OSTBAYERISCHE TECHNISCHE HOCHSCHULE REGENSBURG

NNXT - Tutorials

Tutorial #04 - Das Betriebssystem des NNXT

Grundsätzliches

Betriebssysteme sind - wie in der Vorlesung vorgestellt - dazu da, die Betriebsmittel eines Rechners zu verwalten und auf konkurrierende Nutzer zu verteilen. Für die Resource Rechenzeit wird dieses über den sogenannen Scheduler erledigt, der den verschiedenen Prozessen nach unterschiedlichen Schematas Rechenzeit zuordnet. Prozesse im Kontext von technischen Systemen, etwa in der Robotik, sind parallele Programme, die in einem gemeinsamen Speicherraum ablaufen und um die Rechenzeit konkurrieren. Diese Art von parallel ablaufenden Programmen nennt man im Kontext der technischen System Tasks. Das Betriebssystem legt fest, wie viele dieser Tasks es gleichzeitig geben darf. In unserem Fall ist diese Menge auf 15 beschränkt, d.h. **Sie dürfen niemals mehr als 15 Tasks gleichzeitig aktiv haben**. Es kann in den Programmcodes durchaus mehr als 15 potentielle Tasks geben, aber nur maximal 15 davon dürfen gleichzeitig existieren, alle anderen müssen gelöscht sein.
Die Rechenzeitzuteilung erfolgt für das NNXT-Betriebssystem (welches eine Erweiterung des FreeRTOS Betriebssystems ist) nach einer Reihum-Strategie: Jeder Task bekommt eine Rechenzeitzuteilung von etwa 1 Millisekunde, danach schaltet sich das Betriebssystem ein und teilt einem anderen Task Rechenzeit zu. Während ein Task rechnet, tun alle anderen nichts, sondern stehen in ihrem Ablauf an der unterbrochenen Stelle und warten auf erneute Zuteilung von Rechenzeit. Wichtig: Wir wissen grundsätzlich niemals, wann welcher Task in welchem Teil seines Codes läuft oder steht. Durch die Reihum-Strategie entsteht der Eindruck von parallel ablaufenen Tasks, was jedoch technisch nicht richtig ist und nur durch das schnelle Umschalten erreicht wird.

Um auf der NNXT-Plattform parallele Tasks zu erzeugen, zu starten, zu stoppen und zu löschen, bietet die NNXT-Library wiederum eine API an, die benutzt werden muss.

NNXT-Betriebssystem API

Die unterschiedlichen Tasks auf dem NNXT sind zunächst normale Funktionen, die keinen Rückgabewert und keine Parameter haben. Sie können einem Task grundsätzlich nichts über Parameter übergeben, da die Tasks nicht direkt als Funktionsaufruf gestartet werden, sondern dies implizit über das Betriebssystem erfolgt. auch geben Tasks keine Werte zurück, da sie entweder niemals enden oder aber die Beendigung durch das Betriebssystem erfolgt.Die main-Funktion ist kein eigenständiger Task, sondern wird ausgeführt, bevor das Betriebssystem gestartet wird. Sie können also wie gewohnt alle Konfigurationseinstellungen, wie z.B. Port/Sensor-Einstellungen oder Initialwerte von globalen Variablen dort einstellen. Anschließend müssen Sie aber explizit das Betriebssystem durch einen Funktionsaufruf starten. Nach dem Start des Betriebssystems wird niemals mehr in die main-Funktion zurückgekehrt, da das Betriebssystem ab diesem zeitpunkt die Kontrolle übernimmt und die Tasks abarbeitet. Sollten keine Tasks aktiv sein, so läuft der sogenannte Background-Task, der lediglich aus einer while-Schleife mit keinen Aktionen besteht.
Das Betriebssystem selbst ist Teil Ihres Programmes und wird beim Übersetzungsvorgang mit compiliert. Sie können auch den Quellcode des Betriebssystems sehen: öffnen Sie unter dem ordner "Source" den Ordner "Libs", dann "FreeRTOS", dann "Source". Hier finden Sie die Quellen des Betriebssystems. **Hier bitte niemals etwas verändern!**

Das Betriebssystem starten Sie aus Ihrer main-Funktion mit einem Funktionsaufruf der Funktion:

```
void StartScheduler();
```

Damit wird die Kontrolle abgegeben und alle Tasks, die auf Rechenzeit warten, werden vom dann laufenden Betriebssystem bedient. Beachten Sie: Da die **`main`**-Funktion kein Task ist, wird sämtlich Code, der hinter dem Aufruf von **`StartScheduler()`** steht niemals ausgeführt!

Es werden also nun alle Tasks ausgeführt, die Rechenzeit erwarten, nur wie werden die erkannt? Dafür gibt es unter dem NNXT-Betriebssystem einige Funktionen, mit denen man mit Tasks arbeiten kann. Zunächst ist - wie oben beschrieben - ein Task nichts weiter als eine rückgabelose Funktion ohne Parameter. Dem Betriebssystem muss nun bekannt gemacht werden, dass die fragliche Funktion ein Task sein soll. Dazu existiert eine Funktion, die für das Erzeugen des Task zuständig ist:

```
void CreateTask(nnxt_task task);
```

Die Funktion erwartet ein Argument vom Typ **nnxt_task**. Schauen wir in der Definition von nnxt_task nach, so finden wir die Zeile:

```
typedef void (*nnxt_task)( void * );
```

nnxt_task ist also ein Funktionszeiger auf eine parameterlose Funktion, die nichts zurückgibt. Das bedeutet, dass Sie beim Aufruf von **CreateTask** einfach den Namen der Funktion, die Task sein soll, übergeben (ein Beispiel folgt gleich).

Auf diese Weise haben wir beim Betriebssystem einen Task registriert und innerhalb des Betriebssystems werden alle dafür notwendigen Einstellungen getätigt. Tasks zu registrieren heißt aber nicht, dass Sie schon Rechenzeit haben möchten, sondern legt lediglich die Strukturen dafür im Betriebssystem an. Um einen Task zu starten, bedarf es eines weiteren Funktionsaufrufs:

```
void StartTask(nnxt_task task);
```

Nun ist der angegebene Task bereit, Rechenzeit in Empfang zu nehmen und kann vom Betriebssystem ausgeführt werden. Sie können das Starten der Tasks in der **main**-Funktion vor dem Aufruf des Betriebssystems machen, es ist aber auch möglich, aus laufenden Tasks heraus neue Tasks über **CreateTask** zu erzeugen und diese mittels **StartTask** zu starten. Empfehlung: tun Sie dies zunächst nicht, da das Durchdenken dieser verschränkten, teils parallelen Ausführungen dem menschlichen Verstand einige Schwierigkeiten bereitet. Es empfiehlt sich daher, alle Tasks in der main-Funktion anzulegen und zu starten und dann das Betriebssystem zu starten.
Ein wichtiger weiteres Faktum ist, dass Tasks sich nicht einfach beenden dürfen, da ansonsten das Betriebssystem die Kontrolle verliert. Legen Sie deshalb alle Tasks nicht-terminierend aus, d.h. alle Tasks sollten über eine **while(1)**-Schleife verfügen. Wie in der Vorlesung bereits besprochen, sollten Sie auch nie Tasks anlegen, die ununterbrochen laufen, sondern die Iteration der **while(1)**-Schleife enthält grundsätzlich ein **Delay** mit einer Wartezeit, die über das Abtasttheorem bestimmt wurde.
Zum Abschluss hier ein Beispiel für die typische Struktur von Mehr-Task-Systemen (hier 2 parallele Tasks):

```
void task1() {
    // Initialberechnungen
    while(1) {
        // Hier der Code des Tasks task1
        Delay(N);  // Hier Delay für task1 gemäß Abtasttheorem
    }
}

void task2() {
    // Initialberechnungen
    while(1) {
        // Hier der Code des Tasks task2
        Delay(M);  // Hier Delay für task2 gemäß Abtasttheorem
    }
}

int main() {
    // Hier Systeminitialisierungen, z.B. Ports

    CreateTask(task1);
    CreateTask(task2);
    StartTask(task1);
    StartTask(task2);

    StartScheduler();   // Betriebssystem starten

    // diese Stelle wird niemals erreicht
    return 0;
}
```

Da das Erzeugen und das Starten von Tasks häufig gemeinsam benötigt wird, gibt es in der NNXT-Library ein Makro, womit Sie beides gemeinsam durchführen können:

```
    CreateAndStartTask(task)
```

damit verkürzt sich obige **main**-Funktion zu:

```
int main() {
    // Hier Systeminitialisierungen, z.B. Ports

    CreateAndStartTask(task1);
    CreateAndStartTask(task2);

    StartScheduler();   // Betriebssystem starten

    // diese Stelle wird niemals erreicht
    return 0;
}
```

Es gibt noch einige weitere Funktionen rund um das Thema "Tasks", die wir aber in den Übungen nicht benötigen. Bei Interesse schauen Sie in die Dokumentation der API.

Nun können Sie ihr Programm mit mehreren Tasks schreiben. Solange alle Tasks unabhängig voneinander laufen können, ist dies problemlos möglich. Was aber, wenn Tasks sich auf irgendeine Art und Weise synchronisieren wollen, also beispielsweise erzeugt **task1** ein Ergebnis, das **task2** benötigt? Wie derartiges implmentiert wird, beschreibt das nächste Kapitel.

Synchronisation über globale Variablen

Tasks, die miteinander kommunizieren wollen, also beispielsweise Werte austauschen wollen, können dies nicht über Parameter oder Rückgabewerte erledigen, da Tasks andere Tasks nicht wie Funktionen aufrufen können. Weil Tasks aber im selben Speicherraum ausgeführt werden, können Sie gemeinsame Werte in globale Variablen legen. Dies ist einer der wesentlichen Anwendungen für globale Variablen, auf die man ansonsten in der C-Programmierung ja möglichst verzichten soll.
Da Tasks aber unabhängig voneinander sind (sich also nicht gegenseitig aufrufen) und vom Betriebssystem über Funktionszeiger abgearbeitet werden, ist für den Compiler beim Übersetzungsvorgang nicht ersichtlich, dass die Funktionen, die ja letztlich hinter den Tasks stehen, einen Zusammenhang haben. Das bedeutet, dass ein optimierender Compiler beispielsweise bei einer globalen Variable in einem Task nur lesende Zugriffe erkennt. Dies entspricht der folgenden beispielhaften Situation:

```
uint8_t globVar = 0;

void task1() {
    while(1) {
    if (globVar == 1) {
        // tue irgendetwas, wenn globVar 1 ist
    }
    // Rest des Schleifenrumpfes
    }
}
```

Die globale Variable **globVar** soll nun in einem anderen Task verändert werden, beispielsweise auf 1 gesetzt werden. Da ein Compiler beim Übersetzen versucht zu ergründen, welche Abhängigkeiten zwischen Variablen und Code entstehen, um den Code zu optimieren, aber es keine direkte Abhängigkeit zu einem zweiten Task gibt (da der ja nicht explizit aufgerufen wird), sieht es für den Compiler so aus, als existieren nur lesende Zugriffe auf eine globale Variable, die niemals beschrieben wird und immer den Wert 0 hat. Wenn dem so wäre, dann würde **globVar** immer 0 sein und die Bedingung im **if** in **task1** niemals wahr werden. Folglich würde der Code, der im **if** steht, auch niemals ausgeführt werden. Ein optimierender Compiler (und die setzen wir immer ein, da Programme ja möglichst effizient sein sollen) würde die globale Variable und die **if**-Anweisung aus dem Programm entfernen. Es würde also übrig bleiben:

```
void task1() {
    while(1) {
    // Rest des Schleifenrumpfes
    }
}
```

Wenn nun in einem zweiten Task die globale Variable auf 1 gesetzt werden würde, hätte dies keinen Einfluss mehr auf die Ausführung von **task1**, was wir aber wollten; das Programm würde sich nicht mehr so verhalten, wie vom Programmierer gewünscht und - noch viel schlimmer - es ist am Sourcecode nicht zu erkennen, warum. Diese Arten von Fehler sind die schlimmsten, die einem Programmierer begegnen können, da er im Sourcecode selbst keinen einzigen Hinweis darauf finden kann. Eine Möglichkeit der Verhinderung wäre, den Compiler davon abzuhalten, zu optimieren. Dies ist aber keine brauchbare Alternative, da wir effizienten Code von einem Compiler erhalten wollen. Wir müssen also dem Compiler auf irgendeine Art und Weise mitteilen, dass die globale Variable **globVar** nicht nur von task1 benutzt wird, sondern auch von anderen (von wem ist dabei vollkommen egal). Dies ermöglicht die Programmiersprache C durch ein Schlüsselwort, welches Sie in PG1 schon kennengelernt haben: **volatile**
volatile besagt genau dies und hält den Compiler davon ab, diese Stellen zu optimieren, also genau das was wir möchten. Eine kleine Veränderung des Codes bewirkt also Großes:

```
volatile uint8_t globVar = 0;

void task1() {
    while(1) {
    if (globVar == 1) {
        // tue irgendetwas, wenn globVar 1 ist
    }
    // Rest des Schleifenrumpfes
    }
}
```

Fazit: globale Variablen, die Sie über Funktionen teilen, benötigen keinen weiteren Modifikator, da der Compiler die Abhängigkeiten erkennt. Globale Variablen, die Sie über Tasks teilen, benötigen den Modifikator **volatile**, ansonsten funktioniert es potentiell nicht.

Kritische Bereiche

Dieses Kapitel ist inbesondere für Übungszettel 3 wichtig, in der parallele Tasks sich gegenseitig über einen von Ihnen zu implementierenden Mechanismus synchronisieren sollen.

Wie im letzten Abschnitt gesehen, können globale Variablen als Austauschort von Werten zwischen verschiedenen Tasks fungieren. Sind die Bearbeitungsaufwände für die Inhalte der Variablen jedoch komplex (sprich: mehr als eine Assembler-Anweisung), dann kann es zu Problemen mit der Konsistenz der Daten geben. Der Grund liegt darin, dass das Betriebssystem die einzelnen Tasks zu beliebigen Zeit und beliebigen Stellen unterbrechen und mit der Abarbeitung eines anderen Tasks beginnen kann. Wenn diese Unterbrechungen und die Reihenfolgen der Tasks unglücklich liegen, kann unter Umständen ein Datensatz, der nur zur Hälfte gelesen wurde, geändert werden. Bekommt dann der ursprüngliche Tasks wieder Rechenzeit und setzt den Lesevorgang fort, so hat sich inzwischen womöglich das Datum geändert, aber der lesende Prozess bekommt davon nur die Hälfte mit.

Zur Illustration des Problems der folgende Code: Wir haben eine globale Variable, die eine Struktur ist. Diese Struktur enthält 2 Elemente, eine Kundennummer und einen Preis. Ist die Kundennummer auf 0 gesetzt, so passiert nichts weiter. Wird die Kundennummer hingegen auf einen Wert ungleich 0 gesetzt, so hat ein Kunde (der mit der Kundennummer) etwas gekauft und trägt im zweiten Element der globalen Strukturvariable den Preis ein. Für unser Beispiel nehmen wir nun einen Task, der den Händler darstellt und 2 Tasks für zwei unterschiedliche Kunden mit unterschiedlicher Kundennummer. Der Händlertask beobachtet die globale Variable in deren Kundennummer und sobald dort etwas anderes als 0 drin steht, wird eine Ausgabe mit Kundennummer und Preis gemacht. Anschließend wird die Kundennummer vom Händlertask wieder auf 0 gesetzt, um anzuzeigen, dass diese Kauftransaktion erledigt ist.
Die beiden Kundentasks hingegen kaufen zyklisch etwas. Dazu setzen Sie zunächst die Kundennummer auf ihren spezifischen Wert und anschließend den Preis. Ist das geschehen, legen sie sich für eine bestimmte Zeit schlafen, bevor der Kaufprozess von neuem startet.

Im folgenden sehen Sie den Sourcecode für dieses Beispielprogramm (der Übersicht halber sind hier die 3 Tasks nebeneinander gestellt und farbig markiert):

```
struct einkauf_t {
uint8_t KundenNr;
uint16_t Preis;
}

volatile struct einkauf_t einkauf = {0,0};
```

```
void haendlerTask() {
char msg[20];

while(1) {
  if (einkauf.KundenNr != 0) {
    int kID = (int) einkauf.KundenNr;
    int p = (int) einkauf.Preis;
    einkauf.KundenNr = 0;
    sprintf(msg,"Kunde %d zahlt
%d",kID,p);
    NNXT_LCD_DisplayStringAtLine(0,
msg);
  }
  Delay(100);
  }
}
```

```
void kunde1Task() {

  while(1) {
    einkauf.KundenNr =
123;
    einkauf.Preis = 20;
    Delay(400);
  }
}
```

```
void kunde2Task() {

  while(1) {
    einkauf.KundenNr =
456;
    einkauf.Preis = 50;
    Delay(930);
  }
}
```

```
int main() {
CreateAndStartTask(haendlerTask);
CreateAndStartTask(kunde1Task);
CreateAndStartTask(kunde2Task);
StartScheduler();
return 0;
```

Nun betrachten wir die folgende Situation: Alle 3 Tasks sind bereit mit ihrer Ausführung weil sie aus ihrem jeweiligen Delay zurückgekommen sind. Das Betriebssystem wählt nun zuerst **kunde1Task** aus. Die führt die den Schleifenrumpf seiner while-Schleife aus, setzt also die globale Variable einkauf auf die Werte {123,20} und geht anschließend durch Delay schlafen. Dann wählt das Betriebssystem den Task **haendlerTask** aus, der zunächst die Kundennummer in der globalen Variable auf 0 prüft. Da diese durch **kunde1Task** auf 123 gesetzt wurde, wird **haendlerTask** die Anweisungen im if-Block ausführen. Er liest die Kundennummer in eine lokale Variable. Statt nun den folgenden Code weiterausführen zu können, wird **haendlerTask** vom Betriebssystem unterbrochen und **kunde2Task** wird aktiviert. Dieser schreibt eine neue Kundennummer und einen neuen Preis in die globale Variable und legt sich anschließend schlafen. Nun kommt das Betriebssystem wieder ins Spiel und lässt den Task **haendlerTask** seinen noch verbliebenen Code bis zu dessen Delay ausführen (die anderen beiden Tasks befinden sich ja in ihrem Delay und benötigen zunächst keine weitere Rechenzeit).
Damit ergibt sich der folgende Ablauf (die farblichen Hervorhebungen kennzeichnen den Code der unterschiedlichen Tasks wie oben, Farbwechsel bedeuten, dass das Betriebssystem einen Task unterbrochen und einem anderen Rechenzeit zugewiesen hat):

```
    einkauf.KundenNr = 123;
    einkauf.Preis = 20;
    Delay(400);
```
kunde1Task setzt globale Variable auf seine Werte {123,20}

```
  if (einkauf.KundenNr != 0) {
    int kID = (int) einkauf.KundenNr;
```
haendlerTask liest globale Variable die enthält jetzt {123,20}
kID wird auf 20 gesetzt

```
    einkauf.KundenNr = 456;
    einkauf.Preis = 50;
    Delay(930);
```
kunde2Task setzt globale Variale auf seine Werte {456,50}

```
    int p = (int) einkauf.Preis;
    einkauf.KundenNr = 0;
    sprintf(msg,"Kunde %d zahlt %d",kID,p);
    NNXT_LCD_DisplayStringAtLine(0, msg);
  }
  Delay(100);
```

haendlerTask ist wieder dran
globale Variable ist jetzt {456,50}
kID ist immer noch 123
Ausgabe:

Kunde 123 zahlt 50

Sie sehen, dass bei diesem Ablauf 2 Probleme aufgetreten sind:

- Die Ausgabe ist falsch. Es wird eine Transaktion für den Kunden mit der Kundennummer 123 und einem Preis von 50 ausgegeben, obwohl **`kunde1Task`** (der die Kundennummer 123 hat) einen Preis von 20 eingestellt hatt
- Die Transaktion von **`kunde2Task`** (mit der Kundennummer 456 und einem Preis von 50) wird übersehen (da **`haendlerTask`** nach dem Lesen der globalen Variable die Kundennummer wieder auf 0 zurückgestellt hat)

Woran liegt das? Die Antwort auf diese Frage ist die Unterbrechung des Lesevorganges der globalen Variable in **`haendlerTask`**. Es wird von der ersten Transaktion nur die Kundennummer gelesen und nach der Unterbrechung wird mit dem Preis fortgefahren, obwohl sich in der Zwischenzeit sowohl Kundennummer als auch Preis geändert haben.

Um derartige Inkonsistenzen zu vermeiden, müsste man dem Betriebssystem verbieten, die Tasks in bestimmten Situationen zu unterbrechen. So würde ein Verbot der Unterbrechung beim Lesen der globalen Variable in **`haendlerTask`** diese Situation entschärfen.

Diese Codeabschnitte, die nicht unterbrochen werden dürfen, nennt man **kritische Bereiche**. Da diese kritischen Bereiche insbesondere in Synchronisationsmechanismen bei parallelen System häufig vorkommen, bieten Betriebssysteme Funktionen an, mit denen man diese Bereiche markieren kann. Befindet sich der Code innerhalb dieses Bereiches, kann das Betriebssystem den Task nicht unterbrechen, sondern muss warten, bis der kritische Bereich wieder verlassen wurde.

Das NNXT-Betriebssystem bietet zu diesem Zweck zwei Funktionen:

```
void taskENTER_CRITICAL();

void taskEXIT_CRITICAL();
```

Mit der Funktion **`taskENTER_CRITICAL()`** wird ein kritsicher Bereich eingeleitet, mit der Funktion **`taskEXIT_CRITICAL()`** wird der kritische Bereich beendet. Code zwischen diesen beiden Funktionen kann nicht vom Betriebssystem unterbrochen werden. Dies können wir nun auf unser Beispielprogramm anwenden und die besonders anfälligen Codeabschnitte in kritische Bereiche verwandeln:

```
struct einkauf_t {
uint8_t KundenNr;
uint16_t Preis;
}

volatile struct einkauf_t einkauf = {0,0};
```

```
void haendlerTask() {
char msg[20];

while(1) {
  if (einkauf.KundenNr != 0) {
    taskENTER_CRITCAL();
    int kID = (int) einkauf.KundenNr;
    int p = (int) einkauf.Preis;
    einkauf.KundenNr = 0;
    taskEXIT_CRITICAL();
    sprintf(msg,"Kunde %d zahlt
```

```
void kunde1Task() {

  while(1) {
    taskENTER_CRITICAL();
    einkauf.KundenNr =
123;
    einkauf.Preis = 20;
    taskEXIT_CRITICAL();
    Delay(400);
  }
}
```

```
void kunde2Task() {

  while(1) {
    taskENTER_CRITICAL();
    einkauf.KundenNr =
456;
    einkauf.Preis = 50;
    taskEXIT_CRITICAL();
    Delay(930);
  }
}
```

```
%d",kID,p);
    NNXT_LCD_DisplayStringAtLine(0,
msg);
  }
  Delay(100);
  }
}


int main() {
CreateAndStartTask(haendlerTask);
CreateAndStartTask(kunde1Task);
CreateAndStartTask(kunde2Task);
StartScheduler();
return 0;
```

Betrachten wir mit diesem um kritische Bereiche erweiterten Code die Situation von oben. Wieder möchte das Betriebssystem **haendlerTask** unterbrechen, kann dies nun aber erst, nachdem der Programmablauf den kritischen Bereich verlassen hat. Daraus ergibt sich ein leicht geänderter Ablauf:

```
    taskENTER_CRITICAL();
    einkauf.KundenNr = 123;
    einkauf.Preis = 20;
    taskEXIT_CRITICAL();
    Delay(400);
```

kunde1Task setzt globale Variable
auf seine Werte {123,20}

```
  if (einkauf.KundenNr != 0) {
    taskENTER_CRITCAL();
    int kID = (int) einkauf.KundenNr;
    int p = (int) einkauf.Preis;
    einkauf.KundenNr = 0;
    taskEXIT_CRITICAL();
```

haendlerTask liest globale Variable
die enthält jetzt {123,20}
kID wird auf 123 gesetzt
p wird auf 20 gesetzt
kann erst wieder nach taskEXIT_CRITICAL() unterbrochen werden

```
    taskENTER_CRITICAL();
    einkauf.KundenNr = 456;
    einkauf.Preis = 50;
    taskEXIT_CRITICAL();
    Delay(930);
```

kunde2Task setzt globale Variale
auf seine Werte {456,50}

```
    sprintf(msg,"Kunde %d zahlt %d",kID,p);
    NNXT_LCD_DisplayStringAtLine(0, msg);
  }
  Delay(100);
```

haendlerTask ist wieder dran
globale Variable ist jetzt {456,50}
kID ist immer noch 123, p immer noch 20
Ausgabe:
Kunde 123 zahlt 20

Sie sehen, dass **haendlerTask** nachwievor unterbrochen wird, allerdings erst, nachdem der Inhalt der globalen Variable vollständig gelesen wurde. Aus diesem Grund sind die Inhalte der lokalen Variablen konsistent und die Ausgabe ist ebenfalls korrekt. Da nun das Zurücksetzen der Kundennummer durch **haendlerTask** vor der Ausführung von **kunde2Task** erfolgt, geht die Transaktion von **kunde2Task** auch nicht mehr verloren und wird entdeckt, sobald **haendlerTask** das nächste Mal die globale Variable abfragt.

WICHTIG: Kritische Bereiche unterbinden Taskwechsel durch das Betriebssystem, aber nicht nur die, sondern auch andere Unterbrechungen. Insbesondere kann beispielsweise die Systemuhr nicht aktualisiert werden und eingehende Signale von den Ports

können ebenfalls nicht erkannt werden. Das gleiche gilt für die Ansteuerung der Motoren, da die dafür notwendigen PWM-Signale nicht mehr erzeugt werden können. **Es ist daher ungeheuer wichtig, die kritischen Bereiche so selten wie möglich und insbesondere so klein wie möglich zu halten! Arbeiten Sie also mit sehr viel Umsicht an den kritischen Bereichen!**