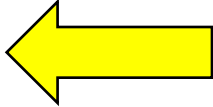


**Folien zur Vorlesung  
Grundlagen systemnahes Programmieren  
Sommersemester 2016  
(Teil 6)**

**Prof. Dr. Franz Korf**  
[Franz.Korf@haw-hamburg.de](mailto:Franz.Korf@haw-hamburg.de)

## Kapitel 6: Interrupt Behandlung

### Gliederung

- Einführung 
- Interrupt Behandlung
- Schritte der Interrupt Behandlung
- Zusammenfassung

# Programmausführung ohne Betriebssystem

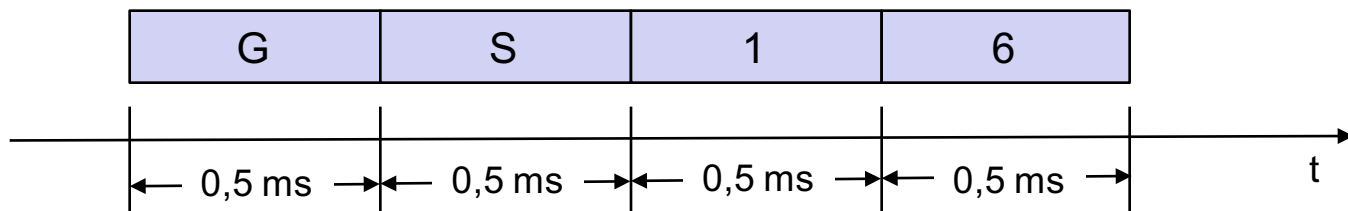
## Typische Technik: **Super Loop**

- Die einzelnen Tasks werden der Reihe nach in einer Hauptschleife abgearbeitet.
- Maximale Reaktionszeit: Summe der maximalen Reaktionszeiten aller Tasks

```
init_system();  
  
while (1) {  
    task_1();  
    task_2();  
    ...  
    task_n();  
}
```

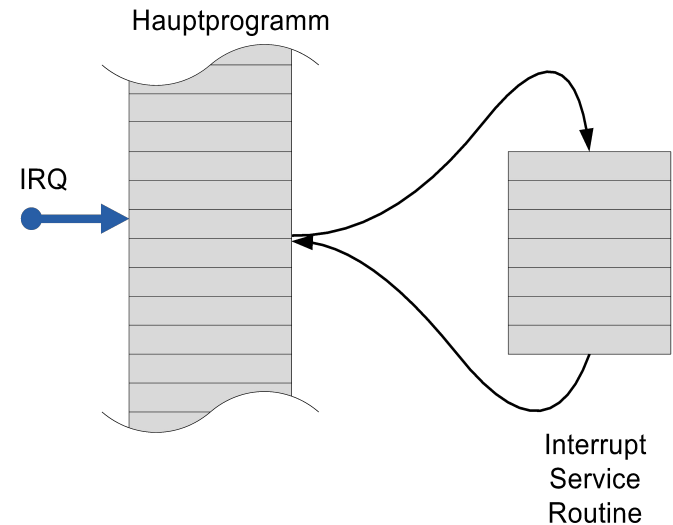
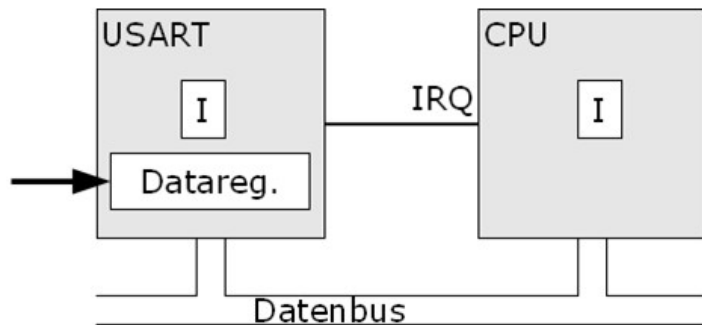
## Reaktion auf externe Ereignisse

- Regelmäßige Abfrage des Eingangsregisters
  - Drehimpulsgeber
  - Zeichen einer seriellen Schnittstelle
- Verarbeitung der neu eingetroffenen Daten (falls welche eingetroffen sind)
- Reaktionszeit auf eintreffende Daten muss so klein sein, dass keine Daten verloren gehen.
- Beispiel Serielle Schnittstelle: Das Datenregister im UART speichert nur 1 Byte
  - Geschwindigkeit 19200 Baud und 11 Symbole für ein Byte
  - Alle  $11/19200 \text{ s} \approx 0,5 \text{ ms}$  ein neues Zeichen in Datenregister
  - Alle 0,5 ms muss ein Zeichen ausgelesen werden (wenn Daten eintreffen).



# Interruptverarbeitung

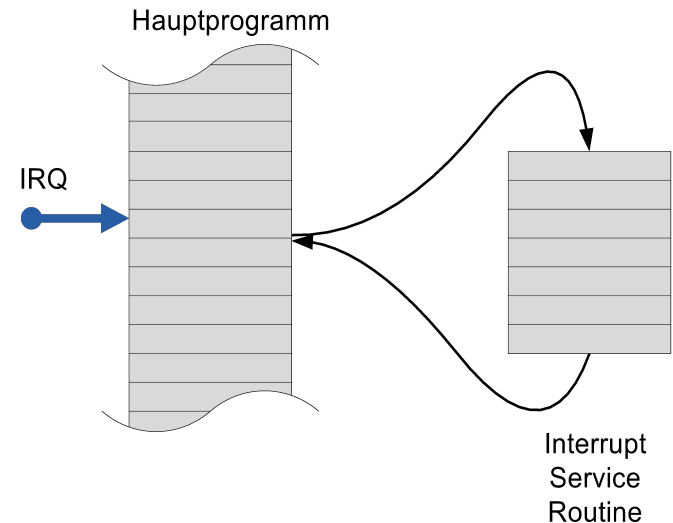
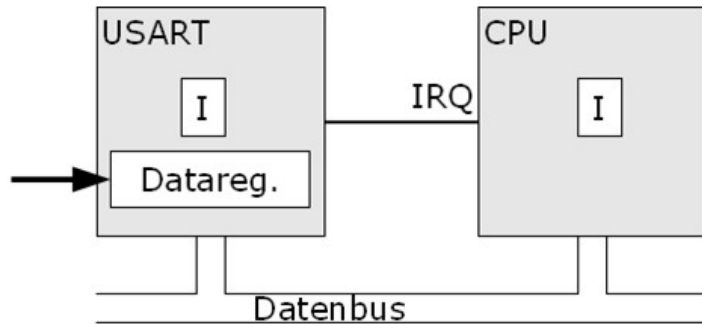
**Idee:** Lese Daten ein, sobald sie eintreffen – egal wo das Hauptprogramm gerade steht.



- Device (UART) löst IRQ (Interrupt Request) aus - Hauptprogramm wird unterbrochen.
  - Aktuelle Instruktion des Hauptprogramms wird noch vollständig abgearbeitet.
  - Zustand der CPU (Kontext) wird gerettet.
  - Abarbeitung der Interrupt Service Routine (ISR).
  - Alter Zustand der CPU wird wieder hergestellt.
  - Fortsetzung des Hauptprogramms an der unterbrochenen Stelle.

# Interruptverarbeitung

ISR Ausführung ist transparent für das Hauptprogramm  
(bis auf gewollte Effekte und Laufzeit der ISR).

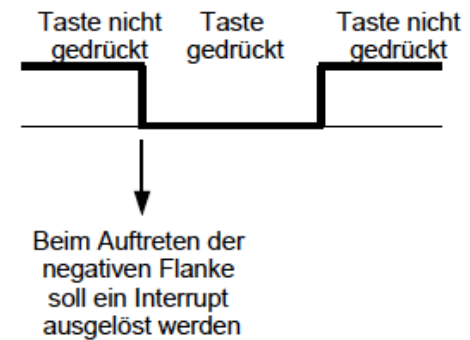
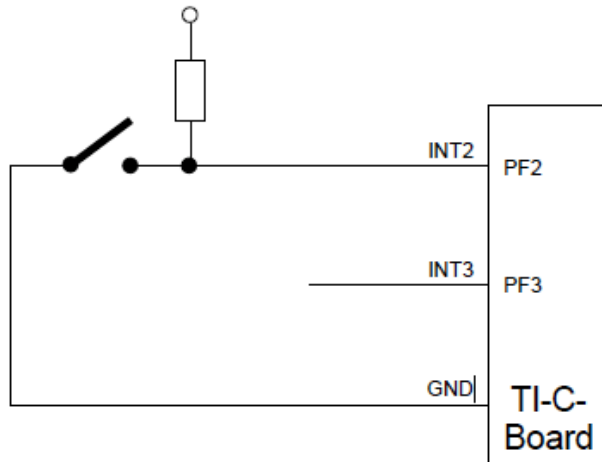


Daher

- ISR Ausführung darf den Zustand der CPU (Kontext) nicht verändern!
  - Alle von der ISR verwendeten Register (z.B.: Statusregister, Datenregister) müssen zuvor gerettet und hinterher wieder hergestellt werden.
  - „Retten“ erfolgt z.B. auf dem Stack oder in zusätzlichen Registersätzen

## Interruptverarbeitung: Prinzipielle Vorgehensweise

Aufgabe: Tastenbetätigungen zählen



- Jeder Tastendruck soll gezählt werden.
- Interruptverarbeitung soll auf Flanken reagieren.
- IRQ bewirkt, dass mit jedem Tastendruck die ISR aufgerufen wird.
- In der ISR wird eine globale Variable counter hochgezählt.
- Das Hauptprogramm gibt den Wert von counter regelmäßig aus.

## Interruptverarbeitung: Prinzipielle Vorgehensweise

### ISR:

```
volatile int counter; // volatile verhindert Fehler, die durch
                        // Optimierungen entstehen koennen
void little_isr(void) {
    EXTI->PR = (1<<2); /* Interrupt bestaetigen:
                        -> Device deaktiviert IRQ-Leitung */

    counter++;
}
```

### Hauptprogramm:

```
int main( void ){
    ... // Interruptverarbeitung initialisieren
    while (1) {
        printf( "Zaehler ist: %d\n", counter );
        delay( 1000 ); // 1 sec
    }
    return 0;
}
```

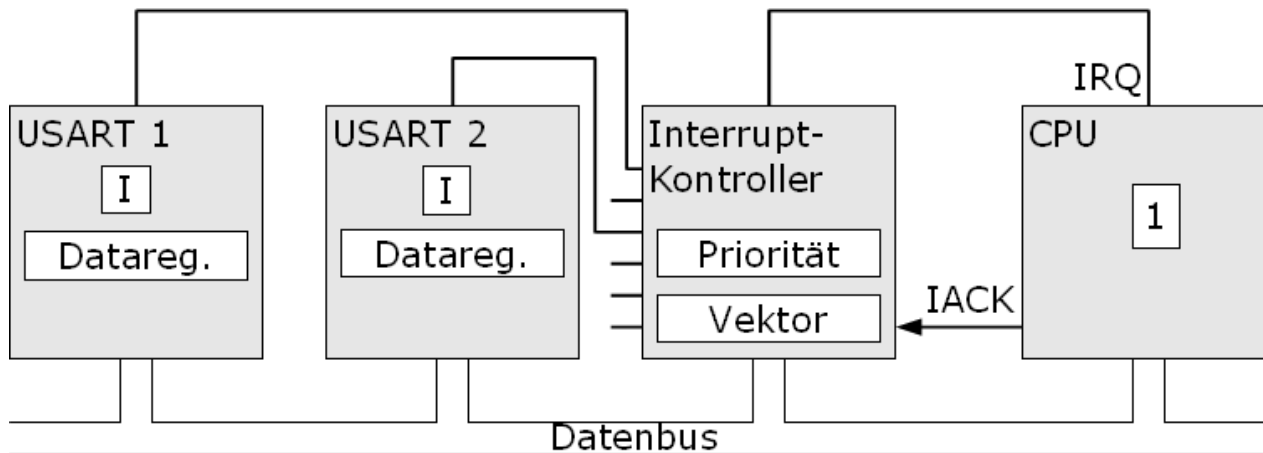


## ISR in C

- Nicht im C-Standard vorgesehen!
- Implementierung abhängig von
  - CPU-Hersteller
  - Compiler-Hersteller
- Compiler
  - rettet alle notwendigen Register und
  - restauriert sie am Ende der ISR.
  - Return am Ende der ISR wird durch „Return from Interrupt“ ersetzt

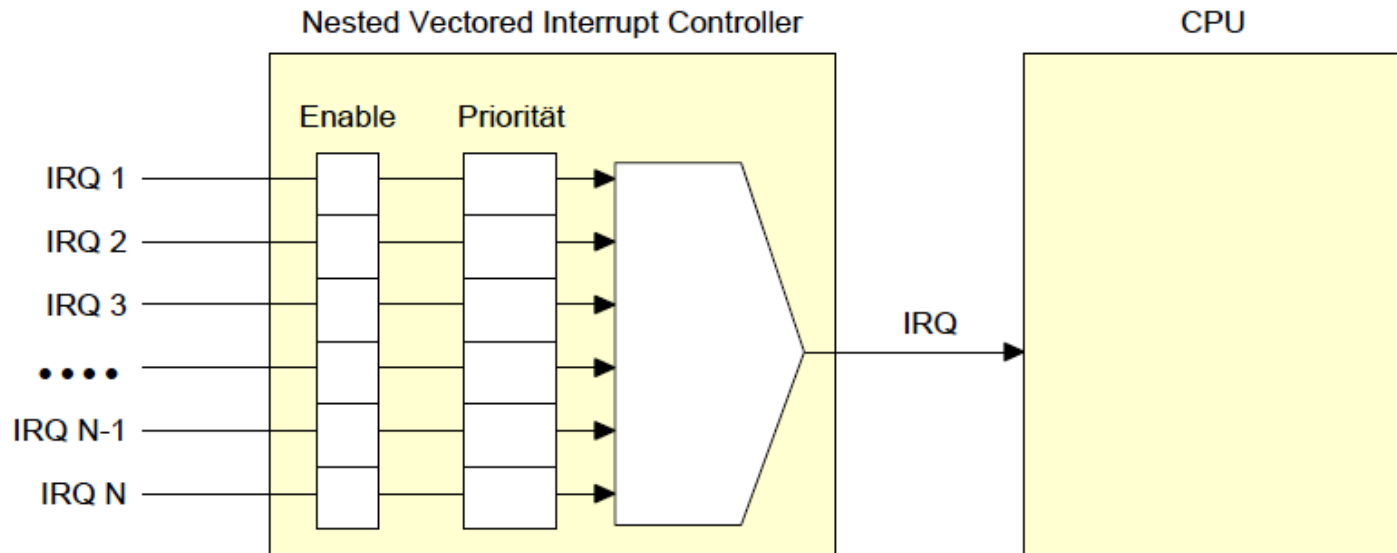
## Mehrere Interrupt-Quellen

### Interrupt-Kontroller:



- Steuert die Abarbeitung der IRQs:
  - Legt die Reihenfolge fest: Vergabe von Prioritäten.
  - Teilt der CPU mit, welche ISR auszuführen ist: Verwendung von Vektornummern oder ISR-Startadressen.
  - Ermöglicht verschachtelte Interrupts: Höher priorisierte Interrupts können laufende ISR unterbrechen.

## Nested Vectored Interrupt Controller (NVIC)



- Bis zu 240 Interrupts
- Bis zu 256 Prioritätsstufen
- Einfache Realisierung von verschachtelten (nested) Interrupts.
- Ermöglicht besonders niedrige Latenzzeiten.

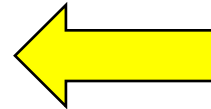
## Interruptquellen des STM32F4xx

SysTick	20	CAN1_RX0	41	RTC_Alarm	62	ETH_WKUP
10 WWDG	21	CAN1_RX1	42	OTG_FS WKUP	63	CAN2_TX
11 PVD	22	CAN1_SCE	43	TIM8_BRK_TIM12	64	CAN2_RX0
12 TAMP_STAMP	23	EXTI9_5	44	TIM8_UP_TIM13	65	CAN2_RX1
13 RTC_WKUP	24	TIM1_BRK_TIM9	45	TIM8_TRG_COM_TIM	66	CAN2_SCE
14 FLASH	25	TIM1_UP_TIM10	46	TIM8_CC	67	OTG_FS
15 RCC	26	TIM1_TRG_COM_TIM	47	DMA1_Stream7	68	DMA2_Stream5
16 EXTI0	27	TIM1_CC	48	FSMC	69	DMA2_Stream6
17 EXTI1	28	TIM2	49	SDIO	70	DMA2_Stream7
18 EXTI2	29	TIM3	50	TIM5	71	USART6
19 EXTI3	30	TIM4	51	SPI3	72	I2C3_EV
20 EXTI4	31	I2C1_EV	52	UART4	73	I2C3_ER
21 DMA1_Stream0	32	I2C1_ER	53	UART5	74	OTG_HS_EP1_OUT
22 DMA1_Stream1	33	I2C2_EV	54	TIM6_DAC	75	OTG_HS_EP1_IN
23 DMA1_Stream2	34	I2C2_ER	55	TIM7	76	OTG_HS_WKUP
24 DMA1_Stream3	35	SPI1	56	DMA2_Stream0	77	OTG_HS
25 DMA1_Stream4	36	SPI2	57	DMA2_Stream1	78	DCMI
26 DMA1_Stream5	37	USART1	58	DMA2_Stream2	79	CRYP
27 DMA1_Stream6	38	USART2	59	DMA2_Stream3	80	HASH_RNG
28 ADC	39	USART3	60	DMA2_Stream4	81	FPU
29 CAN1_TX	40	EXTI15_10	61	ETH		

## Kapitel 6: Interrupt Behandlung

### Gliederung

- Einführung
- Interrupt Behandlung
- Schritte der Interrupt Behandlung
- Zusammenfassung



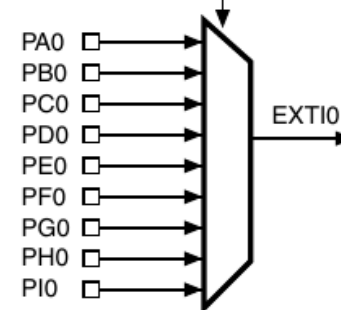
# Implementierung eines Interrupts

## Schritt 1: Routing des Interrupts

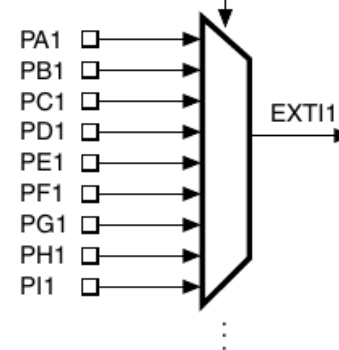
### (Teil 1 der Initialisierung)

- Verbindung der ausgehenden Interrupt Leitung eines Devices mit einem Eingang des Interrupt Controllers
- Hier: Binde GPIO an Interrupt Quelle
  - Jeder GPIO-Pin kann IRQ auslösen
  - 16 Interrupts stehen zur Verfügung (EXTI0 – EXTI15)
  - Über Multiplexer wird von Ports A bis Port I Pin i auf EXTIi geroutet.

EXTI0[3:0] bits in the SYSCFG\_EXTICR1 register



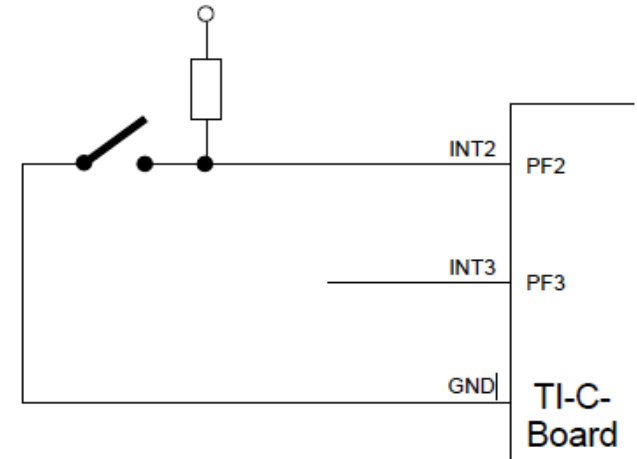
EXTI1[3:0] bits in the SYSCFG\_EXTICR1 register



## Beispiel

- Interrupt soll bei jeder steigenden und fallenden Flanke ausgelöst werden
- Die ISR soll die globale Variable counter um 1 erhöhen.

**Schritt 1:** Routing des Interrupt und Aktivierung des Clock Systems für Port F



```
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOFEN; //Clock for GPIO Port F
RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN; //Digital Interface clock
// Routing Pin 2 of Port F -> EXTI2
SYSCFG->EXTICR[0] &= ~(0x0f << (4*2)); //Remove old selection
SYSCFG->EXTICR[0] |= 0x05 << (4*2); //0x05 : Select port F
```

## Implementierung eines Interrupts

### Schritt 2: Definiere IRQ Event und Unmask IRQ

#### (Teil 2 der Initialisierung)

- Definiere Ereignisse auf der INT Leitung, die einen Interrupt auslösen
- Hier: Flanken-getriggert.
  - Auswahl: positive Flanke
  - negative Flanke
  - beide Flanken
- Maskieren eines IRQ über ein Flag in Interrupt Controller:

Der entsprechende Interrupt wird nicht an die CPU weitergeleitet und somit nicht beachtet.

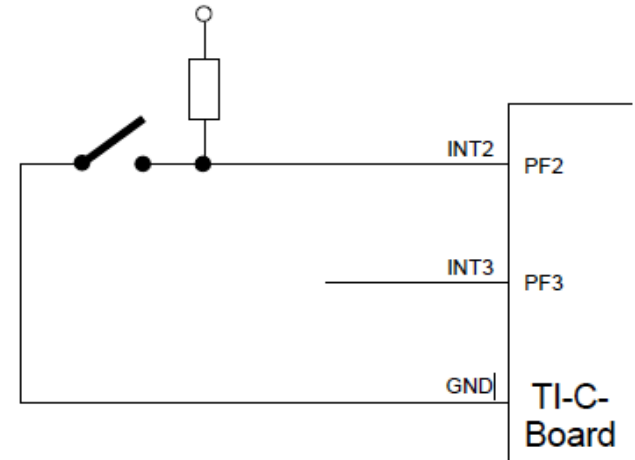
Bei der Aktivierung muss der IRQ unmaskiert werden.



## Beispiel

- Interrupt soll bei jeder steigenden und fallenden Flanke ausgelöst werden
- Die ISR soll die globale Variable counter um 1 erhöhen..

**Schritt 2:** Definiere Events für INT2 und unmask IRQ2



```
EXTI->RTSR |= (1<<2); //select rising trigger for INT2
```

```
EXTI->FTSR |= (1<<2); //select falling trigger for INT2
```

```
EXTI->IMR |= (1<<2); // Unmask INT2
```

# Implementierung eines Interrupts

## Schritt 3: Einstellung des Interrupt Controllers

### (Teil 3 der Initialisierung)

- Priorität des IRQ einstellen:

  - HIER: priority: Definiert Verschachtelung beim Aufruf von IRQs

  - subpriority: Bearbeitungsreihenfolgen von IRQs gleicher Priorität

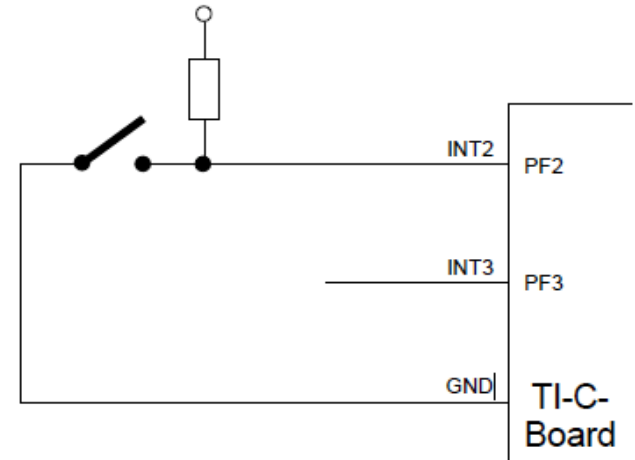
- Interrupt aktivieren

## Beispiel

- Interrupt soll bei jeder steigenden und fallenden Flanke ausgelöst werden
- Die ISR soll die globale Variable counter um 1 erhöhen.

### Schritt 3: Einstellung des Interrupt Controllers

```
NVIC_SetPriorityGrouping(2);    // Setup interrupt controller:
                                // 4 subpriority for each priority
NVIC_SetPriority(EXTI2_IRQn, 8); // Setup EXTI2:
                                // priority = 2, subpriority = 0
NVIC_EnableIRQ(EXTI2_IRQn);    // Enable EXTI2
```



## Implementierung eines Interrupts

### Schritt 4: Programmierung der ISR

- Eine ISR ist eine normale C-Funktion, mit folgenden Randbedingungen:
  - Keine Parameter
  - Kein Rückgabewert
  - Synchronisierung mit dem Hauptprogramm beachten, da stets ein Interrupt auftreten und somit die ISR gestartet werden kann.
  - Verwendete Funktionen müssen reentrant sein (s.u.).
  - ISR hängt von System und Compiler ab.

Teilweise wird die ISR durch ein Attribut gekennzeichnet (Keil nicht)

z.B. `void f () __attribute__ ((interrupt ("IRQ")));`
- ISR setzt den Interrupt direkt am Anfang zurück, damit weitere Interrupts beachtet werden können.

## Implementierung eines Interrupts

### Schritt 4: Programmierung der ISR (Fortsetzung)

- Eintrag der ISR in die Vektortabelle
  - Vektortabelle weist jedem Interrupt eine ISR zu.  
Sie ist im Startcode enthalten.
  - Die Vektortabelle ist mit Standard-ISRs gefüllt.
- Keil Entwicklungsumgebung
  - ISR wird auf Basis eines eindeutig festgelegten Namens implizit in die Vektortabelle eingetragen -> **Eintrag muss nicht explizit programmiert werden!**
  - Wie: Linker überschreibt den Default Eintrag (Weak Symbol) mit der ISR.
  - **Achtung:** Signatur wird nicht überprüft

# Implementierung eines Interrupts

## Schritt 4: Programmierung der ISR (Fortsetzung)

- Kommunikation zwischen ISR und Hauptprogramm über globale Variablen die **volatile** sind.
- **volatile** kennzeichnet Variablen, deren Wert sich außerhalb des aktuellen Programmpfades ändern kann, z.B.:
  - in einer ISR
  - Hardware-Register, z.B. Timer
- Die Kennzeichnung **volatile** verhindert Optimierungen, die davon ausgehen, dass eine Variable sich nur im Programmpfad ändert (z.B.: Laden der Variablen im Cache, Entfernung aus einer Schleife)

```
z.B.:    volatile int counter;
         void EXTI2_IRQHandler (void) {
             ...
             counter++;
             ...
         }
```

## Implementierung eines Interrupts

### Schritt 4: Programmierung der ISR (Fortsetzung)

- Funktionen, die vom Hauptprogramm und einer ISR (oder mehreren ISRs) aufgerufen werden, müssen wiedereintrittsfähig (**reentrant**) sein.
- Eine Funktion heißt **reentrant (wiedereintrittsfähig)**, wenn sie während ihrer Ausführung unterbrechbar ist, erneut in einem anderen Kontext (mehrmals) aufrufbar ist und schließlich fehlerfrei beendet wird (das Ergebnis entspricht der Ausführung der Funktion ohne Unterbrechung).
- Zum Beispiel sind printf und malloc nicht reentrant.

## Implementierung eines Interrupts

### Schritt 4: Programmierung der ISR (Fortsetzung)

```
char buf[100];
char* textUmkehrNotReentrant(char* s){
    int i, j;
    for (i=0, j=strlen(str)-1; str[i]!='\0'; i++, j--){buf[j] = s[i];}
    return buf;
}
volatile char* rev;
void isr(void) {
    ...
    rev = textUmkehrNotReentrant("DA");
    ...
}

// Hauptprogramm
printf("Reverse Text ist: %s\n", textUmkehrNotReentrant("SO"));
```

**Fehler**



## Implementierung eines Interrupts

### Schritt 4: Programmierung der ISR (Fortsetzung)

```
void textUmkehrReentrant( char* s, char* erg){
    int i, j;
    for (i=0,j=strlen(str)-1; str[i]!='\0'; i++,j--){erg[j] = s[i];}
}
volatile char rev[80]; //warum darf dies keine lokale Variable sein?
void ISR() {
    ...
    textUmkehrReentrant("DADA", rev);
    ...
}
// Hauptprogramm
char buf[80];
textUmkehrReentrant( "SOSO", buf );
printf( "Reverse Text ist: %s\n", buf );
...
```

## Implementierung eines Interrupts

### Schritt 4: Programmierung der ISR (Fortsetzung)

- Programmierung, die in der Regel die Reentrant Eigenschaft zerstört:
  - Schreibzugriffe auf statische oder globale Variable.
  - Rückgabe von Adressen auf statische oder globale Variable.
  - Aufrufe von non-reentrant Funktionen
- Daher
  - Bibliotheksfunktionen sollten mit lokalen Variablen oder Daten, die vom Aufrufer zur Verfügung gestellt werden, arbeiten.

## Implementierung eines Interrupts

### Schritt 4: Programmierung der ISR (Fortsetzung)

Synchronisation zwischen ISR und Hauptprogramm

Die Kommunikation findet in der Regel über globale Variablen, die **volatile** sind, statt.

- Problem „gleichzeitige“ Schreibzugriffe zwischen ISR und Hauptprogramm
- Beispiel:
  - Das Hauptprogramm braucht mehrere Maschinenbefehle um die gemeinsame Variable zu überprüfen und sie ggf. anschließend zu verändern.
  - Diese Befehlssequenz wird von einer ISR unterbrochen.
  - Dies führt oftmals zu inkonsistenten Daten.

## Implementierung eines Interrupts

### Schritt 4: Programmierung der ISR (Fortsetzung)

Beispiel:

```
volatile int counter;  
void ISR(void) {  
    ...  
    counter++;  
    ...  
}  
// Hauptprogramm  
...  
while( 1 ){  
    if( counter >= COUNTERMAX ){  
        <mach etwas>  
        counter = 0;  
    }  
}  
...
```

**ISR unterbricht diese  
Befehlssequenz:  
Fehlerquelle: „Verliere  
Werte von counter“**

## Implementierung eines Interrupts

### Schritt 4: Programmierung der ISR (Fortsetzung)

Synchronisation durch **atomare Befehle**, die in einer Maschineninstruktion ausgeführt werden.

```
//atomar:  
counter = 55; // atomar, wenn int und 32-Bit CPUs  
localvalue = counter;  
  
//nicht atomar:  
counter++; //Bei RISC_CPUs: Read-Modify-Write Zyklus  
counter |= (1<<bitnr);
```

## Implementierung eines Interrupts

### Schritt 4: Programmierung der ISR (Fortsetzung)

Synchronisation durch Deaktivierung der Interruptverarbeitung

- Für die Dauer des Zugriffs wird die Interruptverarbeitung deaktiviert:
- Nur notwendig bei Hauptschleife.

ISR ist in der Regel nicht unterbrechbar - bei entsprechend festgelegten Prioritäten.

- Ohne Interrupts bekommt die CPU nichts von der Umwelt mit -> Deaktivierung bedeutet Verlängerung der Latenz -> Ausschaltdauer möglichst kurz halten!

```
#include <armVIC.h>
```

```
...
```

```
int oldirq = disableIRQ();  
if(counter >= COUNTERMAX ){  
    <mach etwas>  
    counter = 0;  
}  
restoreIRQ(oldirq);
```

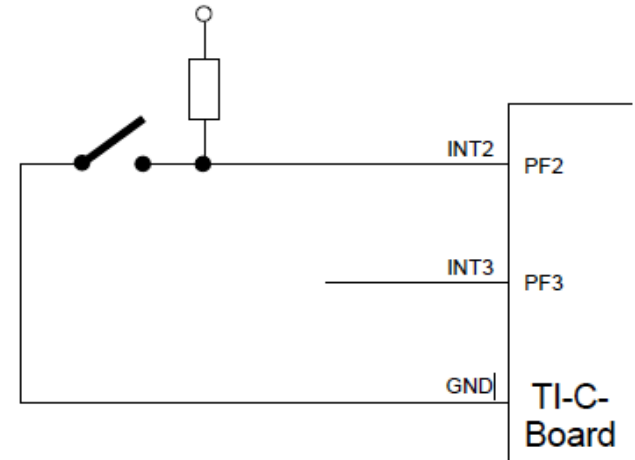
```
...
```

## Beispiel

- Interrupt soll bei jeder steigenden und fallenden Flanke ausgelöst werden
- Die ISR soll die globale Variable counter um 1 erhöhen.

### Schritt 4: Programmierung der ISR

```
volatile unsigned int counter= 0;  
  
void EXTI2_IRQHandler(void){  
    // name according to startup_stm32f4xxx.s  
    EXTI->PR = (1<<2); // Reset INT2  
    counter++;  
}
```



## Beispiel Drehgeber (Aufgabe 5)

### Initialisierung Routing

```
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOFEN; //Clock for GPIO Port F
RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN; //Clock for Syscfg

// Connect EXTI2 with Pin 2 of GPIO F (MASK0x5)
SYSCFG->EXTICR[0] &= ~(0xf << (4*2)); //Remove old selection
SYSCFG->EXTICR[0] |= 0x5 << (4*2); //Select Port F
EXTI->RTSR |= (1<<2); //select rising trigger for INT2
EXTI->FTSR |= (1<<2); //select falling trigger for INT2
EXTI->IMR |= (1<<2); // Unmask Int2

// Connect EXTI3 with Pin 3 of GPIO F (MASK0x5)
SYSCFG->EXTICR[0] &= ~(0xf << (4*3)); //Remove old selection
SYSCFG->EXTICR[0] |= 0x5 << (4*3); //Select Port F
EXTI->RTSR |= (1<<3); //select rising trigger for INT3
EXTI->FTSR |= (1<<3); //select falling trigger for INT3
EXTI->IMR |= (1<<3); // Unmask INT3
```



## Beispiel Drehgeber (Aufgabe 5) Fortsetzung

### Initialisierung Interrupt Controller

```
EXTI->IMR |= (1<<3); // Unmask INT3
```

```
NVIC_SetPriorityGrouping(2); // Setup priorities and subpriorities
```

```
NVIC_SetPriority(EXTI2_IRQn, 8); // INT2: Set Group Prio = 2, Subprio = 0
```

```
NVIC_EnableIRQ(EXTI2_IRQn); // Enable IRQ
```

```
NVIC_SetPriority(EXTI3_IRQn, 8); // INT3: Set Group Prio = 2, Subprio = 0
```

```
NVIC_EnableIRQ(EXTI3_IRQn); // Enable IRQ
```

## Beispiel Drehgeber (Aufgabe 5) Fortsetzung

### ISRs

```
void EXTI2_IRQHandler(void){  
    EXTI->PR = (1<<2); // Reset INT2  
    // Update fsm of rotary pulse generator  
    ...  
}
```

```
void EXTI3_IRQHandler(void){  
    EXTI->PR = (1<<3); // Reset INT3  
    // Update fsm of rotary pulse generator  
    ...  
}
```

## Zusammenfassung

- Einsatz von ISR bei System mit schnellen Reaktionszeiten – kurzen Latenzen
- Latenz: Zeit, die zwischen dem Auftreten eines Ereignisses und bis zur Bearbeitung des Ereignisses vergeht.
- Die Umgebung (controlled object) definiert die Latenzen.  
Beispiel: Drehgeber.
- Bei nicht unterbrechbaren ISRs ergibt sich die Latenz des Systems aus der Bearbeitungsdauer der langsamsten ISR (bei entsprechender Priorität).
- ISRs müssen möglichst schnell abgearbeitet werden.
- In ISRs sollte nicht gewartet werden, z.B.:
  - Warten auf Timer.
  - Warten auf Fertig-Bit einer Hardware-Komponente.
  - Eingabeaufforderung an Benutzer.