

Jefax

Ein Kernel für Atmels XMega

Fabian Meyer

Jens Gansloser

1. August 2014

HTWG Konstanz



Jefax ist ein einfacher Betriebssystemkernel für Atmels XMega Microcontroller, der preemptives Scheduling und damit Multitasking realisiert. Zusätzlich sind Synchronisierungsmechanismen wie Mutex und Condition Variable implementiert. Die dynamische Speicherverwaltung erlaubt die Entwicklung flexibler Datenstrukturen und Tasks. Zusätzlich wurde ein Nachrichtensystem entwickelt, das komfortables und sicheres Senden von Daten über die serielle Schnittstelle erlaubt. Basierend darauf ist über die USART Schnittstelle eine Shell verfügbar.

Inhaltsverzeichnis

1	Einführung	4
2	Tasks	5
2.1	API	5
2.2	Implementierung	6
3	Scheduling	7
3.1	Dispatcher	7
3.1.1	API	7
3.1.2	Implementierung	8
3.2	Scheduler	10
3.2.1	API	10
3.2.2	Implementierung	12
3.2.3	Zustandsautomat	14
3.2.4	Round Robin Scheduler	15
4	Timer	18
4.1	API	18
4.2	Implementierung	19
5	Dynamische Speicherverwaltung	21
5.1	API	21
5.2	Implementierung	22
5.3	Verbesserungsmöglichkeiten	25
6	Message System / Shell	26
6.1	Message Queues	26
6.2	Messages	27
6.3	USART Kommunikation	28
6.4	USART Kommunikation Implementierung	29
6.4.1	Empfangen von Messages	29
6.4.2	Senden von Messages	30
6.5	Shell Task	30
6.5.1	Implementierung	31
7	Tests	32

8 Fazit	33
Literaturverzeichnis	34
Abbildungsverzeichnis	35

1 Einführung

Jefax wurde im Rahmen des Fachs Angewandte Systemprogrammierung entwickelt. Es wurde Wert auf Robustheit und Flexibilität des Kernels gelegt. Erweiterungen sollten einfach möglich sein. Dieses Dokument beschreibt das Application Programming Interface (API) sowie die Implementierung der verschiedenen Komponenten. In Abschnitt 2 bis 4 werden grundlegende Funktionen erläutert, welche essentiell für den jefax Kernel sind. Abschnitt 5 sowie 6 stellen zwei zusätzliche Features des Kernels vor.

2 Tasks

Wie in einem Multitasking-Betriebssystem üblich, können vom Benutzer verschiedene Tasks erzeugt werden, die vom Scheduler geschedult werden. Intern werden Tasks beispielsweise 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

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

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

Der Stack besitzt zur Laufzeit eine feste Größe, welche durch das Symbol `STACK_SIZE` festgelegt werden kann. Dieses Symbol kann zur Kompilierzeit definiert werden, ansonsten wird ein Standardwert verwendet. 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.

2.1 API

Zum Erzeugen einer Task wird die Funktion `initTask()` verwendet. Aufgabe dieser Funktion ist es, den Stack sowie den Stackpointer zu initialisieren. Der Funktionspointer der Taskfunktion muss vor dem Aufruf von `initTask()` gesetzt werden.

```
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     CAR_TASK(1),
3     SHELL_TASK(1),
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)

Der Vorteil dieses `TASKS` Feld ist, dass der Benutzer sehr einfach neue Tasks hinzufügen kann. Er muss lediglich dieses Feld um seine Task erweitern. Der jefax Kernel übernimmt Initialisierung, Scheduling, usw..

2.2 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. Damit auch andere Komponenten (z.B. Scheduler) auf die globale `TASKS` Struktur zugreifen können, wird dieses Feld in den jeweiligen Dateien als extern deklariert.

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

3 Scheduling

Essentieller Bestandteil von Jefax ist das Scheduling. Die Tasks im Kernel laufen pre-emptiv, d.h. sie können unterbrochen und ausgewechselt werden. Um einen sicheren quasi-parallelen Ablauf zu garantieren, muss beim Austausch der Tasks deren Kontext gesichert werden (siehe 3.1 Dispatcher).

Die Kernbestandteile des Scheduling sind der Dispatcher und der Scheduler. Der Standardscheduler von Jefax realisiert ein prioritätengesteuertes Round-Robin-Verfahren. Tasks können somit durch höherpriori Tasks verdrängt werden. Haben mehrere Tasks die gleiche (höchste) Priorität, erhalten sie reihum für eine feste Zeitspanne die CPU. Ist diese Zeitscheibe abgelaufen, werden sie durch den Dispatcher unterbrochen und ausgewechselt, woraufhin die nächste gleichpriori Task an die Reihe kommt.

3.1 Dispatcher

Der Dispatcher übernimmt das Auswechseln und Unterbrechen der Tasks. Das zentrale Element des Dispatchers ist ein Timerinterrupt, der in regelmäßigen Abständen ausgelöst wird. Im Rahmen der Interruptserviceroutine (ISR) wird der Kontext der unterbrochenen Task gesichert, die nächste einzuwechselnde Task ausgewählt (siehe 3.2 Scheduler) und der Kontext der ausgesuchten Task wiederhergestellt.

3.1.1 API

Der Dispatcher unterbricht eine Task immer nach einer festen Zeitspanne. Diese Zeitspanne kann zur Laufzeit mit dem Aufruf der Funktion

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

dispatcher.h

festgelegt werden. Die Funktion bietet eine Auflösung im Millisekundenbereich.

Anmerkung: Der Interrupt kann auch bereits vor Ablauf der Zeitspanne auftreten, wenn beispielsweise eine Task mit einer höheren Priorität als die gerade laufende Task rechenbereit wird.

3.1.2 Implementierung

Bevor der Dispatcher genutzt werden kann muss er mit folgender Funktion initialisiert werden:

```
1 void initDispatcher()
2 {
3     initScheduler(getRRScheduler());
4
5     init32MHzClock();
6     initUsart();
7     initTimeSliceTimer();
8
9     // Save the main context
10    SAVE_CONTEXT();
11    main_stackpointer = (uint8_t *) SP;
12
13    SP = (uint16_t) (getRunningTask()->stackpointer);
14    enableInterrupts();
15    RESTORE_CONTEXT();
16
17    RET();
18 }
```

dispatcher.c

Dieser Aufruf initialisiert zuerst den Standardscheduler (Zeile 3, siehe auch 3.2.2 Implementierung), setzt die Taktfrequenz des Prozessors (Zeile 5), initialisiert die UART-Schnittstelle (Zeile 6) und den Timer, der die Dispatcher Interrupts auslöst (Zeile 7). Danach wird der aktuelle Kontext gespeichert und der Stackpointer als `main_stackpointer` (oder auch Systemstack) gesichert (Zeile 10 - 11). Sämtliche Interruptserviceroutinen innerhalb von Jefax arbeiten auf dem Systemstack.

Während der Initialisierung des Standardschedulers wählt der Scheduler bereits die Task aus, die als erste im System laufen soll (Zugriff über `getRunningTask()`). Der Kontext dieser Task wird in Zeile 13 und Zeile 15 geladen.

Erst im Kontext dieser Task werden die Interrupts systemweit aktiviert (Zeile 15, `enableInterrupts()`). Der Zeitpunkt hierfür ist deshalb so spät gewählt, da sämtliche ISRs in Jefax den Kontext der aktuell laufenden Task sichern. Eine frühere Aktivierung der Interrupts hätte eine Race Condition zur Folge. In diesem Fall könnte bereits ein Interrupt auftreten, bevor das System in dem Kontext einer Task laufen würde.

Der Aufruf dieser Funktion geschieht beim Initialisieren von Jefax (Aufruf `jefax()`).

Die ISRs in Jefax werden durch das folgende Makro definiert:

```

1 #define JEFAX_ISR(vect, func) \
2 ISR(vect, ISR_NAKED) \
3 { \
4     SAVE_CONTEXT(); \
5     getRunningTask()->stackpointer = (uint8_t *) SP; \
6     ENTER_SYSTEM_STACK(); \
7     func(); \
8     SP = (uint16_t) (getRunningTask()->stackpointer); \
9     RESTORE_CONTEXT(); \
10    reti(); \
11 }
```

interrupt.h

Dem Makro `JEFAX_ISR()` muss als erster Parameter der Interruptvektor übergeben werden, der behandelt werden soll. Der zweite Parameter ist eine Funktion, die vorzugsweise vom Typ `void (*func)(void)` sein sollte. Diese Funktion übernimmt die eigentliche Behandlung des Interrupts. Die `JEFAX_ISR()` sichert beim Auftritt des Interrupts den Kontext der aktuell laufenden Task und deren Stackpointer (Zeile 4 - 5). Danach wird der Systemstack betreten (Zeile 6). Auf dem Systemstack wird nun die übergebene Funktion abgearbeitet (Zeile 7). Somit arbeiten alle ISRs in Jefax auf dem Systemstack.

Zum Schluss wird der Kontext der aktuellen (unterbrochenen) Task wieder hergestellt und an deren Unterbrechungspunkt zurückgesprungen (Zeile 8 - 9). Die einzige Ausnahme ist im Falle des Dispatchers: hier muss nicht zwangsläufig zur unterbrochenen Task zurückgesprungen werden. Es wird zu der vom Scheduler ausgewählten Task zurückgesprungen.

Das Sichern des Kontext (`SAVE_CONTEXT()`) umfasst lediglich das Ablegen aller CPU-Register sowie das Statusregister (SREG) auf dem Stack der Task. `RESTORE_CONTEXT()` macht genau das Gegenteil und liest die Register aus dem Stack aus.

Im Rahmen des Dispatchers wird eine `JEFAX_ISR()` verwendet, um den Timerinterrupt zu behandeln. Hiermit werden die Tasks unterbrochen und ausgewechselt. Die Definition der ISR sieht folgender Maßen aus:

```

1 JEFAX_ISR(TCC0_OVF_vect, schedule)
```

dispatcher.c

Während dieser ISR wird der Scheduler aufgerufen, um eine neue Task auszuwählen, die laufen soll (dazu mehr in 3.2.2 Implementierung).

3.2 Scheduler

Der Scheduler des Kernels entscheidet welche Task aus den im System vorhandenen Tasks ausgewählt und als nächstes eingewechselt wird. Daher ist er entscheidend für das Verhalten des Taskschedulings und auch dessen Performance. Als Standardschedulingverfahren wird in Jefax ein prioritätengesteuertes Round-Robin-Verfahren verwendet.

3.2.1 API

Der Scheduler kann zur Laufzeit mit der Funktion

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

scheduler.h

gesetzt 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

Durch die Verwendung einer Struktur für den Scheduler ist es möglich Jefax um selbstimplementierte Schedulingverfahren zu erweitern. Um einen Scheduler in Jefax 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 (Aufruf von `setScheduler()`). Dabei müssen vor allem die `readyList` und die `blockingList` so bearbeitet werden, dass der Scheduler mit den Listen richtig arbeiten kann.

Die Funktion `getNextTask()` wird indirekt durch den Dispatcher im Rahmen des Timerinterrupts aufgerufen (siehe 3.1 Dispatcher). Damit läuft diese Funktion im Interruptkontext. In diesem Kontext ist kein Blockieren oder eine freiwillige Abgabe der CPU möglich (z.B. `lockMutex()`, `yield()`). Die Aufgabe dieser Funktion ist es die Task zu bestimmen, die als nächstes eingewechselt werden soll. Als Rückgabewert wird die einzuwechselnde Task erwartet oder `NULL`, wenn keine Task gefunden wurde. Wird `NULL` zurückgegeben, wechselt Jefax die sogenannte `idleTask` ein, die nichts macht und als Dummy dient.

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 und z.B. die aktuelle Task verdrängen, falls eine höherpriorie Task rechenbereit wurde. Als Parameter wird die Task übergeben, deren Zustand sich verändert hat.

Der Aufruf von `taskWokeUp(task_t*)` scheint auf den ersten Blick dem Aufruf von `taskStateChanged(task_t*)` sehr zu ähneln, da hier eine Task ihren Zustand von `BLOCKING` in `READY` ändert. Jedoch gibt es einen bedeutenden Unterschied: `taskWokeUp(task_t*)` läuft im Interrupt Kontext, es darf also nicht blockiert bzw. auf die CPU verzichtet werden (z.B. durch `sleep()`, `yield()`, etc). Hier ist die übergebene Task diejenige, die aufgewacht ist. 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 zur Liste passenden Zustand haben. Auf die aktuell laufende Task kann mit der Funktion

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

scheduler.h

zugegriffen werden.

Der Status einer Task (`RUNNING`, `BLOCKING`, `READY`) kann über die Funktion

```
1 void setTaskState(task_t *p_task, taskState_t p_state);
```

scheduler.h

geändert werden.

Alternativ können die Funktionen `yield()` und `void sleep(const int p_ms)` verwendet werden, um Task in den Zustand `READY` bzw. `BLOCKING` zu überführen.

Die Funktion `int hasRunningTask()` erlaubt die Überprüfung, ob momentan eine Task läuft. Dies ist genau dann der Fall, wenn nicht die `idleTask` läuft.

Mit `void forceContextSwitch()` kann ein Kontextswitch, also eine Unterbrechung der aufrufenden Task, erzwungen werden.

3.2.2 Implementierung

Bevor der Scheduler genutzt werden kann muss die Funktion

```

1  int  initScheduler(scheduler_t *p_defaultScheduler)
2  {
3      int  ret;
4      initTask(&idleTask);
5
6      ret = initTaskLists();
7      if(ret)
8          return ret;
9
10     ret = initTimerSystem();
11     if(ret)
12         return ret;
13
14     setScheduler(p_defaultScheduler);
15     setFirstRunningTask();
16
17     return 0;
18 }
```

scheduler.c

ausgeführt werden. Diese wird im Rahmen von `initDispatcher()` aufgerufen. Zuerst wird die `idleTask` initialisiert (Zeile 4). Danach werden die Tasklisten, also die `readyList` und die `blockingList`, initialisiert (Zeile 8). Im Rahmen dieser Funktion wird die `readyList` mit den Tasks aus dem `TASKS` Feld gefüllt.

Daraufhin wird das Timersystem initialisiert, welches für den Aufruf von `sleep()` benötigt wird (Zeile 12).

Zuletzt wird der Standardscheduler gesetzt und die erste Task ausgewählt, die gescheddelt werden soll (Zeile 14 - 15).

Die zentrale Funktion des Schedulers, die auch vom Dispatcher (siehe 3.1.2 Implementierung) aufgerufen wird, `schedule()` ist folgender Maßen implementiert:

```

1  void  schedule()
2  {
3      runningTask = scheduler->getNextTask();
4      // no task was found, so schedule idleTask
5      if(runningTask == NULL)
6          runningTask = &idleTask;
7      runningTask->state = RUNNING;
8  }
```

scheduler.c

Zuerst wird die nächste Task durch den Aufruf des Callbacks `getNextTask()` des aktuellen Schedulers ausgewählt (Zeile 3). Die Variable `runningTask` ist dabei global definiert und stellt die aktuell laufende Task dar.

In Zeile 5 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` als `runningTask` ausgewählt.

Beim Ändern des Taskzustands wird folgender Code ausgeführt:

```
1 void setTaskState(task_t *p_task, taskState_t p_state)
2 {
3     uint8_t irEnabled = enterAtomicBlock();
4
5     p_task->state = p_state;
6     scheduler->taskStateChanged(p_task);
7
8     exitAtomicBlock(irEnabled);
9 }
```

scheduler.c

Hierbei wird ununterbrechbar (`enterAtomicBlock()`) der Zustand der Task geändert (Zeile 5) und danach das Callback des Schedulers (Zeile 6) ausgeführt.

Die Funktion `void forceContextSwitch()` hat folgende Implementierung:

```
1 void forceContextSwitch()
2 {
3     // save interrupt enable state
4     uint8_t state = SREG & 0x80;
5     // create interrupt
6     sei();
7     FORCE_INTERRUPT(TCC0);
8
9     // wait to be exchanged
10    while(!TASK_IS_RUNNING(runningTask))
11    { }
12
13    if(!state)
14        cli();
15 }
```

scheduler.c

In Zeile 4 wird der aktuelle Zustand des Interruptenablebits gesichert. Danach wird ein Timerinterrupt des Dispatchers erzwungen. Dazu werden Interrupts aktiviert (Zeile 6) und das Makro `FORCE_INTERRUPT()` aufgerufen (Zeile 7). Diese Makro setzt den Counter des Timers auf den Wert kurz vor dem Auslösen des Interrupts. Nun wird in einer Schleife darauf gewartet, dass die Task wieder den Zustand `RUNNING` annimmt (Zeile 10). Dies impliziert, dass vor dem Aufruf von `void forceContextSwitch()` der Zustand der Task auf einen anderen Zustand als `RUNNING` gesetzt wurde.

3.2.3 Zustandsautomat

In Jefax gibt es 3 Taskzustände: **READY**, **RUNNING**, **BLOCKING**. Der Übergang zwischen diesen Zuständen wird in Abbildung 3.1 dargestellt.

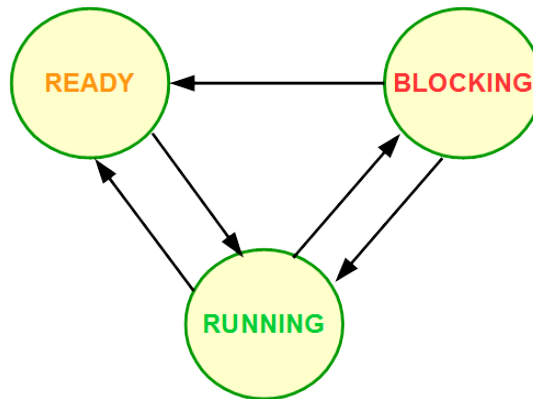


Abbildung 3.1: Zustandsautomat

Alle Tasks starten initial mit dem Zustand **READY**. Aus diesem Zustand heraus können die Tasks nur in den Zustand **RUNNING** wechseln (siehe Abbildung 3.1), 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 welche 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()` (siehe 3.2.1 API) des Schedulers ab.

3.2.4 Round Robin Scheduler

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

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

schedulerRR.h

genutzt werden.

Der Scheduler implementiert sämtliche Callbacks der `scheduler_t` Struktur.

Die `init()` Funktion sortiert die `readyList` nach Priorität der Tasks, damit der Scheduler korrekt 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` ihren Zustand von `RUNNING` in einen der anderen 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 ein Warten auf den Interrupt zu einer Endlosschleife führen.

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 3.2.3 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 `readyList` ausgewählt wird, hat folgende Implementierung:

```

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

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 (bzw. die `idleTask`) eingewechselt werden. Hierbei wird geprüft, ob es überhaupt eine laufende Task gibt. Dies ist genau dann der Fall, wenn eine andere Task als die `idleTask` läuft. Gibt es eine laufende Task, wird weiterhin geprüft, ob die `runningTask` überhaupt weiterlaufen kann, also nicht den Zustand `BLOCKING` hat (Zeile 3). Kann die `runningTask` nicht weiterlaufen wird als Rückgabewert `NULL` festgelegt (Zeile 4), ansonsten die aktuelle `runningTask` (Zeile 7).

Ist die `readyList` nicht leer (else Teil in Zeile 10), muss der Scheduler eine Task aus der `readyList` 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(hasRunningTask() && !TASK_IS_BLOCKING(getRunningTask()) &&
6     CMP_PRIORITY(result, getRunningTask()) < 0) {
7     getRunningTask()->state = RUNNING;
8     result = getRunningTask();
9 } else {
10     popTaskBack(schedulerRR.readyList);
11
12     if(hasRunningTask() && TASK_IS_RUNNING(getRunningTask()))
13         getRunningTask()->state = READY;
14 }
```

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 die Task aus der `readyList` eine niedrigere Priorität (höherer Wert) hat als die aktuelle

`runningTask` (Zeile 5 - 6). 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 7 - 8).

Im anderen Fall wird die gewählte Task aus der `readyList` entfernt (Zeile 10) und falls die `runningTask` noch den Zustand `RUNNING` hat, der Zustand der `runningTask` auf `READY` geändert (Zeile 12 - 13).

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

4 Timer

Jefax verfügt über eine eigene Timerkomponente, mit der Funktionen in relativen Zeitabständen ausgelöst werden können. Diese Schnittstelle wird z.B. auch durch den Scheduler benutzt, um die Funktion `sleep()` zu realisieren (siehe 3.2 Scheduler).

4.1 API

Ein Timer wird durch die Struktur `timer_t` dargestellt. Diese Struktur wird mithilfe von

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

timer.h

initialisiert. Dabei wird als erster Parameter die Timerstruktur übergeben. Der 2. Parameter gibt die relative Zeit an, nach der der Timer ausgelöst werden soll. Der 3. Parameter stellt die Callbackfunktion dar, die beim Zuschlagen des Timers ausgeführt werden soll. Optional kann dieser Funktion noch ein Argument übergeben werden (letzter Parameter).

Mit

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

timer.h

kann der Timer nach der Initialisierung scharf gestellt werden. Die angegebene relative Zeit gilt ab dem Zeitpunkt zu dem diese Funktion aufgerufen wurde. Dabei schlägt der Timer frühestens nach der angegebenen Zeit zu, kann jedoch auch etwas verspätet ausgelöst werden.

4.2 Implementierung

Um die Timerkomponente von Jefax nutzen zu können muss zuerst die Funktion

```

1  int initTimerSystem()
2  {
3      // Set 16 bit timer
4      TIMER_CLOCK.CTRLA = TC_CLKSEL_OFF_gc; // timer off
5      TIMER_CLOCK.CTRLB = 0x00; // select Modus: Normal -> Event/
        Interrupt at top
6      TIMER_CLOCK.PER = MS_TO_TIMER(100, TIMER_PRESCALER);
7      TIMER_CLOCK.CNT = 0x00;
8      TIMER_CLOCK.INTCTRLA = TC_OVFINTLVL_LO_gc; // Enable overflow
        interrupt level low
9
10     return 0;
11 }
```

timer.c

aufgerufen werden. Dabei wird einfach der Hardwaretimer initialisiert, der von dem Timersystem verwendet wird. Der Aufruf dieser Funktion wird bereits durch `initScheduler()` (siehe 3.2 Scheduler) vorgenommen.

Somit ist auch beim Timersystem ein Timerinterrupt der zentrale Bestandteil. Auch hier wird eine `JEFAQ_ISR()` verwendet. Die Funktion, die den Interrupt behandelt heißt

```

1  static void decreaseTimers()
2  {
3      // get elapsed time
4      unsigned int ms = TIMER_TO_MS(TCDO.PER, TIMER_PRESCALER);
5      unsigned int toDec;
6      int i;
7
8      // decrease timer values
9      for(i = 0; i < timerCount; ++i) {
10         // prevent timer[i].ms from getting lower than 0
11         toDec = (timers[i].ms >= ms ? ms : timers[i].ms);
12         timers[i].ms -= toDec;
13     }
14
15     // check for all timers if they elapsed
16     for(i = 0; i < timerCount; ++i) {
17         while(i < timerCount && timers[i].ms <= 0)
18             timerElapsed(i);
19     }
20     if(timerCount > 0)
21         updatePeriod();
22 }
```

timer.c

Hierbei wird die verbleibende Zeit jedes Timers um die Periode des Hardwaretimers verringert (Zeile 4 - 10). Danach wird für jeden Timer überprüft, ob er abgelaufen ist (verbleibende Zeit ist 0) (Zeile 17). Ist der Timer abgelaufen wird sein Callback ausgeführt und er wird aus der Liste der aktiven Timer entfernt.

Am Ende wird der nächste Zuschlagszeitpunkt für den Hardwaretimer festgelegt (Zeile 21). Dabei wird einfach die niedrigste verbleibende Zeit aus allen aktiven Timer ausgewählt und als Periode für den Hardwaretimer gesetzt.

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

5 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. Die dynamische Speicherverwaltung ist schnell und ressourcenschonend. Möglichst wenige Verwaltungsinformationen werden benutzt, um den Verwaltungsaufwand gering zu halten.

5.1 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 allozierten 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 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;

```

memory.h

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 alloziert wurden
- freeListEntries: Anzahl an Einträgen in der Free-List

5.2 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 (siehe Abbildung 5.1).

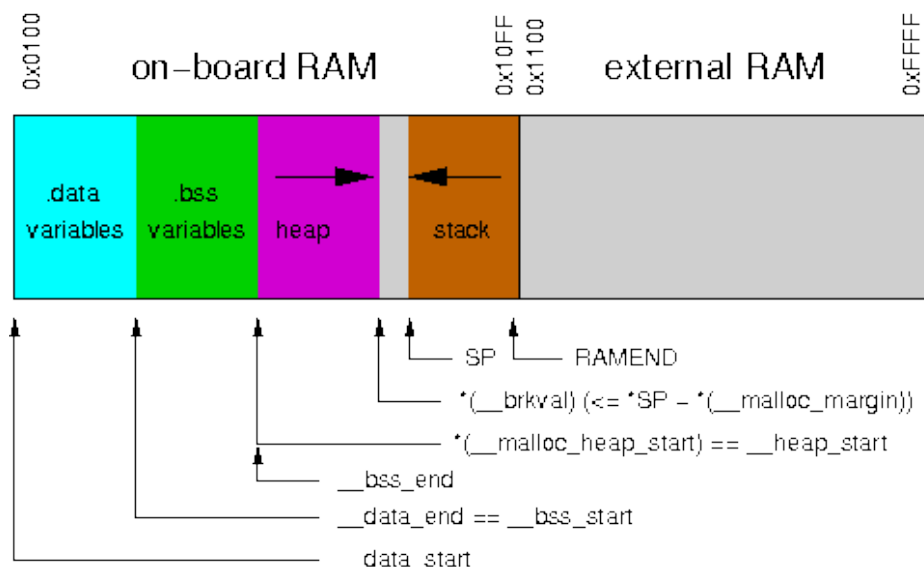


Abbildung 5.1: Sections

Datenstrukturen

Der Anfang des Heaps wird über das Linker Script an den 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;
```

memory.c

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 (Variable `next`) 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 einfach verkettete List der Free-List verwendet. Der Vorteil dieses System ist, dass keine separaten Listen benötigt werden, um die Kontrollinformationen zu speichern. Die einzige Limitierung stellt die `size` Variable dar. Da vor jedem Speicherbereich die Größe gespeichert wird, beträgt der zusätzliche Speicherverbrauch pro Speicherreservierung die Größe dieser Variable (1 Byte).

Bild 5.2 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 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.

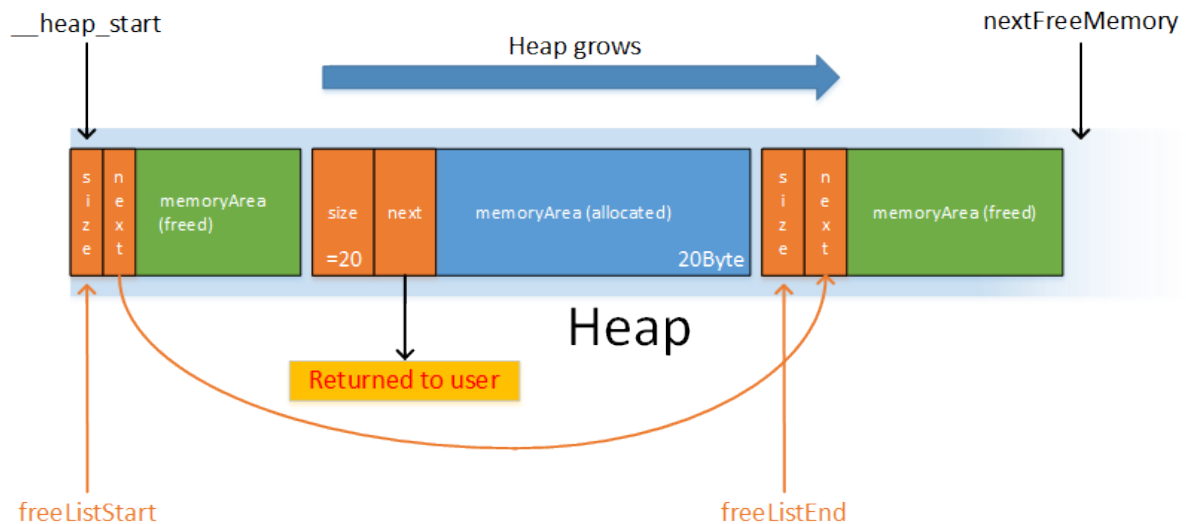


Abbildung 5.2: Dynamische Speicherverwaltung

Speicher reservieren/freigeben

Beim Reservieren von Speicher muss zunächst sichergestellt werden, dass genügend freier Speicher vorhanden ist und keine Heap-Stack Kollision entsteht. Kann Speicher reserviert werden und ist kein passender Eintrag in der Free-List verfügbar (dazu später mehr), wird die Adresse des nächsten verfügbaren Speichers auf einen Pointer auf eine `memoryArea` Struktur gecastet. Anschließend wird die Größe des Speicherbereichs gesetzt (`size`) 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 einfach 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. Beim Suchen nach passenden Free-List Einträgen wird nach dem Best-Fit Prinzip gearbeitet.

Da Heap und Stack aufeinander zu wachsen, besteht hier die Gefahr einer Kollision. Um dies zu vermeiden, wird bei jeder 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.

5.3 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 damit Speicherverbrauch kommen. Auch kann ein Buddy-System implementiert werden, welches durch aufsplitten und zusammenfügen von Speicherbereichen noch effektiver arbeiten kann. Nachteil davon ist der erhöhte Speicherverbrauch durch Verwaltungsinformationen sowie aufwendigere Berechnungen beim Reservieren und Freigeben von Speicher.

6 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 und komfortabel 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 XMeaga über USART. Die Shell läuft als eigener Task, unterstützt verschiedene Steuer- und Debug-Befehle und wird wie alle Tasks gescheduled. Zum Verwalten der USART Nachrichten werden verschiedene Buffer verwendet, welche in den Dateien `usart_message.h` und `usart_queue.h` deklariert sind.

6.1 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 };
```

Message Queues werden mit den Funktionen

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

erzeugt und zerstört. 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. Das Hinzufügen und Auslesen von Messages aus einer Queue erfolgt mit den 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. Ein Aufruf von `destroyMessage()` für die übergebene Message ist nicht erforderlich.

6.2 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 die Art der 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 erleichtert das 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. Zum Senden von Daten 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 dabei ebenfalls die dynamische Speicherverwaltung verwendet. `getMessage()` bekommt als Parameter die Größe der Message in Bytes, sowie den Message Typ, welcher den Stack-Zugriff bestimmt.

```
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);
```

Im folgenden Code-Ausschnitt wird der Vorgang zum Erzeugen von Messages dargestellt:

```
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);
```

6.3 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.

6.4 USART Kommunikation Implementierung

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

`Asynchronous Communication 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 USARTCO
2 #define USART_PORT PORTC
3 #define BAUDRATE 115200
```

Zum Senden und Empfangen werden Low-Level Interrupts verwendet. Wurde ein Byte empfangen, wird der `USARTCO_RXC_vect` Interrupt getriggert. Können Daten in den USART Hardware Buffer kopiert werden, wird der `USARTCO_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.

6.4.1 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 Kopie der `currentReceiveMessage` Message der receive queue hinzugefügt. Eine neue `currentReceiveMessage` wird erzeugt um erneut Empfangene Daten abzuspeichern. 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 6.1 zeigt den groben Ablauf beim Empfangen von Nachrichten.

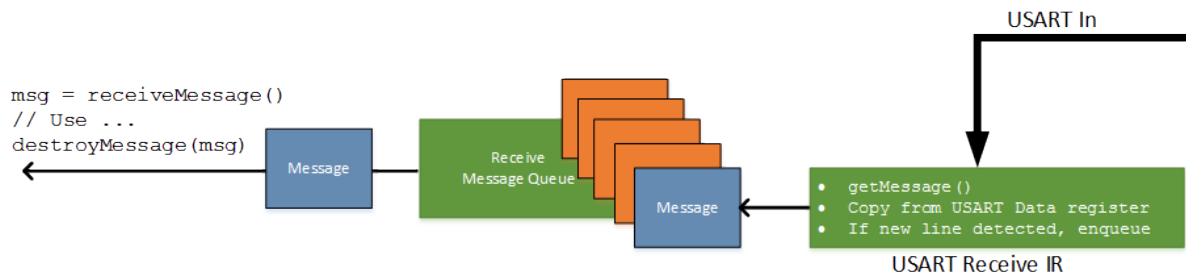


Abbildung 6.1: USART Receive

6.4.2 Senden von Messages

Das Senden von Messages 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. Diese wird zur neuen `currentSendMsg`. Der DRE Interrupt wird aktiviert, sobald eine Nachricht gesendet werden soll. Wurden alle Nachrichten aus der Queue gesendet, wird dieser deaktiviert. Bild 6.2 zeigt das Versenden von Nachrichten.

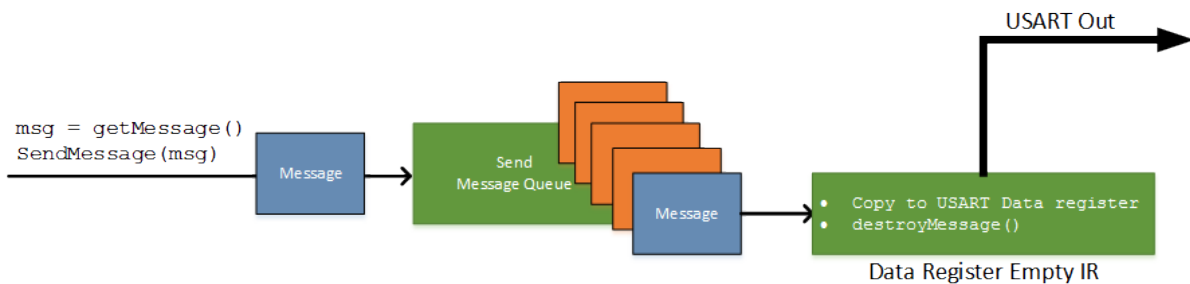


Abbildung 6.2: USART Send

6.5 Shell Task

Die Shell Tasks wird wie alle anderen Tasks gescheduled. Sie 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. Wird kein internes Kommando (zum Beispiel Speicher-Informationen ausgeben) erkannt, werden die Empfangenen Daten an die Callback-Funktion übergeben.

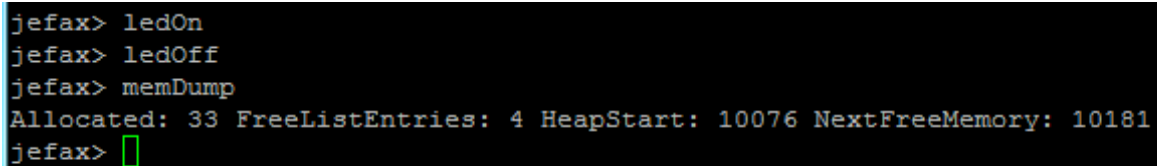
6.5.1 Implementierung

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 6.3: jefax shell

7 Tests

TODO: Carrera Bahn tests

8 Fazit

Jefax bietet eine gute Möglichkeit Tasks quasi-parallel auf einem XMega ablaufen zu lassen. Jedoch werden atomare Zugriffe und der gegenseitige Ausschluss durch viele Interruptsperrern realisiert. Dadurch hat das System eine hohe Interruptlatenzzeit, was vor allem in Realzeitsystemen beachtet werden muss. Das Timer Framework von Jefax wird durch einen Hardware Interrupt realisiert. Somit laufen alle Timer Callbacks im Interruptkontext. Dies verlangt einen gewissen Grad an Erfahrung und Aufmerksamkeit des Nutzers. Intuitiver und weniger fehleranfällig wäre eine Realisierung als Softinterrupt. Die Realisierung des Schedulers als struct bietet einen hohen Grad an Erweiterbarkeit und Flexibilität. Jedoch kann durch einen fehlerhaft implementierten Scheduler das komplette System lahmgelegt und untergraben werden.

Literaturverzeichnis

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

Abbildungsverzeichnis

3.1	Zustandsautomat	14
5.1	Sections	22
5.2	Dynamische Speicherverwaltung	24
6.1	USART Receive	30
6.2	USART Send	30
6.3	jefax shell	31