fhwedel
UNIVERSITY OF APPLIED SCIENCES

Praktikum Echtzeitsysteme

Einführung
Windows Simulator

Timm Bostelmann

Wedel, den 12. Juni 2020

1 Einleitung

Das Praktikum Echtzeitsysteme dient der Vertiefung und Anwendung der Vorlesungsinhalte. Zu diesem Zweck wird eine Aufgaben gestellt, die in Teamarbeit bis zum Abgabetermin bearbeitet werden muss. Die Bearbeitung der Aufgabe erfolgt unter Verwendung des Echtzeitsystems µC/OS-II im Windows Simulator. Die Entwicklungsumgebung ist Visual Studio 2019.
Diese Einführungsaufgabe soll Ihnen den Einstieg ins praktische Arbeiten in der vorhandenen Umgebung erleichtern. Darüber hinaus sind Eigeninitiative und die selbstständige Erarbeitung der Inhalte gefragt. Bei schwerwiegenden Problemen oder Fragen zur Aufgabenstellung stehe ich Ihnen natürlich gerne beratend zu Seite. Die Bearbeitung dieser Einführungsaufgabe erfolgt in Eigenverantwortung, sie wird nicht abgenommen oder testiert.

2 Vorbereitung

Die Entwickler von µC/OS-II bieten einen Simulator an, der es ermöglicht die µC/OS-II Bibliothek unter Windows zu benutzen. Dazu wird Visual Studio 2019 und die eigentliche µC/OS-II Software benötigt.

2.1 Entwicklungsumgebung

Sie können die *Community* Edition von Visual Studio 2019 unter folgender Adresse herunterladen.

https://visualstudio.microsoft.com/de/vs/community/

Installieren Sie die Entwicklungsumgebung und wählen Sie dabei das *Desktopentwicklung mit C++ -Plugin* aus.

2.2 µC/OS-II

Die benötigte Software sowie ein Beispielprojekt befindet sich in Form einer *.zip* Datei, im Moodle unter *Praktikum Echtzeitsysteme*. Laden Sie diese herunter und entpacken Sie sie.

2.3 Projektstruktur

Die Einführungsaufgabe basiert auf dem Projekt *Echtzeitsysteme.sln*, welches sich unter folgendem Pfad befindet.

`\Microsoft\Windows\Kernel\Echtzeitsysteme\VS`

Öffnen Sie zunächst das *Echtzeitsysteme* Projekt in Visual Studio 2019. Nach erfolgreichem Laden des Projektes, erscheint es unter *Echtzeitsysteme* im *Projektmappen-Explorer*. Dabei ist es in folgende Ordner (in Visual Studio 2019 *Filter* genannt) unterteilt.

BSP Bord Supply Package

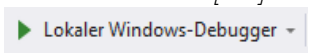
Lib Enthält die µC/OS-II Bibliotheken und die Quellen für den Simulator

Kernel Enthält Konfigurationen für den Kernel

App Enthält den Applikationsquellcode. Hier können Sie weitere Quellen hinzufügen um die Aufgabe zu lösen.

- **app.c** enthält den C Eintrittspunkt. Weitere Tasks können hier hinzugefügt oder in andere Dateien ausgelagert werden.
- **app_cfg.h** enthält die direkt zur Anwendung gehörige Konfiguration, wie Prioritäten und Stackgrößen der Tasks.
- **os_cfg.h** enthält die Betriebssystem-Konfiguration. Hier können die unterschiedlichen Dienste des Systems einzeln (de)-aktiviert und konfiguriert werden. Für die ersten Schritte sollte allerdings die vorhandene Konfiguration ausreichen.

2.4 Prüfen des Vorlageprojektes

Starten Sie nun das Vorlageprojekt durch Drücken der Taste *[F5]* oder durch Klicken auf den grünen Pfeil der Standard Symbolleiste: 

. Nach erfolgreichem Kompilieren, sollte folgender Inhalt in der Konsole angezeigt werden:

```
OSTick    created, Thread ID 11520
Task[63] created, Thread ID  9628
Task[62] created, Thread ID  1456
Task[61] created, Thread ID  7780
Task[61] 'uC/OS-II Tmr' Running
Task[62] 'uC/OS-II Stat' Running
Task[63] 'uC/OS-II Idle' Running
```

Ist dies nicht der Fall, ist wahrscheinlich die Vorlage defekt. Melden Sie sich in diesem Fall bitte bei mir. Die *Thread ID's* sind bei Ihnen höchstwahrscheinlich andere und ist kein Grund zur Sorge. Das Fenster lässt sich auf normalem Wege schließen.

3 Aufgabe

Ziel der Aufgabe ist es, ein Erzeuger- / Verbrauchersystem abzubilden. In diesem System werden genau zwei Benutzertasks laufen. Der Task mit dem Namen *App_TaskCreate* entspricht dem Erzeuger. Das eigentliche *Erzeugen* eines Datums erfolgt durch den Druck einer Taste auf der Tastatur. Der Task *App_TaskCreate* nimmt das Zeichen entgegen und sendet es an den Verbrauchertask *App_TaskConsume*.

3.1 Intertask-Kommunikation

Der Austausch von Daten zwischen Erzeuger und Verbraucher soll in diesem einfachen Beispiel über eine globale Variable erfolgen. Diese Variable ist mit einem Semaphor gegen Mehrfachzugriff zu schützen. Zusätzlich müssen sich Erzeuger und Verbraucher gegenseitig das Lesen und Schreiben signalisieren.

Sempahoren (und andere Kommunikationsdienste) sind unter µC/OS-II von Typ *OS_EVENT **. Für die Deklaration der drei Semaphorenvariablen fügen Sie die folgenden drei Zeilen unter dem Punkt *LOCAL GLOBAL VARIABLES* in die Datei *app.c*:

```
OS_EVENT* SemMutex;
OS_EVENT* SemRead;
OS_EVENT* SemWrite;
```

Definieren Sie direkt im Anschluss die gemeinsame globale Variable. Der Prefix SV steht für *shared variable*.

```
INT16S SVKey;
```

Die Erzeugung und Initialisierung eines Semaphors erfolgt mit der Funktion *OSSemCreate()*. Der Parameter gibt den Zählerstand zum Zeitpunkt der Initialisierung an. Fügen Sie die folgenden drei Zeilen in die *main()* Routine des Programms ein und ergänzen Sie für den Semaphor SemMutex einen sinnvollen Startwert:

```
SemMutex = OSSemCreate( );
SemRead = OSSemCreate(0);
SemWrite = OSSemCreate(1);
```

Prüfen Sie an dieser Stelle erneut, ob sich das Projekt starten lässt um mögliche Fehler frühzeitig zu erkennen. Es sollte sich nichts am Verhalten des Programms geändert haben.

3.2 Der Erzeugertask

Für jeden Task sind in der Datei *app_cfg.h* die Priorität und die größe des Stacks zu definieren. Erinnern Sie sich, dass die Prioritäten unter µC/OS-II eindeutig sein müssen und absteigend vergeben werden. Ein niedrigerer Wert entspricht also einer höheren Priorität. Wir wählen für den Erzeugertask die Priorität 10 und stellen ihm 512 Zeichen Stack zur Verfügung. Fügen Sie hierfür die folgenden beiden Zeilen an den entsprechenden Stellen in die Datei *app_cfg.h* ein:

```
#define  APP_OS_CFG_CREATE_TASK_PRIO                    10
#define  APP_OS_CFG_CREATE_TASK_STK_SIZE               512
```

In der Datei *app.c* muss nun der Speicher für den Taskstack reserviert werden. Hierfür wird ein globales Array vom Basistyp *OS_STK* mit der gewünschten Anzahl von Elementen angelegt. Fügen Sie die folgende Zeile unter dem Punkt *LOCAL GLOBAL VARIABLES* der Datei app.c ein:

```
OS_STK AppCreateTaskStk[APP_OS_CFG_CREATE_TASK_STK_SIZE];
```

Die Taskfunktion machen Sie durch Einfügen des Funktionsprototypens unter dem Punkt *FUNCTION PROTOTYPES* bekannt:

```
static void App_TaskCreate(void* p_arg);
```

Jeder Task ist zwar durch seine Priorität eindeutig gekennzeichnet, es ist jedoch empfehlenswert jedem Task, einen menschenlesbaren Namen zu geben. Hierfür wird in dem C-Präprozessor *#if OS_TASK_NAME_EN > 0u*, in der *main()*, folgende Zeile hinzugefügt:

```
OSTaskNameSet(APP_OS_CFG_CREATE_TASK_PRIO,
              (INT8U*)"Creator Task",
              &os_err);
```

Die hierbei verwendete *OS_TASK_NAME_EN* Konstante ist standartmäßig gesetzt.

3.2.1 Starten der Erzeugertasks

Der Erzeugertask wird mit der Systemfunktion *OSTaskCreate (task, p_arg, ptos, prio)* aus der *main()* erzeugt. Die Parameter haben folgende Bedeutung:

task ist ein Funktionszeiger auf die Taskfunktion *App_TaskCreate*

p_arg ist ein typenloser Zeiger auf ein beliebiges Datum, das dem Task übergeben wird. Dies kann sinnvoll sein, wenn mehrere Tasks aus dem selben Code bzw. der selben Funktion erzeugt werden: *(void *) 0*, da in diesem Fall nicht benötigt

ptos ist ein Zeiger auf den Anfang des Stacks:
&AppCreateTaskStk[APP_OS_CFG_CREATE_TASK_STK_SIZE-1].
Da der Stack rückwärts wächst, muss die letzte Adresse des Feldes angegeben werden.

prio ist die Priorität des zu erzeugenden Tasks: *APP_OS_CFG_CREATE_TASK_PRIO*

Fügen Sie den folgenden Funktionsaufruf an der entsprechenden in der *main()* hinzu:

```
OSTaskCreate(App_TaskCreate,
        (void*)0,
        &AppCreateTaskStk[APP_OS_CFG_CREATE_TASK_STK_SIZE - 1],
        APP_OS_CFG_CREATE_TASK_PRIO);
```

Hierbei ist zu beachten, dass alle Tasks vor dem Aufruf der *OSStart()* erzeugt werden müssen.

3.2.2 Code des Erzeugertasks

Es folgen der Code des Erzeugertasts und seine Beschreibung. Machen Sie sich die Funktionsweise des Codes klar und fügen Sie ihn in die *app.c* ein.

```
1  /*
2   * Der Erzeugertask "erzeugt" bei jedem Tastendruck ein Zeichen und gibt
3   *  es auf dem Bildschirm aus.
4   *
5   *  Arguments : p_arg nicht verwendet
6   */
7  static void App_TaskCreate(void* p_arg) {
8         INT8U err;
9         INT16S key;
10
11        while (1) {
12
13                // Erzeuge Daten
14                while (!PC_GetKey(&key)) {
15                        OSTimeDlyHMSM(0, 0, 0, 100);
16                }
17
18                // Sende Daten
19                OSSemPend(SemWrite, 0, &err);
20                OSSemPend(SemMutex, 0, &err);
21                printf("Schreibe: %c\n", key);
22                SVKey = key;
23                OSSemPost(SemMutex);
24                OSSemPost(SemRead);
25        }
26 }
```

8: Die Variable *err* nimmt Rückgabecodes der Systemfunktionen auf.

9: Die Variable *key* speichert die gedrückte Taste zwischen, bis sie versendet wird.

14: Die Funktion *PC_GetKey* wird gepollt bis eine Taste gedrückt wurde. Der Tastencode befindet sich dann in der Variable *key*.

15: Zwischen den Abfragen wird jeweils *100ms* gewartet. Es handelt sich hierbei nicht um Busywaiting, es können also währen des Wartens andere Tasks ausgeführt werden.

19: DOWN Operation auf den Semaphor *SemWrite*. Der Timeout 0 bedeutet, dass beliebig lange gewartet wird. Der Fehlercode *err* wird in diesem einfachen Beispiel nicht ausgewertet.

20: DOWN Operation auf den Semaphor *SemMutex*.

21: Ausgabe des gedrückten Zeichens *key* auf die Konsole.

22: Schreiben des gedrückten Zeichens *key* in die gemeinsame, globale Variable *SVKey*.

23: UP Operation auf den Semaphor *SemMutex*.

24: UP Operation auf den Semaphor *SemRead*.

3.2.3 Testen des Erzeugertasks

Starten Sie das Projekt und drücken Sie eine Taste. In der letzten Zeile des Kommandozeilenfensters sollte der Text *Schreibe:* gefolgt von dem gedrückten Zeichen erscheinen. Ein erneuter Tastendruck erzeugt keine Ausgabe, da der Puffer *SVKey* belegt ist und der Verbraucher, der diesen leeren könnte noch nicht existiert

3.3 Der Verbrauchertask

Ergänzen Sie dieses Beispiel selbstständig um einen Verbrauchertask. Vergessen Sie nicht die Deklaration und das Starten des Tasks. Es folgt eine mögliche Umsetzung des Verbrauchertasks in Pseudocode:

```
1  App_TaskConsume{
2  INT16S key;
3  while (1){
4      // Empfange Daten
5      DOWN(SemRead);
6      DOWN(SemMutex);
7      key = SVKey;
8      WRITE("Lese: ");
9      WRITELN(key);
10     UP(SemMutex);
11     UP(SemWrite);
12     // Verbrauche Daten
13     DELAY(5s);
14     }
15 }
```

4 Abschlusstest

Wenn Ihr Verbraucher ordnungsgemäß funktioniert, werden direkt nach dem ersten Tastendruck die beiden Zeilen *Schreibe: ...* und *Lese: ...* ausgegeben. Prüfen Sie ob das Verhalten des Programms auch bei mehrfachen Tastendrücken innerhalb kurzer Zeit Ihren

Erwartungen entspricht. Bedenken Sie hierbei, dass gedrückte Tasten bereits vom Betriebssystem in einer Warteschlange gespeichert und dann in der richtigen Reihenfolge an das Programm weitergeben werden.