

# Jefax

**Ein Kernel für Atmels XMega**

FaMe

JeGa

22. Juni 2014

HTWG Konstanz

Jefax ist ein einfacher Betriebssystemkernel, der preemptives Scheduling und damit Multitasking realisiert. Zusätzlich sind einfache Synchronisierungsmechanismen wie Mutex und Condition Variable implementiert. Die dynamische Speicherverwaltung erlaubt die Entwicklung flexibler Datenstrukturen und Tasks. Über die serielle UART Schnittstelle ist eine Shell zur Ausgabe von ASCII Strings verfügbar.

# Inhaltsverzeichnis

<b>1</b>	<b>Tasks</b>	<b>3</b>
1.1	API . . . . .	3
1.2	Implementierung . . . . .	4
<b>2</b>	<b>Scheduling</b>	<b>5</b>
2.1	Dispatcher . . . . .	5
2.2	Scheduler . . . . .	7
2.2.1	Zustandsautomat . . . . .	8
2.2.2	Round Robin Scheduler . . . . .	9
<b>3</b>	<b>Timer</b>	<b>12</b>
<b>4</b>	<b>Dynamische Speicherverwaltung</b>	<b>14</b>
4.1	API . . . . .	14
4.2	Implementierung . . . . .	15
4.3	Verbesserungsmöglichkeiten . . . . .	17
<b>5</b>	<b>Message System / Shell</b>	<b>18</b>
5.1	Message Queues . . . . .	18
5.2	Messages . . . . .	19
5.3	USART Kommunikation . . . . .	20
5.4	USART Kommunikation Implementierung . . . . .	21
5.4.1	Empfangen von Messages . . . . .	21
5.4.2	Senden von Messages . . . . .	22
5.5	Shell Task . . . . .	22
5.5.1	Implementierung . . . . .	23
<b>6</b>	<b>Fazit</b>	<b>24</b>
	<b>Literaturverzeichnis</b>	<b>25</b>
	<b>Abbildungsverzeichnis</b>	<b>26</b>

# 1 Tasks

Wie in einem multitasking Betriebssystem üblich, können vom Benutzer verschiedene Tasks erzeugt werden die vom Scheduler gescheduled werden. Intern werden Tasks für den Dispatcher sowie optional für die Shell verwendet. Jede Task besitzt ihren eigenen Task-Kontext. Bei einem Kontextwechsel wird dieser gesichert bzw. wiederhergestellt. Verschiedene Schedulingverfahren erlauben das quasi-parallele Ausführen der Tasks.

```
1 typedef struct {
2     int (*function)();
3
4     unsigned int priority;
5     volatile taskState_t state;
6
7     uint8_t *stackpointer;
8
9     uint8_t stack[STACK_SIZE];
10 } task_t;
```

task.h

Ein Task wird durch die Struktur `task_t` repräsentiert. Diese besteht aus:

- Pointer auf die Task Funktion
- Priorität der Tasks
- Taskstatus (READY, RUNNING, BLOCKING)
- Stackpointer, der auf die nächste freie Speicheradresse auf dem Stack zeigt.
- Ein Feld von `uint8_t`, welches den Stack darstellt.

Der Stack besitzt eine feste Größe, welche durch das define `STACK_SIZE` festgelegt werden kann. Wichtig zu wissen ist hierbei, dass der Stack sich nicht im Compiler-spezifischen Stackbereich befindet (stack section - beginnend von der letzten Adresse im SRAM), sondern in der data Section.

## 1.1 API

Zum Erzeugen einer Task wird die Funktion `initTask()` verwendet. Aufgabe dieser Funktion ist es, den Stack sowie den Stackpointer zu initialisieren.

```
1 void initTask(task_t *task);
```

Um zu bestimmen, welche Tasks gescheduled werden sollen, wird das Feld `TASKS` in `jefax.c` verwendet. Ein Benutzer kann hier die zu schedulenden Tasks eintragen. Als letztes Element in dem Feld muss sich immer ein leeres `task_t` Element befinden. Optional kann das Makro `SHELL_TASK` zum aktivieren der Shell verwendet werden.

```
1 task_t TASKS[] = {  
2     {counterTask2, 2, READY, 0, {0}},  
3     SHELL_TASK,  
4     {0, 0, READY, 0, {0}}  
5 };
```

jefax.c

Ein Eintrag im `TASKS` Feld besteht aus:

- Pointer auf die Task-Funktion
- Priorität der Task
- Initialer Status der Tasks (meist `READY`)
- Stackpointer (muss immer 0 sein)
- Stack (muss immer 0 sein)

## 1.2 Implementierung

Da bei den XMega Microcontrollern der Stack von der höchsten zur niedrigsten Adresse wächst, wird der Stackpointer mit

```
1 task->stackpointer = task->stack + STACK_SIZE - 1;
```

initialisiert. Anschließend wird die Adresse der Task Funktion auf dem Stack abgelegt (bei den XMega128 3 Byte). Zuletzt werden noch die 32 Arbeitsregister sowie das Statusregister auf dem Stack benötigt.

Um vom Dispatcher/Scheduler/... auf die globale `TASKS` Struktur zuzugreifen, wird dieser in den jeweiligen Dateien als extern deklariert.

```
1 extern task_t TASKS[];
```

## 2 Scheduling

Essentieller Bestandteil von Jefax ist das Scheduling. Die Tasks im Kernel laufen preemptiv, d.h. sie können unterbrochen und ausgewechselt werden. Damit die Tasks ihre Unterbrechung nicht bemerken, wird der Kontext der unterbrochenen Task gespeichert (siehe 2.1 Dispatcher).

Die 2 Kernbestandteile des Scheduling sind der Dispatcher und der Scheduler. Der Standardscheduler realisiert ein prioritätengesteuertes Round-Robin-Verfahren. Tasks erhalten eine feste Zeitscheibe, in der sie die CPU erhalten, danach werden sie durch den Dispatcher unterbrochen.

### 2.1 Dispatcher

Der Dispatcher übernimmt das auswechseln der Tasks. Kernteil des Dispatcher ist dabei ein Timerinterrupt. Der Timer wird initial auf einen bestimmten Wert gestellt, welcher die Zeitscheibe jeder Task repräsentiert. Dieser kann jedoch auch manuell, nach der Initialisierung des Dispatcher, mit

```
1 void setInterruptTime(unsigned int p_msec);
```

gesetzt werden.

Während der Interrupt Service Routine (ISR) wird zuerst der Kontext der unterbrochenen Task gesichert. Zum Kontext gehören folgende Speicherbereiche:

- Sämtliche Register des XMega (R0 bis R31)
- Das Statusregister (SREG)
- Der Stackpointer

Die Register sowie das SREG werden auf dem Task eigenen Stack abgelegt. Der Stackpointer wiederum wird direkt im Task Control Block (TCB) gesichert. Außerdem wird beim Auftritt der ISR der Program Counter (PC) automatisch auf dem Stack der Task abgelegt.

Nach dem Sichern des Kontext wird in der ISR die Dispatcher Task geladen. Dabei handelt es sich nicht um eine Task, die vom Scheduler berücksichtigt wird. Der Dispatcher besitzt lediglich einen eigenen TCB und damit auch einen Stack, auf dem er

arbeiten kann.

Folgender Code Ausschnitt zeigt die ISR:

```
1 ISR(TCC0_OVF_vect , ISR_NAKED)
2 {
3     SAVE_CONTEXT();
4     getRunningTask()->stackpointer = (uint8_t *) SP;
5
6     TODO: main stack context
7
8     // set stackpointer to default task
9     initTask(&dispatcherTask);
10    SP = (uint16_t) (dispatcherTask.stackpointer);
11
12    DISABLE_TIMER(TCC0);
13    RESTORE_CONTEXT();
14    reti();
15 }
```

dispatcher.c

In Zeile 3 und 4 wird der Kontext der aktuellen Task gesichert. Zeile 7 reinitialisiert die Dispatcher Task. Der Dispatcher soll bei jedem Aufruf von vorne beginnen, sodass ein Speichern und Wiederherstellen des Kontext unnötig ist. Durch Zeile 8 und 11 wird der Kontext der Dispatcher Task geladen. Der Aufruf von `reti()` in Zeile 12 holt den PC vom Stack des Dispatcher und springt an diese Adresse. `DISABLE_TIMER(TCC0)` deaktiviert den Timer, der den Dispatcher Interrupt auslöst. So wird verhindert, dass der Dispatcher und Scheduler Zeit von der Zeitscheibe der nächsten Task verbraucht.

Die eigentliche Taskfunktion der Dispatcher Task sieht folgender Maßen aus:

```
1 static int runDispatcher()
2 {
3     task_t* toDispatch = schedule();
4     dispatch(toDispatch);
5     return 0;
6 }
```

dispatcher.c

Zuerst wird der Scheduler aufgerufen (Zeile 3). Dieser entscheidet, welche Task als nächstes eingewechselt werden soll (siehe 2.2 Scheduler). Die erhaltene Task wird daraufhin eingewechselt (Zeile 4).

Das Einwechseln der Task sieht dem letzten Teil der oben genannten ISR sehr ähnlich. Der Stackpointer wird auf denjenigen der ausgewählten Task gewechselt und die Register werden vom Stack geholt. Außerdem wird der Timer TCC0, der in der ISR ausgeschaltet wurde, wieder aktiviert.

## 2.2 Scheduler

Der Scheduler des Kernels entscheidet welche der vorhandenen Tasks als nächstes eingewechselt werden soll. Daher ist er entscheidend für das Verhalten des Kernels und auch dessen Performance. Der Scheduler kann zur Laufzeit durch die Funktion

```
1 void setScheduler(scheduler_t *p_scheduler);
```

geändert werden. In Jefax wird der Scheduler durch die Struktur `scheduler_t` repräsentiert.

```
1 typedef struct
2 {
3     void (*init)();
4     task_t* (*getNextTask)();
5     void (*taskStateChanged)(task_t*);
6     void (*taskWokeUp)(task_t*);
7     taskList_t *readyList;
8     taskList_t *blockingList;
9 } scheduler_t;
```

scheduler.h

Um einen eigenen Scheduler zu nutzen, müssen die Callback Funktionen der Struktur implementiert werden. Die Funktion `init()` wird aufgerufen, sobald der Scheduler als neuer Scheduler des Kernels bestimmt wurde. Dabei müssen vor allem die `readyList` und die `blockingList` so bearbeitet werden, dass der Scheduler mit den Listen arbeiten kann.

Die Funktion `getNextTask()` wird vom Dispatcher aufgerufen (siehe 2.1 Dispatcher) und bestimmt welche Task als nächste eingewechselt wird. Als Rückgabewert wird die einzuwechselnde Task oder NULL, wenn keine Task gefunden wurde, erwartet. Wird NULL zurückgegeben wird die sogenannte `idleTask` eingewechselt, die nichts tut.

Mit dem Callback `taskStateChanged(task_t*)`, wird dem Scheduler signalisiert, dass eine Task ihren Zustand geändert hat. Der Scheduler sollte entsprechend auf diese Änderung reagieren. Als Parameter wird die betroffene Task übergeben.

Der Aufruf von `taskWokeUp(task_t*)` scheint auf den ersten Blick dem Aufruf von `taskStateChanged(task_t*)` sehr zu ähneln. Jedoch gibt es einen bedeutenden Unterschied: `taskWokeUp(task_t*)` läuft im Interrupt Kontext, es darf also nicht auf den nächsten Timer interrupt gewartet werden (z.B. durch schlafen, yield, etc).

Die beiden Listen `readyList` und `blockingList` werden gesetzt sobald der Scheduler mithilfe von `setScheduler(scheduler_t*)` als neuer Scheduler bestimmt wird. Sie beinhalten die Tasks, die den entsprechenden Zustand haben. Auf die aktuell laufende Task kann mit der Funktion

```
1 task_t *getRunningTask();
```

zugegriffen werden.

Die zentrale Funktion des Schedulers, die auch vom Dispatcher aufgerufen wird `schedule()` ist folgender Maßen implementiert:

```
1 task_t* schedule()
2 {
3     task_t *result;
4     runningTask = scheduler->getNextTask();
5     if(runningTask == NULL)
6         result = &idleTask;
7     else
8         result = runningTask;
9     result->state = RUNNING;
10    return result;
11 }
```

scheduler.c

Zuerst wird die nächste Task durch den Aufruf des Callbacks `getNextTask()` des aktuellen Schedulers ausgewählt (Zeile 4). Die Variable `runningTask` ist dabei global definiert und stellt die aktuell laufende Task dar.

In Zeile 5 bis 8 wird überprüft, ob der Scheduler eine lauffähige Task gefunden hat. Dies wird durch den Rückgabewert `NULL` des Callbacks signalisiert. Wird keine Task gefunden, wird die sogenannte `idleTask` dem Dispatcher zum einwechseln gegeben.

Der Rückgabewert `result` (siehe Zeile 3) der Funktion stellt schlussendlich die Task dar, die vom Dispatcher eingewechselt wird (vergleiche 2.1 Dispatcher).

### 2.2.1 Zustandsautomat

In Jefax gibt es 3 Taskzustände: `ready`, `blocking`, `running`. Der Übergang zwischen diesen Zuständen wird in Abbildung 2.1 dargestellt.



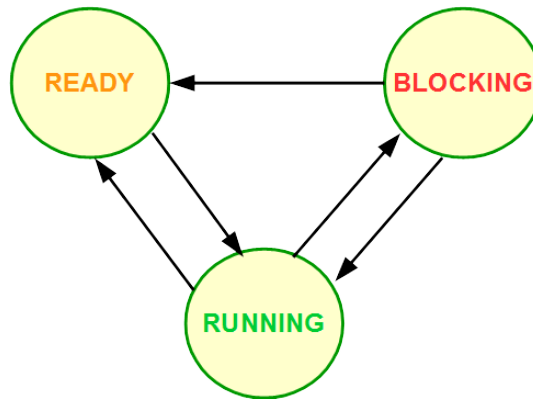


Abbildung 2.1: Zustandsautomat

Alle Tasks starten initial mit dem Zustand ready. Aus diesem Zustand heraus können die Tasks nur in den Zustand running wechseln (siehe Abbildung 2.1), indem sie vom Scheduler ausgewählt und durch den Dispatcher eingewechselt werden.

Aus dem Zustand running kann die Task in die Zustände blocking oder ready wechseln, je nach dem welcher Ereignisse zur Laufzeit auftreten. Von blocking gibt es entweder die Möglichkeit in den Zustand ready oder direkt in den Zustand running zu wechseln.

Hierbei handelt es sich jedoch nur um eine Empfehlung für die Zustandsänderung. Wie der Zustandsautomat tatsächlich aussieht hängt von der Implementierung der Funktion `taskStateChanged(task_t*)` des Schedulers ab.

## 2.2.2 Round Robin Scheduler

Der einzige standardmäßig implementierte Scheduler in Jefax - und damit auch der Defaultscheduler - ist ein prioritätengesteuerter Round Robin Scheduler. Dieser kann über die Funktion

```
1 scheduler_t *getRRScheduler();
```

genutzt werden.

Der Scheduler implementiert sämtliche Callbacks der `scheduler_t` Struktur. Die `init()` Funktion sortiert die `readyList` nach Priorität der Tasks, damit Scheduler vernünftig mit der Liste arbeiten kann. Dabei steht die Task mit höchster Priorität (niedrigstem Wert) an letzter Stelle der Liste.

Die Implementierung der Funktion `void taskStateChanged(task_t*)` prüft, ob die übergebene Task in den Zustand ready gewechselt hat, und wenn ja, ob die Task eine höhere Priorität als die momentan laufende Task besitzt. In diesem Fall wird die aktuell laufende Task ausgewechselt. Außerdem prüft die Funktion, ob die `runningTask` selbst ihren Zustand

weg von running geändert hat. Auch in diesem Fall wird die runningTask ausgewechselt.

Die Funktion `taskWokeUp(task_t*)` funktioniert ähnlich. Auch hier wird geprüft, ob die aufgewachte Task eine höhere Priorität als die laufende Task hat. Ist dies der Fall wird ein Kontextwechsel erzwungen, jedoch nicht auf den Interrupt gewartet. Da diese Funktion im Interruptkontext abläuft, würde so ewig auf den Interrupt gewartet werden.

Die wohl komplexeste Funktion des Schedulers ist `getNextTask()`. Aufgrund ihrer Länge wird diese Funktion hier Teil für Teil besprochen. Die grobe Struktur sieht in Pseudocode so aus:

```
1 static task_t* getNextTaskRR()
2 {
3     readyUpBlockingTasks();
4     result = chooseFromReadyTasks();
5     addRunningTaskToList();
6     return result;
7 }
```

Im ersten Schritt in Zeile 3, werden alle Tasks in der blockingList, deren Zustand nicht mehr blocking und damit ready ist (siehe 2.2.1 Zustandsautomat), in die readyList einsortiert. Dabei werden sie auch wieder entsprechend ihrer Priorität in die Liste eingefügt.

Der nächste Schritt (Zeile 4), in dem die nächste Task aus der Ready List ausgewählt wird hat folgende Implementierung:

```
1 //ready list is empty
2 if(isEmpty(schedulerRR.readyList))
3 {
4     if(NO_TASK_SCHEDULED() || RUNNING_TASK_IS_BLOCKING())
5         result = NULL;
6     else
7     {
8         getRunningTask()->state = RUNNING;
9         result = getRunningTask();
10    }
11 }
12 else
13 {
14     //...
15 }
```

schedulerRR.c

Zuerst wird unterschieden, ob die readyList leer ist (Zeile 2). Ist dies der Fall, kann keine neue Task ausgewählt werden. Es kann lediglich die aktuelle Task weiterlaufen oder keine Task eingewechselt werden. Hierbei wird geprüft, ob überhaupt eine Task gescheduled ist und ob die runningTask überhaupt weiterlaufen kann, also nicht den Zustand blocking hat (Zeile 4 bis 10). Kann die runningTask nicht weiterlaufen wird als

Rückgabewert NULL festlegt (Zeile 5), ansonsten die runningTask (Zeile 8).

Ist die readyList nicht leer (else Teil in Zeile 12 bis 15), muss der Scheduler eine Task aus der Liste herausuchen. Der Code dazu lautet:

```
1 // get next task with highest priority
2 result = getLast(schedulerRR.readyList);
3
4 //next task would have lower prio, keep running task
5 if(RUNNING_TASK_IS_RUNNING() && result->priority > getRunningTask()->
   priority)
6     result = getRunningTask();
7 else
8 {
9     popTaskBack(schedulerRR.readyList);
10
11     if(RUNNING_TASK_IS_RUNNING())
12         getRunningTask()->state = READY;
13 }
```

schedulerRR.c

Zuerst holt sich der Scheduler die Task mit der höchsten Priorität aus der readyList ohne sie daraus zu entfernen (Zeile 2). Die Task mit der höchsten Priorität steht immer an der letzten Stelle der Liste.

Als nächstes wird untersucht, ob die aktuell laufende Task noch lauffähig ist und falls ja, ob sie eine höhere Priorität besitzt als die Task aus der readyList (Zeile 4). Ist dies der Fall, wird die runningTask nicht ausgewechselt und die andere Task auch nicht aus der readyList entfernt. Der Rückgabewert der Funktion ist dann die runningTask (Zeile 5).

Im anderen Fall wird die gewählt Task aus der readyList entfernt (Zeile 8) und falls die runningTask noch den Zustand running hat, der Zustand der runningTask auf ready geändert (Zeile 11 und 12).

Im letzten Schritt der `getNextTask()` Funktion des Round Robin Schedulers wird die runningTask entsprechend ihres Zustandes in die blocking- oder readyList eingeordnet.

**Anmerkung:** Die hier benutzten Makros `RUNNING_TASK_IS_...()` berücksichtigen den Fall, dass runningTask `NULL` ist und geben in diesem Fall `false` zurück.

## 3 Timer

Jefax verfügt über eine eigene Timer Komponente, mit der Funktionen in relativen Zeitabständen ausgelöst werden können. Ein Timer wird durch die Struktur `timer_t` dargestellt.

```
1 typedef struct{
2     void (*callBack) (void*);
3     void *arg;
4     volatile unsigned int ms;
5 } timer_t;
```

timer.h

Ein Timer verfügt über eine Callback Funktion, die aufgerufen wird, sobald der Timer abgelaufen ist. Der Funktion kann auch ein Argument vom Typ `void*` übergeben werden. Die verbleibende Zeit (in Millisekunden) des Timers wird in der Variable `ms` gespeichert.

Um einen eigenen Timer zu initialisieren, wird die Funktion

```
1 int initTimer(timer_t *p_timer, int p_ms, void (*p_callBack) (void*),
    void * p_arg);
```

timer.h

aufgerufen, wobei die Callbackfunktion, deren Argument und die Zeit, nach der der Timer zuschlagen soll, übergeben werden.

Über den Aufruf

```
1 int addTimer(timer_t p_timer);
```

timer.h

kann ein Timer aktiviert werden. Nach dem Aufruf dieser Funktion schlägt der Timer frühestens nach der bei `initTimer()` angegebenen Zeit zu.

Bei beiden Funktionen gilt ein Rückgabewert von 0 als Erfolg.

Intern werden die Timer in einer Liste verwaltet und es wird ein Hardware Timer genutzt, um nach der entsprechenden Zeit zuzuschlagen. Dieser Hardware Timer wird auf den niedrigsten Wert aller Timer aus der Timerliste gesetzt, sodass das System tickless arbeitet. Dies geschieht immer dann, wenn eine Änderung in der Timerliste vorgenommen wird.

Während der ISR des Timers wird zuerst der Kontext der unterbrochenen Task gespeichert. Daraufhin werden alle Timerwerte um den Wert des Hardwaretimers erniedrigt.

Daraufhin werden alle Timer darauf überprüft, ob sie abgelaufen sind. Ist dies der Fall, wird der Timer aus der Liste entfernt und dessen Callback ausgeführt.

**Anmerkung:** Damit laufen die Callback Funktionen im Interruptkontext, der Benutzer muss also auf Schlafen und ähnliches verzichten.

## 4 Dynamische Speicherverwaltung

Für die dynamische Speicherverwaltung wurden eigene malloc und free Implementierungen erstellt. Diese erlauben das dynamische Reservieren von Speicher auf dem Heap durch verschiedene Tasks. Die Speicherverwaltung wird durch Interrupt-sperren geschützt, um race conditions zu verhindern. Alle Funktionsaufrufe sind nicht blockierend und dürfen auch von ISRs aufgerufen werden. Ist kein Speicher mehr verfügbar, geben die Funktionen einen Fehlerwert zurück. Die `allocateMemory()` Funktion prüft bei Speicheranforderungen den vergebenen und noch verfügbaren Speicher, um Kollisionen mit dem Stack zu vermeiden. Freigegebener Speicher wird in einer Free List verwaltet, bei weiteren Speicheranforderungen wird zunächst der Speicher der Free List verwendet.

### 4.1 API

Um Speicher zu reservieren, wird die Funktion `allocateMemory()` verwendet. Als Argument bekommt diese die Größe des Speicherbereiches in Byte. Der Rückgabewert ist bei Erfolg ein Pointer auf den allokierten Speicherbereich. Bei einem Fehler bei der Speicheranforderung (kein Speicher mehr verfügbar) ist dieser 0.

```
1 void *allocateMemory(uint8_t size);
```

memory.h

Um den angeforderten Speicher wieder freizugeben, wird die Funktion `freeMemory()` benutzt, welche als Argument einen Pointer auf den freizugebenden Speicherbereich bekommt. Nach dem Aufruf der Funktion sollte nicht mehr über den Pointer auf den Speicherbereich zugegriffen werden.

```
1 void freeMemory(void *mem);
```

memory.h

Um Informationen zum aktuellen Speicherverbrauch abzufragen, kann die Funktion `dumpMemory()` verwendet werden. Diese liefert ein Objekt der Struktur `memoryInfo` zurück.

```
1 typedef struct
2 {
3     char *heapStart;
4     char *nextFreeMemory;
5     int heapAllocated;
6     int freeListEntries;
7 } memoryInfo;
```

Die Felder haben folgende Bedeutung:

- heapStart: Start Adresse des Heaps (durch linker übergeben)
- nextFreeMemory: Nächste freie Speicheradresse die vergeben werden kann (Ohne berücksichtigung der Free-List)
- heapAllocated: Anzahl an Bytes die auf dem Heap allokiert wurden
- freeListEntries: Anzahl an Einträgen in der Free-List

## 4.2 Implementierung

Die dynamische Speicherverwaltung ist lose an die avr-gcc malloc Implementierung gehalten. Es wurde darauf geachtet, möglichst wenig Overhead durch erforderliche Zusatzinformationen zu erzeugen. Bei einem Aufruf von `allocateMemory()` wird Speicher auf dem Heap reserviert. Der Heap beginnt nach der `.bss` section und wächst richtung Stack (4.1).

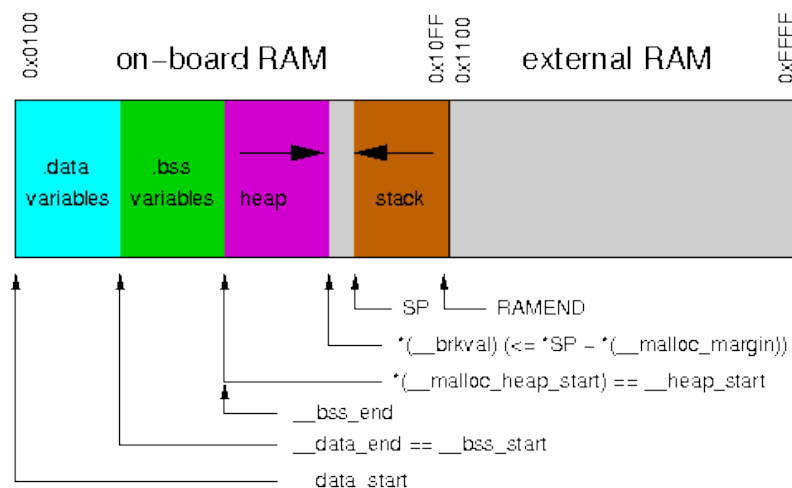


Abbildung 4.1: Sections

Der Anfang des Heaps wird über das Linker Script an den Compiler/Linker übergeben. Die Start-Adresse kann über die Variable `__heap_start` abgefragt werden.

```
1 // Defined from linker script
2 extern char __heap_start;
3 char *heapStart = &__heap_start;
```

Der allozierte Speicher wird durch die `memoryArea` Struktur verwaltet.

```
1 typedef struct memoryArea {  
2     uint8_t size;  
3     struct memoryArea *next;  
4 } memoryArea;
```

memory.c

Vor jedem angeforderten Speicherbereich wird die Größe dieses Speicherbereichs (Variable size) abgelegt. Dadurch weiß `freeMemory()`, wie viel Byte freigegeben werden müssen. Die Variable `next` erfüllt zwei Zwecke. Bei erfolgreicher Reservierung ist dies die Speicheradresse, die dem Benutzer zurück gegeben wird. Beim Freigeben des Speichers wird dieser Pointer für die verkettete List der Free-List verwendet. Der Vorteil dieses System ist, dass keine separaten Listen benötigt werden, um die Kontrollinformationen zu speichern. Die einzige Restriktion stellt die `size` Variable dar. Da vor jedem Speicherbereich die Größe gespeichert wird, ist der Overhead pro Speicherreservierung die Größe dieser Variable (1 Byte).

Bild 4.2 verdeutlicht diesen Zusammenhang nochmals. Es sind drei Speicherblöcke (vom Typ `memoryArea`) abgebildet. Der mittlere (blaue) Speicherblock wurde vom Benutzer reserviert und ist in Benutzung, die beiden grünen Speicherblöcke wurde wieder freigegeben und befinden sich in der Free List. In jedem Speicherbereich sind die Variablen `size` und `next` abgebildet (benötigt für die Kontrollinformationen). Beim reservieren des Speicherblocks blau wurde die Adresse der Variable `next` an den Benutzer gegeben. Die einzige Kontrollinformation stellt die Variable `size` (= 20Byte) dar. In der Free List befinden sich zwei Elemente, die Variable `next` zeigt jeweils auf den nächsten Eintrag der Free List.

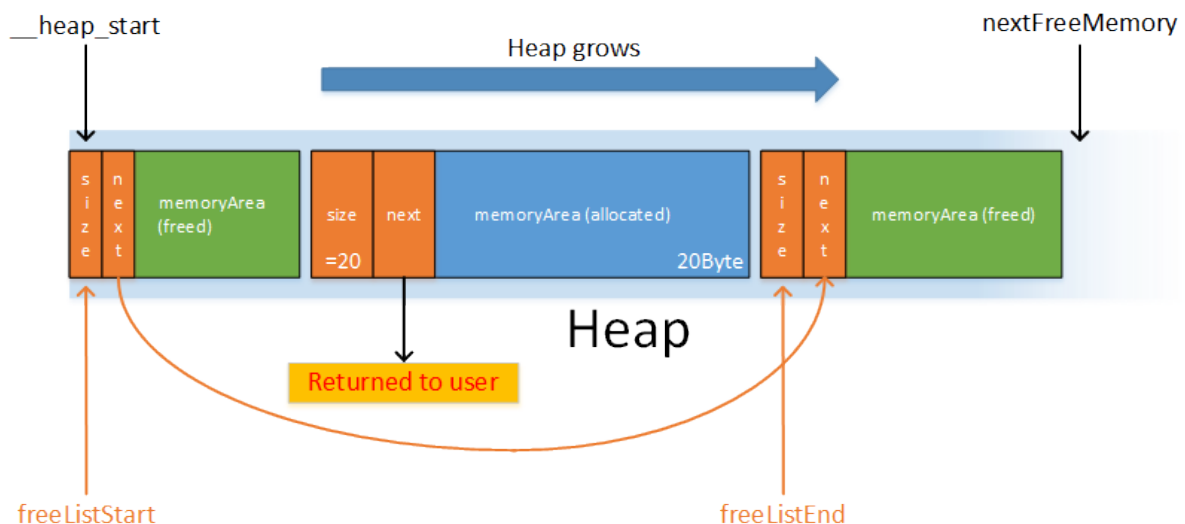


Abbildung 4.2: Dynamische Speicherverwaltung



Beim reservieren von Speicher wird zunächst der verfügbare Speicher sowie auf Stack-Kollision geprüft. Kann Speicher reserviert werden, wird die Adresse des nächsten verfügbaren Speichers auf ein Pointer auf eine `memoryArea` Struktur gecastet. Anschließend wird die Größe des Speicherbereichs gesetzt sowie die nächste freie Speicheradresse berechnet. Die Funktion gibt die Adresse der next Variable an den Benutzer als benutzbaren Speicher auf dem Heap zurück.

```
1 newMemoryArea = (memoryArea *) nextFreeMemory;
2 newMemoryArea->size = size;
3
4 nextFreeMemory += size + sizeof(uint8_t);
5
6 // ...
7
8 return &(newMemoryArea->next);
```

memory.c

Freigegebener Speicher wird in der Free-List verwaltet. Die Free-List besteht aus einer verketteten Liste von `memoryArea` Elementen. Bei der Speicheranforderungen wird zuerst die Free-List betrachtet. Wird ein passender Eintrag in der Free-List gefunden, wird dieser aus der Free-List entfernt und dem Benutzer zurück geliefert. Wird kein passender Eintrag gefunden, wird überprüft, ob durch ein größeres Element in der Free-List die Speicheranforderung erfüllt werden kann. Ist dies nicht der Fall, wird neuer Speicher aus dem Heap zurück geliefert.

Da Heap und Stack aufeinander zu wachsen, besteht hier die Gefahr einer Kollision. Um dies zu vermeiden, wird bei Speicheranforderungen auf eine Kollision überprüft. Der Heap kann nicht weiter als bis zu einem festgelegten Abstand zum Stack wachsen. Dieser Abstand bestimmt die Variable `margin`. Ist diese Grenze erreicht, kann `allocateMemory()` keinen Speicher mehr allozieren und liefert 0 zurück. Obwohl es keinen freien Speicher mehr auf dem Heap gibt, kann die Anwendung weiterhin mit dem Stack (der noch `margin` Bytes frei hat) arbeiten und stürzt nicht ab.

Um race-conditions zu vermeiden und die Speicherverwaltung vor Interrupts zu schützen, werden beim allozieren und freigeben Interrupts gesperrt.

## 4.3 Verbesserungsmöglichkeiten

Eine Verbesserungsmöglichkeit der Speicherverwaltung wäre eine Funktion zum defragmentieren des Speichers. Durch häufiges allozieren und freigeben von Speicher unterschiedlicher Größe kann es zu Fragmentierung und Speicherbrauch kommen. Zusätzlich zum Defragmentieren kann ein Buddy-System implementiert werden, welches durch aufsplitten und mergen von Speicherbereichen noch effektiver arbeiten kann.

## 5 Message System / Shell

Zur seriellen Kommunikation wird die USART Schnittstelle des XMega verwendet. Das Senden und Empfangen erfolgt Interrupt-gesteuert. Ein Message System wurde implementiert, um einfach Nachrichten über USART senden und empfangen zu können. Um zu verhindern, dass Messages verloren gehen oder Buffer überlaufen, wird die dynamische Speicherverwaltung zum Verwalten der Messages und Queues verwendet. Eine Shell ermöglicht das komfortable Kommunizieren mit dem XMega über USART. Die Shell läuft als eigener Task und wird wie alle Tasks geschedult.

Zum Verwalten der USART Nachrichten werden verschiedene Buffer verwendet, welche in der Dateien `usart_message.h` und `usart_queue.h` deklariert sind.

### 5.1 Message Queues

Zum Empfangen und Senden werden Message Queues (verkettete Listen) vom Typ `messageQueue` verwendet.

```
1 struct messageQueue
2 {
3     message *tail;
4     message *head;
5     int size;
6 };
```

Zum Erzeugen und Zerstören von Message Queues werden die Funktionen

```
1 messageQueue *getMessageQueue();
2 void destroyMessageQueue(messageQueue *queue);
```

verwendet. Diese verwenden intern zum Reservieren/Freigeben des Speichers der Queue die dynamische Speicherverwaltung. Beim Freigeben des Speichers der Queue wird auch der Speicher aller Elemente in der Queue freigegeben.

Zum Hinzufügen und Auslesen von Messages aus einer Queue werden die Funktionen

```
1 void enqueue(messageQueue *queue, message *msg);
2 message *dequeue(messageQueue *queue);
```

benutzt. Wichtig ist, dass die übergebene Message nach einem Aufruf von `enqueue()` von der Message Queue verwaltet wird und keine Kopie dieser Message gemacht wird. Es wird kein Aufruf von `destroyMessage()` für die übergebene Message benötigt.

## 5.2 Messages

Um Daten zu senden und zu empfangen werden Messages verwendet.

```
1 struct message
2 {
3     char *data;
4     int size;
5
6     int stackIndex;
7     MSG_TYPE type;
8
9     message *next;
10};
```

Die Variable type bestimmt dabei, ob die Art des Messages.

- TX\_MSG: Messages um über USART zu senden.
- RX\_MSG: Messages um über USART zu empfangen.

Die Variable stackIndex wird verwendet, um eine Message als Stack benutzen zu können. je nach MSG\_TYPE können Byte-weise Daten gepusht oder gepopt werden. Dies erlaubt ein komfortables Senden und Empfangen bei der USART Kommunikation, da über USART immer nur 1 Byte gesendet/empfangen werden kann. Beispielsweise kann in jedem USART data-receive interrupt jeweils ein Byte auf die aktuelle Message gepusht werden. Es entfällt das manuelle Berechnen des nächsten freien Bytes in der Message. Um Daten zu Senden, können je nach bedarf verschiedene Funktionen benutzt werden.

Wie bei Message Queues gibt es jeweils eine Funktion zum erzeugen und zerstören von Messages. Es wird ebenfalls die dynamische Speicherverwaltung verwendet. `getMessage()` bekommt als Parameter die Größe der Message in Bytes, sowie den Message Typ, welcher den Stack-Zugriff vorgibt.

```
1 message *getMessage(int dataSize, MSG_TYPE type);
2 void destroyMessage(message *msg);
```

Zum setzen des Inhalts der Message wird `setMessageData()` verwendet. Die übergebenen Daten werden dabei in den Message Buffer kopiert.

```
1 int setMessageData(message *msg, char *data, int size);
```

Um die Daten einer Message auszulesen werden die folgenden Funktionen verwendet:

```
1 char getMessageDataCopy(message *msg, char **data, int *size);
2 char *getMessageData(message *msg);
```

`getMessageDataCopy()` liefert eine Kopie der Daten zurück (`char **data`). Das bedeutet, diese Daten müssen nach Benutzung mit `freeMemory()` freigegeben werden. `getMessageData` liefert einen Pointer auf die Daten der Message zurück. Dieser Speicher darf nicht mit `freeMemory()` freigegeben werden. Auch muss beim Zugriff auf diese Adresse sichergestellt sein, dass die dazugehörige Message noch nicht gelöscht wurde. Wird eine Kopie einer Message inklusive Daten benötigt kann folgende Funktion verwendet werden:

```
1 message *copyMessage(message *msg);
```

Vorgang zum Erzeugen von Messages:

```
1 char character = '\n';
2 message *msg;
3
4 msg = getMessage(1, TX_MSG);
5 if (msg == 0)
6     error();
7
8 setMessageData(msg, &character, 1);
9
10 // Use message
11
12 destroyMessage(msg);
```

## 5.3 USART Kommunikation

Die Datei `usart.h` bietet Funktionen zum Initialisieren der USART-Schnittstelle sowie zum Senden und Empfangen von Messages. Um direkt Messages zu empfangen/senden werden folgende Funktionen verwendet:

```
1 void sendMessageUsart(message *msg);
2 message *receiveMessageUsart();
```

Jede Empfangene Message besteht aus einem Null-terminierten String. Nach einem Aufruf von `sendMessageUsart()` wird die Message vom USART System verwaltet. Der Benutzer sollte nicht mehr auf die Message zugreifen. Für die Verwaltung (Speicher freigeben) der zurückgelieferten Message von `receiveMessageUsart()` ist der Benutzer selbst zuständig. Zusätzlich können zum Senden von Strings komfortablere Funktionen verwendet werden. Der Benutzer muss hierbei selbst keine Messages erzeugen.

```
1 void print(char *string);
2 void printChar(char character);
```

`print` erwartet als Paramter einen Null-terminierten String, welcher in den internen Buffer kopiert wird.

## 5.4 USART Kommunikation Implementierung

Die USART Kommunikation erfolgt Interrupt-gesteuert. Folgender Frame Format wird verwendet:

*Asynchronous Communicatin Mode, no parity, 1 Stop Bit, 8 Bit data*

Als Standard-Wert ist eine Baudrate von 115200 eingestellt. Baudrate und USART Geräte können über die folgenden Defines eingestellt werden:

```
1 #define USART USARTC0
2 #define USART_PORT PORTC
3 #define BAUDRATE 115200
```

Zum Senden und Empfangen werden Low-Level Interrupts verwendet. Wurde ein Byte empfangen, wird der USARTC0\_RXC\_vect Interrupt getriggert. Können Daten in den USART hardware Buffer kopiert werden, wird der USARTC0\_DRE\_vect Interrupt getriggert (data register empty). Empfangene und zu sendende Messages werden in Message Queues gespeichert. Da das Message System mit dynamischer Speicherverwaltung arbeitet, können keine Messages verloren gehen.

```
1 static messageQueue *rxQueue;
2 static messageQueue *txQueue;
```

Der Zugriff auf die Message Queues wird durch Interrupt-sperren gesichert, da von Tasks sowie von Interrupts auf diese zugegriffen wird. Beide USART Interrupts arbeiten auf dem System Stack.

### 5.4.1 Empfangen von Messages

Es werden so lange Bytes empfangen und in die Message `currentReceiveMessage` gepusht, bis eine neue Zeile erkannt wurde oder eine bestimmte maximale Länge der Message (`MAX_RECEIVE_MSG_SIZE`) erreicht wurde. Erst dann wird eine Message in die receive queue hinzugefügt und es wird eine neue `currentReceiveMessage` erzeugt. Die Empfangsroutine prüft zusätzlich auf verschiedene escape Sequenzen wie DEL, CR, ESC, usw,. Um zu verhindern dass der command prompt gelöscht werden kann wird bei DEL die cursor position abgefragt. Verschiedene Funktionen garantieren, dass die empfangene Message immer nur die minimale Größe besitzt. Bild 5.1 zeigt den groben Ablauf beim Empfangen von Nachrichten.

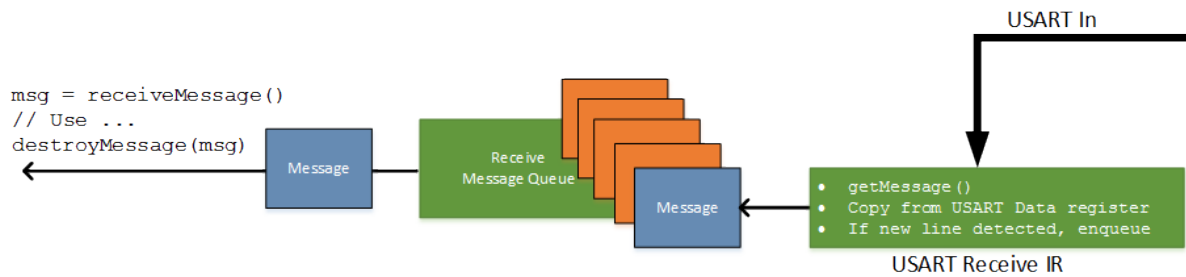


Abbildung 5.1: USART Receive

### 5.4.2 Senden von Messages

Senden erfolgt analog zum Empfangen. Die variable `currentSendMsg` enthält die gerade zu sendende Message. Im DRE Interrupt wird immer 1 Byte von dieser Message gepopt. Wurde diese vollständig gesendet kann der Speicher der Message freigegeben werden und es kann die nächste Message aus der send queue gesendet werden. Der DRE Interrupt wird aktiviert, sobald eine Nachricht gesendet werden soll. Wurden alle Nachrichten aus der Queue gesendet, wird dieser deaktiviert. Bild 5.2 zeigt das Versenden von Nachrichten.

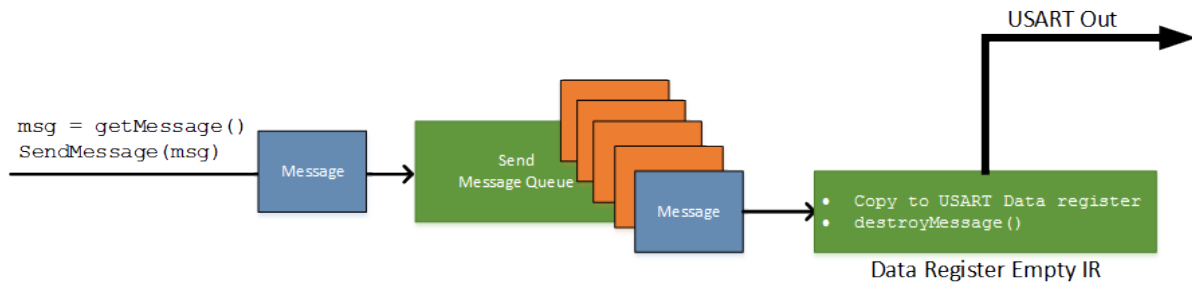


Abbildung 5.2: USART Send

## 5.5 Shell Task

Die Shell Tasks wird wie alle anderen Tasks geschedult und verarbeitet und interpretiert die empfangenen Messages. In der Headerdatei `shell.c` kann der command prompt der Shell konfiguriert werden. Die Funktion

```
1 void setMessageCallback(void (*processMessageCB)(char *msg));
```

erlaubt das Setzen einer Callback-Funktion. Die Shell-Task verarbeitet die Empfangenen Daten und verarbeitet ggf. die Befehle. Alle nicht verarbeiteten Daten werden an die Callback-Funktion übergeben.

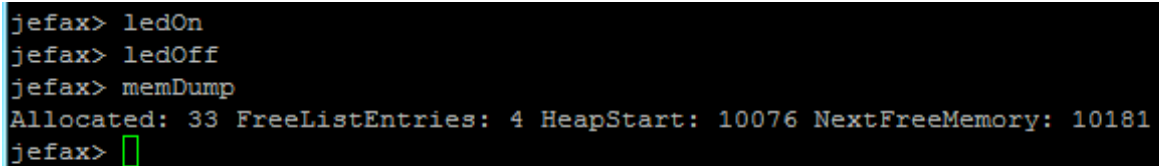
## 5.5.1 Implementierung

Die Funktion `waitForMessage` wird verwendet um aktiv auf eine Message zu warten:

```
1 static message *waitForMessage()
2 {
3     message *msg;
4
5     while (1) {
6         msg = receiveMessageUsart();
7         if (msg != 0)
8             break;
9     }
10
11     return msg;
12 }
```

Wurde eine Message empfangen wird `parseMessage` aufgerufen, welche die Message verarbeitet. Folgende Befehle sind verfügbar:

- `ledOn` Schaltet die led1 an
- `ledOff` Schaltet die led1 aus
- `memDump` Gibt Informationen der dynamischen Speicherverwaltung aus



```
jefax> ledOn
jefax> ledOff
jefax> memDump
Allocated: 33 FreeListEntries: 4 HeapStart: 10076 NextFreeMemory: 10181
jefax> 
```

Abbildung 5.3: jefax shell

## 6 Fazit



# Literaturverzeichnis

- [1] <http://www.nongnu.org/avr-libc/user-manual/malloc.html>

# Abbildungsverzeichnis

- 2.1 Zustandsautomat . . . . . 9
- 4.1 Sections . . . . . 15
- 4.2 Dynamische Speicherverwaltung . . . . . 16
- 5.1 USART Receive . . . . . 22
- 5.2 USART Send . . . . . 22
- 5.3 jefax shell . . . . . 23