

C/C++ Materialpaket (Level C)

02b_FLOW_intr – Control Flow (Interrupts)

Prof. Dr. Carsten Link

Zusammenfassung

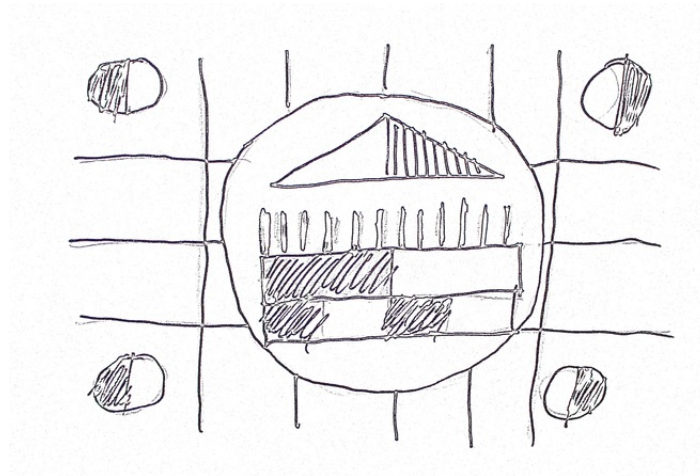


Abbildung 1: Testbild

Inhaltsverzeichnis

1 Kompetenzen und Lernergebnisse	2
2 Konzepte	2
2.1 Asynchron unterbrochener Kontrollfluss (Interrupts)	2
2.2 Datenstrukturen für Interrupt-Handler	5
2.3 Exkurs: alternierender Kontrollfluss (Koroutinen)	6
2.3.1 Generatoren in Python	6
3 Material zum aktiven Lernen	8
3.1 Aufgabe: Grundgerüst	8
3.2 Aufgabe: Modifikationen	8
3.3 Verständnisfragen	9
4 Nützliche Links	9

1 Kompetenzen und Lernergebnisse

Durch das Bearbeiten dieses Materialpaketes erwerben Sie diese Kompetenzen (Wissen, Fähigkeiten und Fertigkeiten zur Problemlösung):

Sie können Interrupt-Handler implementieren, welche Daten mit dem Hauptprogramm austauschen.

Die oben genannten Kompetenzen erwerben Sie, indem Sie Lernziele erreichen, welche sich prüfen lassen. Lernergebnisse: Sie können nachweislich¹:

- mit den Besonderheiten des Programmablaufs bei der Mikrocontrollerprogrammierung umgehen, indem sie in einem Programm asynchrone Ereignisse mit den adäquaten Sprachmitteln behandeln und die geeigneten Datenstrukturen auswählen
- einen Interrupt Handler implementieren
- die Unterschiede und Gemeinsamkeiten zwischen Funktionsaufrufen und Interrupt Handlers analysieren
- Datenstrukturen für den Datenaustausch zwischen Hauptprogramm einsetzen oder implementieren
- den Nutzen erläutern, den Koroutinen bei bestimmten Anwendungsszenarien haben und diese von Prozeduren/Funktionen sowie Threads abgrenzen

2 Konzepte

Im Folgenden wird auf die weiterführende Kontrollstrukturen von C++ eingegangen. Diese Kontrollstrukturen erlauben es dem Programmierer, Programme zu schreiben, die auf Hardware-Ereignisse reagieren (Interrupts).

2.1 Asynchron unterbrochener Kontrollfluss (Interrupts)

Bei der Programmierung von Mikrocontrollern ist eine Art des Kontrollflusses besonders wichtig: Interrupts mit Interrupt Service Routinen (ISR, auch interrupt handler genannt). Interrupts werden von Hardwarekomponenten ausgelöst, um dem Programm (Betriebssystem, -bibliothek) die Behandlung von externen Ereignissen zu ermöglichen (z.B. das Eintreffen eines Zeichens auf einer seriellen Schnittstelle). Stellt die CPU das Vorhandensein eines Interrupts fest, so stoppt sie die Ausführung des aktuellen Programms und springt die Interrupt Service Routine an. Diese kümmert sich um das externe Ereignis (z.B. liest das Zeichen aus der seriellen Schnittstelle) und läßt die CPU wieder in den normalen Betriebsmodus im zuvor ausgeführten Programm zurückkehren (z.B. mit dem CPU-Befehl `interrupt return`). Das Programm, das durch den Interrupt und

¹Sie können das Erzielen der einzelnen Lernergebnisse beispielsweise bei einem Testat im Praktikum oder einer Aufgabe in der Modulprüfung nachweisen.

der damit verbundenen Ausführung der ISR unterbrochen wurde, bemerkt nichts davon – die Ausführung wird schlicht nach der kurzen Unterbrechung wieder aufgenommen.

Damit nach der Ausführung das unterbrochene Programm unbeeinträchtigt weiter ausgeführt werden kann, muss die ISR den Zustand sichern (CPU-Register etc., insbes. aktueller Aktivierungsrecord). Hier gibt es also eine Parallele zum Funktionsaufruf, bei dem der Aktivierungsrecord der aufrufenden Funktion auf dem call stack gesichert wird. Zwei wesentliche Unterschiede ergeben sich jedoch²:

- der Aufruf einer ISR wird nicht vom Programm ausgelöst und dessen Zeitpunkt ist beliebig
- daraus ergibt sich, dass es keinen direkten Weg gibt, Daten vom der ISR an das Hauptprogramm zu übergeben (`return` steht nicht zur Verfügung)

Das folgende Beispiel illustriert das Zusammenspiel aus Hauptprogramm und ISR. Dabei wird der Einfachheit halber auf den `signal`-Mechanismus von UNIX-Betriebssystemen zurückgegriffen.

Die Funktion `ISR()` stellt die Interrupt Service Routine dar. Interrupts werden von außen mit dem `kill`-Befehl im Shell Skript `run.sh` ausgelöst. Es wird das Signal `USR1` mit dem Wert 30 an den Prozess gesendet (vgl. `install_ISR()`).

```

1  #include <thread>
2  #include <chrono>
3  #include <future>
4  #include <functional>
5  #include <signal.h>
6  #include <unistd.h>
7  #include <time.h>
8  #include "AnsiConsole.hpp"
9  #include "println.hpp"
10 #include "printSteps.hpp"
11
12 const int maxSignals = 3;
13 const bool doPrintDebug = false;
14
15 struct TimeStamps {
16     int slotsUsed;
17     time_t slots[maxSignals];
18 };
19
20 volatile int j;
21 volatile TimeStamps timeStamps;
22 volatile sigset_t blockedSignals;
23
24 void ISR(int iarg);

```

²Ausnahme: Software Interrupts; siehe Vorlesung Betriebssysteme

```

25
26 void install_ISR(void){
27     signal(SIGUSR1, ISR);
28 }
29
30 //extern "C" {
31 void ISR(int iarg){
32     if (doPrintDebug) println("ISR j=",j);nop;
33     if(timeStamps.slotsUsed<maxSignals){nop;
34         time((time_t*)&timeStamps.slots[timeStamps.slotsUsed++]);nop;
35         nop;}
36     nop;}
37 //}
38
39 void mainloop(){
40     for(int i=0; i<20; i++){
41         sigprocmask(SIG_BLOCK, (sigset_t *)&blockedSignals, NULL);
42         int slotsUsed = timeStamps.slotsUsed;nop;
43         sigprocmask(SIG_UNBLOCK, (sigset_t *)&blockedSignals, NULL);
44         if(slotsUsed>=maxSignals){
45             if (doPrintDebug) println("finished at ", i);
46             break;
47         }
48         j=i;
49         if (doPrintDebug) println("main for loop at i=", i);
50         std::this_thread::sleep_for(std::chrono::milliseconds(1000));
51     }
52 }
53
54 void start_ISR(){
55     if (doPrintDebug) println("process ID = ", getpid());
56     timeStamps.slotsUsed = 0;
57     sigemptyset ((sigset_t *)&blockedSignals);
58     sigaddset((sigset_t *)&blockedSignals, SIGUSR1);
59     install_ISR();
60     mainloop();
61     if (doPrintDebug) println("done.");
62 }

```

Das Skript `run.sh` startet das obige Programm und sendet dann an dessen Prozess-ID das Signal `USR1`, wodurch der installierte Handler `ISR()` ausgeführt wird:

```

#!/bin/bash
./flow_b.out &
#echo $!

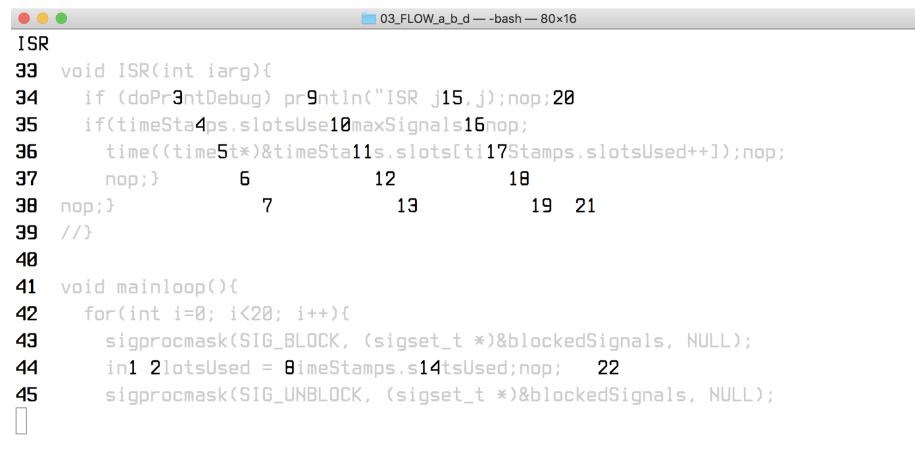
```

```

sleep 1.5
kill -USR1 $!
sleep 1.3
kill -USR1 $!
sleep 1.1
kill -USR1 $!
#echo ...
kill -USR1 $!
#sleep 1.7
#kill -USR1 $!
#kill -USR1 $!

```

Bei Ausführung ergibt sich beispielsweise die in Abbildung 2 dargestellte Ausgabe.



```

03_FLOW_a_b_d -- -bash -- 80x16
ISR
33 void ISR(int iarg){
34     if (doPrintDebug) printf("ISR j15_j);nop;20
35     if(timeStampSlotsUsed>maxSignals;nop;
36         time((timeStampSlotsUsed+1));nop;
37         nop;}          6          12          18
38 nop;}          7          13          19 21
39 //}
40
41 void mainloop(){
42     for(int i=0; i<20; i++){
43         sigprocmask(SIG_BLOCK, (sigset_t *)&blockedSignals, NULL);
44         int slotsUsed = timeStampSlotsUsed;nop; 22
45         sigprocmask(SIG_UNBLOCK, (sigset_t *)&blockedSignals, NULL);

```

Abbildung 2: Handler für Unterbrechungen (ISR), dessen Ausführung durch Signals ausgelöst wird.

Es ist zu sehen, dass die Zeile 09 in ISR() ausgeführt wird, obwohl in der Iteration in Zeile 21 keine Aufruf zu sehen ist.

Wesentlicher Unterschied ISR vs. Funktionsaufruf: der unterbrochene Code hat bei der ISR deren Aufruf nicht ausgelöst (nicht explizit und in der Regel auch nicht implizit) und kann daher keinen Rückgabewert in Empfang nehmen.

2.2 Datenstrukturen für Interrupt-Handler

Falls die ISR Daten an das Hauptprogramm übergeben soll, so ist dies weniger schwierig als beim Datenaustausch zwischen Threads, da nur zwei Akteure im Spiel sind und beide (ISR und Hauptprogramm) sicherstellen können, nicht unterbrochen zu werden.

Teilweise sind also Mutual Exclusion und Synchronisation notwendig (wenn main Daten bearbeitet, darf der Handler sie nicht bearbeiten). Es lässt sich leicht erreichen, dass keine Unterbrechungen von ISR oder Hauptprogramm stattfinden, indem bei der CPU die Interruptverarbeitung kurzzeitig deaktiviert wird.

Es gibt *wait free data structures* hierbei ist allerdings folgendes zu beachten:

- es wird davon ausgegangen, dass der Puffer nicht überläuft
- Voraussetzung: atomarer Zugriff auf int und pointer durch die CPU

Weiterführende Informationen finden sich im Abschnitt “Nützliche Links”.

2.3 Exkurs: alternierender Kontrollfluss (Koroutinen)

Bei Threads arbeiten verschiedene Ausführungspfade gleichzeitig (nebenläufig) daran, ein Problem zu lösen. Da bei Threads die Ausführung auf Mehrkern-CPUs tatsächlich parallel stattfindet, ist wechselseitiger Ausschluss und Synchronisation notwendig. Dies macht die Programmierung mit Threads schwierig und Fehleranfällig.

Bei Koroutinen arbeiten mehrere Ausführungspfade an der Lösung eines Problems – jedoch ist immer nur einer gleichzeitig in Ausführung. Daher sind kein wechselseitiger Ausschluss und Synchronisation notwendig, was die Programmierung erheblich vereinfacht.

Der Wechsel von einem Ausführungspfad zu einem anderen geschieht bei Koroutinen programmgesteuert und ist daher im Quelltext direkt ersichtlich.

Koroutinen stellen eine mächtige Möglichkeit dar, Zustand zu kapseln (im Aktivierungsrecord einer Funktion, statt z.B. in einem Objekt (-baum))

In C++ ist derzeit noch keine brauchbare Unterstützung für Koroutinen enthalten (siehe ³). Es existieren jedoch Bibliotheken, mit denen Koroutinen verwendet werden können (z.B. `libcoroutine`⁴). Daher wird im nächsten Kapitel der Nutzen von Koroutinen mit Python-Code dargelegt.

2.3.1 Generatoren in Python

In Python wird eine Form der Koroutinen häufig verwendet, die Generator⁵ genannt wird. Kernidee ist, dass eine Funktion im Rahmen einer `for`-Schleife aufgerufen wird, einen Wert zurückliefert und hierbei jedoch seinen Aktivierungsrecord beibehält. Statt mit `return` wird per `yield` die Ausführung der (Generator-) Funktion eingefroren und das `yield`-Argument an die `for`-Iteration weitergegeben. Im Gegensatz zu den allgemeineren Koroutinen ist es bei Python-Generator nicht möglich, beim `yield` zu spezifizieren, an welche Koroutine die Ausführung zurückgegeben wird.

³<https://www.slideshare.net/cppfrug/async-await-in-c>

⁴<https://github.com/stevedekorte/coroutine>

⁵<https://wiki.python.org/moin/Generators>

Hier ein einfaches Beispiel:

```
1 def weekdayIterator():
2     yield "Monday"
3     yield "Tuesday"
4     yield "Wednesday"
5     yield "Thursday"
6     yield "Friday"
7     yield "Saturday"
8     yield "Sunday"
9
10 for day in weekdayGenerator():
11     print(day)
```

Ergibt die Ausgabe:

```
1 Monday
2 Tuesday
3 Wednesday
4 Thursday
5 Friday
6 Saturday
7 Sunday
```

Dabei ist zu beachten, dass `yield` von der Funktion `weekdayIterator` mehrfach verwendet wird (`return` würde nur einmal ausgeführt) und bei jeder Verwendung ein anderer Wert an die `for`-Iteration übergeben wird.

In folgendem Beispiel ist zu sehen, dass auch komplexere Kontrollstrukturen innerhalb einer Generatorenfunktion verwendet werden können:

```
1 def timesTwoGenerator(endValue):
2     i = 1
3     while i <= endValue:
4         yield i * 2
5         i += 1
6
7 for j in timesTwoGenerator(5):
8     print(j)
```

Ergibt die Ausgabe:

```
1 2
2 4
3 6
4 8
5 10
```

Hier wird ersichtlich, dass zwischen den mehrfachen `yields` der Kontext der

Funktion `timesTwoGenerator` eingefroren und wiederverwendet (engl.: resumed) wird. Zum Kontext gehören die Variablen `i` und `endValue` sowie die Position der Ausführung (der nächste Durchlauf beginnt hinter dem letzten `yield`).

3 Material zum aktiven Lernen

Regelmäßiger Hinweis: Da eine Programmiersprache nur durch aktive Verwendung erlernt werden kann, werden im Folgenden Aufgaben zum praktischen Üben vorgestellt. Zunächst wird ein Grundgerüst (C/C++-Programm) erstellt, welches dann auf mehrere Arten modifiziert wird. Insbesondere die Modifikationen ermöglichen es dem Lernenden (und auch dem Lehrenden), die Qualität des Kompetenzerwerbs bzgl. dieses Materialpakets bewerten zu können.

3.1 Aufgabe: Grundgerüst

Nehmen Sie die Datei `main_ISR.cpp` als Ausgangspunkt. Erweitern Sie es so, dass der Interrupt Handler in einer globalen Datenstruktur einen Zeitstempel ablegt. Das Hauptprogramm befindet in einer Dauerschleife, die beendet wird, sobald in der globalen Datenstruktur mehr als fünf Einträge vorhanden sind. Diese werden dann beim Programmende ausgegeben. Testen Sie ausgiebig und stellen Sie sicher, dass die Datenstruktur nicht durch den verzahnten Zugriff durch den Interrupt Handler inkonsistent wird.

3.2 Aufgabe: Modifikationen

Regelmäßiger Hinweis: Weiter unten ist eine Liste mit Modifikationen gegeben, die zwei Zwecken dienen: 1) Sie dienen als Richtschnur für das Praktizieren und Üben der Inhalte dieses Materialpakets. 2) Die Modifikationen können im Rahmen eines Testats als Aufgabe verwendet werden, durch deren Lösung Studierende nachweisen können, dass sie den Stoff dieses Materialpakets beherrschen. Stellen Sie sicher, dass Sie jede einzelne der nachfolgenden Modifikationen innerhalb weniger Minuten (ca. 5 - 10) vor Zuschauern (Testatsituation) umsetzen können. Konkret sollen Sie im Testat in der Lage sein, das gegebene Grundgerüst um mindestens eine zufällig ausgewählte Modifikation zu erweitern. Bereiten Sie dazu auf ihrer Arbeitsumgebung ein Verzeichnis vor, welches ausschließlich das Grundgerüst enthält. **Arbeiten Sie also auf einer Kopie des Verzeichnisses `src-cpp-student/`!** Achten Sie darauf, dass der Text auf Ihrem Bildschirm in heller Umgebung aus einem Meter Abstand heraus gut lesbar ist (light mode, große Schrift).

Modifikationen:

1. Verwenden Sie `sigprocmask`, um sicherzustellen, dass das Hauptprogramm nicht durch einen Interrupt unterbrochen werden kann, wenn es die globale Datenstruktur verwendet oder Manipuliert (vergleiche *Critical Section*)

2. Machen Sie die Registrierung der Handler-Funktion vor Programmende rückgängig

3.3 Verständnisfragen

Nach Bearbeitung des Kapitels “Konzepte”, der Erstellung des Grundgerüsts sowie dem Üben der Modifikationen sollten Sie in der Lage sein, die folgenden Fragen zu beantworten.

1. Warum sind besondere Datenstrukturen notwendig für den Datenaustausch zwischen Hauptprogramm und ISR?
2. Warum sind Garantien von der Hardware (CPU) notwendig für den Datenaustausch zwischen Hauptprogramm und ISR?
3. Was sind die Gemeinsamkeiten zwischen Funktionsaufrufen und ISR?
4. Was sind die Unterschiede zwischen Funktionsaufrufen und ISR?
5. Freiwillig: Zeigen Sie auf, inwiefern das Piktogramm auf der Titelseite dieses Materialpaketes den Inhalt zusammengefasst darstellt.

4 Nützliche Links

- QNX: Writing an Interrupt Handler⁶
- Microcontrollers: Interrupt-safe ring buffers⁷
- Simply embedded, LESSON 10: UART RECEIVE BUFFERING⁸

5 Literatur

- [PPP] Stroustrup, Bjarne: Programming - Principles and Practice using C++
- [TCPL] Stroustrup, Bjarne: The C++ Programming Language, Fourth Edition

⁶<http://www.qnx.de/developers/docs/6.4.0/neutrino/prog/inthandler.html>

⁷<https://www.downtowndougbrown.com/2013/01/microcontrollers-interrupt-safe-ring-buffers/>

⁸<http://www.simplyembedded.org/tutorials/interrupt-free-ring-buffer/>