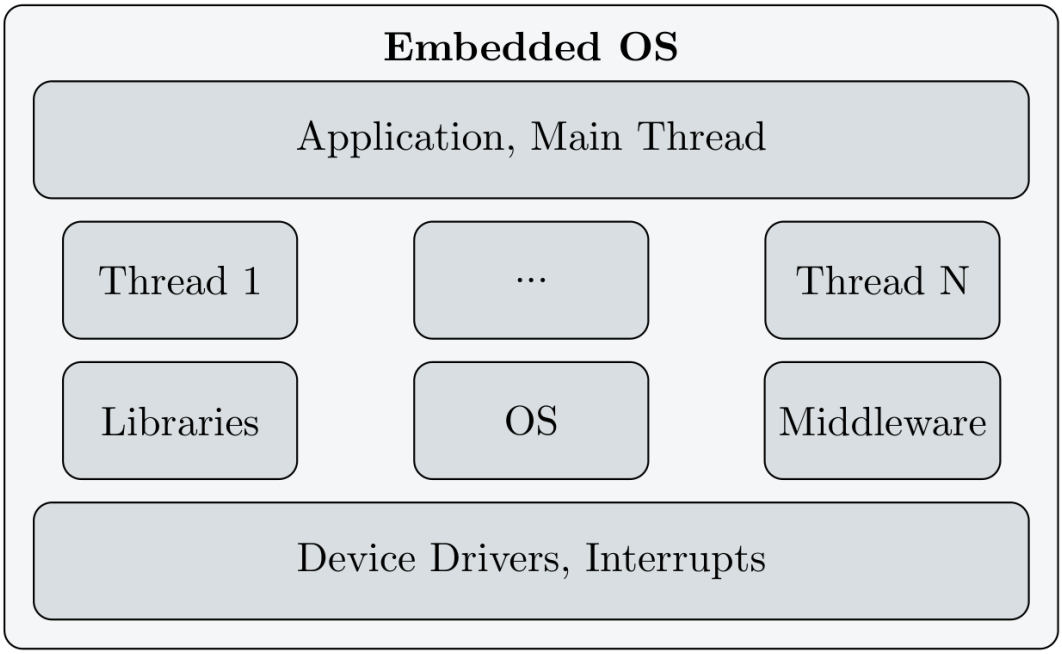
Herausforderung

FSM oder Jobs sollten nicht direkt in den Interrupts ausgeführt werden, da dies andere Interrupts blockieren kann und Latenz einführt oder durch einen höher priorisierten Interrupt unterbrochen werden.

- Für **kleine Jobs** kann dies direkt **im Interrupt-Event** ausgeführt werden.
- Für **alle anderen Jobs** sollte eine **Queue** oder einen Benachrichtigungsmechanismus wie den **Dispatcher** verwendet werden.

Mischform

Es gibt auch Mischformen von *Super Loops* und *Event Loops*, welche eine Balance zwischen Komplexität und Latenzzeit erreicht.



5.1.3 Embedded OS

Ein *Embedded OS* sind für Mikrocontroller oder kleine Computer ausgelegt und arbeitet mit **Threads**. Jobs werden in Threads verteilt und das Betriebsystem (*OS*) verwaltet die Ausführung der Threads. Dies bringt grosse Erweiterbarkeit und Flexibilität.

```
void mainTask(void) {
  CreateTask(sensorTask);
  CreateTask(otherTask);
  /* ... */
  for (;;) {
    /* do work */
  }
}

void main(void) {
  InitHardware();
  InitDriversAndInterrupts();
  CreateTask(mainTask);
  StartOS();
}
```

Vorteil dieses Ansatzes ist, dass jeder Job für sich arbeitet und Zugriff zur Hardware hat. Die Jobs arbeiten "parallel" miteinander.

> **Herausforderung**
>
> Eine Herauserforderung ist die Art, wie die Tasks beim OS-*Scheduler* "angemeldet" werden. Eine Alternative zu *Main Task erstellt alle anderen Tasks*, da dieser bei Resourcenmangel zu einem Abbruch des Systems führen kann, ist, dass alle Tasks vor dem Start des OS-*Scheduler* erstellt werden.

5.2 Modularisierung

6. Module

6.1 Anforderungen

- **Interface** –
- **Synchronisation** –
- **Organisation** –
- **Konfiguration** –
- **Funktionaliät**, **Init** und **Deinit** –

7. Echtzeit

Echtzeit wie wir es (durchschnittlich) betrachten unterscheidet sich grundsätzlich mit der technischen Echtzeit. Echtzeit für uns beschreibt im Bezug zur Raum-Zeit-Kontinuum eine Kontinuität und Gleichzeitigkeit, was im technischen Sinne nicht möglich ist.

Technische Echtzeit fokussiert sich auf die Rechtzeitigkeit und Richtigkeit. Das richtige Resultat muss zur richtigen Zeit rechtzeitig produziert werden.

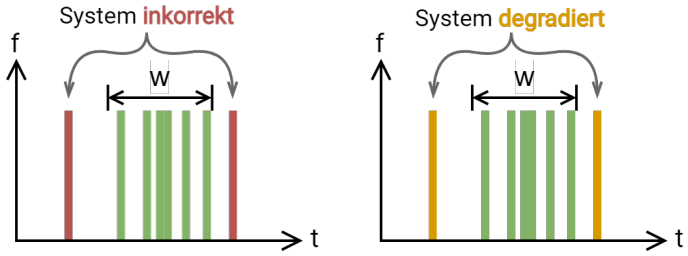
> **!** Definition Echtzeitsystem
>
> Ein Computer ist als Echtzeitsystem klassifiziert, wenn er auf externe Ereignisse in der echten Welt reagieren kann: mit dem richtigen Resultat, zur richtigen Zeit, unabhängig der Systemlast, auf eine deterministische und vorhersehbare Weise.

Da ein Rechner nicht unendlich viele Prozesse gleichzeitig laufen kann, entsteht eine **Systemlast** basierend auf **# gleichzeitige Events & Tasks**, **Intervall**, **Reaktionszeit**, **Verarbeitungszeit**. Je höher die Last, desto unzuverlässiger ist das System im Bezug zur Echtzeit-Anforderung.

> **!** Attribute Echtzeitsystem
>
> - **Rechtzeitigkeit** für alle Stufen: *Eingabe* ⇒ *Verarbeitung* ⇒ *Ausgabe*
> - **absolute** und **relative** Rechtzeitigkeit
>   - **Absolut**: Einschalten der Bewässerung jeden Tag um 05:30 am Morgen, ± 1 Minu**Relativ**: Nachdem ein trockener Boden festgestellt wurde, soll in der darauffolgenden Nacht um 22:00 Uhr die Bewässerung einschalten, ± 5 Minuten. Nach Einschaltung, schaltet die Bewässerung nach 30 Minuten, (± 10s) automatisch aus.

7.1 Harte und Weiche Echtzeit

Ein **hartes** Echtzeit-System ist extrem zeitkritisch. Wird die Zeitbedingung in einem Zeitbereich verpasst, so gilt bei



8. FreeRTOS

8.1 Kernel API

8.1.1 Scheduler starten

```
void vTaskStartScheduler(void);
```

8.1.2 Scheduler beenden

```
void vTaskEndScheduler(void);
```

8.1.3 Kernel/Scheduler anhalten

```
void vTaskSuspendAll(void);
```

8.1.4 Kernel/Scheduler fortsetzen

```
portBASE_TYPE xTaskResumeAll(void);
```

- `pdTRUE`
- `pdFALSE`

8.1.5 Kontext Switch forcieren

```
#define taskYIELD() portYIELD()
```

8.2 Queues

9. Kernel

9.1 FreeRTOS Interrupts

FreeRTOS benötigt zwei Interrupts:

- `SysTick` oder irgendein Timer Interrupt
- Software Interrupts
  - `SVCall` - sofortige Ausführung
  - `PendableSrvReq` - führt aus, sobald möglich (daher *pendable*)

Wer bearbeitet welche Interrupts?

Dies ist **Port-abhängig** und daher nicht definitiv, ausser das FreeRTOS mindestens die zwei nötigen Interrupts von vorher behandelt.
Im Port für die AEMBS-Mikrocontroller behandelt das OS nur die beiden Interrupts, die anderen sind unter Kontrolle von der Anwendung.

9.2 Hardware Stack

Bei Programmstart wird mit dem **MSP** gearbeitet, welcher für **Interrupts** und **Exception Handling** zuständig ist. Wird der Scheduler gestartet werden für **Tasks** der PSP verwendet.

Vorteil davon ist, dass die **Stacks separat** sind und somit kann der Task nicht in den Hauptstack hinein*pfuschen*.

9.3 Critical Sections

Standardmässig verwendet das FreeRTOS keine Critical Sections für seine Datenstrukturen. Das (De)aktivieren von Interrupts benötigt Zeit und kann je nach dem kritische Verzögerungen einführen.

Facts

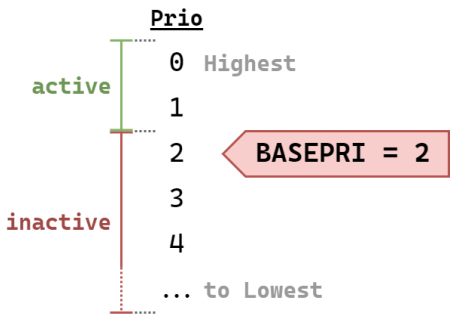
- Interrupts bleiben eingeschaltet
- Effizienz und weniger Interrupt Latenz
- RTOS API mit FromISR Suffix benutzten Critical Sections

9.4 `PRIMASK` & `BASEPRI`

Diese Register werden für Interrupt-Disable/Enable in FreeRTOS verwendet, damit logischerweise die Interrupts aktiviert und deaktiviert werden können.

`PRIMASK` – 1-Bit Register, welches bei `1` die Ausführung der Interrupts im ISR deaktiviert. `0` aktiviert die Ausführung wieder.

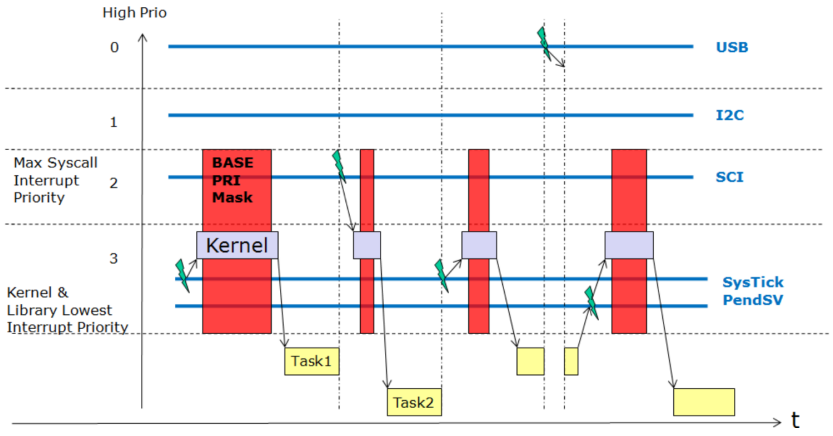
`BASEPRI` – Ein nicht-null Wert, welcher einer Aufwärtsmaskierung von Interrupts angibt, welche deaktiviert werden. `0` ist die höchste Priorität und kann nicht deaktiviert werden, daher ein nicht-null Wert.



LPC845 & TinyK22

Der LPC845 (Cortex M0+) verfügt über kein `BASEPRI` und deaktiviert daher bei einem Interrupt-Disable/Enable alle Interrupts. Der TinyK22 (Cortex M4F) verfügt über ein `BASEPRI`-Register.

> **Atomic Operation**
>
> Verfügt der MCU über *Atomic Operations* (1-Zyklus Operationen von anderen Threads oder von Aussen betrachtet), werden gewisse Critical Sections wegdefiniert (z.B. bei `TICK_TYPE`).



10. Synchronisation

11. Nachrichten

Grosse Systeme werden meist modularisiert und aufgeteilt. Die Teile des Systems können nur miteinander über Daten- & Nachrichtenaustausch kommunizieren (Interprozesskommunikation). In FreeRTOS werden dafür *Queues* & *Timer* verwendet.

> **Direct Task Notification**
>
> Falls Queues nicht verwendet werden, kann die *Direct Task Notification* verwendet werden → Direkte Kommunikation zwischen Tasks & Interrupt zu einem Task möglich

11.1 Queues

11.2 Timer

12. Parallelität

12.1 Reentrancy

12.1.1 Dis-/Enable Interrupts

> **Nachteil**

12.1.2 Critical Sections

```
#define CriticalVariable() \
  uint8_t cpuSR                                                    ①
```

① definiert lokale Variable `cpuSR` für Sicherung des aktuellen Interrupt-Zustandes

```
#define EnterCritical()      \
  do {                       \
    __asm (                  \
      "mrs   r0, PRIMASK \n" \ ①
      "cpsid i           \n" \ ②
      "strb r0, cpuSR    \n" \ ③
    );                       \
  } while(0)
```

① `PRIMASK` wird in `R0` abgespeichert
② Interrupts deaktivieren
③ `R0` in `cpuSR` abspeichern

```
#define ExitCritical()    \
  do {                    \
    __asm(                \
      "ldrb r0, cpuSR \n" \ ①
      "msr PRIMASK,r0 \n" \ ②
    );                    \
  } while(0)
```

① Inhalt von `cpuSR` in `R0` laden
② `R0` nach `PRIMASK` kopieren.

13. Benutzer

14. Grafik

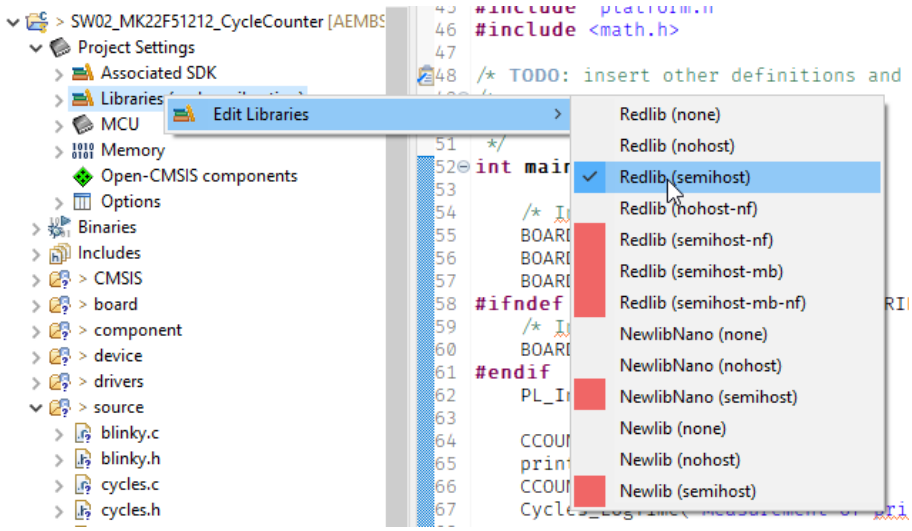
15. MCUXpresso

15.1 Semihosting

Um Semihosting zu aktivieren (falls es schon nicht ist), muss die *Library* des Projektes geändert werden.

`aktuelles Projekt` → `Project Settings` → auf `Libraries` Rechtsklick machen → `Edit Libraries`

Alle Optionen mit *semihost* können ausgewählt werden.



15.1.1 Unterschied none, nohost and semihost

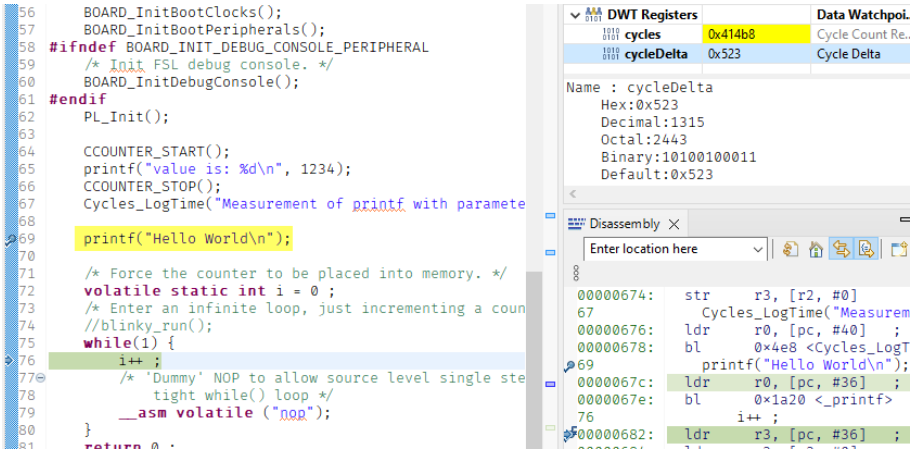
- **Semihosting** implementiert alle Funktionen, inklusive *File I/O*. *File I/O* wird über den Debugger weitergeleitet und auf dem Hostsystem ausgeführt. Zum Beispiel verwenden printf/scanf das Konsolenfenster des Debuggers und fread/fwrite arbeiten mit Dateien auf dem Hostsystem.
- **Nohost** implementiert String- und Speicher-Handling Funktionen und ein paar *File*-basierten I/O Funktionen. Es wird davon ausgegangen, dass es kein Debugging Host existiert.
- **None** hat den kleinsten Speicherplatzbedarf. Es schließt Low-Level-Funktionen für alle *File*-basierten I/O und einige String- und Speicherverarbeitungsfunktionen aus.

15.2 DWT

Der MK22FN512VLH12 (oder allgemein Cortex-M4) verfügt über ein DWT-Unit (**D**ata **W**atchpoint and **T**race), welches die Anzahl abgelaufenen Zyklen angibt und die Delta-Zyklen zwischen zwei Breakpoints.

`cycles` gibt die Anzahl abgelaufenen Zyklen
`cycleDelta` gibt die Anzahl Zyklen zwischen zwei Breakpoints

Folgendes Beispiel zeigt die Anzahl Zyklen, welche für `printf` verwendet wurde.



16. Fragen

16.1 SW01 Einführung

16.1.1 Administratives

1. Was bedeutet *Embedded*?

Embedded bedeutet 'eingebettet' und weist darauf hin, dass ein Komponente oder Objekt Teil eines Ganzen ist.

2. Was ist ein *Embedded System*?

Embedded Systems bestehen aus Rechner (zum Beispiel Mikrocontroller), welche in einem grösseren System 'eingebettet' sind, also Teil eines Ganzen. Obwohl diese Rechner sind, sind diese nicht als 'normale' Rechner wie ein Desktop Computer oder Laptop erkennbar. Diese besitzen keine typischen Merkmale, wie Bildschirm, Maus, Tastatur, sondern sind für einen speziellen Zweck optimiert. Durch die Spezialisierung besitzt meistens ein Embedded System beschränkte Ressourcen oder sind auf gewisse Faktoren wie Grösse, Kosten oder Energieverbrauch ausgelegt.

3. Was bedeutet *IPC*?

Inter-**P**rocess **C**ommunication, wie zum Beispiel eine Queue. Dies beschreibt die Kommunikation zwischen zwei Prozessen in einem RTOS.

4. Beurteile, ob ein Raspberry Pi ein *Embedded System* ist.

Ein Raspberry Pi kann für beide Fälle verwendet werden, entweder als ein Modul in einem System, welche zum Beispiel Bildverarbeitung für ein Bilderkennungssystem macht, oder als Desktop Computer verwendet werden. Der Übergang ist fliessend. Man kann es als Mini-Computer bezeichnen.

5. Erkläre, was man unter *Build Tools* versteht.

Unter Build Tools versteht man die Werkzeuge, welche das Programm in den Maschinen Code übersetzt. Funktionen sind: Compiler, Linker, Standard Libraries, Debugger und zusätzliche Tools. Das Build Environment übernimmt die Anwendung dieser Werkzeuge.

16.1.2 Software & Tools

1. Eclipse ist eine sehr universell einsetzbare IDE, was vielleicht auch problematisch sein kann. Was wären mögliche Kritikpunkte?

Weniger Performant, da es in Java geschrieben ist. Die Komplexität von Eclipse führt ebenfalls zu Performance-Einbussen.

2. Was hat wohl wesentlich dazu beigetragen, dass Eclipse als Open Source Projekt erfolgreich wurde?

Durch das Eclipse Open Source Konsortium wurde die Entwicklung von Eclipse stark gefördert und in die Öffentlichkeit gebracht. Ebenfalls hat es das Plugin-System eingeführt.

3. Was ist die Rolle einer Foundation wie die der für Eclipse? Inwiefern unterscheidet sich eine Foundation von einer Firma wie IBM?

*Foundation*s sind non-profit Organisationen.

4. Hersteller bieten oft 'Eval Boards' an (ähnlich wie das tinyK22). Was ist der Sinn und Zweck davon?

Ein Eval-Board besitzt das Minimum der kritischen Komponenten, damit der Haupt-Komponenten (z.B. Sensor) funktioniert. Es wird zum Evaluieren/Austesten der Haupt-Komponent verwendet, aber auch für die direkte Hardware Integration. Ebenfalls reduziert es den Hardware-Designaufwand, da man keine eigene Eval-Boards machen muss.

5. Welche Komponenten finden Sie typischerweise in einem SDK?

Beispiel-Projekte, Dokumentation, Treiber, Lizenzinformationen.

6. Das Raspberry Pi ist weder das beste, schnellste, modernste noch das billigste Board, trotzdem ist es ein Erfolg. Was könnten die Erfolgsfaktoren sein?

Die Raspberry Foundation legt grossen Wert auf die Dokumentationen ihrer Geräte/Produkte. Die Datenblätter & andere Dokumentation sind sehr detailliert beschrieben und besteht ebenfalls aus vielen Anleitungen, wie man den Raspberry Pi.

7. Was ist der Grund, dass man im Pins Tool für einen Pin einen Identifier verwendet?

Damit im Code nicht mit *Magic Numbers* gearbeitet wird, also im Sinne dass man direkt sieht, mit welchem Pin man es zu tun hat. Man gibt dem Pin eine Bedeutung.

8. Sie wollen von Ihrer Firma ein Projekt in die Open Source Domäne 'entlassen'. Was müssten Sie dabei berücksichtigen, damit es ein Erfolg wird?

Alle verwendeten Software-Komponenten müssen öffentlich zugänglich sein (nicht closed source) und die entsprechende Lizenz muss einer Open Source Lizenz entsprechen (z.B. GNU GPLv3, MIT, Creative Commons).

9. Sie realisieren ein neues Embedded System: Was ist der Unterschied zwischen *Design* und *Architektur*?

Die Architektur beschreibt die Beziehung zwischen den Komponenten (z.B. Kommunikation) und Design beschreibt die Implementation bezüglich der Architektur.

10. Für welche Anforderungen oder Anwendungen eignet sich eher ein FPGA als ein Mikrocontroller? Was sind die Gründe dafür?

FPGAs werden für datenintensive, parallele und reaktionsschnelle Systeme verwendet. Beispiel wäre ein Digital Oszilloskop.

11. Was versteht man unter einer Debug Probe?

Eine Debug-Probe ist ein Gerät, welches die Verbindung zwischen Target und Host. Unter anderem, verwenden diese Geräte als Schnittstellen JTAG, SWD oder ISP.

12. Was ist *CMSIS-DAP*?

Common **M**icrocontroller **S**oftware **I**ndustry **S**tandard - **D**ebug **A**ccess **P**ort

CMSIS-DAP ist ein standardisiertes Kommunikationsprotokoll für Debug-Zwecken. Die Firmware von CMSIS-DAP wird DAPLink verwendet.

13. Was ist *CMSIS*?

Common **M**icrocontroller **S**oftware **I**ndustry **S**tandard ist eine Kollektion von verschiedenen Industrie-Standards für Mikrocontroller Systemen. Es bietet Schnittstellen zu Prozessoren und Peripheriegeräten, Echtzeitbetriebssystemen und Middleware-Komponenten. CMSIS umfasst einen Liefermechanismus für Geräte, Platinen und Software und ermöglicht die Kombination von Softwarekomponenten verschiedener Anbieter.

16.2 SW02 Architektur

16.2.1 System

1. Nenne drei gute Beispiele eines transformierenden Systems.

MP3/MPEG De-/Encodierung, Verschlüsselung, Noise Canceling, Quarto → Latex

2. Nenne drei gute Beispiele eines reaktiven Systems.

ABS, Roll-Over-Detection, Regelkreis eines positiongesteuerten Motors, PID-Regler, Regelkreis

3. Nenne drei gute Beispiele eines interaktiven Systems.

Ticketautomat, Selfscan-Gerät, Taschenrechner

4. Inwiefern unterscheiden sich transformierende Systeme von reaktiven Systemen?

Reaktive Systeme sind typisch Echtzeitsysteme und reagieren auf externe Events. Transformierende Systeme verarbeiten alles, was am Input anliegt und geben es verarbeitet am Output aus.

reaktiv → closed loop ; transformativ → open loop

5. Beschreibe ein gutes Beispiel eines transformierenden Systems, welches über einen Eingabestrom und zwei Ausgabeströme verfügt.

Soundkarte → MP3 Decodierung gibt Stereo-Sound aus

6. Gibt es ein Beispiel eines Embedded Systems ohne Benutzerschnittstelle?

Ein elektronischer Regelkreis, automatisiertes System (Abpackmaschine)

7. Zu welcher System Klasse gehört ein 'Embedded System'?

Abhängig von Anwendungszweck, aber kann grundlegend alle Systeme annehmen.

8. Viele Systeme sind eine Kombination von transformierenden, reaktiven und interaktiven Systemen. Bestimme diese am Beispiel eines Smartphones

Interaktiv: Homescreen- & App-Bedienung

Reaktiv: Helligkeitssensor für automatische Bildschirmhelligkeit

Transformativ: Telefongespräche (Digital zu Analog)

9. Wieso ist die Verarbeitungsqualität für transformierende Systeme so wichtig?

Damit der Informationsverlust gering gehalten werden kann.

10. Wieso sind transformierende Systeme typischerweise optimiert für eine optimale Systemausnutzung?

Transformierte Systeme sind meist für eine spezifische Aufgabe ausgelegt/optimiert, da die Systemlast bekannt (kann als "Worst-Case" betrachtet).

11. Interaktive Systeme sind typischerweise optimiert für eine schnelle Antwortzeit. überlege typische Antwortzeiten für interaktive Systeme geben: Wovon hängen diese ab?

Die Antwortzeit ist abhängig von der Applikation/Funktion. Die Antwortzeit muss ein bisschen schneller sein, als eine Person eine "Langsamkeit" wahrnehmen kann.

12. Was bedeutet 'Verarbeitungsqualität' bei einem Audio Encoder System?

Das genügend Informationen in die Encodierung einberechnet werden, bzw. dass man die Audio-Qualität/Information (Spektrum) nicht verliert.

13. Klassifiziere die folgenden Systeme nach reaktiv, interaktiv und transformativ: Digital-Uhr, Airbag, Polizei-Radar, Feuer Alarmsystem, Geldautomat, Tankanzeige im Flugzeug.

Interaktiv: Geldautomat

Reaktiv: Airbag, Polizei-Radar(kasten; wartet auf zu schnelles Fahrzeug), Feuer Alarmsystem (wartet auf zu viel Rauch), Tankanzeige im Flugzeug

Transformativ: Tankanzeige im Flugzeug, Digital-Uhr, Polizei-Radar (Laserpistole; Geschwindigkeitsmessung wird zu Foti tranformiert)

14. Nenne einige Systeme, welche keinen Computer oder Mikroprozessor verwenden.

transformativ: Widerstand (Strom zu Wärme & Potentialunterschied), Velo

reaktiv: Mimose (Pflanze),

interaktiv: Lebewesen

15. Mit mehr Speicher können Systeme oft schneller rechnen. Probiere Beispiele dazu zu finden.

Kamera mit Serieauslöser (Bufferspeicher), Lookup-Tables

16.2.2 Rechner

1. Ist ein Intel basiertes Notebook eher eine von Neumann oder Harvard Architektur?

No answer here!

2. Ist der Instruktionssatz des tinyK22 CISC oder RISC?

No answer here!

3. Wie kann eine RISC Architektur einen Rechner beschleunigen, da doch dabei mehr Instruktionen ausgeführt werden müssen?

No answer here!

4. Wieso eignen sich SIMD Instruktionen vor allem für Signalverarbeitung?

No answer here!

5. Was sind die Grenzen eines SoC Ansatzes, und wie können diese überwunden werden?

No answer here!

6. Wieso benötigt man ein XiP Verfahren bei einem externen Programmspeicher? Hinweis: Adressbereiche.

No answer here!

16.2.3 Cortex

1. Eine Anwendung verwenden viele 32bit Multiplikationen und Divisionen. Eignet sich ein ARM Cortex-M0+ dafür? Was sind Alternativen?

No answer here!

2. Was ist der Grund, dass beim M7 oft ein externer Speicher zum Einsatz kommt?

No answer here!

3. Wieso ist eine MMU für den Einsatz eines Linux nötig?

No answer here!

4. Wieso wurde ARM mit den ARM11 so erfolgreich?

No answer here!

5. Der M3 war und ist sehr erfolgreich. Was waren die Gründe für den M0 und M4?

No answer here!

6. Was ist das Konzept von TrustZone?

No answer here!

7. Was ist der Unterschied zwischen einer MPU und einer MMU?

No answer here!

8. Bringe ein Beispiel für eine Sättigungsarithmethik.

No answer here!

9. Welche Schlüsseleigenschaften wurden von welchen Firmen in die Gründung des ARM Joint Venture eingebracht und von wem?

No answer here!

10. Wieso wurde die mögliche Übernahme von ARM durch Nvidia kontrovers diskutiert?

No answer here!

11. Was sind die wichtigsten Erfolgsfaktoren von ARM Prozessoren aus Sicht der Anwender?

No answer here!

12. Was könnte ein guter Kritikpunkt an ARM und deren Prozessoren sein? Gibt es Alternativen?

No answer here!

13. RISC-V ist in 'aller Munde': Beschreibe in ein paar kurzen Sätzen was RISC-V ist.

No answer here!

14. Welcher Vorteil hat Arm gegenüber einer Konkurrenz wie Intel?

No answer here!

15. Ein Temperatur Sensor unterstützt einen Temperaturbereich von -45 Grad Celsius bis 125 Grad Celsius mit einer Auflösung von 0.1 Grad? Ist dafür eine Gleitkomma-Repräsentation mit den zugehörigen Operationen angebracht? Was wäre eine Möglichkeit?

No answer here!

16.3 SW03 Entwicklung

16.3.1 Prozess

1. Was ist der Unterschied zwischen Verifikation und Validierung? Erläutere es mit einem Beispiel.

Verifikation hat ihre Anwendung im internen Produktionsprozess und prüft, ob ein Produkt den Spezifikation entspricht (*are you building the right thing*). *Validation* prüft, ob die letztendliche Anwendbarkeit funktioniert, also dem geplanten Zweck dienlich ist (*are you building the right thing*).

Beispiel: Defibrillator gibt 3000V für 3 Millisekunden aus, um ein Herz wieder in den normalen Rhythmus zu bringen. In der Situation, dass der **Defibrillator 3000V für 3ms** ausgibt, **aber** das **Herz fehlschlägt**, ist die Verifizierung erfolgreich, aber die Validation fehlgeschlagen.

2. Erkläre den Unterschied zwischen *Unit Test*, *Integration Test* und *System Test*?

Ein *Unit-Test* überprüft die Funktionialität von einzelnen Funktionen (Units), entspricht also dem untersten Test-Level des V-Modells. Der *Integration-Test* überprüft die einzelnen Units im Zusammenspiel, also Integriert. Der *System-Test* schlussendlich überprüft das gesamte System.

3. Wieso geht die Phase der Ausserbetriebnahme eines Produktes oft vergessen? Was sind mögliche Konsequenzen?

Weil, dies Kosten verursacht, ohne einen direkten benefit zu erhalten.

4. Das Wasserfall Modell wird oft als 'schlecht' dargestellt? Ist das berechtigt?

Ändernde Anforderungen sind nur sehr schwer zu implementieren und auch bei einem Fehler ist der Schritt zurück sehr gross.

5. Welche grundlegenden Vorteile führt das V Modell gegenüber dem Wasserfall Modell ein?

Das *V-Modell* verfügt über eine kleinere Granularität und hat den Fokus auf Tests. So erhält man mehr Sicherheit, dass die einzelnen Stufen richtig implementiert wurden.

6. Welche Anforderungen stellt das Agile Modell an das Entwicklungsteam?

Gute Kommunikation und Abschprachen.

7. Unter welcher Annahme wird das Agile Modell nur einmal durchlaufen?

Das alles beim ersten mal nach Kundenwünschen implementiert wurde.

8. Wieso braucht es beim Agile Modell ein Backlog?

Übersicht über noch offene Arbeiten.

16.3.2 Werkzeuge

1. Was versteht man unter einem Refactoring und was ist das Ziel davon?

Bei einem Refactoring wird etwas neu umgeschrieben oder geändert, ohne die eigentliche Funktionalität zu ändern. Ein Refactoring hat häufig eine Verbesserung von Lesbarkeit und Wartbarkeit als Ziel.

2. Was ist der Unterschied zwischen Coverage und Profiling?

Die *Coverage* zeichnet die Testabdeckung auf. Bei dieser Instrumentation wird aufgezeichnet, welche Code Zeilen wie oft ausgeführt wurden.

Das *Profiling* zeichnet periodisch die Position des Programmpointers auf um Rückschlüsse darauf zu ziehen, wo sich das Programm am meisten aufhält.

3. Was ist der grosse Vorteil von statischen Analyse Werkzeugen gegenüber den dynamischen?

Die *statische Analyse* kann on the fly geschehen (z.B. in der IDE) und gibt so Informationen währen man das Programm schreibt.

4. Welcher Vorteil ergibt eine Commit Phase mit einem VCS?

Falls meine Änderung einen Fehler hervorruft, kann ich auf einen Commit zuvor zurücksetzten. Zudem kann ich auch Tage später noch sehen, was ich zu diesem Zeitpunkt implementiert habe.

5. Gib ein Beispiel, wo es schwierig ist eine 100% Coverage zu erreichen?

Zum Beispiel Error-Routinen, wenn das Programm nie in einen Error fällt.

16.4 SW04 Firmware

16.4.1 Architektur

1. Welche Aufgaben hat ein *Device Driver*?

Ein *Device Driver* ist eine Hardware-Abstraktion und übernimmt die Hardware-Konfigurierung, -Ansteuerung, etc.

2. Was versteht man unter *Latenz*?

Verzögerung bei Versenden von Informationen, bei Verarbeitungen...

3. Wieso darf man möglicherweise im Event Loop Modell nicht alle Arbeit im Interrupt machen?

Damit man keine grossen Unterbrüche/Latenz entstehen, wenn ein Task zu lange zum Verarbeiten hat. Meistens wird mit einem Flag gearbeitet, welcher dann im Hauptprozess den Task ausführt.

4. Welche Funktionen stellt die Standard Library der Programmiersprache C zur Verfügung?

No answer here!

5. Wie kann beim Event Loop Modell die Latenz oder Ausführungszeit in den Interrupts minimiert werden?

No answer here!

6. Wieso ist eine gute Hardware Abstraktion wichtig für eine Firmware?

No answer here!

7. Wieso spricht man von Firmware und nicht von Software im Bereich von Embedded Systems?

No answer here!

8. Was ist wohl der Grund, dass Middleware so heisst und nicht anders?

No answer here!

9. Welche Funktionalität bietet CMSIS-DSP und was ist der Vorteil gegenüber der Standard Library?

No answer here!

10. Welche Herausforderungen stellen sich beim Super Loop wenn man neue Jobs hinzufügt? Was wären Lösungsmöglichkeiten?

No answer here!

11. Wie kann man beim Super Loop Modell auch in einen stromsparenden Modus gehen?

No answer here!

12. Anstatt dass man vor dem Starten des Betriebssystem in einem Embedded System alle Tasks erstellt, kann man dies euch in einem einzelnen Startup Task machen? Diskutiere Vor- und Nacheile.

No answer here!

13. Was ist mit Quasi-Gleichzeitig in einem System gemeint?

No answer here!

14. Welches fundamentale Prinzip wird durch Setter und Getter realisiert?

No answer here!

15. Nenne eine typische Anwendung für einen Super Loop?

No answer here!

16. Nenne eine typische Anwendung für einen Event Loop?

No answer here!

17. Nenne eine typische Anwendung für ein Embedded OS?

No answer here!

18. Beschreibe die Aufgabe des Dispatcher im Event-Loop Modell? Wieso braucht es den und was genau macht dieser?

No answer here!

16.4.2 Module

1. Auf welche verschiedene Arten kann man einen Software oder Hardware Treiber konfigurieren?

No answer here!

2. Was meint man damit, dass in C nur ein Namespace existiert? Was bedeutet dies? Welche Lösungsmöglichkeiten gibt es?

Ein Namespace wird als *Scope-Wechsel* verwendet, wobei C nur ein Hauptnamespace hat. In C++ können Namespaces verwendet werden, falls es mehrere Funktionen mit der gleichen Signatur und Funktionsnamen hat. Damit können die Funktionen unterschieden werden.

3. Erkläre das Konzept der Kapselung anhand eines Treibers für eine UART.

No answer here!

4. Erkläre das Konzept der Abstraktion anhand eines Treibers für eine UART.

No answer here!

5. Welches grosse Problem tritt auf, wenn eine Schnittstelle nicht selfcontained ist?

No answer here!

6. Inwiefern realisiert ein Device Handle einen objektorientierten Ansatz?

No answer here!

7. Wir haben einen Ansatz mittels GetDefaultConfiguration() in der Programmiersprache C angeschaut: was ist die Alternative davon in der OOP Welt?

No answer here!

16.4.3 Bibliotheken

1. Finde heraus: welche Dateierweiterung haben vor-kompilierte Bibliotheken bei der GNU Toolchain? Für was steht die Erweiterung?

No answer here!

2. Was könnte der Grund oder die Gründe sein, dass die GNU Standard Bibliotheken ohne Debug Information installiert sind?

No answer here!

3. Was ist der Grund dafür, dass der Startup Code die main() Routine nicht direkt aufruft?

No answer here!

4. Wieso ruft der Compiler Runtime Routinen auf, anstatt die nötigen Instruktionen direkt in der übersetzten Anwendung zu platzieren?

No answer here!

5. Obwohl die Standard Library einen eigenen Startup Code hat, wird dieser oft nicht benutzt. Wieso?

No answer here!

6. Probiere die GNU newlib in der MCUXpresso Installation zu lokalisieren. Wo ist diese installiert? (Hinweis: libc.a)

No answer here!

7. Bei der nohost Variante sind die Stubs leer implementiert und machen gar nichts. Was ist der Sinn davon?

No answer here!

8. Verglichen mit newlib verbraucht newlib-nano für printf() Ausgaben weniger Speicher, ist aber auch viel langsamer. Wieso?

No answer here!

9. Standard Library Headers soll man mit <...> inkludieren. Was ist der Grund dafür?

No answer here!

10. Man hat die Wahl zwischen der Verwendung von `glibc`, `newlib`, `newlibnano` oder einer proprietären Bibliothek wie RedLib. Welche verwenden man, mit welcher Begründung?

No answer here!

11. Wenn man eine Bibliothek als Archive bekommt, bedeutet das auf jeden Fall dass man nur auf Assembly Code Stufe ohne zusätzliche Informationen debuggen muss? Begründe die Antwort.

No answer here!

12. Probiere mindestens drei verschiedene Runtime Funktionen (Namen) im Code zu finden. Hinweis: Assembly Code anschauen, gute Gelegenheiten sind Operationen mit double oder Zuweisungen von double zu int.

No answer here!

13. Was brauchen man mit einer C++ Anwendung: newlib oder newlib-nano?

No answer here!

14. CMSIS-Core stellt auch eine Startup Routine zur Verfügung. Wie heisst diese?

No answer here!

15. Mit CMSIS-Core kann man die Interrupts der CPU ein- und ausschalten. Finde die zwei Funktionen? Hinweis: cmsis gcc.h

No answer here!

16.5 SW05 RTOS

16.5.1 Echtzeit

16.5.2 FreeRTOS

16.5.3 Archtiektur

16.5.4 Kernel API

16.5.5 Tasks

16.6 SW06 Kernel

16.6.1 Interrupts

16.6.2 Visualisierung

16.7 SW07 Synchronisation

16.7.1 Synchronisierung

16.7.2 FreeRTOS & Interrupts

16.8 SW08 Nachrichten

16.8.1 Queues

16.8.2 Timer

16.9 SW09 Parallelität

16.9.1 Reentrancy

16.9.2 Sema

16.10 SW10 Benutzer

16.10.1 Benutzerschnittstellen

16.11 SW11 Grafik

16.11.1 Graphical User Interface