MiniOS

Ein Minibetriebssystem für Mikrocontroller

Konfiguration und Anwendung von MiniOS

Version 2014, Hj. Kern, Winterthur
Druck 2014

Inhalt

MiniOS *Ein Minibetriebssystem für Mikrocontroller* ... 1
Inhalt ... 2
Abkürzungen und Begriffe ... 3
Multitasking-OS für Mikrocontroller ... 3
Sequentielles Programm ... 4
Multitasking-OS ... 4
MiniOS Konfiguration und Anwendung ... 6
Was ist MiniOS? ... 6
Eine Applikation unter MiniOS ... 8
Ablauf einer MiniOS-Applikation ... 8
MiniOS-Installation und Dateien ... 9
Installation ... 9
MiniOS-Dateien ... 9
Header-Datei os_minios.h ... 9
Das Hauptprogramm main() ... 10
Anwender-Task konfigurieren und implementieren ... 10
Task, TCB und Prioritäten ... 11
Task Status (task state) ... 12
MiniOS-Scheduler: 2-Level-Scheduling ... 13
Ebene "Ereignisbehandlung" ... 13
Ebene "Hintergrundbehandlung" ... 13
Die Makros SCHEDULE() und SCHED_PRIO() ... 13
Prioritätsmodus vor Round-Robin-Modus ... 14
OS Ruhezustand, System Idle Mode ... 15
OS-Systemzeit, System-Timing ... 16
Konfiguration ... 16
System-Timer-Interrupt, OS-Timer-Tick ... 17
Task-SW-Timer TIMER ... 17
Timer, Beispiele ... 18
OS und Anwendung initialisieren ... 19
Ereignisbehandlung und -steuerung ... 19
Ereignisüberlauf, Systemüberlastung ... 20
System-Timeout verhindern ... 20
MiniOS und Interrupts ... 21
Kommunikation und Synchronisation ... 23
Warum Semaphore und Messages? ... 23
Semaphore und Messages benutzen, Uebersicht ... 24
Semaphore ... 24
Semaphore in MiniOS ... 25
Ein zählender Byte-Semaphor... ... 25
Semaphor als Event-Flag ... 26
Semaphore zur Synchronisation von Tasks ... 26
Semaphor zur Zugriffssteuerung von Ressourcen, Mutex ... 27
Zählende Semaphore als Indikatoren für freie Ressourcen ... 27
Mailbox ... 28
Mailbox in MiniOS ... 29
Hook-Funktionen ... 30
User-Hooks ... 30
Debug-Hooks ... 31
Fehlerbehandlung, OS_Exception() ... 33
Fehlerbehandlung, vom Anwender definiert ... 34
Quellen, Literatur ... 34

Abkürzungen und Begriffe

HW	Hardware
SW	Software
OS	Operating System, Betriebssystem
RTOS	Real Time Operating System, echtzeitfähiges Betriebssystem
Polling	dauerndes Abfragen einer HW oder SW um deren Zustand zu erfassen, Gegensatz: Interrupt-Technik
ISR	Interrupt Service Routine
Task	Programmteil der eine fest zugewiesene Aufgabe erledigt (z.B. in Form einer Funktion)
API-Funktion	Eine Funktion die im OS implementiert ist und vom Anwender aufgerufen wird, API = Anwender-Programmier-Interface
Callback-Funktion	Eine Funktion, die vom OS aufgerufen wird, jedoch vom Anwender implementiert wurde (wie z.B. die Event-Handler im C++-Builder)
TCB	Task Control Block, eine Datenstruktur in der die Task-Daten eines OS verwaltet werden.
FIFO-Buffer	First-In-First-Out: ein Datenspeicher, bei dem die ältesten Daten zuerst ausgelesen werden, *Warteschlange (queue) oder Pipeline*.
LIFO-Buffer	Last-In-First-Out: ein Datenspeicher, bei dem die jüngsten Daten zuerst ausgelesen werden, *Stack*.

Multitasking-OS für Mikrocontroller

Bei vielen Arten von Prozesssteuerungen müssen feste Zeitgrenzen eingehalten werden. Die Steuersoftware muss "echtzeitfähig" sein und sich im Ablauf vorhersagbar verhalten.

Die Anwendungen verlangen eine *parallele Verarbeitung* von verschiedenen Funktionen: Eine Tastatur soll gleichzeitig während einer im Hintergrund laufenden Berechnung abgefragt und die Eingabe am Display angezeigt werden...

"Embedded-Systeme" - also Mikrocontroller-Anwendungen - sind meistens Ein-Prozessorsysteme. Der Prozessor kann sich nur einer Aufgabe gleichzeitig widmen. Eine Parallelverarbeitung ist nur "quasi-parallel": Die Prozessorzeit wird in kurze Zeitintervalle zerlegt, die den einzelnen Funktionen reihum zugeteilt werden.

Ein häufig realisierter Ansatz für eine Applikation ohne Multitasking-OS sieht dann etwa so aus:

Sequentielles Programm

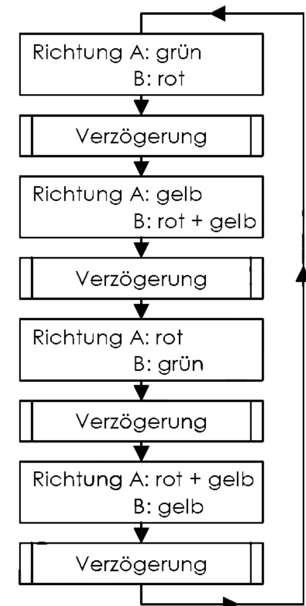
Beispiel, Bild - Verkehrsampel ausgeführt als sequentieller Main-Loop

Programm mit einer sequentiellen Main-Loop-Architektur, "Super-Loop":

Anwendungsfunktionen und Berechnungen werden bei jedem Loop-Durchlauf zyklisch aufgerufen und entsprechend einiger globaler Zustandsvariablen abgearbeitet.

Bei jedem Durchlauf werden Ein- und Ausgänge bedient ("polling system"), das Display neu "gefüttert" und die Zustandsvariablen dem aktuellen Stand angepasst. *Blockierende* Wartefunktionen sind eingefügt im Ablauf.

Falls Interrupts verwendet werden:
ISR-Funktionen übernehmen im Vordergrund zeitkritische Jobs und liefern neue Systemereignisse.



In der sequentiellen Main-Loop-Architektur laufen alle Programmteile in fester Reihenfolge hintereinander ab. Der Prozessor verbringt viel Zeit mit Aufrufen von momentan inaktiven Funktionen und dem Testen von Zustandsvariablen.

Die Anwendung läuft mit der *vollen Geschwindigkeit des Prozessors*, was für die Steuerung von Prozessen eine zeitliche Strukturierung erzwingt: die benötigten Timer- und Wartefunktionen blockieren das System oder müssen mittels Timer-Interrupts und globalen Variablen realisiert werden.

Der Programmcode (source code) - somit das Lösungskonzept - ist schlecht nachvollziehbar und unübersichtlich: Fehlerkorrekturen und Erweiterungen sind nur mit Mühe einzubringen.

Multitasking-OS

Ein *Multitasking-OS* behebt die Schwachstellen der sequentiellen Main-Loop-Architektur und arbeitet vereinfacht etwa nach folgenden Grundsätzen:

Prozessorzeit ist kostbar: Funktionen - ab jetzt "Tasks" genannt - welche momentan "nichts zu tun haben", weil sie nicht rechenbereit sind oder auf ein Ereignis warten, dürfen keine Prozessorzeit beanspruchen.

Warte- und Abfrage-Tasks (polling task functions) dürfen die Anwendung nicht blockieren (also keine Delay-Endlosschleifen zum Warten in der Anwendung!). Der Prozessor kann während den allfälligen Wartezeiten eines Tasks nämlich andere Tasks bearbeiten. Dazu stellt das OS nicht-blockierende Delay/Timer-Funktionen zur Verfügung.

Echtzeit-Abfragen: Abfrage-Tasks, welche z.B. eine Taste oder ein Signal zyklisch abfragen (polling), tun dies in einem fixen Zeitraster mittels OS-Timer, z.B. alle 20 ms. Damit bleibt zwischen den Abfragen genug Zeit übrig für die Bearbeitung anderer Tasks (während 20 ms ist schon eine Menge Programmcode ausführbar...).

Task-Status: Jede Task besitzt eine Art Zustandsetikette die dem Kernel mitteilt, was mit der jeweiligen Task zu tun ist.
Bearbeitet (d.h. ausgeführt) werden nur Tasks, deren momentaner Zustand READY (rechenbereit) ist. Wartet eine Task auf ein Ereignis, z.B. auf einen Delay-Timer der abläuft oder eine Nachricht von einer anderen Task, verbraucht er keine Prozessorzeit, da er sich im Zustand WAITING befindet und vom Kernel nicht "beachtet" wird.

Prioritäts-Scheduling (priority scheduling): Befinden sich mehrere Tasks im Zustand READY, wird zuerst diejenige Task abgearbeitet, die die höchste Priorität besitzt.
und/oder
Round Robin Scheduling: Die READY-Tasks werden nicht entsprechend der Reihenfolge ihrer Prioritäten, sondern in fester Reihenfolge hintereinander ausgeführt.

Kooperatives Multitasking: Die *Task entscheidet*, ob und wann die Kontrolle zurück an den Kernel gehen soll, um einer anderen Task Prozessorzeit einzuräumen (z.B. Aufruf von OS_Yield() oder SCHEDULE() im Task-Code).

Preemptives Multitasking: Der *Kernel entscheidet* auf Grund eintreffender Ereignisse, Interrupts, Task-Prioritäten etc., ob die Bearbeitung der aktuellen, zu Gunsten einer nächsten Task zu unterbrechen sei.

Bei einem *Task-Wechsel* (task context switching) wird die Task-Umgebung (task context), d.h. Stackpointer und Program-Counter der laufenden Task, im Speicher gesichert und der Kontext der nächsten auszuführenden Task geladen, um deren Bearbeitung weiterzuführen. Mittels Task-Context-Switching wird eine "quasi-parallele" Task-Ausführung möglich.

Tasks und Interrupts können miteinander *kommunizieren und sich synchronisieren* mittels OS-Funktionen: Semaphore (globale Event-Flags z.B. für die Interrupt-Auswertung) und Nachrichten (messages), die über eine Mailbox versandt und empfangen werden. Eine Task versendet eine Message oder signalisiert (aktiviert) einen Semaphor. Eine andere Task wartet - ohne die Anwendung zu blockieren - bis die Message in der Mailbox eintrifft oder bis der Semaphor aktiviert wird.

Ein *API (Anwender Programmier-Interface)* stellt Timer- und Kommunikationsfunktionen zur Verfügung. Der Anwender braucht sich nicht mehr um die Implementierung von grundlegenden Systemfunktionen zu kümmern.

Programmentwicklung: Eine Task kann grundsätzlich als einfaches sequentielles System entworfen und programmiert werden. Warteschleifen mit den OS-Timer-Funktionen sind nicht-blockierend. Dies vereinfacht die Entwicklung und Wartung der ganzen Applikation.

Listing - Beispiel einer Multitasking-Anwendung mit zwei Tasks

```
// Task_0
// Tasten abfragen
// ****************************
...

while (1) {

   OS_WaitTimer(2);
   SCHEDULE()
   FrageTastaturAb();

   if (Tastendruck) {
      // Entprellung
      OS_WaitTimer(2);
      SCHEDULE()
      TestUndSpeichere(KeyBuf);
      OS_PostMessage(KEY_DOWN);
   }

}
```

```
// Task_1
// Zeichen anzeigen
// *************************
...

while (1) {

   OS_WaitMessage(KEY_DOWN);
   SCHEDULE()
   Display(KeyBuf);

}
```

Zwei quasi-parallele Tasks kommunizieren miteinander mittels Messages. Im OS-Makro SCHEDULE() wird der Kernel aufgerufen, der weitere rechenbereite Tasks ausführen lässt. Beim Wiederaufruf der Task durch den Kernel setzt der Ablauf beim ersten Befehl nach SCHEDULE() fort.

MiniOS Konfiguration und Anwendung

Was ist MiniOS?

- Ein "Mini"-Multitasking-RTOS für Mikrocontrolleranwendungen in Embedded-Systemen.
- Für 8-Bit-Mikrocontroller mit begrenzten Ressourcen konzipiert.
- MiniOS ist sehr einfach zu konfigurieren und anzuwenden.

- MiniOS benötigt wenig Speicherplatz: eine Beispielkonfiguration mit
  - 4 Task-Funktionen (mit je 2 Kontext-Switches)
  - eine globale Mailbox für Byte-Messages, 16 Byte-Puffergrösse
  - 4 zählende Byte-Semaphoren

  benötigt ca. 2 kB Code-Memory und ca. 100 Bytes Data-RAM (Keil-C51-Umgebung mit 8 Bit-Mikrocontroller).

- Mikro-Kernel-Konzept mit kooperativem Task-Switching.
- MiniOS unterstützt maximal 250 Tasks, indexiert als Task[0...250].
- Die Task-Prioritäten im Bereich 0...250 sind fix und identisch mit dem Task-Index, höchste Priorität hat die Task[0].
- Der MiniOS-Kernel unterstützt ein 2-Level-Scheduling (Vordergrund- und Hintergrund-Tasks) mit einer Kombination aus Prioritäts- und Round-Robin-Scheduling.

- MiniOS ist "weich-echtzeitfähig". Für "harte" Echtzeitvorgaben, können zeitkritische Teile der Anwendung in Interrupt-Routinen (ISR) implementiert werden, da der Kernel die Interrupts zu 90-100% seiner Laufzeit freigibt.

- Scheduling und Task-Switching erfolgen Ereignis-gesteuert, OS-Events sind:
  - Task-Timer-Event
  - Semaphor signalisiert (z.B. innerhalb einer ISR)
  - Eintreffen einer Message in der globalen Mailbox
  - Starten einer Task durch eine andere Task
- Warten auf Events mit Timeout ist möglich.

- Das OS-Timing wird mittels einer Anwender-Timer-ISR erzeugt (z.B. 10 ms Ticks), welche auch der Applikation zur Verfügung steht.
- Kommunikations- und Synchronisationsmittel: MiniOS unterstützt zählende Semaphore und eine globale Mailbox mit Byte-Messages.

- Task-Blockaden und System-Timeout werden detektiert und eine Exception ausgelöst.
- OS-Exception-Handler für die Fehlerbehandlung.
- OS-Hooks für Polling-Aufgaben und zur Systemerweiterung.

- MiniOS ist eine reine ANSI-C-Implementation und benötigt *keine* Prozessor- oder Target-spezifischen Erweiterungen oder Assemblersequenzen.

- Der Source-Code von MiniOS steht zur freien Verfügung und ist gut geeignet für Schulungs- und Ausbildungszwecke.

Hinweise:

- Da MiniOS als OS für Embedded-Systeme mit Mikrocontrollern konzipiert wurde, wird das System *zur Kompilierzeit skaliert und konfiguriert*, alle Task- und Event-Objekte sind statisch und nicht zur Laufzeit änderbar.

- Um den Speicherbedarf klein zu halten, arbeitet MiniOS mit dem HW-Stack, der vom C-Compiler zugewiesen wird. Dieser wird von *allen Tasks und dem Kernel gemeinsam genutzt.*

- Lokale Task-Variablen bleiben nach einem Kontext-Switch erhalten, sofern sie als **static** deklariert werden.

- Jede Task muss mindestens **einen sich wiederholenden** Task-Wechsel - z.B. SCHEDULE() - enthalten, um die weiteren Tasks nicht zu blockieren.

- Ein Kontextwechsel wie SCHEDULE(), darf nur in der Task-Hauptfunktion (nicht in einer Task-Unterfunktion) aufgerufen werden.

- Für das Kontext-Switching benötigt MiniOS die ANSI-C-Bibliothek SETJMP.H.

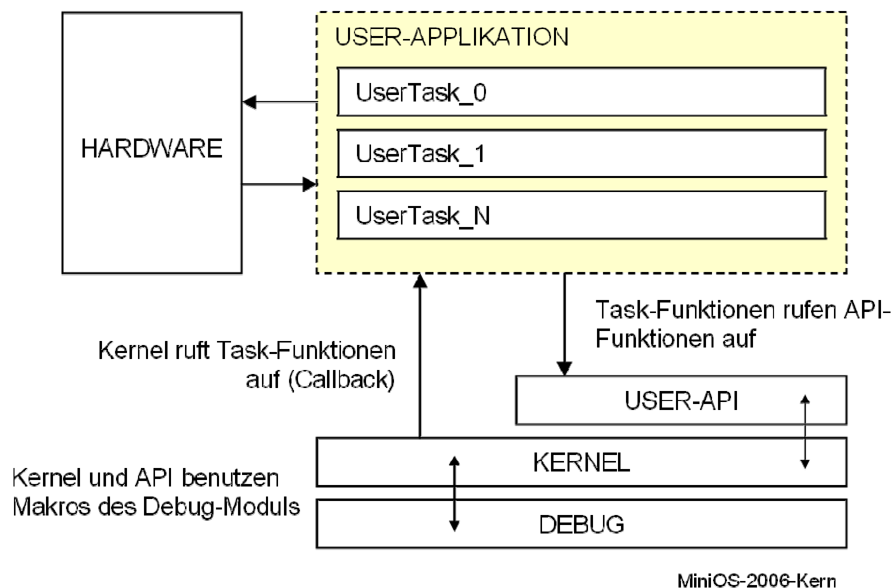
- Nur die dafür bezeichneten API-Funktionen dürfen in einer ISR aufgerufen werden, siehe API-Doku.

Eine Applikation unter MiniOS

MiniOS ist als Mikro-Kernel aufgebaut und vollständig HW-unabhängig. Die HW-Anbindung geschieht auf der Applikationsebene, d.h. Gerätetreiber und IO-Funktionen sind Teil der Nutzer-Applikation und müssen als User-Tasks entworfen und implementiert werden.

MiniOS besteht aus den Teilen
- Scheduler (Kernel)
- Ereignisbehandlung/Dispatcher (Kernel)
- Kommunikation und Synchronisation (Kernel)
- User-API
- Definitionen (Debug-Hooks) für die Fehlersuche, das Modul ist optional nutzbar.

Bild - MiniOS-Module mit zugehöriger Applikation



Ablauf einer MiniOS-Applikation

Der grobe Ablauf einer Applikation unter MiniOS geschieht in der Reihenfolge:

- Applikation initialisieren — Anwenderfunktion
- OS initialisieren — OS_Execute()
- Tasks initialisieren — OS_Execute()
- Tasks ausführen, Multitasking — OS_Execute() → OS_Schedule()
- Exit, Fehlerbehandlung — Anwenderfunktion

- Schliessen und Aufräumen der Applikation bei Power-Down innerhalb einer Anwenderfunktion.

MiniOS-Installation und Dateien

Installation

Die OS-Dateien im Ordner os_lib werden in der Entwicklungsumgebung (IDE, z.B. Keil) wie gewöhnliche Projektdateien im Projekt eingebunden:

1) Projektordner in der verwendeten IDE anlegen.
2) Alle Dateien im Ordner os_lib in den Projektordner kopieren und die OS-Files zum neu eröffneten Projekt hinzufügen, z.B. Menü Project/AddFile...
3) Evtl. Beispiele und Vorlagen kopieren (z.B. keil_c51\main.c).

MiniOS-Dateien

Tabelle - Dateien einer Beispielapplikation unter MiniOS

Dateiname	*konfigurieren?*
os_minios.h	enthält: inkludierte System- und Anwendungs-Header, Deklarationen Anwender: - Prozessor-abhängige Deklarationen (z.B. sbit) einfügen - Benötigte Header-Dateien für das ganze Projekt inkludieren oder ausschliessen
os_kernel.h	Anwender: OS und Anwendung konfigurieren: Tasks, Events etc.
os_kernel.c	keine Einträge vornehmen
os_userapi.h	keine Einträge vornehmen
os_userapi.c	keine Einträge vornehmen
os_debug.h	Anwender optional: Definitionen zu Hooks einfügen (*)
os_debug.c	Anwender optional: Hook-Funktionen implementieren (*)

Der Anwender erstellt:

apptask.c/h	Anwendungs-Tasks implementieren
main.c/h	Anwendung, main()

(*) Die Dateien os_debug.c/h können aus dem Projekt entfernt werden, falls die Debug-Hooks (siehe da) nicht benötigt werden.

Header-Datei os_minios.h

Im File os_minios.h sind die für das Projekt und das Zielsystem nötigen Deklarationen und Header aufgeführt.

Zusätzlich sind vom Anwender einzufügen:
- neu erstellte Header-Files der Anwendung
- Header für die CPU-Registerdeklaration
- System- und Prozessor-spezifische Bezeichner
- Compiler-Extensions etc.

Die vom Anwender editierte Header-Datei os_minios.h muss in alle Applikationsdateien inkludiert werden.

Das Hauptprogramm main()

Listing - Das Hauptprogramm main()

```
void main(void)
{
    while (1) {                 // Restart im Fehlerfall
        InitAnwendung();        // App initialisieren, auch ISR
        OS_Execute();           // OS und Anwendung ausführen
        Fehlerbehandlung();     // falls Fehler in OS_Execute()
    }
}
```

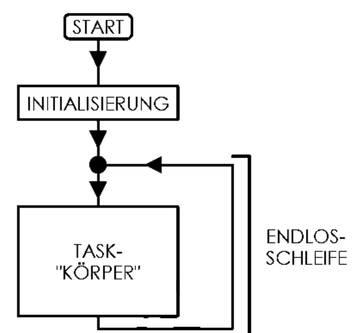
Das main()-Programm besteht im einfachsten Fall aus drei Zeilen. Die User-Applikation wird innerhalb der Endlosfunktion OS_Execute() ausgeführt. OS_Execute() intialisiert das OS und alle Tasks und ruft den Scheduler auf, der das Task-Switching ausführt.

Anwender-Task konfigurieren und implementieren

- ***Die Konfiguration der Tasks ist in der Datei os_kernel.h vorzunehmen.***

Listing, Bild - Aufbau einer Task-Funktion

```
// Aufbau einer minimalen Anwender-Task

void Task1_Function(void)
{
   InitTask();
   SCHEDULE()        // Switch zum Kernel

   while (1) {       // Task Endlos-Loop
      ...
      SCHEDULE()     // Switch zum Kernel
      ...
   }                 // no Return
}
```



Eine MiniOS-Task wird als C-Funktion implementiert. Sie besteht aus einem Initialisierungsteil und einer Endlosschleife mit dem eigentlichen Task-Code.

- Jede Task *muss mindestens einen Kontextwechsel* (wie z.B. SCHEDULE()) enthalten, um dem Kernel die "Chance" zu geben, auch andere Tasks auszuführen.

- Ein Kontextwechsel darf nur in der Task-Hauptfunktion und nicht in einer Task-Unterfunktion aufgerufen werden.

- Task-Funktionen können keine Parameter empfangen oder zurückgeben

- Task-Variablen, die den Task-Wechsel überdauern sollen, müssen *static* deklariert sein. Static-Variablen besitzen - wie globale Variablen - ihre eigene Adresse im RAM und werden nicht auf den Stack gelegt. Sie bleiben nach einem Task-Switch bestehen.

  In zwei Ausnahmefällen dürfen lokale Variablen deklariert und verwendet werden:
  - Lokale Variablen in Unterfunktionen der Task
  - Lokale Variablen, die nur zwischen zwei Kontext-Switches der Task überleben müssen und nachher neu initialisiert werden.

- Nur die dafür bezeichneten API-Funktionen dürfen in einer ISR aufgerufen werden (die Wiedereintrittsfähigkeit des Codes ist eingeschränkt).

Task, TCB und Prioritäten

MiniOS kann maximal 250 Tasks verarbeiten, die als Task[0...250] indexiert angesprochen werden. Die Task-Daten werden im TCB verwaltet, der als globales Struktur-Array implementiert ist. Tasks können nicht dynamisch erzeugt oder gelöscht werden, sie werden zur Kompilierungszeit erzeugt, gemäss den User-Deklarationen im Quelltext-Header os_kernel.h

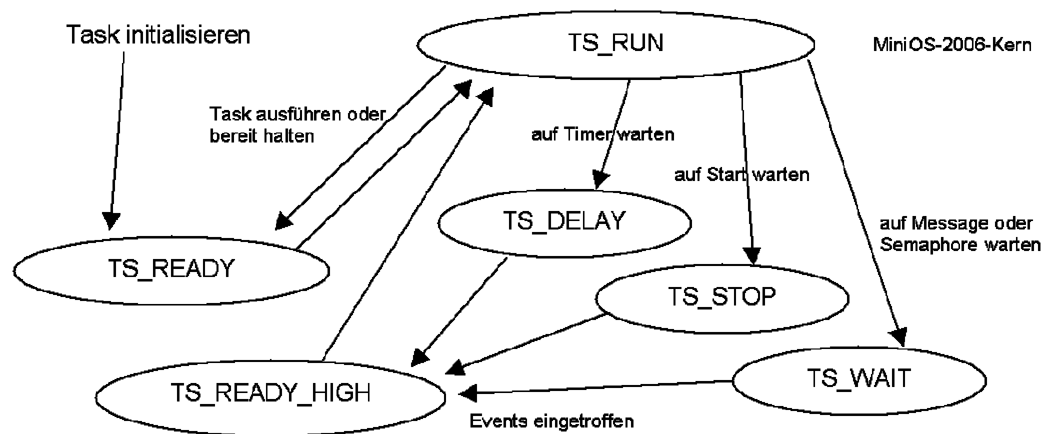
Die Task-Priorität ist identisch mit dem Index der Task im TCB-Array, also zwischen 0...250. Somit besitzt jede MiniOS-Task eine eigene Priorität. Prioritäten sind statisch (festgelegt zur Kompilierungszeit) und können nicht dynamisch (zur Laufzeit) verändert werden. Der TCB-Array hat gleichzeitig die Funktion einer Warteschlange, rechenbereite Tasks werden direkt im Array selektiert und entsprechend dem gerade gültigen Scheduling-Verfahren ausgeführt.

Listing - TCB-Struktur von MiniOS (vereinfacht)

```
// MiniOS, TCB-Array


typedef struct {                // TCB struct
   void (* pTaskFn)(void);      // task function pointer
   jmp_buf        Label;        // task context
   UBYTE          State;        // task state
   T_TIMER        Timer;        // task timer
   UBYTE          Event;        // message or semaphore task waits
}  OS_TCB;


(extern) OS_TCB Task[MAX_TASK];    // globales TCB-Array


   Tasks[0]        → höchste Priorität
   Tasks[1]
   ...
   Tasks[248]
   Tasks[249]      → niedrigste Priorität, max. 250 Tasks
```

Task Status (task state)

Bild - MiniOS Task-States und Uebergänge



Der Task-Status TS_READY_HIGH zeigt eine Ereignisbehandlung an. Ein Ereignis für diese Task ist eingetroffen und der Scheduler wird sie im Prioritätsmodus ausführen. Tasks mit Status TS_READY werden im Round-Robin-Modus ausgeführt.

MiniOS-Scheduler: 2-Level-Scheduling

Der 2-Level-Scheduler von MiniOS vermag eintreffende Ereignisse mittels Prioritäts-Scheduling effizient zu beantworten, währenddem "gleichzeitig" Hintergrund-Tasks im Round-Robin-Verfahren rechnen.

Ebene "Ereignisbehandlung"

Die *Ereignisbehandlung* erfolgt im Prioritäts-Scheduling Modus, d.h. alle Tasks, die durch ein eingetroffenes Ereignis rechenbereit (= TS_READY_HIGH) sind, werden vom Scheduler gemäss ihrer Priorität ausgeführt.

Ebene "Hintergrundbehandlung"

Bei der *Hintergrundbehandlung* mit Round-Robin-Scheduling werden die *Task-Prioritäten nicht beachtet*. Der Scheduler ruft alle TS_READY-Tasks der Reihe nach auf, gemäss ihrem Array-Index.
Falls die gerade rechnende Task sich für einen Kontextwechsel entscheidet und dem Scheduler die Kontrolle mittels dem Makro SCHEDULE() zurückgibt, wird *die nächste* im Array liegende TS_READY-Task aufgerufen, sofern nicht inzwischen eine weitere Task durch ein Ereignis rechenbereit (= TS_READY_HIGH) geworden ist und darum vorgezogen wird.

Die Makros SCHEDULE() und SCHED_PRIO()

Eine noch rechnende Task (Status TS_RUN), die den Scheduler mittels dem Makro...

... **SCHEDULE()** aufruft, erhält den Status **TS_READY**. Sie sie "vorgemerkt" für das Round-Robin-Scheduling.

... **SCHED_PRIO()** aufruft, erhält den Status **TS_READY_HIGH**. Sie wird mittels Prioritäts-Scheduling weiterbehandelt.

→ Falls alle Kontextwechsel einer Anwendung mittels SCHED_PRIO() erfolgen, stellt sich ein reines Prioritäts-Scheduling ein.

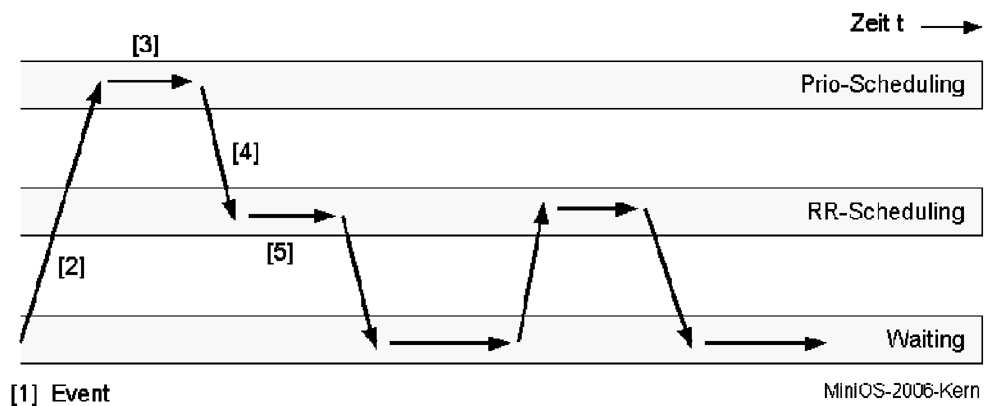
Prioritätsmodus vor Round-Robin-Modus

Prioritäts-Scheduling hat Vorrang vor dem Round-Robin-Scheduling:

Eine READY_HIGH-Task wird einer READY-Task vorgezogen, d.h. solange Ereignisbehandlungen anstehen, werden im Hintergrund keine Round-Robin-Tasks ausgeführt.

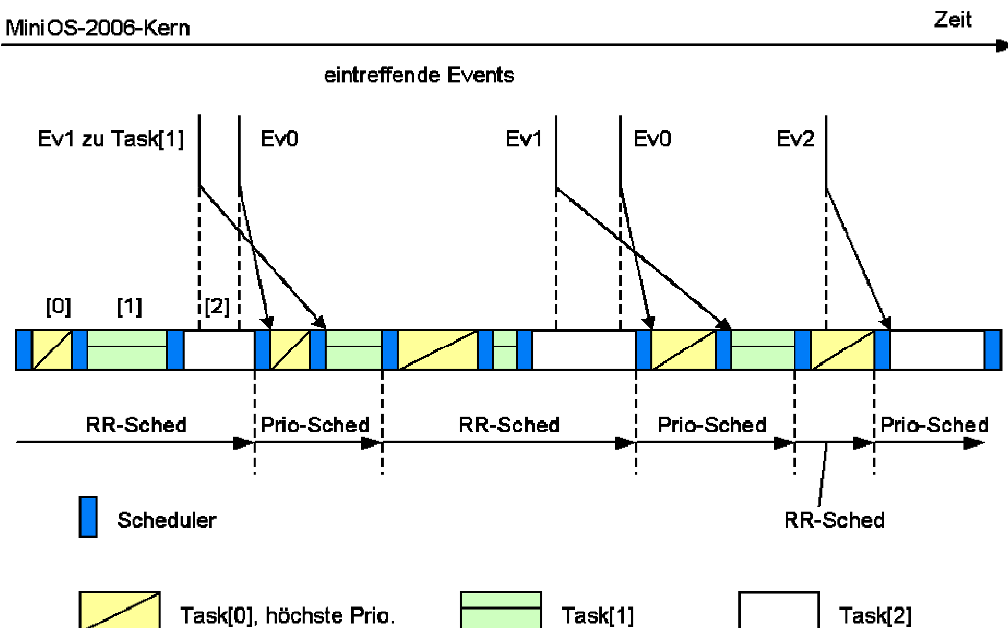
Beim Round-Robin-Scheduling werden die *Task-Prioritäten nicht beachtet*, sondern jede rechenbereite TS_READY-Task berücksichtigt.

Bild - Das 2-Level-Scheduling schematisch am Beispiel einer Task



Ein eintreffendes Ereignis [1] aktiviert den Task-Status TS_READY_HIGH und somit die Ereignisbehandlung im Prioritätsmodus: [2] und [3]. Der folgende Scheduleraufruf [4] mit "SCHEDULE()" führt zum Status TS_READY und einer Weiterbehandlung der Task im Round-Robin-Modus, ab [5].

Bild - Das 2-Level-Scheduling im Timing-Diagramm



Eintreffende Ereignisse werden den darauf wartenden Tasks zugeordnet und mittels Prioritäts-Scheduling behandelt Dazwischen laufen die Hintergrund-Tasks im Round-Robin-Modus.

Tabelle - Arbeitsweise des Schedulers

Der Scheduler wird von einer Task aufgerufen, mit den Makros SCHEDULE() oder SCHED_PRIO():

1) Solange nicht-initialisierte Tasks vorhanden sind:
   Initialisiere diese Tasks, d.h. Task-Funktionen aufrufen

2) Solange TS_READY-HIGH-Tasks vorhanden sind:
   diese Tasks mit Prioritäts-Scheduling ausführen

3) Sonst, solange TS_READY-Tasks vorhanden sind:
   diese Tasks mit Round-Robin-Scheduling ausführen

4) Sonst (falls alle Tasks wartend oder gestoppt sind):
   Ruhezustand in der Idle-Loop bis ein Ereignis (z.B. Interrupt) eintrifft
   (Idle-Mode: Idle-Hook-Funktion für Power-Management oder Polling-Jobs)

OS Ruhezustand, System Idle Mode

- ***Die Deklaration der User-Hook-Funktionen ist in der Datei os_kernel.h vorzunehmen.***

Wenn keine Task rechenbereit ist (alle warten oder sind gestoppt) führt der Scheduler die Idle-Loop ("idle" = untätig) aus und wartet auf das nächste Ereignis (z.B. einen Interrupt). Bei jedem Durchlauf der Idle-Schleife ruft der Kernel eine *Idle-Hook-Funktion* auf. Diese Funktion kann der Anwender nutzen und z.B. zyklische Abfragen (polling) von HW oder SW-Variablen (Flags) implementieren, Power-Management-Funktionen aktivieren oder die Auslastung des Systems erfassen. Siehe auch unter "User-Hook-Funktionen".

Listing - Beispiel einer Idle-Hook-Funktion des Anwenders

```
// Beispiel einer Idle-Hook-Funktion, die zum OS zurückkehrt,
// falls ein Interrupt eintrifft

// os_kernel.h:
// **********************************************************
#define OS_HOOK_SCHEDULE_IDLE  OnScheduleIdle();  // Hook-Fn


// Anwender-Modul:
// **********************************************************

void OnScheduleIdle(void) // ist in os_kernel.h deklariert
{
   while (!NOTIFY) {      // solange kein Interrupt-Event

      DoIdleJob();        // Idle-Job nicht blockierend!
   }
   return;                // zurück zum Scheduler
}
```

Um den Ruhezustand des Systems zu nutzen, kann auch die niedrigst priorisierte Task als *Idle-Task* (Leerlaufprozess) genutzt werden. Dies erfordert jedoch ein *reines Prioritäts-Scheduling*: die Task mit der niedrigsten Priorität wird aktiv, wenn alle anderen Tasks wartend oder gestoppt sind. Dies bedeutet jedoch gegenüber einer Idle-Hook-Funktion einen Mehraufwand an Speicher und Ausführungszeit.

OS-Systemzeit, System-Timing

Konfiguration

- ***Die Konfiguration der Timer-Datentypen ist in der Datei os_kernel.h vorzunehmen.***

Listing - Timer-Datentypen in os_kernel.h

```
// os_kernel.h:
//
// SYSTEM TIMER DATA TYPES
// ************************************************************
// if you wish to extend task-timer or tick range: modify
// this data types

typedef unsigned int  T_TICK;   // sys time, tick counter type
typedef unsigned int  T_TIMER;  // task timer type
```

Die Datentypen werden dem benötigten Zählbereich angepasst (Speicherbedarf optimieren).

System-Timer-Interrupt, OS-Timer-Tick

Alle Echtzeitvorgänge einer MiniOS-Applikation benötigen eine Zeitreferenz. Diese wird mit Hilfe einer Timer-Interrupt-Routine erzeugt:

Listing - Die System-Timer-ISR erzeugt den System-Timing-Tick

```
// vom Anwender zu implementieren:

void timer0_sysint (void) interrupt 1   // ISR, Sys-Timer
{
   ...                    // Interrupts sperren etc.

   OS_TimerTick();        // OS-Funktion einbinden

   ...                    // weitere User-Jobs
   ...                    // Int. wieder freigeben
}
```

Eine Timerüberlauf-ISR muss vom Anwender implementiert und darin die Funktion OS_TimerTick() aufgerufen werden. Die ISR wird z.B. alle 10 ms aufgerufen (10 ms Tick) und steht auch der User-Anwendung zur Verfügung. Die Systemzeitvariable TICK zählt die Anzahl der eintreffenden System-Timer-Interrupts und kann als Systemzeit ausgewertet werden, z.B.:

```
if (TICK == VorgabeWert) { MachWas() }
```

Der maximale TICK-Zählwert hängt vom verwendeten Datentyp T_TICK für den Timer ab.

Task-SW-Timer TIMER

MiniOS verwaltet für jede Task die Timer-Variable Task[tid].Timer, welche in der Timer-ISR dekrementiert wird, solange ihr Wert grösser Null ist. Wert Null reaktiviert die wartende Task - Status TS_DELAY - und sie wird rechenbereit (TS_READY_HIGH).

Der maximale Zählwert der Task-Timer-Variablen hängt vom verwendeten Datentyp T_TIME ab. Wird T_TIME als **unsigned char** (UBYTE) typisiert, erstreckt sich der Zählbereich über 255 (+0/-1) Timer-Ticks. Die Typen **unsigned int** oder **unsigned long** erhöhen den Bereich entsprechend.

Der Ausdruck TIMER bezeichnet die Task-Timer-Variable der gerade rechnenden Task (Task[OS.CurrentTask].Timer). Der Timer kann auch vom Anwender geladen und ausgewertet werden: siehe Beispiele.

Timer, Beispiele

Listing - Task-Timer für Wartezeit (Delay) und zyklische Funktionsaufrufe benutzen

```
// Zyklischer Funktionsaufruf
// Beispiel: Der System-Tick sei 10 ms, dies führt zum
// zyklischen Aufrufen der Funktionen im 1 Sekunden-Takt
...
   while (1) {                     // Task-Loop
      ...
      OS_WaitTimer(100);           // Task-Timer starten
      SCHEDULE()                   // warten
      DoFunction1();               // zurück vom Warten
      DoFunction2();
      DoFunction3();
      ...
   }
```

Listing - Task-Timer für Event mit Timeout benutzen

```
// Warten auf EventMessage mit Timeout
...
   while (1) {
      ...
      OS_WaitMessage(MSG_XY, 100); // wait with timeout
      SCHEDULE()
      if (ON_TIMEOUT) {
         DoTimeoutHandling();
      }
      else {
         DoEventHandling();
      }..
   }
```

Listing - Task-Timer für Timeout (Alarm) benutzen

```
// Timeout/Alarm setzen und abfragen
...
   while (1) {
      ...
      OS_SetTimer(50);          // Task-Timer starten

      while (TIMER) {           // solange Timer läuft
         DoJob();               // Job mit Timeout erledigen
         SCHEDULE()             // andere Tasks behandeln
         ...
      }
      ...
   }
```

OS und Anwendung initialisieren

Tabelle - Ablauf der System-Initialisierung beim System-Start-Up

Step	*Funktion*	*OS.SysState, OS-Zustand*	*Anw.-Impl.*
1	main()	OS_STOPPED	
	HW, z.B. Ports initialisieren		**x**
	System/OS-Timer-Interrupt initialisieren und freigeben/starten		**x**
	weitere Anwender-Interrupts initialisieren und freigeben		**x**
2	OS_Execute() OS starten und ausführen	OS_STOPPED	
3	OS_Init() alle OS-Variablen initialisieren	OS_RUN_SYSTEM	
4	OS_HOOK_USERINIT_OS (Anw.-Hook)	OS_RUN_SYSTEM	
	evtl. weitere Anwender-Interrupts freigeben		**x**
5…n	TaskFunktion() alle Appl.-Tasks initialisieren, z.B.	OS_RUN_USER	
	- weitere HW initialisieren - Display - Tastatur etc.		**x**
	OS_Schedule() Appl.-Tasks wechseln und ausführen	OS_RUN_SYSTEM OS_RUN_USER	

Ereignisbehandlung und -steuerung

Ereignissteuerung liegt dann vor, wenn die Tasks eines Systems auf Ereignisse (events) warten und erst bei deren Eintreffen aktiviert (= rechenbereit) werden. Grundlage von Ereignissen sind oft HW-Interrupts oder das Ergebnis einer zyklischen Abfrage (polling) der HW oder SW.

MiniOS kennt folgende Events:
- Task-Timer gleich Null
- Semaphor signalisiert (z.B. innerhalb einer ISR)
- Eintreffen einer Message in der globalen Mailbox

Neu eintreffende Ereignisse werden im Kernel registriert und den wartenden Tasks zugeordnet. Trifft ein Event ein, wird der darauf wartende Task rechenbereit (Status TS_READY_HIGH). Die Ereignisbehandlung geschieht innerhalb des Kernels, d.h. ohne Task-Wechsel werden keine Ereignisse mehr bearbeitet und das System blockiert.

Ereignisüberlauf, Systemüberlastung

Ein System-Timeout-Error (event overflow exception) entsteht, wenn pro Zeitintervall zu viele Ereignisse eintreffen und nicht mehr in "Echtzeit" verarbeitet werden können, da der Scheduler zu wenig häufig oder nicht mehr aufgerufen wird. Der Event-Notifier zeigt dies an, indem er eine Systemüberlastung, resp. Event-Overflow-Exception auslöst.

Mit dem folgenden Eintrag in os_kernel.h definiert der Anwender die Anzahl der Ereignisse, die zwischen 2 Task-Wechseln (SCHEDULE()-Aufrufe) eintreten dürfen, ohne dass das System einen Timeout-Error generiert. Die globale Variable OS.NotifyCount zählt die eintreffenden Ereignisse. Sie wird vom Kernel überwacht und mit dem Grenzwert OS_NOTIFY_MAX verglichen.

Listing - Maximale Anzahl unbearbeiteter Ereignisse, bevor das OS "Alarm auslöst".

```
// Datei os_kernel.h:

#define OS_NOTIFY_MAX  25   // max value 250 (UBYTE)
```

Achtung: Als Ereignis zählt auch das Eintreffen eines System-Ticks. Hier würde also nach 25 System-Ticks ein System-Timeout-Error generiert, andere in diesem Zeitintervall auftretende Events noch nicht mitgezählt.

Jeder SCHEDULE()-Aufruf setzt den Event-Notify-Zähler zurück und veranlasst normalerweise auch die Verarbeitung der anstehenden Ereignisse.

System-Timeout verhindern

kann man nur mittels häufigem Aufruf des Schedulers (z.B. mit dem Makro SCHEDULE() und anderen).

In Ausnahmeanwendungen oder bei länger dauernden Schleifen, verhindert OS_ClearNotifier() das Auslösen des Timeout-Errors, falls der Scheduler aus irgendwelchen Gründen nicht häufig genug aufgerufen werden kann.
Während dieser Zeit bleiben jedoch alle anderen Tasks unbearbeitet!

Listing - System-Timeout verhindern

```
// Task-Funktion, System Timeout verhindern
...
   while (1) {                      // Task-Loop
      ...
      OS_SetTimer(50);              // Task-Timer starten

      while (TIMER) {               // solange Timer läuft
         DoJob();                   // Job mit Timeout erledigen
         OS_ClearNotifier();        // Notifier zurücksetzen
         ...
      }
      ...
   }
```

MiniOS und Interrupts

Ausser einem periodischen Timer-Interrupt als Zeitreferenz (siehe Abschnitt System-Timing) benötigt MiniOS keine weiteren System-Interrupts. Die Implementation weiterer Interrupts ist somit abhängig von der Applikation und wird dem Anwender überlassen.

Der MiniOS-Kernel sperrt alle Interrupts während ca. 0-10% seiner Laufzeit. Dies ist abhängig davon, welche Art von Ereignissen abgewartet und bearbeitet werden. Innerhalb der Idle-Loop - wenn das System auf Ereignisse wartet - werden die Interrupts zu keinem Zeitpunkt gesperrt, d.h. die Latenzzeit ist minimal. Teile der Anwendung, die Echtzeitanforderungen genügen müssen, können daher in ISRs implementiert werden.

- ***Die Konfiguration der Interrupts (disable und enable) ist in der Datei os_kernel.h vorzunehmen.***

Listing - User-Makros in os_kernel.h: Interrupts im Kernel sperren und freigeben

```
// os_kernel.h:
// define the interrupt-disable and enable instructions

#define OS_DISABLE_INT    // EA = 0; // C51
#define OS_ENABLE_INT     // EA = 1;
```

Der Anwender muss in der Datei os_kernel.h die jeweiligen Befehle für das Sperren und Freigeben der Interrupts eintragen, diese sind Zielsystem- und Prozessor-abhängig.

Hier kann auch Code definiert werden, um die Interrupt-Sperrzeiten im Zielsystem für die jeweilige Anwendung zu messen.

Allgemein müssen für die Interrupt-Behandlung einige Regeln und Massnahmen beachtet werden:

ISR: Einige Regeln um Aerger mit Interrupts zu vermeiden

- Die ISR ist eine von den Tasks (*) losgelöste Funktion, deren Ausführung jederzeit und überall im Programmablauf (asynchron) eintreten kann. Die ISR hat weder Uebergabe- noch Rückgabeparameter.
- Eine ISR kann nicht auf Ereignisse warten.
- Die ISR so kurz wie möglich halten, damit keine anderen Interrupts "ausgebremst" werden.
- Eine ISR kann durch andere Interrupts oder Tasks blockiert werden (interrupt disabled) und darum möglicherweise zeitliche Anforderungen nicht erfüllen.
- Die ISR so entwerfen, dass keine kritischen Bereiche entstehen, Code "reentrant" halten.
- Keine System-Ressourcen nutzen, auf die auch innerhalb von Tasks zugegriffen wird (shared resources), es sei denn, die Mittel sind dafür vorgesehen (z.B. Semaphore, Event-Flags).
- Ein Semaphor als Event-Flag in der ISR signalisiert der darauf wartenden Task das Eintreffen des Interrupts.

(*) bedeutet immer auch main()-Programm!

Innerhalb einer ISR muss eine der folgenden Funktionen aufgerufen werden:

Beide Aufrufe veranlassen den Kernel die Idle-Loop zu beenden und die zum Interrupt zugehörige Task weiterzuführen, sowie ein allfälliges System-Timeout zu überwachen.

Falls eine Task zu benachrichtigen ist, wird ein Semaphor als Event-Flag eingesetzt:

OS_SignalSem_ISR(SEM_NAME)

sonst:

OS_Notify_ISR()

Listing - Portschalter einlesen mit ISR und Semaphor

```
// ISR: Port-Schalter und Semaphoren-Signalisierung
// ******************************************************
void ISR_GetPortSwitch(void) interrupt_handler_x
{
   OS_SignalSem_ISR(SEM_SW1); // Semaphor signalisieren
   return;
}


// Task: Schalter lesen und auswerten
// ******************************************************
void Task_Switch1(void)
{
   static UBYTE bSwitch1State;
   ...
   while (1) {
      ...
      OS_WaitSem(SEM_SW1, Timeout); // warten auf Semaphor
      SCHEDULE()
      bSwitch1State = PORT_PIN;     // Schalter einlesen
      SwitchAuswerten();            // und auswerten
      ...
   }
}
```

Der Semaphor übernimmt die Rolle eines Event-Flags (allfällige Timeout-Behandlung nicht gezeigt).

Kommunikation und Synchronisation

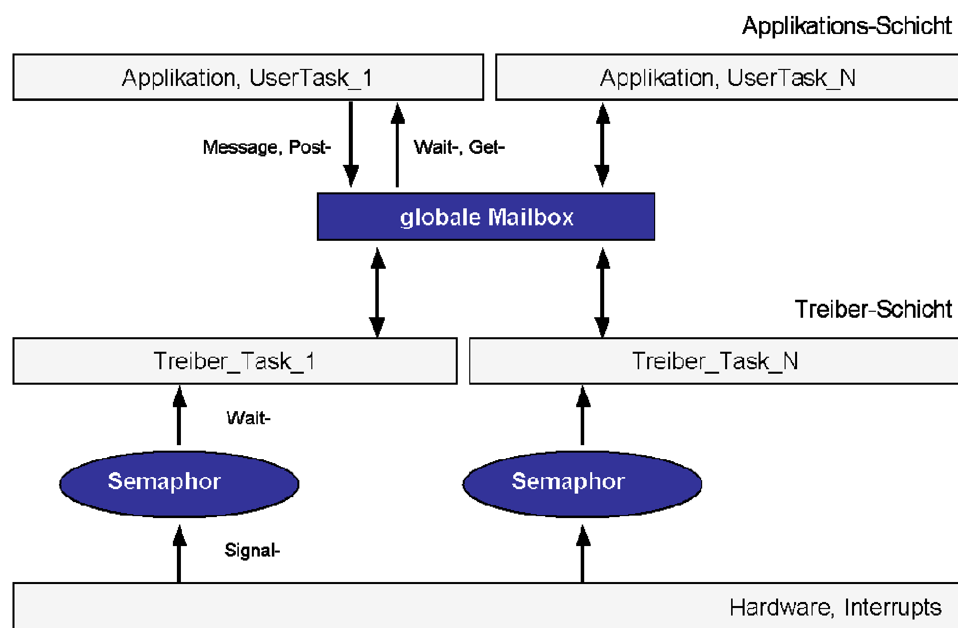
MiniOS unterstützt zwei grundsätzliche Einrichtungen zur Kommunikation und Synchronisation:

- zählende Byte-Semaphore
- eine globale Mailbox mit Single-Byte-Messages

Der erwartete Semaphor- oder Message-Event wird der Task im TCB-Array zugeordnet: Task[tid].Event = SemaphorID oder MessageName. Die Ereignisbearbeitung im Kernel vergleicht den von der Task erwarteten Event mit den effektiv eintreffenden Ereignissen. Trifft der erwartete Event ein, bekommt die Task den Status TS_READY_HIGH zugewiesen und wird mittels Prioritäts-Scheduling ausgeführt.

Warum Semaphore und Messages?

Bild - Eine mögliche Anwendung von Semaphoren und der globalen Mailbox



Ein Beispiel:
Implementation einer Treiberschicht zwischen der Hardware und der Applikationsschicht. Die Semaphore sind für die Anzeige der Interrupts in der Treiberschicht zuständig. Die Mailbox vermittelt die Meldungen zwischen der Applikation und den Treibern.

Semaphore und Messages benutzen, Uebersicht

- Semaphor signalisieren innerhalb einer Task, nicht blockierend
  `OS_SignalSem(SEM_NAME);`

- Semaphor signalisieren innerhalb einer ISR, nicht blockierend
  `OS_SignalSem_ISR(SEM_NAME);`

- Message an Mailbox senden, innerhalb einer Task, nicht blockierend
  `OS_PostMessage(MSG_NAME);`

- Auf Semaphor oder Message warten, nur innerhalb einer Task (nicht in ISR)
  `OS_WaitSem(SEM_NAME, Timeout);`
  `OS_WaitMessage(MSG_NAME, Timeout);`

- Semaphor prüfen ohne ihn zu verändern, innerhalb einer Task oder ISR, nicht blockierend
  `ByteVar = OS_PeekSemaphore(SEM_NAME);`

- Eine bestimmte (bekannte) Message in der Mailbox abholen und löschen, nicht blockierend, nur innerhalb einer Task
  `ByteVar = OS_GetMessage(MSG_NAME);`

- Existenz einer Message in der Mailbox überprüfen ohne sie zu verändern, nicht blockierend, nur innerhalb einer Task
  `ByteVar = OS_PeekMessage(MSG_NAME);`

Kurz:

- Eine ISR kann nur Semaphore signalisieren oder ihren Wert lesen ohne zu blockieren (peek), aber nicht auf Semaphore warten.

- Eine Task kann Semaphore signalisieren oder darauf warten.
  Messages können *nur von Tasks* versandt oder erwartet werden.

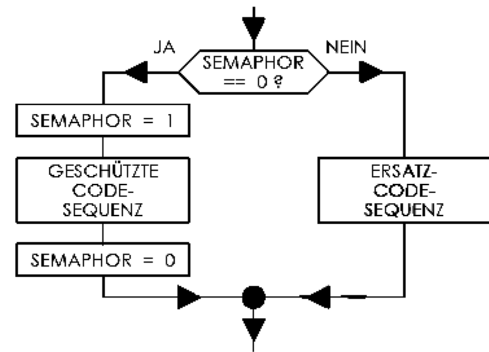
Semaphore

- ***Die Konfiguration der Semaphore ist in der Datei os_kernel.h vorzunehmen.***

Bild - Ein Semaphor steuert den Zugriff auf einen geschützten Codebereich

Semaphore sind OS-Variablen, die u.a. eingesetzt werden für:

- die Signalisierung von Ereignissen
- die Steuerung des Programmablaufs
- die Steuerung des Zugriffs auf Ressourcen



Prüfen und Setzen/Rücksetzen (Test and Set) des Semaphors sollten in einer einzigen unteilbaren ("atomaren") Instruktion erfolgen, um Zugriffsunsicherheiten zu verhindern, falls eine andere (preemptive) Task gleichzeitig den Semaphor besitzen möchte.

Semaphore in MiniOS

- MiniOS verwaltet max. 250 zählende Byte-Semaphore.
- Der maximale Zählwert (MAX_SEM_COUNT) eines Semaphors ist konfigurierbar zwischen den Grenzen 2 und 255. Wird der Zählgrenzwert erreicht, löst dies eine Semaphor-Overflow-Exception aus.
- Die Semaphore werden in einem Array zusammengefasst und über ihren Index angesprochen: OS.Semaphore[0...250].
- Die Beispielfunktion OS_SignalSem (**SEM_NAME**) spricht den Semaphor OS.Semaphore[**SEM_NAME**] an.
- Den Indizes SEM_NAME werden zur Kompilierzeit konstante Namen zugewiesen. Diese können die Funktion des Semaphors oder Ereignistypen bezeichnen, wie z.B.
  - SEM_EVENT_FLAG_A
  - SEM_ISR_EXT
  - SEM_MUTEX1
  - SEM_BUFFER_X

  etc.

Ein zählender Byte-Semaphor...

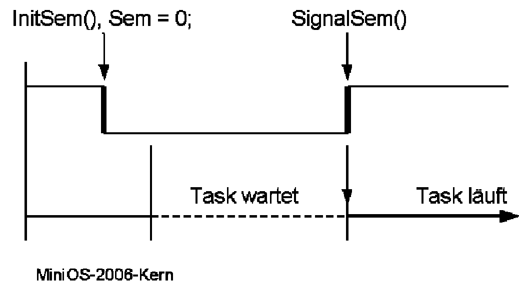
- *..."signalisieren"* bedeutet:
  Die Semaphor-Variable wird vom OS inkrementiert, falls sie kleiner als der vorgegebene Ueberlaufwert (z.B. 255) ist.
- *..."warten und auswerten"* bedeutet:
  Falls die Semaphor-Variable grösser als 0 ist, wird sie vom OS dekrementiert und die darauf wartende Task aktiviert. Falls sie gleich 0 ist, wartet die Task weiterhin, bis der Semaphor signalisiert wird.
- *...intialisieren* (falls keine Task auf den Semaphor wartet) mit der Funktion OS_InitSem (SEM_NAME, InitWert);

Semaphor als Event-Flag

werden häufig in einer ISR (siehe da) eingesetzt um der wartenden Task das Eintreffen des Interrupts anzuzeigen (zu signalisieren).

Bild - Semaphor als Event-Flag

Ein Semaphor mit dem Wert 0, zeigt an, dass ein erwartetes Ereignis noch nicht eingetroffen ist. Beim Eintreffen des Events wird der Semaphor signalisiert - auf 1 gesetzt - und die wartende Task ausgeführt.



Semaphore zur Synchronisation von Tasks

Listing - Task-Synchronisation mit Semaphoren

```
// Task1
// ************************
...
   while (1) {
      OS_WaitSem(SYNC_2, 0);
      SCHEDULE() ;
      DoJob11();
      OS_SignalSem(SYNC_1);
   }
```

```
// Task2
// ***********************
...
   while (1) {
      DoJob21();
      DoJob22();
      OS_SignalSem(SYNC_2);
      OS_WaitSem(SYNC_1, 0);
      SCHEDULE()
   }
```

Task1 wartet bis Task2 seine Jobs abgearbeitet hat, dann werden sie wieder gestartet.

Semaphor zur Zugriffssteuerung von Ressourcen, Mutex

Listing - Zugriff auf eine gemeinsame Ressource (Display)

```
// Semaphor mit 1 initialisieren: Ressource ist frei
// *******************************************************
   OS_InitSem(MUTEX, 1);
   ...

// Task1
// ************************
...
   while (1) {
      ...
      DatenVorbereiten();
      OS_WaitSem(MUTEX, 0);
      SCHEDULE()
      WriteDisplay();
      OS_SignalSem(MUTEX);
      ...
   }

// Task2
// ***********************
...
   while (1) {
      ...
      DatenVorbereiten();
      OS_WaitSem(MUTEX, 0);
      SCHEDULE()
      WriteDisplay();
      OS_SignalSem(MUTEX);
      ...
   }
```

Zugriffssteuerung mittels des Prinzips des gegenseitigen Ausschlusses (mutual exclusion, mutex): "Wer zuerst kommt schreibt zuerst…"

Hinweis: In kooperativen Systemen kann der Zugriff auf eine Ressource geschützt (exklusiv) ausgeführt werden, indem kein Task-Wechsel veranlasst wird. Zugriffe auf die Ressource, während denen ein Task-Switching nötig ist, können mit einem Mutex-Semaphor geschützt werden.

Zählende Semaphore als Indikatoren für freie Ressourcen

Beispiel: Mehrere Tasks möchten auf einen Datenpuffer BUF (z.B. FIFO) mit 16 Bytes Grösse zugreifen mit den Aktionen "put" (Byte speichern) und "get" (Byte lesen und löschen).

Put: Puffer schreiben, ein Semaphor zeigt den freien Speicherplatz an.
Get: Puffer lesen, ein Semaphor zeigt an, ob zu lesende Bytes im Puffer vorhanden sind.

Semaphore initialisieren mit der Anzahl freien und lesbaren Bytes im Puffer:

```
...
OS_InitSem(BUF_PUT, 16); // Puffergrösse in Bytes
OS_InitSem(BUF_GET, 0);  // Anzahl lesbarer Bytes
```

Eine Task schreibt ein Byte in den Puffer (put):

```
...
// Testen und warten bis Platz im Puffer vorhanden ist
OS_WaitSem(BUF_PUT, Timeout);
SCHEDULE()               // Platz-Sem. dekrementieren
PutByteInBuffer(InByte); // schreiben
OS_SignalSem(BUF_GET);   // Inhalt-Sem. inkrementieren
...
```

Eine Task liest und löscht ein Byte im Puffer (get):

```
...
// Testen und warten bis lesbare Bytes im Puffer vorhanden sind
OS_WaitSem(BUF_GET, Timeout);
SCHEDULE()               // Inhalt-Sem. dekrementieren
OutByte = GetByteFromBuffer(void);    // lesen
OS_SignalSem(BUF_PUT);   // Platz-Sem. inkrementieren
...
```

Mailbox

- ***Die Konfiguration der Mailbox ist in der Datei os_kernel.h vorzunehmen.***

Einige Begriffe:

Message-Passing (Nachrichtenübermittlung):
Eine Task sendet eine Nachricht an eine andere Task, dabei wird zwischen zwei Kommunikationsarten unterschieden:

Direkte Kommunikation:
Die Sende-Task (Sender) schickt *direkt* an die Empfänger-Task (Empfänger) eine Nachricht oder Meldung (Punkt-zu-Punkt-Verbindung). Falls der Uebertragungskanal kein Datenpuffer aufweist, ist nur synchrones Senden und Empfangen möglich (mit gegenseitigem Warten).

Indirekte Kommunikation:
Die Kommunikation erfolgt über eine Mailbox. Der Sender stellt die Nachricht in die Mailbox, wo sie der Empfänger abholt, dies ermöglicht asynchrones Senden und Empfangen.

Synchrones Senden und Empfangen (blocked sending, receiving):
Der Sender sendet eine Nachricht an den Empfänger und geht in den Wartezustand, bis der Empfänger ihm den Empfang quittiert. Der wartende Empfänger wird aktiv und beginnt mit der Abarbeitung der Task. Danach kehrt auch er wieder in den Wartezustand zurück bis die nächste Nachricht eintrifft.

Asynchrones Senden und Empfangen (non-blocked sending, receiving):
Der Sender wartet nicht auf eine Empfangsbestätigung des Empfängers, sondern setzt die Abarbeitung des Tasks fort. Der Empfänger kann jederzeit nicht-blockierend überprüfen, ob eine Nachricht für ihn vorliegt. Die Empfangs-Task kann die Ueberprüfung der Mailbox auch an den Kernel delegieren, um zu "schlafen" (WAITING) und sich beim Eintreffen der Nachricht aktivieren zu lassen. Asynchrones Senden und Empfangen erfordert eine gemeinsame Mailbox als Datenpuffer (oder Puffer auf der Sende- und Empfangsseite).

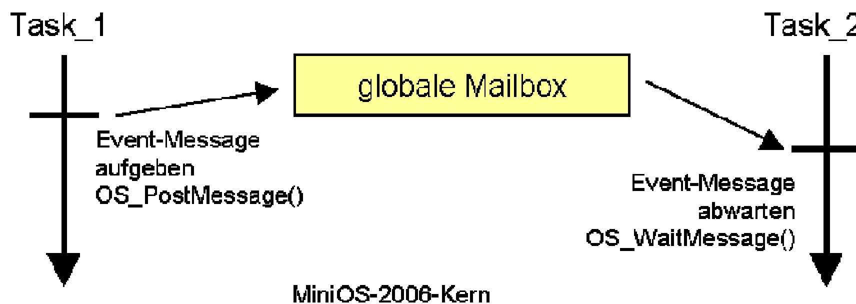
Mailbox in MiniOS

MiniOS verwendet für die Inter-Task-Kommunikation ein indirektes Message-Passing mit Single-Byte-Messages über eine globale System-Mailbox. Die Zeitpunkte des Sendens und Empfangens einer Message sind voneinander unabhängig (asynchron). Der Absender einer Message kennt den Empfänger nicht, d.h. die Message in der Mailbox steht allen Tasks als globaler Wert zur Verfügung. Die Empfangs-Task wartet auf eine spezifische, ihr bekannte Message (z.B. eines Ereignisses), auf die sie entsprechend reagieren kann.

Die Mailbox wird periodisch (bei jedem Scheduler-Aufruf) "geleert". Die zeitliche Reihenfolge der eingegangenen Nachrichten wird nicht beachtet. Wartende Tasks, für die eine Message eingetroffen ist, erhalten von der Ereignisbearbeitung im Kernel den Status TS_READY_HIGH zugewiesen. Sie werden entsprechend ihrer Priorität aktiviert und weiter ausgeführt. Die Mailbox kann auch von der Task selbst auf eingegangene Nachrichten überprüft werden (ohne Blockieren).

(Die aktuelle Version von MiniOS unterstützt kein direktes Message-Passing zwischen Tasks.)

Bild - Globale Mailbox



MiniOS: Globale Mailbox als Nachrichtenpuffer für die indirekte Kommunikation.

- Die globale Mailbox kann max. 250 Single-Byte-Messages aufnehmen. Diese Puffergrösse wird mit der Konstanten MAX_MSGBUF zur Kompilierzeit vorgegeben.

- Die Beispielfunktion OS_PostMessage (**MSG_NAME**) behandelt die Byte-Message MSG_NAME. Diese kann spezifische Nachrichten oder Ereignistypen bezeichnen, wie z.B.
  - MSG_KEY_DOWN
  - MSG_UART_RXD
  - MSG_SIGNAL1
  - MSG_EVENT_X

  etc.

- Eine Byte-Message kann max. 253 verschiedene Namen darstellen. Die Byte-Werte 0 und 255 sind nicht als Message-Namen nutzbar

---

Zugriffe auf die globale Mailbox von MiniOS dürfen *nicht* innerhalb von Interrupt-Routinen (ISR) erfolgen, da die Mailbox Wiedereintritte nicht zulässt.

(Mögliche Datenfehler beim Lesen und Schreiben in der Mailbox, falls sich Zugriffe innerhalb einer Task und einer eintretenden ISR überlagern.)

Hook-Funktionen

Eine *Hook-Funktion* (hook = Haken, "Eingriff") ist eine Callback-Funktion, die der Kernel an wichtigen Stellen im Ablauf aufruft. Sie wird vom Anwender implementiert und genutzt, um die Funktionalität des OS zu erweitern oder Fehler zu suchen.

Hooks in MiniOS sind Makros (#define ...), die zur Kompilierzeit eine vom Anwender definierte Programmsequenz (z.B. einen Funktionsaufruf) in den Code einfügen. Bleibt die Definition leer (z.B. #define OS_HOOK_EXCEPTION <SPACE>), wird kein Code eingesetzt.

MiniOS verwendet User- und Debug-Hooks. Beides sind Makro-Definitionen, welche sich jedoch in der Anwendung unterscheiden.

User-Hooks

- ***Die Deklaration der User-Hook-Funktionen ist in der Datei os_kernel.h vorzunehmen.***

User-Hooks sind vom Anwender definierte Erweiterungen des Kernels und *fester Bestandteil der Applikation*. Beschreibung der wichtigsten User-Hooks:

OS_HOOK_USERINIT_OS	Dieser Hook wird nach der OS-Initialisierung, aber vor der Initialisierung der Tasks aufgerufen.
OS_HOOK_EXCEPTION	Wird in der Funktion OS_Exception() beim Auftreten eines Fehlers aufgerufen.
OS_HOOK_SCHEDULE_ENTRY	Wird beim Eintritt in die Scheduler-Funktion aufgerufen.
OS_HOOK_SCHEDULE_IDLE	Wird im Ruhezustand (idle mode) des Systems, bei jedem Durchgang der Idle-Schleife aufgerufen.

Listing - Beispielanwendung eines User-Hooks

Flags und Hardware zyklisch abfragen (polling) und eintreffende Ereignisse mittels Semaphoren signalisieren.

```
// In der Datei os_kernel.h definieren:
// ***********************************************************

// Hier kann noch differenziert werden, an welcher Stelle im
// Scheduler die Abfrage erfolgen soll.

#define OS_HOOK_SCHEDULE_ENTRY  ScheduleHook(); // Sched-Eing
#define OS_HOOK_SCHEDULE_IDLE   ScheduleHook(); // Sched-Idle


// In der User-Applikation implementieren:
// ***********************************************************

void ScheduleHook(void) // Polling Jobs
{
// Mehrfachauswertung vermeiden, indem Flags gelöscht,
// resp. Semaphore vor dem Signalisieren abgefragt werden.

   if (Flag_A) {
      OS_SignalSem(SEM_FLAG_A); // System-Event auslösen
      Flag_A = 0;               // ohne Mehrfachauswertung
   }

   if (PORT_PIN && !OS_PeekSemaphore(SEM_PORT_PIN)) {
      OS_SignalSem(SEM_PORT_PIN);
   }
}
```

Debug-Hooks

- ***Die Definitionen der Debug-Hooks finden sich in der Datei os_debug.h, zugehörige Funktionen sind in os_debug.c zu implementieren.***

Debug-Hooks sind Wegmarken oder "Checkpoints" an wichtigen Stellen innerhalb des Kernels, um z.B. ohne Eingriffe in den Source-Code das Debuggen der Applikation zu vereinfachen.

Im Unterschied zu den User-Hooks haben Debug-Hooks temporären Charakter. Sie können alle aus dem Projekt entfernt werden, indem die Datei os_debug.h ausgeschlossen wird (siehe Header os_minios.h). Der Ausdruck DEBUG_ON in os_debug.h bewirkt eine bedingte Kompilation.

Listing - Beispiel eines Debug-Hooks

```
// User-Makro-Definition in os_debug.h
// z.B. Status auf LCD ausgeben und Applikation anhalten

#define ON_OS_TERMINATE                 \
    {                                   \
        LCDWrite("BEENDEN...");         \
        while (1);                      \
    }
```

Listing - Debug-Hook-Definitionen in os_kernel.h und os_debug.h

Hook-Define	*Wozu?*
// DEBUG_HOOKS	
// omitted if no debug-mode-file	bedingte Kompilation von
#define DEBUG_ON // in os_debug.h	os_debug.h,
#ifdef DEBUG_ON // in os_debug.h	falls die Datei vorhanden ist...
#ifndef DEBUG_ON // in os_kernel.h	...sonst alle Makros leer
// api debug hooks	API-Hooks
// **************************************	
// os control	OS-Steuerung
#define ON_OS_EXCEPTION	
#define ON_OS_TERMINATE	
// task control	Task-Steuerung
#define ON_OS_STOP_CURRENT_TASK	
#define ON_OS_STOP_TASK	
#define ON_OS_START_TASK	
#define ON_OS_START_TASK_ISR	
#define ON_OS_START_BACKGROUND_TASK	
// timer	Timer-Hooks
#define ON_OS_TIMERTICK	
#define ON_OS_TIMERTICK_TIMER_EXP	
#define ON_OS_SET_TIMER	
#define ON_OS_WAIT_TIMER	
// semaphore	Semaphor-Hooks
#define ON_OS_SIGNAL_SEM	
#define ON_OS_SIGNAL_ISR_SEM	
#define ON_OS_WAIT_SEM	
// message box	Mailbox-Hooks
#define ON_OS_PEEK_MESSAGE	
#define ON_OS_WAIT_MESSAGE	
#define ON_OS_POST_MESSAGE_SUCCESS	
#define ON_OS_GET_BUFMESSAGE_SUCCESS	
#define ON_OS_GET_BUFMESSAGE_EMPTY	
#define ON_OS_GET_BUFMESSAGE_NOMSG	
//	
// kernel debug hooks	Kernel Hooks
// **************************************	
// set task ready on event (ev dispatch)	
#define ON_EVENT_DISPATCH_SEM_SIGNALED	Semaphor-Event eingetroffen
#define ON_EVENT_DISPATCH_MSG_FOUND	Message-Event
#define ON_EVENT_DISPATCH_SEM_TIMED	Semaphor-Event oder Timeout
#define ON_EVENT_DISPATCH_MSG_TIMED	Message-Event oder Timeout
#define ON_EVENT_DISPATCH_TIMER_EXP	Timer-Event
// schedule	
#define ON_SCHEDULE_ENTRY	Eintritt in den Scheduler
#define ON_SCHEDULE_TASK_INIT_JUMP	Start-up: Tasks initialisieren
#define ON_SCHEDULE_EVENT_PRIO_MODE_RESUME	Task wiederaufnehmen wegen Event
#define ON_SCHEDULE_RR_NEW_RESUME	Task Round Robin wiederaufnehmen
#define ON_SCHEDULE_NO_TASK_RESUME	keine Task bereit
#define ON_SCHEDULE_IDLE_LOOP_ENTRY	
#define ON_SCHEDULE_IDLE_LOOP	Idle-Loop
#define ON_SCHEDULE_IDLE_LOOP_EXIT	
// os execute	
#define ON_EXECUTE_ENTRY	OS-Start-up
#define ON_EXECUTE_SETJMP_ENTRY	Kontext-Save für Restart
#define ON_EXECUTE_INIT_ALL	re-initialisieren
#define ON_EXECUTE_INITCALL_TASK_FN	Task-Funktionen aufrufen, Init
#define ON_EXECUTE_TERMINATE_OS	OS beenden
#endif **// DEBUG_HOOKS**	

Fehlerbehandlung, OS_Exception()

- ***Anwender-Error-Codes sind in der Datei os_kernel.h zu definieren.***

Bei jedem auftretenden Fehler im System wird die Funktion OS_Exception() aufgerufen. Sie beendet die Ausführung von OS_Execute() - also der Applikation. In der Variable OS.ExceptionCode (Kurzform ERROR) steht ein globaler Fehlercode zur Auswertung bereit. Die Fehlerbehandlung muss *nach* der Funktion OS_Execute() eingefügt werden.

Tabelle - Fehlercodes

Fehlercode	*Beschreibung, Ursache*
ERR_VOID	kein Fehler
ERR_EXIT_OS	unbekannter Fehler
ERR_ARRAY_BOUNDARY	Array-Grenze überschritten
ERR_TASK_RETURN	Task-Funktion ohne Endlosschleife
ERR_EVENT_NOTIFY	Ueberlauf des Ereigniszählers, eine Task blockiert
ERR_ALL_TASKS_BLOCKED	alle Tasks sind gestoppt
ERR_SEM_OVFL	Semaphor-Ueberlauf
ERR_MSGBUF_OVFL	Die Mailbox ist voll (Buffer Overflow)
Fehler in einer ISR	
ERR_ISR_NOTIFY	System-Timeout, eine Task blockiert
ERR_ISR_SEMOVFL	Semaphor-Ueberlauf in einer User-ISR

Beim Auftreten einer dieser Fehler wird OS_Execute() verlassen. Die Fehlercodes sind enthalten in der globalen Systemvariablen ERROR.

Listing - Fehlerbehandlung

```
// Fehlerbehandlung

void main(void)   // Applikation
{
    while (1) {
        InitApp();
        OS_Execute();

        // Fehlerbehandlung
        Display("Task = ", CURRENT);   // Fehler in dieser Task
        if (ERROR == ERR_...) {        // kein ISR-ERROR
            FehlerbehandlungNormal();  // und Restart
        }
        else if (ERROR == ERR_ISR_...) {
            FehlerbehandlungISR();     // ISR-ERROR
            while (1);                 // Stopp
        }
    }
}
```

Die Fehlerbehandlung kann unterscheiden, ob der Fehler in einer ISR oder in einer Task auftrat, dazu wird der OS-Fehlercode abgefragt (ERROR).

Hinweis: Ein Restart der Applikation erfolgt mittels einer Endlosschleife, d.h. einem Neuaufruf von InitApp() und OS_Execute() wie im Beispiel.

Fehlerbehandlung, vom Anwender definiert

Der Anwender kann in der Datei os_kernel.h auch selbst definierte Error-Codes einfügen und die Funktion OS_Exception(USER_ERRCODE) in der Anwendung als Error-Trap (Halt bei Fehlern) einsetzen.

Quellen, Literatur

Grundlage der Dokumentation sind Texte und Artikel von verschiedenen Autoren u.a.

J. Labrosse: *Embedded Systems Building Blocks*, CMP Books, 2002
Kern, Hj.: *MiniOS, Betriebssysteme, einige Begriffe und Grundlagen*, 2005