Ziel des Multitaskings ist es mehrere Prozesse (Threads) quasi gleichzeitig auszuführen. Die Aufgabe kann nur durch hin und herschalten zwischen den Prozessen erreicht werden.

Der Kontext

Damit man einen Prozess unterbrechen kann und später wieder weiter ausführen kann muss man den exakten Zustand (Kontext) rekonstruieren können. Der Zustand eines Prozesses besteht aus den Daten auf dem Stack und dem Inhalt der CPU-Register. Die einfachste Art den Zustand bei einer Unterbrechung zu sichern ist die CPU-Register auf den Stack zu schreiben. Wenn der Thread später weiter ausgeführt werden soll, können sie CPU Register wieder von Stack geladen werden. Da jeder Thread seinen eigenen Stack hat kommt dabei nichts durcheinander.

Das Sichern des Kontextes ist in Assembler geschrieben und nennt sich saveContext() , die umgekehrte Routine zur Wiederherstellung nennt sich loadContext()
.

Der Kontext:

Addresse	Register
SP+36	PC_LO
SP+35	PC_HI
SP+34	R0
SP+33	SREG
SP+32	RAMPZ
SP+31	R31
SP+30	R30
SP+29	R29
...	...
SP+3	R3
SP+2	R2
SP+1	R1
SP+0	*- frei -*

Die Reihenfolge der Register auf dem Stack ergibt sich aus folgenden Gründen:

Als erstes wird die aktuelle Position im Programm (der Programmcounter - PC) auf den Stack geschrieben. Da es sich um einen 16 Bit Wert handelt benötigt er 2 Byte (Lo+Hi). Die Speicherung des PC erfolgt automatisch durch den Prozessor, entweder durch den Funktionsaufruf der task_yield() Funktion oder durch die Ausführung des Timer-Interrupts.

Danach wird zunächst das Register R0 gesichert, damit danach ein Register zur freien Verfügung bereit steht.

Als nächstes wird das Status Register (die CPU-Flags) gesichert, damit es im weiteren Verlauf verwendet werden kann. Da es sich hierbei um ein IO-Register handelt muss es zunächst in ein CPU Register geladen werden. Kein Problem, da das Register R0 ja schon gesichert wurde!

Anschliessend muss bei AVR Modellen mit mehr als 64k Speicher noch das RAMPZ Register gesichert werden.

Nachdem noch alle übrigen CPU Register (R1-R31) gesichert worden sind ist der Kontext gespeichert.

Das Wiederherstellen eines Kontextes geschiet genau in der umgekehrten Reihenfolge!

Die Thread Datenstruktur

Zusätzlich werden noch ein paar Daten für jedem Thread benötigt:

```
typedef struct _thread_s {
 cdll_t list;
 uint8_t priority;
 uint8_t flags;
 uint16_t signal_cnt;
 uint16_t wait_timeout;
 void * stackpointer;
} thread_t;
```

Durch den cdll_t Eintrag **list** kann unser Thread in eine doppelt verkettete Liste eingefügt werden - er kann damit 'Mitglied' in genau einer Gruppe werden.

Jeder Thread hat eine Priorität, je kleiner der Wert von priority ist desto wichtiger ist der Thread. Threads mit der Priorität 0 werden immmer vor Threads mit einer höheren Priorität ausgeführt.

- **Multitasking Teil 1**

- **Multitasking Teil 3** **(TODO)**