AUFGABENBLOCK 2 24. März 2004

Teilaufgabe 1: Geigerzähler für Licht

Kennenlernen und Befassen mit BrickOS: Architektur, API, Task Management (z.B. Scheduling), allokieren von Ressourcen.
Implementieren Sie den „Geigerzähler für Licht" aus Aufgabe 1 in „C" unter BrickOS unter Verwendung der BrickOS API. Funktionalität: Erkennen von Licht mit dem Lichtsensor. Hohe Lichteinstrahlung hohe Piepfrequenz des Systempiepers, geringe Lichteinstrahlung geringe Frequenz. Anzeige des gemessenen Wertes als Integer am LCD Display.

Teilaufgabe 2: Static Cycling Scheduling

Bauen Sie einen fahr- und lenkfähigen Legoroboter. Der Roboter soll durch ein Kontrollprogramm, das zufällig Fahranweisungen generiert, gesteuert werden. Wenn der Roboter auf ein Hindernis stößt soll zurückgesetzt werden und durch (wiederum zufällige) Wahl einer neuen Fahrtrichtung versucht werden, das Hindernis zu umsteuern. Das Erkennen von Hindernissen erfolgt mittels der Berührungssensoren.

Das Kontrollprogramm soll basierend auf dem Prinzip des „static cycling scheduling" aufgebaut werden: Die zur Ausführung der Kontroll bzw. Steuerungsaufgabe notwendigen Berechnungen werden periodisch (zyklisch) wiederholt ausgeführt. Dazu werden in der Hauptschleife die einzelnen Berechnungstasks aufgerufen:

```
while(1)  /* Scheduler loop */
{
   task1(...);    /* Function call */
   task2(...);    /* Function call */
   task3(...);    /* Function call */
   ...
   taskN(...);    /* Function call */
   delay_until_next_time_unit();  /* Periodic schedule */
}
```

Jeder Task erledigt seine Berechnung und kehrt unmittelbar zur Hauptschleife zurück! In solchen Modellen ist kein busy waiting keine ‚tight loops' und keine interne Synchronisation erlaubt (simple tasks).

Das Kontrollprogramm soll in folgende Teilaufgaben (Tasks) untergliedert werden: `motor_control`, `forward_run`, `direction_change` und `obstacle_handling`. `motor_control` ist der einzige Task, der direkt auf die Motoren zugreifen darf. Er erhält Steueranweisungen von allen anderen Tasks. Jede Steueranweisung wird solange ausgeführt, bis ihre Dauer abgelaufen ist oder die Steueranweisung durch die eines anderen Tasks überschrieben wurde. `forward_run` versucht das Fahrzeug in geradeaus Fahrt zu halten. `direction_change` generiert zufällig Richtungsänderungen (geringe Wahrscheinlichkeiten wählen, sonst kommt der Roboter nicht von der Stelle!) und `obstacle_handling` erkennt ob ein Hindernis berührt wurde und versucht das Hindernis durch Abdrehen des Fahrzeuges zu umgehen.

Dem `motor_control` Task werden die Steuerbefehle über folgende Kommandostruktur weiter gegeben:

```
/* The duration in ms for a time unit */
#define TIME_UNIT_IN_MSEC 100

/* The different directions of the vehicle (add more yourself) */
enum dir_t {turn_left, turn_right, forward, backward, stop, idle};

/* The type of the global data structure */
struct driving_command {
  int        priority;
  enum dir_t direction;
  int        speed;
  long int   duration;
}
```

```
/* Function for changing the command in the global data structure */
void change_driving_command(int prio, enum t_dir dir, int speed, long int dur)
{
  /* Update driving_command data structure if allowed */
}
```

Jeder der Tasks hat eine bestimmte Priorität betreffend die Antriebskontrolle. Die Priorität gibt an, ob ein Steuerkommando dieses Tasks im Vergleich zu Steuerkommandos anderer Tasks bevorrangt ist: Wie würden Sie z.B. die Priorität des Tasks `forward_run` in Relation zu `obstacle_handling` setzen? Die Implementierung der Prioritäten könnte einfach über eine Funktion `change_driving_command()` erfolgen, die eine definierte Schnittstelle zum Zugriff auf die Motorkontrollstruktur wie folgt darstellen könnte: Für jedes neu zu setzende Steuerkommando wird geprüft, ob dessen Priorität höher ist als das des gerade ausgeführten. Wenn nicht wird das aktuelle Kommando nicht ersetzt.

Die Dauer jedes Steuerkommandos als auch jene des Kontrollzyklus (1 Schleifendurchlauf des Schedulers) wird in *Zeiteinheiten* angegeben. Die Dauer der Zeiteinheit wird in der Zeile `#define TIME_UNIT_IN_MSEC xxx` spezifiziert.
Jedes Mal, wenn `motor_control` ausgeführt wird werden die Steuerkommandos für die Motoren gesetzt und die Dauer für das aktuelle Kommando um eine Zeiteinheit verringert. Das aktuelle Kommando (die aktuelle Fahrtrichtung) soll auch auf dem LCD angezeigt werden. Ist die Dauer für das aktuelle Kommando abgelaufen werden die Motoren angehalten und die Kommandopriorität auf den Wert niedrigster Priorität gesetzt. Um die Dauer für die periodische Abarbeitung des Kontrollprogrammes und damit die Dauer für Ausführung der Kommandos exakt auf die Zeiteinheit einzustellen sollte die Funktion `msleep()` verwendet werden.

Der Parameter `speed` in der Kontrollstruktur erlaubt zusätzlich zu der Richtungsangabe Schnell- und Langsamfahrt. Die Interpretation ist dem `motor_control` überlassen.

Hinweis: Ist es notwendig, die Kommandodatenstruktur, auf die alle Tasks zugreifen, gesondert zu schützen z.B. durch Semaphore?