Mikroprozessortechnik

Prof. Dr. Michael Lipp

fbeit
FACHBEREICH ELEKTROTECHNIK
UND INFORMATIONSTECHNIK

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Super Loop (Aktuelle Anwendungs-architektur)

Super Loop Architektur

- **Bislang: Reine Super-Loop Architektur**
  - Initialisierung
  - Endlosschleife
    - Abfragen, ob Bedingung 1 eingetreten ist
    - Wenn ja, bearbeiten
    - Abfragen, ob Bedingung 2 eingetreten ist
    - Wenn ja, bearbeiten
    - Abfragen, ob Bedingung 3 eingetreten ist
    - Wenn ja, bearbeiten
    - ...

```
// Labor 1:

void task7() {
    CPolledTimer twoHz(500);
    CPolledTimer threeHz(333);
    CPolledTimer fourHz(250);
    while (true) {
        if (twoHz.timeReached()) {
            leds = leds ^ (1 << 0);
        }
        if (threeHz.timeReached()) {
            leds = leds ^ (1 << 1);
        }
        if (fourHz.timeReached()) {
            leds = leds ^ (1 << 2);
        }
    }
}
```

Super Loop Architektur

- **Nachteile**
  - Offensichtlich
    - Bearbeitung einer eingetretenen Bedingung blockiert den Prozessor
      - → Behandlung von wichtigen Ereignissen kann „beliebig“ verzögert werden
    - Komponenten können nicht vollständig gekapselt werden
      - Beispiel 7-Segment-Anzeige: `CPolledTimer` für die Anzeige in `main` definiert
  - Weniger offensichtlich
    - …
- **Verbesserung für zeitnahe Bearbeitung von (wichtigen) Ereignissen → Interrupts**

Interrupts

Interrupts

- **Ein Interrupt ist eine Unterbrechung des „regulären“ Programmablaufs**
- **Ein Interrupt wird durch einen Zustand oder Zustands-wechsel einer Peripherieeinheit ausgelöst („Ereignis“)**
  - Beispiel: Grenzwertüberschreitung des ADC
- **Der Interrupt führt zur sofortigen (genauer: möglichst zeitnahen) Ausführung von Code**
  - Dieser Code (Interrupt Handler) behandelt den Zustand oder Zustands-wechsel und setzt ihn zurück (oder deaktiviert die erneute Auslösung des Interrupts)

Interrupts

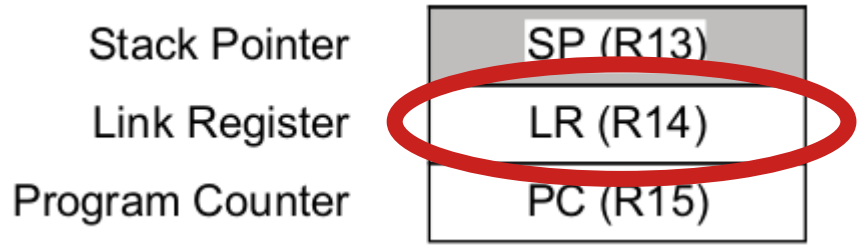
- **Aufruf des Interrupt Handler ist einem „erzwungenen“ Unterprogrammaufruf vergleichbar**

  … Unterprogrammaufruf … ???

Unterprogramm-aufrufe (Wdh./Vertiefung)

Wdh: Unterprogrammaufruf

- **Unterprogrammaufruf**
  - Beispiel:




```
void main(void){
    int i,k;
    i = sub(13);
    k = sub(14);
     // ...
    }
```

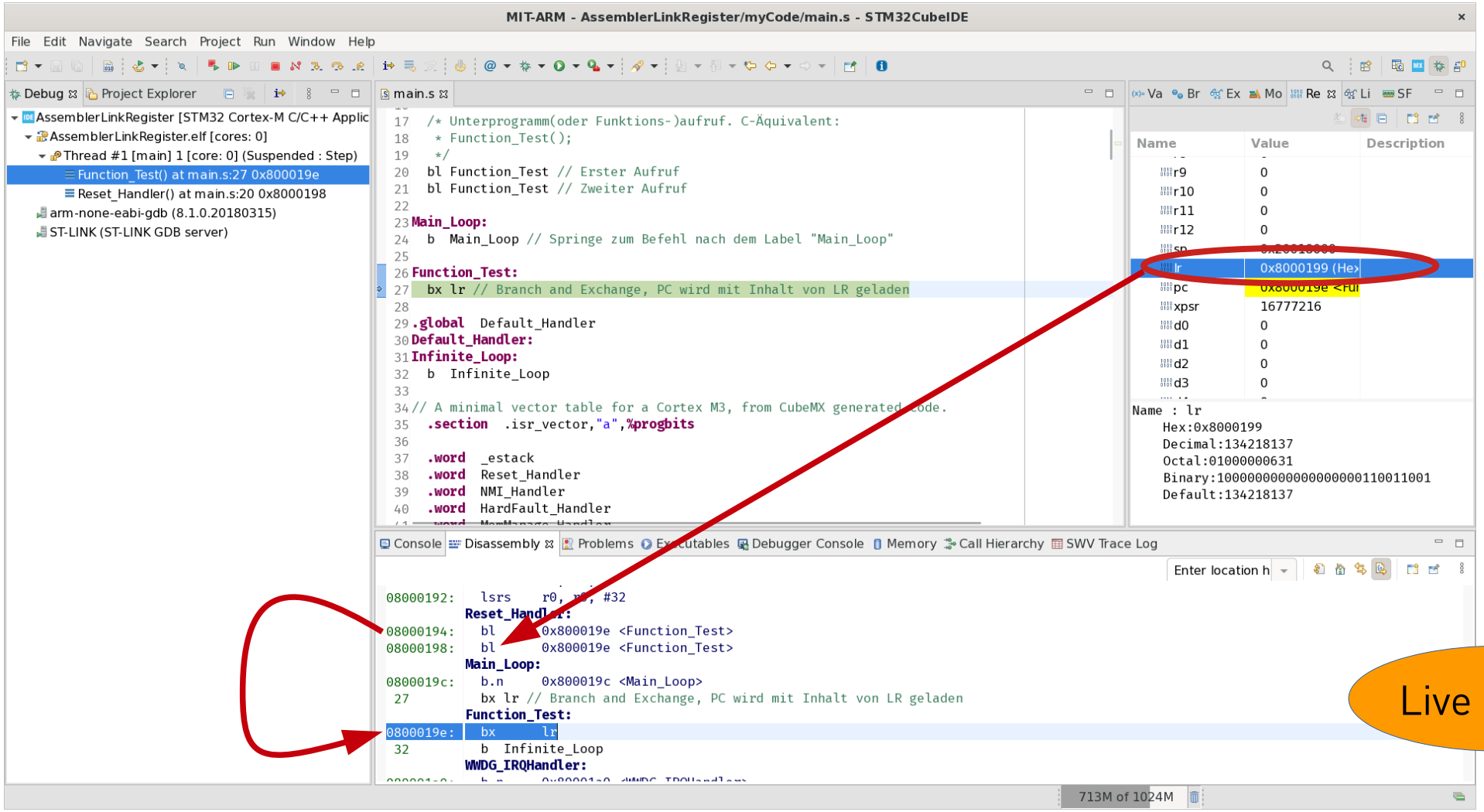
```
int sub(int i){
    return i*2;
}
```

  - Spring aus main nach `sub` (PC ← sub)
  - Welchen Wert lädt `return` in den PC?

Wdh.: Unterprogrammaufruf

- **Lösung**
  - Sprung zu `sub` mit „branch with link“ („bl“)
    - Speichert die Adresse des nächsten Befehls (nach „bl“) im Link-Register
    - Lädt PC mit Adresse des Sprungziels
  - Rücksprung
    - Lade den Wert von `lr` in `pc` („bx `lr`“)

Wdh.: Unterprogrammaufruf



Live Coding

Registerbehandlung

- **ARM Calling Convention**
  - Offiziell: AAPCS (ARM Architecture Procedure Call Standard)
  - R0-R3: Argumente / Rückgabewert(e), können von der Subroutine überschrieben werden
  - R4-R11: Lokale Variablen, dürfen von der Subroutine **nicht** überschrieben werden, d.h. wenn benötigt, Werte auf dem Stack sichern und vor Rücksprung wiederherstellen
  - R12: Hilfsregister für Aufruf der Subroutine, Wert muss nicht erhalten werden

Beispielfunktion

- **C-Code:**

```
int32_t add(int32_t a, int32_t b) {
    return a + b;
}
```

- **Assembler-Code:**

```
add  r0, r1
bx   lr
```

Für diese einfache Funktion reichen die verfügbaren Register aus.

Beispielfunktion

- **C-Code:**

```
int32_t calculate(int32_t a, int32_t b, int32_t c, int32_t d) {
    return add(a, b) * add(c, d);
}
```

- **Assembler-Code:**

```
// Es werden 3 zusätzliche Register benötigt …
push {r4, r5, r6, lr}  // … und die return-Adresse muss gespeichert werden.
mov  r5, r2    // r2 in r5 speichern …
mov  r6, r3    // … und r6 in r3 …
bl   add       // … da der Unterprogrammaufruf r2 und r3 ändern darf.
mov  r4, r0    // Zwischenergebnis in r4 speichern.
mov  r1, r6    // In r5 und r6 gespeicherte Argumente in r1 …
mov  r0, r5    // … und in r0 übertragen
bl   add
muls r0, r4    // Zwischenergebnisse multiplizieren, Endergebnis in r0
pop  {r4, r5, r6, pc}  // „Geschicktes“ return, lr → pc
```

(Zurück zum Interrupt)

Interrupt

- **Aufruf Interrupt Handler (Reaktion auf eingetretene Bedingung)**
  - Zustandsregister, PC, LR, R12 und R3-R0 auf dem Stack speichern (dürfen von „normalem“ Unterprogramm verändert werden)
  - LR wird mit „Spezialwert“ geladen, so dass beim „Return“ (PC ← LR) auf dem Stack gespeicherten Informationen wiederhergestellt werden
  - Register, die von einem „normalen“ Unterprogramm nicht verändert werden dürfen, dürfen auch von Funktionen zur Interrupt-Behandlung nicht verändert werden (müssen auf dem Stack „gerettet“ werden)
    - D. h. der Code für Funktionen für die Interrupt-Behandlung unterscheidet sich nicht von dem Code „normaler“ Funktionen
  - Funktionen für die Interrupt-Behandlung haben keine Parameter
    - (Woher sollten auch die Argumente kommen?)

Interrupt

- **Interrupt Handler**
  - Unterschiedliche Interrupt Handler für unterschiedliche Hardware-Komponenten
    - Sonst müsste man im Handler erst alle möglichen Auslöser prüfen
  - Zuordnung „Ereignis“ zu Interrupt Handler über den Vector Table
    - „Fortsetzung“ der beim Booten verwendeten Werte an den Adressen 0x0 und 0x4
  - Vector Table enthält die Start-Adressen der Interrupt Handler

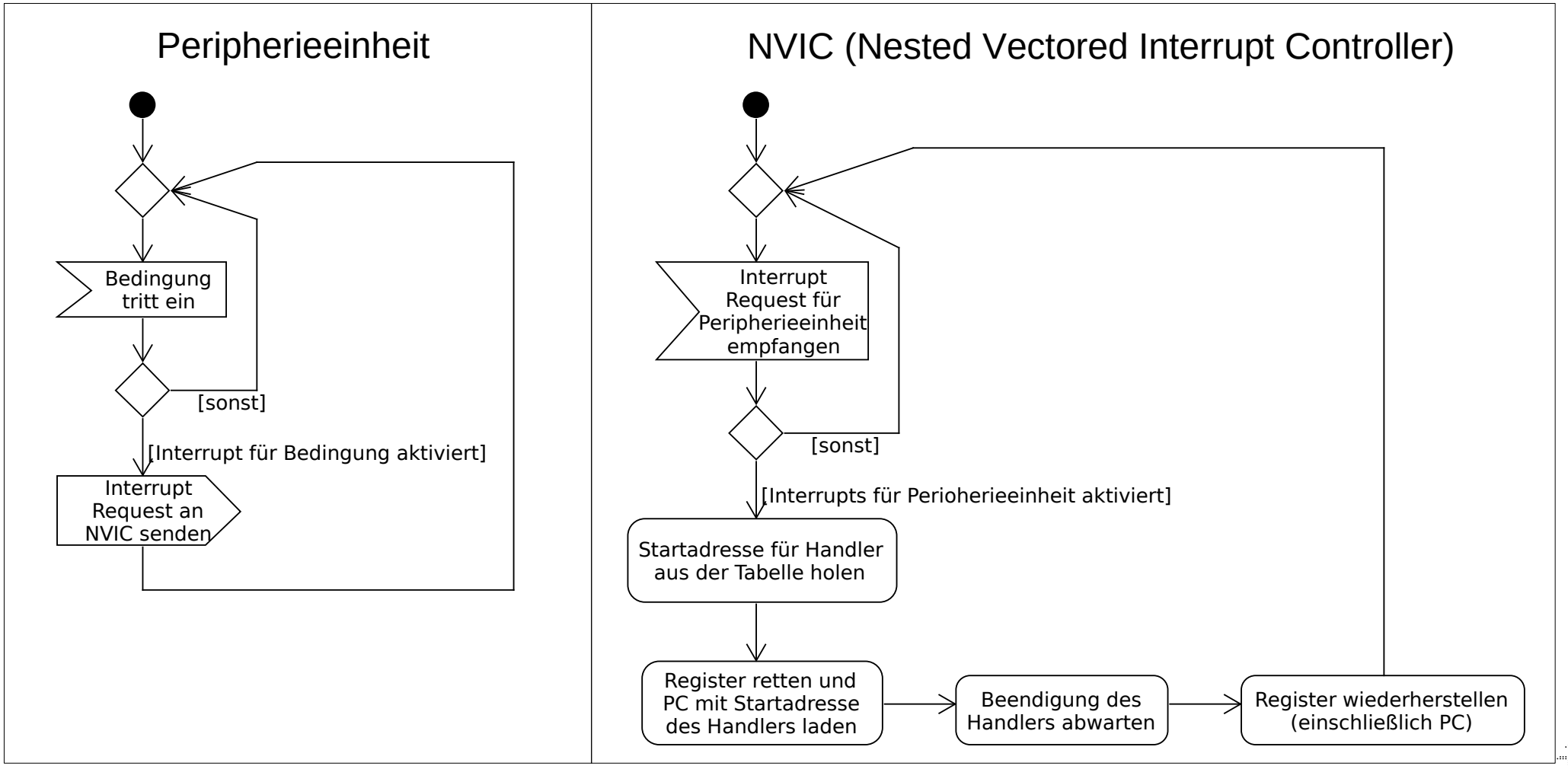
Vector Table

Table 38. Vector table for STM32F401xB/CSTM32F401xD/E

Position	Priority	Type of priority	Acronym	Description	Address
...					
0	7	settable	WWDG	Window Watchdog interrupt	0x0000 0040
1	8	settable	EXTI16 / PVD	EXTI Line 16 interrupt / PVD through EXTI line detection interrupt	0x0000 0044
2	9	settable	EXTI21 / TAMP_STAMP	EXTI Line 21 interrupt / Tamper and TimeStamp interrupts through the EXTI line	0x0000 0048
3	10	settable	EXTI22 / RTC_WKUP	EXTI Line 22 interrupt / RTC Wakeup interrupt through the EXTI line	0x0000 004C
4	11	settable	FLASH	Flash global interrupt	0x0000 0050
5	12	settable	RCC	RCC global interrupt	0x0000 0054
6	13	settable	EXTI0	EXTI Line0 interrupt	0x0000 0058

Quelle:
[F401-RM]

Gesamtablauf (Verhalten der Hardware)



Beispiel ADC-Interrupt

- **Mbed OS unterstützt keine ADC Interrupts**
  - ADC muss manuell konfiguriert werden
  - Minimalkonfiguration
    - Beschreibungen im Reference Manual bzw. Datenblatt lesen
    - GPIO-Pin als Analogeingang konfigurieren (vergl. Vorlesung 2)
    - ADC mit Takt versorgen
    - ADC einschalten
    - Zu konvertierenden Kanal festlegen
- **(Takt für Peripherieeinheiten)**
  - Versorgung der Peripherieeinheiten mit Takt wird später im Detail behandelt, im Moment Beispielvorgehen verwenden

ADC-Konfigurationstest mit Polling

```
int main() {
    keys.mode(PullDown);

    // Eingang PB0 als "Analog" konfigurieren (vergl. Vorlesung 2).
    MODIFY_REG(GPIOB->MODER, 0, GPIO_MODER_MODER0);
    MODIFY_REG(GPIOB->OTYPER, GPIO_OTYPER_OT0, 0);
    MODIFY_REG(GPIOB->OSPEEDR, GPIO_OSPEEDER_OSPEEDR0, 0);
    MODIFY_REG(GPIOB->PUPDR, GPIO_PUPDR_PUPD0, 0);

    // Sicherstellen, dass ADC mit Takt versorgt wird und ADC einschalten
    RCC->APB2ENR |=  RCC_APB2ENR_ADC1EN;
    ADC1->CR2 |= ADC_CR2_ADON;

    // PB0 ist mit ADC1_IN8 verbunden ([F401-DS] Table 8). Länge der Sequenz
    // der zu konvertierenden Eingänge auf 1 setzen und ADC1_IN8 als ersten
    // (und einzigen) abzufragenden Kanal eintragen (s. F401-RM 11.3.3
    // und 11.12.9).
    MODIFY_REG(ADC1->SQR1, ADC_SQR1_L, (1 << ADC_SQR1_L_Pos));
    MODIFY_REG(ADC1->SQR3, ADC_SQR3_SQ1, (8 << ADC_SQR3_SQ1_Pos));

    // Prescaler (bestimmt Takt, s. Vorlesung)
    MODIFY_REG(ADC1_COMMON->CCR, 0, (1 << ADC_CCR_ADCPRE_Pos));

    task1();
}
```

```
/**
 * Beispiel für die Nutzung des direkt programmierten ADC im
 * "polling" (Abfrage-)Modus.
 */
void task1() {
    while (true) {
        // Polling: Konvertierung starten und auf Ende warten
        ADC1->CR2 |= ADC_CR2_SWSTART;
        while (!(ADC1->SR & ADC_SR_EOC)) {
        }

        // Ergebnis anzeigen
        leds = ADC1->DR >> 4;
    }
}
```

Live Coding

Beispiel ADC Interrupt

- **Aktivierbare ADC-Interrupts**

11.12.2 ADC control register 1 (ADC_CR1)

Address offset: 0x04

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20
Reserved					OVRIE	RES		AWDEN	JAWDEN		
					rw	rw	rw	rw	rw		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DISCNUM[2:0]			JDISCEN	DISCEN	JAUTO	AWDSGL	SCAN	JEOCIE	AWDIE	EOCIE	AWDCH[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 26 **OVRIE:** Overrun interrupt enable
This bit is set and cleared by software to enable/disable the Overrun interrupt.
0: Overrun interrupt disabled
1: Overrun interrupt enabled. An interrupt is generated when the OVR bit is set.

Bit 7 **JEOCIE:** Interrupt enable for injected channels
This bit is set and cleared by software to enable/disable the end of conversion interrupt for injected channels.
0: JEOC interrupt disabled
1: JEOC interrupt enabled. An interrupt is generated when the JEOC bit is set.

Bit 6 **AWDIE:** Analog watchdog interrupt enable
This bit is set and cleared by software to enable/disable the analog watchdog interrupt.
0: Analog watchdog interrupt disabled
1: Analog watchdog interrupt enabled

Bit 5 **EOCIE:** Interrupt enable for EOC
This bit is set and cleared by software to enable/disable the end of conversion interrupt.
0: EOC interrupt disabled
1: EOC interrupt enabled. An interrupt is generated when the EOC bit is set.

Beispiel ADC Interrupt

Table 38. Vector table for STM32F401xB/CSTM32F401xD/E

Position	Priority	Type of priority	Acronym	Description	Address
				...	
18	25	settable	ADC	ADC1 global interrupts	0x0000 0088

- **„Anmelden“ eines Interrupt Handlers**
  - Vector Table enthält Adressen von „Default Handlern“
  - Werden durch Definition einer C-Funktion mit gleichem Namen ersetzt
  - Standard-Eintrag bei `0x00000088`: `ADC_IRQHandler`

Einschub: C- und C++ Funktionen

- **C- und C++-Funktionen sehen gleich aus**
  - C-Funktion: `void f();`
  - C++-Funktion: `void f();`
- **Tatsächlich befinden sich die Funktionen aber in unterschiedlichen „Namensräumen“**
  - Eine in C++ Quelltext definierte Funktion `void f()` kann aus C nicht aufgerufen werden, außer sie ist definiert (oder deklariert) als

    ```
    extern "C" void f() {…}
    ```

    d. h. „extern“ wird sie wie eine C-Funktion behandelt

Beispiel ADC-Interrupt

- **Umstellen auf Interrupt-Betrieb (allgemein)**
  - Interrupt-Handler definieren
    - Im Interrupt-Handler sicherstellen, dass die Ursache des Interrupts beseitigt wird (sonst wird der Interrupt-Handler immer wieder aufgerufen)
  - Am zentralen Interrupt-Controller (NVIC) die Verarbeitung von Interrupts der Peripherieeinheit aktivieren
    - Flag in einem Register, kann über Makro `__NVIC_EnableIRQ(`*`InterruptNummer`*`);` gesetzt werden
  - In der Konfiguration der Peripherieeinheit das Auslösen eines Interrupts unter den gewünschten Bedingungen aktivieren

Beispiel ADC-Interrupt

```
/**
 * Beispiel für die Nutzung des direkt programmierten ADC im
 * Interrupt-Modus.
 */
void task2() {
    // Interrupt-getrieben: Interrupts einschalten ...
    __NVIC_EnableIRQ(ADC_IRQn); // ... ADC-Interrupts allgemein und ...
    ADC1->CR1 |= ADC_CR1_EOCIE; // ... speziell den EOC-Interrupt

    // (Erste) Konvertierung starten
    ADC1->CR2 |= ADC_CR2_SWSTART;

    while (true) {
    }
}
```

Live Coding

```
/**
 * Interrupt-Handler für den ADC. Gibt den Konvertierten Wert auf den
 * LEDs aus und startet eine neue Konvertierung.
 */
extern "C" void ADC_IRQHandler() {
    // Prüfen, ob der Interrupt wegen eines abgeschlossenen
    // Konvertierungsvorgangs ausgelöst wurde.
    if (ADC1->SR & ADC_SR_EOC) {
        // Lesen des DR Registers setzt das Interrupt-Flag zurück.
        leds = ADC1->DR >> 4;
        ADC1->CR2 |= ADC_CR2_SWSTART;
    }
}
```