Systemnahe Programmierung in C (SPiC)

Teil D Betriebssystemabstraktionen

Daniel Lohmann, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2012

http://www4.cs.fau.de/Lehre/SS12/V_SPIC

Überblick: Teil D Betriebssystemabstraktionen

Programmausführung, Nebenläufigkeit

Programmablauf in einer µC-Umgebung

- Programm läuft "nackt" auf der Hardware
  - ➥ Compiler und Binder müssen ein vollständiges Programm erzeugen
    - ➤ keine Betriebssystemunterstützung zur Laufzeit
  - Funktionalität muss entweder vom Anwender programmiert werden oder in Form von Funktionsbibliotheken zum Programm dazugebunden werden
  - Umgang mit "lästigen Programmierdetails" (z. B. bestimmte Bits setzen) wird durch Makros erleichtert
- Es wird genau ein Programm ausgeführt
  - Programm kann zur Laufzeit "niemanden stören"
  - Fehler betreffen nur das Programm selbst
  - keine Schutzmechanismen notwendig
    - ➥ ABER: Fehler ohne direkte Auswirkung werden leichter übersehen

Programme laden

- generell bei Mikrocontrollern mehrere Möglichkeiten
  - Programm ist schon da (ROM)
  - Bootloader-Programm ist da, liest Anwendung über serielle Schnittstelle ein und speichert sie im Programmspeicher ab
  - spezielle Hardware-Schnittstelle
    - ➤ "jemand anderes" kann auf Speicher zugreifen
    - ➤ Beispiel: JTAG
      spezielle Hardware-Komponente im AVR-Chip, die Zugriff auf die Speicher hat und mit der man über spezielle PINs kommunizieren kann

Programm starten

- Reset bewirkt Ausführung des Befehls an Adresse 0x0000
  - dort steht ein Sprungbefehl auf die Speicheradresse einer start-Funktion, die nach einer Initialisierungsphase die main-Funktion aufruft
  - alternativ: Sprungbefehl auf Adresse des Bootloader-Programms, Bootloader lädt Anwendung, initialisiert die Umgebung und springt dann auf main-Adresse

Fehler zur Laufzeit

- Zugriff auf ungültige Adresse
  - es passiert nichts:
    - ➤ Schreiben geht in's Leere
    - ➤ Lesen ergibt zufälliges Ergebnis
- ungültige Operation auf nur-lesbare / nur-schreibbare Register/Speicher
  - hat keine Auswirkung

Interrupts

- An einer Peripherie-Schnittstelle tritt ein Ereignis auf
  - ➤ Spannung wird angelegt
  - ➤ Zähler ist abgelaufen
  - ➤ Gerät hat Aufgabe erledigt (z. B. serielle Schnittstelle hat Byte übertragen, A/D-Wandler hat neuen Wert vorliegen)
  - ➤ Gerät hat Daten für die Anwendung bereit stehen (z. B. serielle Schnittstelle hat Byte empfangen)
- ? wie bekommt das Programm das mit?
  - ➤ Zustand der Schnittstelle regelmäßig überprüfen (= ***Polling***)
  - ➤ Schnittstelle meldet sich von sich aus beim Prozessor und unterbricht den Programmablauf (= ***Interrupt***)

Polling vs. Interrupts

- Polling
  - \+ Pollen erfolgt **synchron** zum Programmablauf, Programm ist in dem Moment auf das Ereignis vorbereitet
  - – Pollen erfolgt explizit im Programm und meistens umsonst — Rechenzeit wird verschwendet
  - – Polling-Funktionalität ist in den normalen Pragrammablauf eingestreut — und hat mit der "eigentlichen" Funktionalität dort meist nichts zu tun
- Interrrupts
  - \+ Interrupts melden sich nur, wenn tatsächlich etwas zu erledigen ist
  - \+ Interrupt-Bearbeitung ist in einer Funktion kompakt zusammengefasst
  - – Interrupts unterbrechen den Programmablauf irgendwo (**asynchron**), sie könnten in dem Augenblick stören
    - ➥ durch die Interrupt-Bearbeitungentsteht **Nebenläufigkeit**

Implementierung von Interrupts

- typischerweise mehrere Interrupt-Quellen
- Interrupt-Vektor
  - Speicherbereich (Tabelle), der für jeden Interrupt Informationen zur Bearbeitung enthält
    - ➤ Maschinenbefehl (typischerweise ein Sprungbefehl auf eine Adresse, an der eine Bearbeitungsfunktion (***Interrupt-Handler***) steht) oder
    - ➤ Adresse einer Bearbeitungsfunktion
  - feste Position im Speicher — ist im Prozessorhandbuch nachzulesen
- Maskieren von Interrupts
  - Bit im Prozessor-Statusregister schaltet den Empfang aller Interrupts ab
  - zwischenzeitlich eintreffende Interrupts werden gepuffert (nur einer!)
  - die Erzeugung einzelner Interrupts kann am jeweiligen Gerät unterbunden werden

Interrupts: Ablauf auf Hardware-Ebene

① Gerät löst Interrupt aus,
Ablauf des Anwendungsprogramms wird unmittelbar unterbrochen

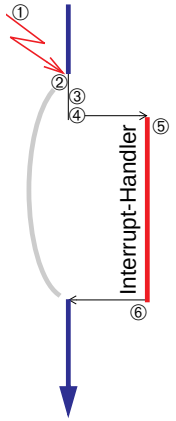
② weitere Interrupts werden deaktiviert

③ aktuelle Position im Programm wird gesichert

④ Eintrag im Interrupt-Vektor ermitteln

⑤ Befehl wird ausgeführt bzw. Funktion aufrufen
(= Sprung in den Interrupt-Handler)

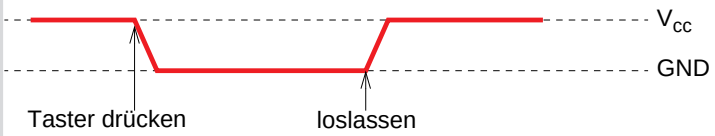
⑥ am Ende der Bearbeitungsfunktion bewirkt ein Befehl "Return from Interrupt" die Fortsetzung des Anwendungsprogramms und die Reaktivierung der Interrupts



! Der Interrupt-Handler muss alle Register, die er ändert am Anfang sichern und vor dem Rücksprung wieder herstellen!

Pegel- und Flanken-gesteuerte Interrupts

- Beispiel: Signal eines (idealisierten) Tasters



- Flanken-gesteuert
  - ➤ Interupt wird durch die Flanke (= Wechsel des Pegels) ausgelöst
  - ➤ welche Flanke einen Interrupt auslöst kann bei manchen Prozessoren konfiguriert werden
- Pegel-gesteuert
  - ➤ solange ein bestimmter Pegel anliegt (hier Pegel = GND) wird immer wieder ein Interrupt ausgelöst

Nebenläufigkeit – Überblick

- Definition von Nebenläufigkeit:
zwei Programmausführungen sind nebenläufig, wenn für zwei einzelne Befehle a und b aus beiden Ausführungen nicht feststeht, ob a oder b tatsächlich zuerst ausgeführt wird
- Nebenläufigkeit tritt auf
  - ➤ bei Interrupts
  - ➤ bei parallelen Abläufen (gleichzeitige Ausführung von Code in einem Mehrprozessorsystem mit Zugriff auf den gleichen Speicher)
  - ➤ bei quasi-parallelen Abläufen (wenn ein Betriebssystem verschiedenen Prozesse den Prozessor jeweils für einen Zeitraum zuteilt und ihn nach Ablauf der Zeit wieder entzieht)
- Problem:
  - ➤ was passiert, wenn die nebenläufigen Ausführungen auf die gleichen Daten im Speicher zugreifen?

Nebenläufigkeit durch Interrupts

- Interrupts unterbrechen Anwendungsprogramme "irgendwo"
- Interrupts haben Zugriff auf den gleichen Speicher
- Szenario:
  - ➤ eine Lichtschranke soll Fahrzeuge zählen und alle 10 Sekunden soll der Wert ausgegeben werden

```
static volatile uint16_t a;

void main(void) {
  volatile uint32_t i;
  while(1) {
    for (i=0; i<2000000; i++)
    /* Zählen dauert 10 Sek. */;
    print(a);
    a=0;
  }
}
```

```
/* Lichtschranken-
   Interrupt */
void count(void) {
        a++;
}
```

Nebenläufigkeit durch Interrupts (2)

- Auf C-Ebene führt die Interrupt-Behandlung nur einen Befehl aus: a++
  - nur scheinbar ein Befehl
  - auf Maschinencode-Ebene (Bsp. AVR) sieht die Sache anders aus

```
...
    print(a);
    a=0;
...
```

```
void count(void) {
        a++;
}
```

```
...
.L5:
        lds r24,a
        lds r25,(a)+1
        rcall print
        sts (a)+1,__zero_reg__
        sts a,__zero_reg__
        rjmp .L2
...
```

```
...
        lds r24,a
        lds r25,(a)+1
        adiw r24,1
        sts (a)+1,r25
        sts a,r24
...
```


Nebenläufigkeit durch Interrupts (3)

- Annahme1: Interrupt trifft folgendermaßen ein:

```
.L5:
H1      lds r24,a
H2      lds r25,(a)+1
H3      rcall print
H4      sts (a)+1,__zero_reg__
H5      sts a,__zero_reg__
H6      rjmp .L2
```

```
I1      lds r24,a
I2      lds r25,(a)+1
I3      adiw r24,1
I4      sts (a)+1,r25
I5      sts a,r24
```

  - Folge: ein Fahrzeug wird nicht gezählt
    - ➥ **Lost-Update-Problem**
- Details des Szenarios zeigen mehrere Problemstellen:
  - uint16-Wert wird in zwei Schritten in zwei Register geladen (uint32: 4 Register)
  - Operationen erfolgen in Registern, danach wird in Speicher zurückgeschrieben

Nebenläufigkeit durch Interrupts (3)

- Skizze zu Annahme 1, a habe initial den Wert 5

Code-zeile	Variable **a**		Prozessor-Register		gesicherte Registerinhalte		Ausgabe von print
	oberes Byte	unteres Byte	**r25**	**r24**	**r25**	**r24**	
initial	**00**	**05**					
H1		05		**05**			
H2	00		**00**				
INT			00	05	**00**	**05**	
I1		05		**05**			
I2	00		**00**	05			
I3			**00**	**06**			
I4	**00**		00				
I5		**06**		06			
ret			**00**	**05**	00	05	
H3			00	05			**5**
H4	**00**						
H5		**00**					

Nebenläufigkeit durch Interrupts (4)

- Annahme 2: Interrupt trifft folgendermaßen ein:

```
.L5:
H1      lds r24,a
H2      lds r25,(a)+1
H3      rcall print
H4      sts (a)+1,__zero_reg__
H5      sts a,__zero_reg__
H6      rjmp .L2
```

```
I1      lds r24,a
I2      lds r25,(a)+1
I3      adiw r24,1
I4      sts (a)+1,r25
I5      sts a,r24
```

  - Folge: möglicherweise werden 255 Fahrzeuge zuviel gezählt
    - Variable a ist auf 2 Register verteilt →a = 0 nicht atomar zuerst wird obere Hälfte auf 0 gesetzt
    - falls a++ im Interrupt-Handler a zufällig von 255 auf 256 zählt → Bitüberlauf vom "unteren" in's "obere" Register
    - nach Interrupt wird nur noch untere Hälfte auf 0 gesetzt → a = 256

Nebenläufigkeit durch Interrupts (4)

- Skizze zu Annahme 2, a habe initial den Wert 255

Code-zeile	Variable **a**		Prozessor-Register		gesicherte Registerinhalte		Ausgabe von print
	oberes Byte	unteres Byte	**r25**	**r24**	**r25**	**r24**	
initial	**00**	**ff**					
H1		ff →		**ff**			
H2	00 →		**00**				
H3			00	ff			**255**
H4	**00**						
INT			00	ff →	**00**	**ff**	
I1		ff →		**ff**			
I2	00 →		**00**	ff			
I3			**01**	**00**			
I4	**01** ←		01				
I5		**00** ←		00			
ret			**00**	**ff** ←	00	ff	
H5	01	**00**					

Nebenläufigkeit durch Interrupts (5)

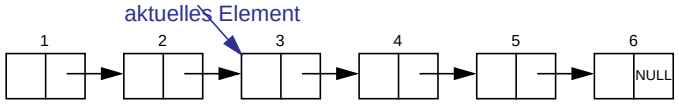
- weiteres Problem bei Zugriff auf globale Variablen:
  - AVR stellt 32 Register zur Vefügung
  - Compiler optimiert Code und vermeidet Speicherzugriffe wenn möglich
    - ➤ Variablen werden möglichst in Registern gehalten
  - Registerinhalte werden bei Interrupt gesichert und am Ende restauriert
    - ➤ Änderungen der Interrupt-Funktion an einer Variablen gehen beim Restaurieren der Register wieder verloren
- Lösung für dieses Problem:
  - Compiler muss Variablen vor jedem Zugriff aus dem Speicher laden und anschließend zurückschreiben
    - ➤ Attribut `volatile`

```
volatile uint16_t a;
```

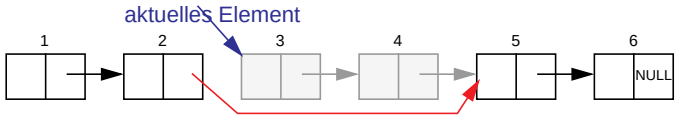
  - Nachteil: Code wird umfangreicher und langsamer
    - ➤ nur einsetzen wo unbedingt notwendig!

Nebenläufigkeitsprobleme allgemein

- Zugriff auf gemeinsame Daten ist bei nebenläufigen Ausführungen generell kritisch
  - ➤ selbst bei einfachen Variablen (siehe vorheriges Beispiel)
  - ➤ Problem bei komplexeren Datenstrukturen
    (z. B. Einketten einer Struktur in verkettete Liste)
    noch gravierender: Datenstruktur kann völlig zerstört werden
- Beispiel: Programm läuft durch eine verkettete Liste



  - Interrupt-Handler oder parallel laufendes Programm entfernt Elemente 3 und 4 und gibt den Speicher dieser Elemente frei



Umgang mit Nebenläufigkeitsproblemen

- Gemeinsame Daten möglichst vermeiden
  - ➤ Interrupt-Funktionen sollten weitgehend auf eigenen Daten arbeiten
  - ➤ Parallele Abläufe sollten ebenfalls möglichst eigene Datenbereiche haben
- Kommunikation zwischen Anwendungsabläufen erfordert aber oft gemeinsame Daten
  - ➤ solche Daten sollten deutlich hervorgehoben werden
    z. B. durch entsprechenden Namen

```
volatile uint16_t INT_zaehler;
```

    - betrifft nur globale Variablen
    - lokale Variablen sind unkritisch
      (nur in der jeweiligen Funktion sichtbar)
  - ➤ Zugriff auf solche Daten sollte in der Anwendung möglichst begrenzt sein
    (z. B. nur in bestimmten Funktionen,
    gemeinsames Modul mit Interrupt-Handlern, vgl. Kap. 12)

Umgang mit Nebenläufigkeitsproblemen (2)

- Zugriffskonflikte mit Interrupt-Handlern verhindern
  - das Programm muss vor kritischen Zugriffen auf gemeinsame Daten Interrupts sperren
    - ➤ Beispiel AVR:
      Funktionen `cli()` (blockiert alle Interrupts)
      und `sei()` (erlaubt Interrupts)
  - Problem: Interrupt-Verluste bei Interrupt-Sperren
    - ➤ trifft ein Interrupt während der Sperre ein, wird im zugehörigen Register das entsprechende Bit gesetzt
    - ➤ treffen weitere Interrupts ein, geht diese Information verloren

➥ Zeitraum von Interruptsperren muss möglichst kurz bleiben!

- ➤ alternativ kann es sinnvoll sein, nur Interrupts des Geräts zu sperren, dessen Handler auch auf die kritischen Daten zugreift
  (hängt vom Einzelfall und von Details der Hardware ab!)

Umgang mit Nebenläufigkeitsproblemen (3)

- Warten auf einen Interrupt
  - Häufiges Szenario: im Programm soll auf ein bestimmtes Ereignis gewartet werden, das durch einen Interrupt signalisiert wird
    - ➤ Warten erfolgt meist passiv (Sleep-Modus des Prozessors)
  - Problem: Abfrage ob Ereignis bereits eingetreten ist, ist ein kritischer Zugriff auf gemeinsame Daten mit der Interrupt-Behandlung

```
volatile static uint8_t event = 0;
ISR(INT2_vect) { event = 1; }

void main(void) {
       sleep_enable();

       while(1) {
              while(event == 0) { /* Warte auf Ereignis */
                     sleep_cpu();
              }
              /* bearbeite Ereignis */
              ...
```

  - Synchronisation erforderlich?

Umgang mit Nebenläufigkeitsproblemen (4)

- ... Warten auf einen Interrupt
  - Was passiert, wenn der Interrupt an dieser Stelle eintrifft?

```
volatile static uint8_t event = 0;
ISR(INT2_vect) { event = 1; }

void main(void) {
       sleep_enable();

       while(1) {

              while(event == 0) { /* Warte auf Ereignis */
                     sleep_cpu();
              }

              /* bearbeite Ereignis */
              ...
```

Interrupt!

➥ **Lost-Wakeup-Problem**

Umgang mit Nebenläufigkeitsproblemen (5)

- ... Warten auf einen Interrupt
  - kritischer Abschnitt

```
volatile static uint8_t event = 0;
ISR(INT2_vect) { event = 1; }

void main(void) {
       sleep_enable();

       while(1) {
                     kritischer Abschnitt
              while(event == 0) { /* Warte auf Ereignis */
                     sleep_cpu();
              }

              /* bearbeite Ereignis */
              ...
```

Interrupt!

  - können hier Interruptsperren helfen?

Umgang mit Nebenläufigkeitsproblemen (6)

- ... Warten auf einen Interrupt
  - Problem: Interruptsperre muss vor dem `sleep_cpu()` aufgehoben werden

```
volatile static uint8_t event = 0;
ISR(INT2_vect) { event = 1; }

void main(void) {
       sleep_enable();

       while(1) {
              cli();   kritischer Abschnitt
              while(event == 0) { /* Warte auf Ereignis */
                     sei();                    Interrupt!
                     sleep_cpu();

              }

              /* bearbeite Ereignis */
              ...
```

    - ➤ aber Interrupt darf nicht zwischen `sei()` und `sleep_cpu()` kommen
    - ➤ Lösung: `sei()` und die Folgeanweisung werden atomar ausgeführt (Hardware-Mechanismus des AVR-Prozessors!)

Umgang mit Nebenläufigkeitsproblemen (7)

- Einseitige Synchronisation
  - Besonderheit bei Nebenläufigkeit durch Interrupts:
    - ➤ der Interrupt kann den normalen Programmablauf unterbrechen
    - ➤ aber nicht umgekehrt
    - ➥ die Interruptbehandlung wird nie unterbrochen (höchstens durch Interrupts mit höherer Prioriät)
- Mehrseitige Synchronisation
  - Standardsituation bei parallelen Abläufen (z. B. auf Mehrkern-Prozessoren)
    - ➤ Interruptsperren helfen hier nicht
  - Lösungen
    - ➤ spezielle atomare Maschinenbefehle (z. B. test-and-set oder compare-and-swap bei Intel-Architekturen)
    - ➤ Software-Synchronisation (lock-Variablen, Semaphore, etc.)
    - ➤ Kommunikation mittels Nachrichten statt gemeinsamer Daten