MultiTasking Handbuch

E-LAB AVRco

Pascal Multi-Tasking für Single Chips

Version für

AVR

Blaise Pascal Mathematiker 1623-1662

Inhaltsverzeichnis

Multi Tasking mit dem E-LAB AVRco

von Gunter Baab, revidiert von "Harry"

Prinzipielles

MultiTasking und MultiTasking Betriebssysteme / -Compiler sind ein extrem vielschichtiges Thema, das immer wieder kontrovers diskutiert wird. Selbst für extrem leistungsfähige CPUs, die typischerweise bei Netzwerk Servern eingesetzt werden, gibt es keinen "Königsweg" für die Konzeption der System Software. Noch gespaltener sind die Meinungen, ob man bei MikroControllern –angesichts der beschränkten Ressourcen- ein MultiTasking Konzept verfolgen sollte. Konsequenter Weise ist es beim AVRco MultiTasking auch "nur" eine Option.

Um das MultiTasking System des AVRco zu verstehen und optimal einzusetzen, ist es unumgänglich, sich mit der Umsetzung auf Assembler Ebene –zumindest grob und prinzipiell- vertraut zu machen.
Ist MultiTasking importiert (durch ***Import** Tasks*, ***Import** Processes*), bindet der AVRco den *Scheduler* in den Code ein. Der Scheduler sorgt dafür, dass die im Programm definierten eigenständigen Module (Main, Prozesse und Tasks) scheinbar parallel abgearbeitet werden.
Dies geschieht indem jedes Modul (maximal) eine gewisse Zeit "läuft" und dann das nächste Modul "drankommt". Somit wird der Scheduler zum eigentlichen "Hauptprogramm", dessen Aufgabe es ist, ein Modul nach dem anderen zu starten und auch zu beenden.
Das "Hauptprogramm" ist damit vorgegeben (-> der Scheduler), sodass auch das Pascal *Main* zu einem normalen Modul (genauer: zu einem Prozess) "degradiert" werden muss.

Um es gleich ganz deutlich herauszustellen:
Prozesse und Tasks bilden die Module des Multitasking Systems und werden ausschließlich durch das System verwaltet. Sie können und dürfen niemals wie ein Unterprogramm aufgerufen werden, auch wenn sie formal den Aufbau eines Unterprogramms haben bzw. haben können.

Prozesse

obwohl ein Prozess formal die Struktur eines Unterprogramms hat, ähnelt er in seiner Arbeitsweise eher einem Main, wo mittels **Loop – Main Prog - EndLoop** eine Endlosschleife definiert ist.
Die Endlosschleife von Prozessen wird im AVRco Pascal nicht explizit programmiert. Dennoch erstellt der Compiler daraus eine Codesequenz, deren letzter Befehl ein Jump auf den Anfang der Codesequenz ist.

Wird ein Prozess zum ersten Mal vom Scheduler gestartet, entspricht das (vereinfacht gesagt – s.u.) einem einfachen Assembler Jump Befehl zum Anfang des Prozess. Ab dann würde dieser Prozess auf alle Ewigkeiten seine Endlosschleife abarbeiten und der Scheduler (und somit ein anderer Prozess/Task) bekäme niemals die Kontrolle.

An dieser Stelle kommt der *SysTick* ins Spiel.

SysTick

Der SysTick ist ein periodischer Timer Interrupt, der das aktive Modul (z.B. den gerade besprochenen Prozess) regelmäßig (Empfehlung: alle 10 msek.) unterbricht. Dazu wird ein 8-Bit Timer (Timer 0 oder Timer 2) des Controllers exklusiv verwendet.
In der Interrupt Service Routine dieses Interrupts (dem *"SysTick"*) werden, abhängig von den importierten Treibern (SysTimer, ADC, ...), vielfältige Funktionen ausgeführt, die an dieser Stelle jedoch nicht von Interesse sind.

Danach wird die Kontrolle jedoch an den Scheduler übergeben.
Der Scheduler entscheidet jetzt, welches Modul als nächstes weiter arbeitet. Dies kann der gerade unterbrochene Prozess oder auch ein völlig anderes Modul sein.
Die Entscheidung des Schedulers ist u.A. von den eingestellten *Prioritäten* (s.u.) abhängig.

Wie oben erwähnt, läuft jeder Prozess (mit Unterbrechungen) endlos "im Kreis". Das heisst jedoch, dass jeder Prozess, wenn er nach einer Unterbrechung wieder die Kontrolle erhält, seinen alten Zustand z.B. seine Register und seinen Stack (s.u.: *Stack, Frame*) wieder vorfinden muss.
Dies wiederum bedingt, dass **jeder** Prozess seinen eigenen Bereich für diese Laufzeitumgebung haben muss (also einen Bereich im RAM wo diese Informationen zwischengespeichert werden).
Bevor der Scheduler einem Prozess die Kontrolle zurück gibt, muss er also auch dessen Laufzeit-umgebung wieder herstellen bzw. –beim ersten Aufruf- diese initialisieren. Wenn ein Prozess unterbrochen wird, muss der Scheduler andererseits dessen Laufzeitumgebung sichern

Bemerkung:
Dies ist leider aufwändig, da die ATMEL Controller nur einen Registersatz besitzen. Controller mit mehreren Register-Sätzen vereinfachen ein Multitasking erheblich, da dann einfach und schnell zumindest der richtige Registersatz eingeschaltet werden kann.

Ein Prozess läuft also so lange bis die ihm zugeteilte Rechenzeit aufgebraucht ist (s.u.: *Prioritäten*). Hat jedoch ein Prozess gerade nichts zu tun, wäre dies eine Ressourcen-Verschwendung. Deshalb gibt es einige Techniken um dies zu verhindern. Beispielsweise kann eine Prozess mit dem Befehl *Schedule* vorzeitig beendet (und dem Scheduler die Kontrolle übergeben) werden oder mit *Suspend (self)* sogar komplett abgebrochen / deaktiviert werden (bis er mit *Resume* wieder aktiviert wird).
Somit ist i.A. nicht vorhersagbar ist wie oft ein Prozess läuft. Das wäre nur dann möglich. wenn es nur Prozesse gäbe, diese immer aktiv wären und alle ihre volle Rechenzeit ausnutzen würden.
Andererseits gibt es Aufgabenstellungen die erfordern, dass gewisse Module in einem genauen zeitlichen Raster abgearbeitet werden. Aus diesem Grund wurden *Tasks* implementiert.

Tasks

ein Task wird vom Scheduler in einem festen zeitlichen Raster aufgerufen (z.B. alle 50ms – s.u.: *Prioritäten*). Tasks sind von ihrer Assembler Struktur her einfache Unterprogramme, die mit einem "Return from Subroutine" enden und damit dem Scheduler (der den Task gestartet = der dieses Unterprogramm aufgerufen hat) die Kontrolle zurückgeben .

Ein "ordentlicher" Task läuft also von Anfang bis Ende durch und setzt, im Gegensatz zu einem Prozess, seine Arbeit niemals später fort, da er ja nicht durch den Scheduler unterbrochen wurde.
Daraus ergibt sich, dass es für alle Tasks nur **einen einzigen Bereich für die Laufzeitumgebung** gibt, was eine enorme Ersparnis an Ressourcen bedeuten kann.
Weiterhin folgt daraus, dass jeder Task nur **kürzer als einen Systick** laufen darf.

Läuft ein Task zu lange, wird er vom Systick abgebrochen und startet bei seinem nächsten Aufruf wieder von seinem Beginn. Ein derartiger Task wird seine komplette Arbeit also nie beenden können.

Prioritäten

Jedes Modul (jeder Task, jeder Prozess und auch der Main-Prozess) hat eine *Priorität*. Die Priorität ist eine ganze Zahl und kann bei der Definition des Tasks/Prozess oder später mit *SetPriority* festgelegt/ geändert werden.

Beispiele:
```
Process Proc1 (32, 32 : iData; 5, suspended);   // prio= 5, kein automatischer Start
Task Task1 (iData, 8, suspended);               // prio= 8, kein automatischer Start
```

Bemerkung:
Die Priorität eines Moduls und dessen Status (*suspended/resumed*) sind System Variablen, die den Scheduler steuern. Diese können mit den entsprechenden Funktionen (*SetPriority, Suspend, Resume*) jederzeit geändert werden, unabhängig davon ob das Modul bzw. der Scheduler gerade aktiv sind.

Ohne explizite Angabe einer Priorität bekommen Tasks die Priorität 5, Prozesse die Priorität 3 und der Main Prozess ebenfalls die Priorität 5. Die Einheit der Prioritäten ist SysTicks (s.u.).

Wesentlich ist nun die unterschiedliche Bedeutung der Priorität bei Tasks und Prozessen:

- hat ein Task die Priorität *n*, so wird er alle *n* SysTicks aufgerufen. Je geringer die Priorität eines Tasks ist, desto häufiger wird er aufgerufen.

- hat ein Prozess die Priorität *n*, läuft er maximal *n* SysTicks bevor der Scheduler den nächsten Prozess startet. Je geringer die Priorität eines Prozess ist, desto weniger CPU Zeit bekommt er zugeteilt.

Nach dem Start des Multitasking Systems (s.u. *Start_Processes*) bekommt zunächst der *Main Prozess* die Kontrolle bis dieser durch den ersten SysTick unterbrochen wird. Im SysTick werden immer zunächst die Hintergrund Aufgaben erledigt und dann wird der Scheduler gestartet.

Der Scheduler startet im **1.SysTick den 1.Task** (den der Compiler vorgefunden hat = der als erster Task in der Source definiert ist). Sobald der seine Arbeit beendet hat, kommt der aktuell laufberechtigte Prozess dran, bis dieser durch den 2.SysTick unterbrochen wird.

Der Scheduler startet dann im **2.SysTick den 2.Task** ... dann den aktuell laufberechtigten Prozess, bis dieser durch den 3.SysTick unterbrochen wird.

Der Scheduler startet dann im **3.SysTick den 3.Task** ... dann den aktuell laufberechtigten Prozess bis dieser durch den 4.SysTick unterbrochen wird.
usw.

Wurden alle Tasks einmal gestartet, bestimmt deren Priorität wann sie wieder aufgerufen werden und es wird i.A. nicht mehr bei jedem SysTick ein Task gestartet (s.u. "Randbedingungen").

Die restliche Zeit (in der kein Task läuft) steht den Prozessen zur Verfügung. Hat der Main Prozess z.B. die Priorität=3, so ist (nach dem 3.Task im 3.SysTick) der nächste Prozess (den der Compiler vorgefunden hat = der als erster Prozess in der Source definiert ist) laufberechtigt und bekommt die Kontrolle.

Bemerkung:
Daraus folgt, dass die Priorität eines Prozess dessen *maximale* Rechenzeit angibt. Davon geht die Zeit für die Berechnungen im SysTick und ggf. die Laufzeit für einen Task ab. Diese Zeiten fallen aber typischerweise gegenüber der Laufzeit eines Prozess nicht ins Gewicht.

Wichtig: **pro SysTick kann maximal ein einziger Task laufen.**

Randbedingungen für die Prioritäten

Sind mehrere Tasks aktiv, so gibt es wichtige Randbedingungen an deren Prioritäten, da ansonsten ein zyklischer Aufruf nicht möglich ist:

1.: Die Prioritäten aller Tasks müssen einen gemeinsamen Nenner haben

warum? Schauen wir uns ein (falsches !) Beispiel mit Task1/Priorität=3 und Task2/Priorität=4 an.
Jetzt müsste

*Task1 bei Tick Nummer 1, 4, 7, **10**, 13, 16, 19, **22**, ... und*
*Task2 bei Tick Nummer 2, 6, **10**, 14, 18, **22**, ... laufen.*
d.h. beim 10. und 22. (usw.) SysTick müssten eigentlich zwei Tasks laufen, was nicht möglich ist.

Gibt man jedoch beiden Tasks eine (korrekte) Priorität=3, so läuft

Task1 bei Tick Nummer 1, 4, 7, 10, 13, 16, 19, 22, ... und
Task2 bei Tick Nummer 2, 5, 8, 11, 14, 17, 20, 23, ...

Warum aber nicht beiden Tasks die Priorität=2 geben? Das würde doch heissen

Task1 bei Tick Nummer 1, 3, 5, 7, 9, ... und
Task2 bei Tick Nummer 2, 4, 6, 8, 10, ...

Auch dann müssten ebenfalls niemals zwei Tasks laufen.

Wenn aber nun beide Tasks ihre maximal erlaubte Rechenzeit (= 1 SysTick!) fast komplett ausnutzen, würde für die vorhandenen Prozesse (zumindest der Main Prozess ist ja immer vorhanden) praktisch keine Rechenzeit mehr übrig bleiben.
Im zweiten Beispiel jedoch haben die Prozesse die SysTicks 3, 6, 9, ... exklusiv für deren Bedarf.
Daraus folgt die 2. Randbedingung:

2.: Die kleinste Priorität muss (zumindest um 1) größer als die Anzahl der Tasks sein

wie man im zweiten Beispiel sieht, ist für alle Prozesse nur jeder dritte SysTick garantiert als Laufzeit reserviert (plus die Zeit, den die Tasks nicht verbrauchen).
Dies kann, insbesondere wenn mehrere Prozesse aktiv sind, immer noch dazu führen, dass diesen zu wenig Rechenzeit zur Verfügung steht.
Auf der sicheren Seite ist man, wenn man die
kleinste erlaubte Priorität größer als (Anzahl Tasks + Anzahl Prozesse)
setzt.
Also z.B. bei 2 Tasks, 2 Prozessen + Main Prozess sollte die kleinste erlaubte Priorität=6 sein.

> Bemerkung.:
> Müssten (bei einer falschen Prioritäten Vergabe) in einem SysTick mehrere Tasks laufen, so bestimmt die Reihenfolge der Task Definition (in der Source) welcher Task verzögert wird.
> Durch "*besonders ungeschickte Wahl*" der Prioritäten kann man es sogar "*erreichen*", das ein ganz "unten definierter" Task völlig zum Erliegen kommt, weil -jedes Mal wenn er eigentlich laufen müsste- ein "weiter oben definierter" Task die Kontrolle erhält.
>
> Negativ Beispiel:
> definiert sind
> Task1/Priorität=2, und
> Task2/Priorität=4. Dann wird
> (vielleicht nach einiger Zeit bei einer schnellen Erweiterung und ohne groß nachzudenken)
> Task3/Priorität=4 definiert.
> Somit müsste
> *Task1 bei Tick Nummer 1, 3, 5, 7, 9, 11, 13, 15 ...,*
> *Task2 bei Tick Nummer 2, 6, 10, 14, 18, ... und*
> *Task3 bei Tick Nummer 3, 7, 11, 15, ...*
> laufen.
> Wie man sieht, wird Task3 in diesem Fall vollständig von Task1 am Laufen gehindert.

Ich hoffe, es ist nun klar, dass man keine einfache und allgemein gültige Vorgabe für die optimale Vergabe der Prioritäten machen kann. Es kann durchaus Applikationen geben, wo von den obigen Regeln abgewichen werden kann. Man muss sich dann aber ganz genau über die Zusammenhänge und Konsequenzen im Klaren sein:

z.B.
- laufen alle Tasks garantiert nur extrem kurz
- ist in den Prozessen kaum etwas zu tun, sodass diese auch kaum Rechenleistung benötigen
- ist vielleicht garantiert, dass immer nur ein Teil der Module gleichzeitig aktiv ist

usw.

Wenn nötig kann man dann sogar die Prioritäten während der Laufzeit an den jeweiligen Betriebszustand der Applikation dynamisch anpassen und dadurch jederzeit die optimale Performance erzielen. Die Wartung und Erweiterung solcher komplexen Programme artet jedoch schnell zu einem Albtraum aus.

Start_Processes

Wie erwähnt, wird der Scheduler durch die Definition von Tasks und/oder Prozessen (in den Assembler Code) eingebunden und regelmäßig durch den SysTick aufgerufen.
Der SysTick ist ein Timer Interrupt. Damit dieser arbeitet muss das Interrupt System normalerweise mit *EnableInts* gestartet werden. Ist jedoch MultiTasking implementiert, muss *Enable_Ints* durch *Start_Processes* **ersetzt** werden, damit auch der Scheduler aufgerufen wird.

Diverse System Funktionen des Multi Tasking Systems

Der AVRco beinhaltet eine Vielzahl von Funktionen um das Verhalten des MultiTasking System zu steuern.
Diese sind im Compiler Handbuch im Kapitel "*Multi-Task Funktionen*" beschrieben und sollten mit obigem Hintergrundwissen dort hinreichend erläutert sein.

..\E-LAB\DOCs\DocuCompiler.pdf - Kapitel "Multi-Task Funktionen"

An dieser Stelle soll daher nur noch auf einige ausgewählte Themen eingegangen werden, deren Sinn und Einsatzmöglichkeiten weniger offensichtlich sind.

Stack und Frame

Der Stack wird im AVRco für die gleichen Aufgaben wie in einem Non-MultiTasking System benutzt: zum Speichern der Return-Adressen bei Unterprogramm Aufrufen / Interrupts und zur Übergabe von Variablen (bzw. deren Adresse) an Unterprogramme.
Der Frame wird zur Speicherung von lokalen Variablen und, falls Unterprogramme aufgerufen werden, auch zum temporären Speichern von Übergabe Parametern benutzt.

Wie oben erläutert, gibt es für alle Tasks einen gemeinsamen Stack- und Frame Bereich, der im Programmkopf definiert wird.
Z.B.:

```
Define
  ...
 TaskStack     =     $0020, iData;
 TaskFrame     =     $0010; //TaskFrame benutzt den gleichen Speicher wie TaskStack !
```

Der Task mit dem größten Bedarf bestimmt dabei die notwendige Größe des Stack- und Frame.

Da Prozesse unterbrochen werden und später ihre alte Umgebung wieder vorfinden müssen, hat **jeder** Prozess seinen **eigenen** Stack- und Frame Bereich.
Für den Main Prozess werden diese Bereiche ebenfalls im Programmkopf definiert.
Z.B.:

```
Define
  ...
 StackSize     =     $0064, iData;
 FrameSize     =     $0064, iData;
```

Für alle weiteren Prozesse werden deren jeweiliger Stack- und Frame Bereiche bei der Definition individuell festgelegt:

```
Process ProcessName (StackSize, FrameSize : word; DataPage);
```

Auch hier wird für Stack und Frame stets der gleiche Speicher Bereich ("DataPage") benutzt.

Die notwendige Größe der Stack- und Frame Bereiche werden (unabhängig vom MultiTasking) also einzig durch den Aufbau der jeweiligen Programm Module bestimmt:

- benutzen diese Unterprogramme und wie tief sind die verschachtelt
- wie groß ist der Bedarf für Übergabe Parameter
- wie groß ist der Bedarf für lokale Variable

Eine generelle Empfehlung ist somit nicht möglich.

Sollten Stack/Frame jedoch nicht ausreichen, werden die Daten anderer Prozesse überschrieben und ein System Absturz ist unausweichlich. Eine ständige Überprüfung des freien Speichers ist jedoch aus Performance Gründen bei einem uC-System nicht möglich.

Der einfachste Ausweg aus dieser Zwickmühle ist es, das Programm intensiv im Simulator zu testen und dabei möglichst alle denkbaren Zustände nachzustellen. Im Fenster *"Processes"* kann man sich dann mittels des Buttons *"State"* den maximalen Verbrauch an Stack/Frame anschauen.

Weiterhin gibt es einige System Routinen, die zur Laufzeit den aktuellen Verbrauch ermitteln.
Diese sind im Kapitel "*Multi-Task Funktionen*" im Kapitel "*Stack und Frame Verbrauch*" beschrieben und sollten in schwierigen Fällen ausreichend sein, um derartige Probleme einzugrenzen.

Locked Variable

Ein Problem bei MultiTasking ist ein konkurrierender Zugriff auf Daten

Beispiel:
Ein Prozess schreibt gerade eine globale Variable (was bei komplexeren Typen wie *integer* oder gar *float* einige Assembler Befehle benötigt) und wird -während des Schreibens- durch einen anderen Prozess unterbrochen. Wenn der unterbrechende Prozess diese Variable dann liest, ist diese noch nicht vollständig aktualisiert und es kommt Schott dabei raus.

Lösungen:
Entweder definiert man die Variable als *"locked"*, wodurch jeder Zugriff in "**DisableInts**" – (Zugriff) – "**EnableInts**" gekapselt wird.

```
var xyz : integer, locked;
```

Nachteil:
auf diese Variable darf nicht permanent (in einer Schleife) zugegriffen werden (polling) weil dann fast ständig die Interrupts gesperrt sind und das MultiTasking nicht mehr ordentlich funktionieren kann.

Oder der Prozess verhindert, dass er unterbrochen wird. Dies ist mit

```
Lock (self);
xyz := ...
Unlock (self);
```

leicht möglich.
Nachteil:
der zyklische Aufruf von Tasks wird dadurch ggf. gestört.

DeviceLock

Ein weiteres Problem ist ein konkurrierender Zugriff auf die Hardware.
z.B. schreibt Prozess A in die erste Zeile eines Displays und Prozess B in die zweite. Während des Schreibens wird nun Prozess A durch Prozess B unterbrochen, der Schreibcursor wird durch Prozess B in die zweite Zeile positioniert und Zeile 2 wird geschrieben. Nun bekommt Prozess A die Kontrolle zurück und beendet das gerade unterbrochene Schreiben - der Cursor steht dann jedoch irgendwo in Zeile 2 ==> Chaos.
In derartigen Fällen kann ein DeviceLock (Bool'sche Variable) eingesetzt werden. Beide Prozesse müssen den Lock beachten (dürfen nur Schreiben wenn das Display nicht gerade durch den anderen Prozess gelockt ist) und anwenden (setzten / <dann schreiben> / löschen).

Pipes

Pipes ermöglichen einen einfachen Datenaustausch zwischen einzelnen Modulen.
z.B. empfängt/generiert Modul A Daten, die dann Modul B bearbeitet.
Dazu könnte Modul A einen Speicherbereich reservieren, die Daten dort schreiben und Modul B die Daten dort wieder lesen.
Dies ist jedoch wenig performant, da Modul B regelmäßig aufgerufen werden muss - nur um die Existenz neuer Daten zu checken (*"polling"*).
Eleganter ist es wenn der Scheduler Modul B nur dann startet wenn wirklich Daten vorhanden sind.
Dazu kann man eine Pipe definieren, einen Speichertyp (FIFO), der auch dem Scheduler bekannt ist.
In Modul B reicht der Befehl *WaitPipe(<Pipe-Name>)* und der Scheduler erledigt die Aufgabe, das Modul B erst dann wieder zu starten wenn in der Pipe auch Daten von Modul A abgelegt wurden.
Man muss sich dabei auch nicht um den "Füllstand" der Pipe kümmern: Modul A kann schreiben bis alles erledigt ist (vorausgesetzt die Pipe ist groß genug) und Modul B liest bis die Pipe "leer" ist.

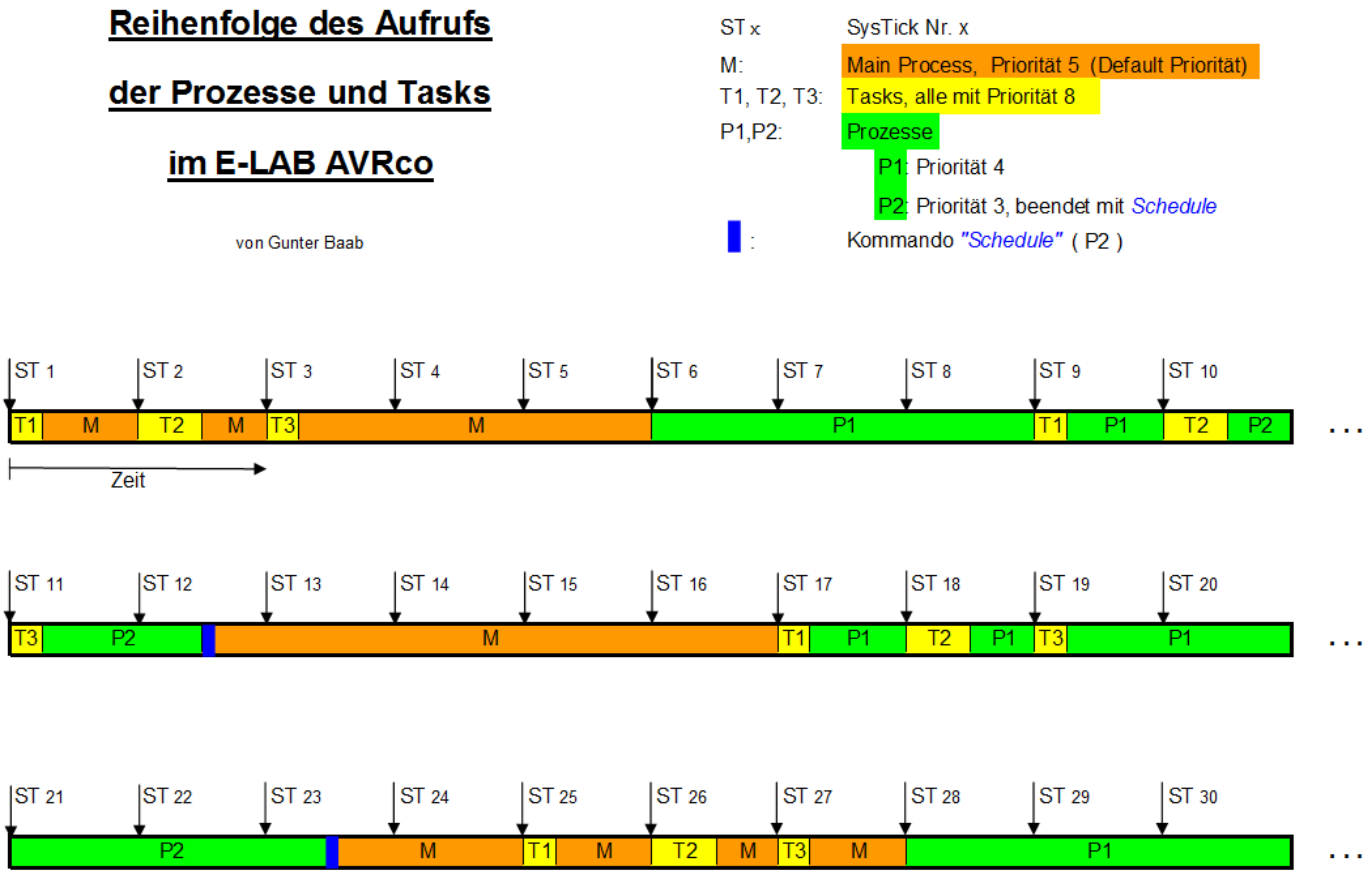
Semaphore

Semaphore werden eigentlich zur Synchronisation von konkurrierenden Ressourcen verwendet.
Auch wenn diese eher bei großen Systemen (mit Multi Kernel CPUs, Multi Path Datenkanälen usw.) zum Einsatz kommen, haben sie auch bei Mikrocontrollern Einsatzmöglichkeiten.
Die Semaphore werden intern als Byte gespeichert und haben einen Wert (0..255). Insoweit ist auch ein DeviceLock eine spezielle Semaphore, die nur zwei Werte (False/True) annehmen kann.
Analog zu WaitPipe existiert ein *WaitSema(<Sema-Name>)*. Sobald die Semaphore > 0 ist, aktiviert der Scheduler das wartende Module und dekrementiert die Semaphore.

Eine Anwendung könnte z.B. beim Einsatz eines WLAN Chips mit mehreren Kanälen ("Sockets") sein. Es ist gleichgültig welchen der Kanäle ein Prozess benutzt und eine Semaphore könnte benutzt werden um die Anzahl der freien Sockets zu verwalten. Prozesse die senden wollen warten dann mit *WaitSema* bis ein freier Kanal zur Verfügung steht.

Aufrufreihenfolge

Das folgende Diagramm zeigt den Wechsel der Tasks und Prozesse im MultiTasking



Notizen

Grombacherstr. 27
D74906 Bad Rappenau

Tel. 07268/9124-0
Fax. 07268/9124-24

Internet: www.e-lab.de
e-mail: info@e-lab.de