Ausarbeitung AtxOS

Konstantin Wurster

Ruben Gonzalez

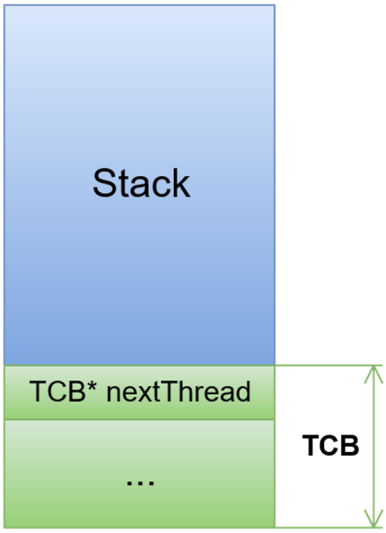
Angewandte Systemsoftware

WS15/16

INHALTSVERZEICHNIS

1 THREADS

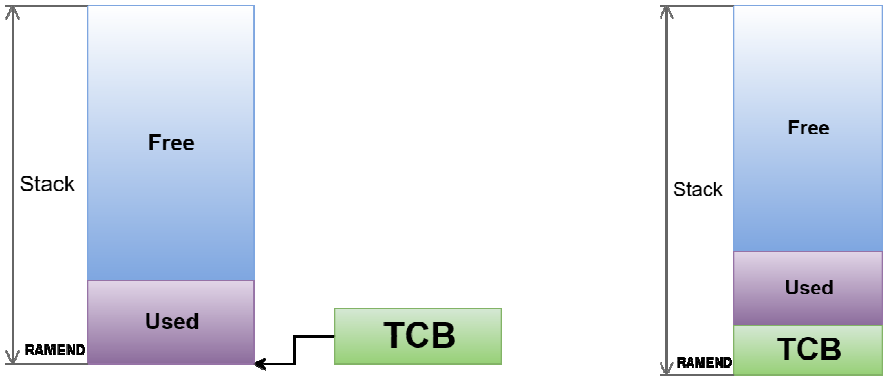
Ein Thread wird durch einen Thread Control Block definiert. In ihm sind alle wichtigen Daten über den Thread gespeichert. Der TCB wird ganz unten im dem zur Verfügung gestellten Thread-Stack geschrieben.



Vorteile	Nachteile
OS muss sich nicht darum kümmern wohin es den TCB schreibt. (da weder dynamische Allokation noch Array mit fester Größe). Die Maximale Anzahl an Threads ist somit nur abhängig davon, wie sparsam der Benutzer des OS mit dem Speicher umgeht.	Man muss bei der Wahl der Stackgröße zusätzlich zum Platz den das Sichern der Register benötigt auch die Größe des TCB berücksichtigen. Ein Amok laufender Thread kann den TCB beschädigen. (Vernachlässigbares / nicht Verhinderbares Risiko bei AVRs.)

1.1 INITIALISIERUNG

Im Standartzustand sieht der Stack wie folgt aus. Etwas bereits benutzter Speicher und darüber freier Speicher. Unser Ziel ist es den initialen Stack in einen für unser OS verwendbaren Stack mit TCB zu verwandeln (welcher später für die idle Task verwendet wird). Dazu verschieben wir den Bereits genutzten Speicher nach oben. Während Initialisierungsroutine werden auch HW Timer eingestellt.



Vorteile	Nachteile
Bessere Speicherausnutzung. Ein Kontrolliertes Herunterfahren des OS ist möglich.	Komplizierter als einfach in einen fertigen OS kompatiblen Stack zu wechseln. Die Stackgröße ist so groß wie der freie RAM -> Gefahr eines Overflows bei vollem RAM.

Mit dieser Vorgehensweise wird unser eigener Stack während der Laufzeit modifizieren. Da (besonders im nicht optimieren Modus) variablen nicht immer in Register verschoben werden sondern auf den Stack ausgelagert sind, kann es leicht zu ungewolltem Verhalten kommen. Um das Vorzubeugen hat man mehrere Möglichkeiten zur Auswahl.

Methode	Vorteile	Nachteile
Routine in ASM schreiben	• Funktioniert garantiert immer	• Aufwendig • Schwer lesbarer Code • Schlecht portierbar
Variablen als static deklarieren	• Einfach	• Speicher wird verschwendet • Es ist unsicher ob der Compiler nicht doch mit dem Stack arbeitet
register Keyword verwenden	• Einfach • Tipp an Compiler Register zu verwenden	• Compiler ist nicht gezwungen Register zu verwenden und kann den Tipp ignorieren.

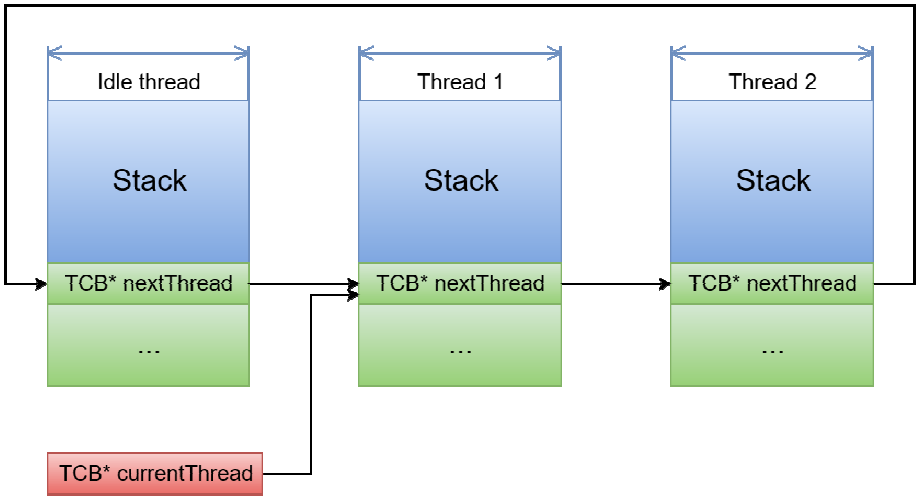
Da der avr gcc Compiler das register Keyword nicht grundsätzlich ignoriert und genug Register zur Verfügung hat, wurde das Problem somit gelöst (obwohl ASM die bessere Lösung darstellt).

```
//copy existing stack
register BYTE* from = SP;
register BYTE* to = (BYTE*)(SP - sizeof(TCB));
register WORD size = ATXOS_PORT_RAMSTART + ATXOS_PORT_RAMSIZE - SP;
for (register WORD n = size; n; n--) {
	*to++ = *from++;
}
SP = (WORD)(to - size);
```

2 SCHEDULER

Jeder TCB enthält eine Referenz auf einen Folgenden TCB. Es bildet sich somit ein Kreis. Der erste (idle) Thread enthält eine Referenz auf sich selbst.

Desweiteren gibt es eine globale TCB Referenz auf den aktuell ausgeführten Thread.



Um einen RR Scheduler mit Prioritäten implementieren zu können wurden zwei variablen prioLevel und prioCurrent als bitfields zu je 4 bit definiert wobei prioLevel das definierte prioritätslevel enthält.

Eine Priorität von 0 entspricht der höchsten Priorität, eine Priorität von 15 der niedrigsten.

Wenn der Scheduler angewiesen wird sich den nächsten Thread auszusuchen iteriert er (startend beim currentThread->nextThread) in einer Endlosschleife über alle Threads.

Wenn die prioCurrent eines Threads der definierten Priorität entspricht wird der Thread ausgewählt (durch setzten der currentThread Referenz) und die prioCurrent auf 0 gesetzt. Die Iteration wird beendet.

Falls nicht wird die prioCurrent um eins erhöht.

- ➔ Ein Thread der Priorität 0 wird immer als nächster aktive Thread gewählt.
- ➔ Ein Thread der Priorität 1 wird jedes zweite Mal als nächster aktiver Thread gewählt.
- ➔ Ein Thread der Priorität 15 wird jedes 16. Mal als nächster aktiver Thread gewählt.

Vorteile	Nachteile
Denkbar einfachster scheduling Algorithmus. Keine Starvation von Threads. Nur 1 Byte für die Implementierung von Prioritäten per TCB benötigt. Wenig benötigter Stack.	Wenn viele niedrigpriore Tasks aufeinander warten wird viel Zeit im Scheduler mit sinnlosem durchiterieren verbracht. Keinerlei Berücksichtigung von Faktoren wie z.B. die tatsächlich benötigte CPU Zeit.

3 DISPATCHER

Der Dispatcher wird entweder nach einer definierten Zeit durch einen Interrupt ausgelöst oder manuell durch aufrufen der ContextSwitch Methode.

Der Dispatcher führt folgende Schritte aus:

1. Deaktiviert alle Interrupts
2. Sichert den aktuellen Thread Kontext durch pushen auf den Stack
3. Ruft den Scheduler auf
4. Stellt den Thread Kontext für den neuen currentThread wieder her.
5. Aktiviert alle Interrupts

Schritte 2 und 4 sind komplett als Makros in ASM geschrieben.

In der TCB Struktur (muss) sich an oberster Stelle ein Pointer auf den top of Stack des Threads befinden.

Beim wiederherstellen des Kontexts wird dieser Wert als allererstes in den Stackpointer geladen, somit wird von da an auf dem neuen Stack gearbeitet.

Die restlichen Register können nun gepopt werden. Zum Schluss wird noch das Statusregister wiederhergestellt.

```
#define RESTORE_CONTEXT() asm volatile (\
"lds   r26, currentThread          \n\t"\
"lds   r27, currentThread + 1      \n\t"\
"ld     r0, x+                     \n\t"\
"out   __SP_L__, r0                \n\t"\
"ld     r0, x+                     \n\t"\
"out   __SP_H__, r0                \n\t"\
"pop   r31                         \n\t"\
"pop   r30                         \n\t"\
"pop   r29                         \n\t"\
...
"pop   r2                          \n\t"\
"pop   r1                          \n\t"\
"pop   r0                          \n\t"\
"out   __SREG__, r0                \n\t"\
"pop   r0                          \n\t"\
);
```

Der Aufruf der ContextSwitch Methode aus der ISR erfolgt mittels eines relativen jumps und wird vom Compiler wegoptimiert sodass die ISR direkt in die ContextSwitch Methode springt.

```
ISR(ATXOS_PORT_SCHEDULE_VECT, ISR_NAKED) {
	asm volatile ("rjmp ContextSwitch");
}

//perform a context switch
void __attribute__ ((naked)) ContextSwitch(void)
{
	cli();
	SAVE_CONTEXT();
	...
```

Da der ContextSwitch nicht nur von ISRs aus aufgerufen wird, ist das deaktivieren aller Interrupts zu Beginn notwendig.

Das rückspringen aus der ContextSwitch Routine und gleichzeitige aktivieren der Interrupts wurde mit einer reti Instruktion gelöst.

Vorteile	Nachteile
Tut was es soll und das effizient. Man muss sich nicht um das setzten des topOfStack pointers kümmern.	Selber ContextSwitch für ISR und Usercalls kann zu potentiellen Programmierproblemen führen da die Routine nie mit deaktiviertem Interrupt verlassen werden kann. →Einführung eines soft Context Switch mit vorherigem sichern des Interrupt Flags mögliche Lösung. topOfStack pointer muss ganz oben im TCB stehen.

4 Starten des OS

Um das OS zu starten wird zuerst die Initialisierung mit einem Aufruf von `AtxOSInit` durchgeführt.

Nach diesem Aufruf sind alle benötigten Hardwaretimer eingestellt, der aktuelle Stack wurde in einen Thread umgebaut, die currentThread Referenz zeigt auf den Thread und ein paar wenige OS apis (wie z.B. `CreateThread`) können verwendet werden.

In dieser Phase können nun hauptsächlich Threads gestartet werden. Es wird jedoch noch nicht geschedult da alle Interrupts deaktiviert sind!

Mit einem Aufruf von `AtxOSStart` werden alle Interrupts aktiviert und das OS ist voll funktionstüchtig. Die `AtxOSStart` Methode kehrt niemals zurück, sie wird zum idle Task.

Vorteile	Nachteile
Strukturierter Startprozess, kein extra initprozess welcher Threads startet notwendig.	Es müssten nicht unbedingt alle Interrupts Deaktiviert werden, dann wäre die Auswahl an erlaubten OS Funktionen größer.

5 Idle Task

Die Idle Task ist ein Thread mit der niedrigsten Priorität. Ihr TCB wird in der `AtxOSInit` Methode erstellt und sie ist somit der erste Thread. Ihr einziger Sinn und Zweck ist es die `ContextSwitch` Methode des Dispatchers aufzurufen.

Vorteile	Nachteile
Vereinfacht den Bau eines Schedulers da es immer einen Task gibt der ausgeführt werden kann. Lässt Raum für eventuelle Energiesparoptionen wie z.B. das Heruntertakten der MCU oder deaktivieren von UART / externer Hardware.	Aktuell unnötiger Context Switch. Ein kurzes aktivieren/nop/deaktivieren aller Interrupts würde ein (leicht) besseres Ergebnis erzielen.

6 SCHLAFENDE THREADS

Um einen Thread Schlafen legen zu können wir eine weitere Variable in den TCB eingebaut. Sie bestimmt ob der Thread aktiv oder im waiting state ist.

Threads im waiting state werden vom Scheduler übersprungen.

Wenn ein Thread auf etwas wartet und ein weiterer Thread auf dieselbe Sache warten will, wartet er praktisch darauf, bis der erste Thread seine Aufgabe erledigt hat. Aus diesem Grund wurde eine weitere TCB Referent nextWaiting in den TCB eines jeden Threads gebaut.

Das funktioniert, da ein Thread nur auf eine Sache gleichzeitig warten kann. Warten mehrere Threads auf dieselbe Sache kann man mit dieser Referenz eine Verkettete Liste bauen.

6.1 MUTEX

Einen Mutex darf nur ein Thread zur selben Zeit besitzen. Ein Mutex besteht lediglich aus einer TCB Referenz auf den ersten Wartenden Thread (head). Das OS implementiert einen Mutex Lock wie folgt:

1. Schauen ob der Thread welcher zu locken versucht nicht schon gelockt hat (sich selbst aussperren verhindern)
2. Wenn aktuell kein Thread den Mutex beansprucht (head==NULL) wird der head=currentThread gesetzt und der Mutex gehört dem aktuellem Thread. Die Methode beendet sich ohne zu blockieren.
3. Falls bereits Threads auf den Mutex warten (head!=NULL) wird bis ans Ende der Verketteten Liste gegangen und sich dort eingehängt. Anschließend wird der Threadstatus auf Wartend gesetzt und ein Kontextwechsel herbeigerufen.

Ein Mutex Unlock ist wie folgt Implementiert:

1. Wenn aktuell ein weiterer Thread darauf wartet den Mutex zu nehmen, wecke diesen auf und aktualisiere die Liste der wartenden Threads.
2. Wenn kein weiterer Thread auf den Mutex wartet setzte den head=NULL

6.2 SEMAPHORE

Eine Semaphore ist in Ihren Grundsätzen gleich wie ein Mutex aufgebaut. Sie enthält lediglich ein weiteres Byte an Daten welches die Anzahl der gleichzeitig zugreifenden Threads reguliert.

Bei jedem Semaphore Down wird dieser Wert dekrementiert. Falls er bereits 0 ist, trägt sich der Thread in die Warteliste ein und Schläft bis er aufgeweckt wird.

Bei jedem Semaphore Up wird geschaut ob Threads warten. Falls ja wird der nächste aufgeweckt. Falls nein wird der Wert inkrementiert.

Vorteile	Nachteile
Es funktioniert	Eventuell einen int8_t anstelle eines uint8_t nehmen um schnell sehen zu können wie viele Threads warten.

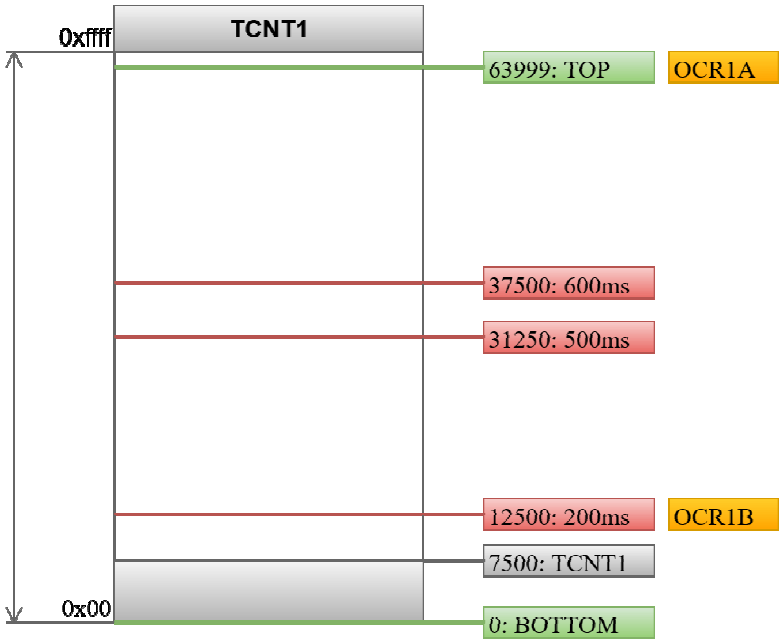
7 TIMING

Das OS enthält ein vom Scheduler separiertes Timing Modul mit eigenem Counter. Der Counter wird wie der des Dispatchers in der `AtxOSInit` Methode konfiguriert.

Der Counter wird im CTC Modus betrieben wobei der TOP Wert einer Zeit von 1024ms entspricht. 1024ms da somit rechenlastige Operationen wie etwa Dividieren mit AND/OR Operationen vom Compiler durch Optimierung ersetzt werden können.

Wenn der Counter den TOP Wert erreicht wird er durch den CTC Modus wieder auf 0 gesetzt und ein Interrupt A ausgelöst. Es entsteht somit keinerlei Zeitverlust.

Die ISR von A inkrementiert eine statische Variable, einen „Sekundenzähler“.



Da meist noch ein zweites OCR Register für einen 16 bit Counter existiert wurde dies verwendet um auf Mikrosekunden genau die Aufweckzeit eines Threads zu bestimmen.

Die durch Erreichen des OCRB Registers ausgelöste ISR weckt dann den schlafenden Thread auf und setzt OCRB neu bzw. deaktiviert den Interrupt falls keine weiteren Threads auf ein Zeitevent warten.

Schlafende Threads sind Zeitlich aufsteigend in einer Verketteten Liste gespeichert.

Um eine zeitlich sortierte Liste zu erstellen wurde im TCB ein weiterer Wert gespeichert, die wakeuptime. Sie besteht aus den noch zu wartenden „Sekunden“ und den „parts“, also den Partiellen Sekunden gerechnet in notwendigen Timer Counter Inkrementierungen. Da sie nur verwendet wird wenn ein Thread auf ein Zeitevent wartet wurde sie in eine union geschrieben und nicht standartmäßig in den TCB gepackt.

Vorteile	Nachteile
Extrem genau. Es wäre denkbar damit Linux-ähnliche Arten von Timern und softirqs zu erstellen. Vom Scheduler unabhängiger Interrupt welcher einen Kontextwechsel einleiten Könnte.	Komplex zu programmieren, Komplexe Fehlerquellen, Komplex den Timer richtig einzustellen. Setzt in der Aktuellen OS Implementierung lediglich den Threadstatus auf aktiv und die prioCurrent auf prioLevel, verspielt damit praktisch all ihre Vorteile.

7.1 Probleme und Lösungen

Wenn der Scheduler Läuft Kann es zu mehreren Interrupts kommen. Diese werden jedoch nicht ausgeführt da alle Interrupts deaktiviert sind. Bei reaktivieren der Interrupts kann es nun vorkommen, dass ISR A, obwohl nach ISR B ausgelöst, vor B ausgeführt wird. Dieses Problem gilt es bei MCUs ohne programmierbare Interruptprioritäten zu lösen.

- ➔ In ISR A Überprüfen ob ISR B ausgelöst wurde durch lesen der Interrupt flags und dementsprechend Maßnahmen ergreifen.
- ➔ Wenn ISR A aufgerufen wird, wird der secs Counter aller wartender Threads dekrementiert. Falls ein Counter bereits 0 ist, heißt das, dass er bereits abgearbeitet hätte sein müssen. Es sind dementsprechend Maßnahmen zu ergreifen.

Wenn Eine ISR läuft können gesetzte Interrupts überrannt werden, da schneller Interrupts erzeugt werden als man abarbeiten kann.

- ➔ Überprüfen ob die parts der wartenden Threads kleiner gleich dem aktuellen Zählerstand sind und sie dementsprechend aufwecken.
- ➔ Überprüfung der parts nach dem Aufwecken, falls während des Aufweckvorgangs ein Interrupt hätte stattfinden müssen.

Im Dauertest über 12 Stunden mit einem atmega328p und 5 Threads konnten keine Fehler mehr festgestellt werden.

8 Portabilität des Codes

Es wurde darauf geachtet, dass alle vom OS benötigten hardwarespezifischen Definitionen an einer zentralen Stelle einstellbar sind. Je nach eingestelltem Devicetyp wird das richtige Includefile eingebunden.

```
#if defined (__AVR_ATmega328P__)
#include "atxOS_m328p.h"
#elif defined (__AVR_ATxmega128A1__)
#include "atxOS_x128a1.h"
#endif
```

Es können Beispielsweise die Größe der Rücksprungsddresse, ISR Vektoren und Frequenz der MCU eingestellt werden.

9 Basis OS Funktionen

Alle Funktionen sind bereits im Source Dokumentiert. Hier eine Kurze Übersicht.

9.1 Generell

- `void AtxOSInit()`
  Erster Aufruf Zur Initialisierung des OS. Danach kann CreateThread verwendet werden.
- `void AtxOSStart()`
  Startet Das OS. Funktion kehrt niemals zurück.
- `void ContextSwitch()`
  Führt einen Kontextwechsel durch.
- `ATXVALUE CreateThread(void *start, BYTE* stack, int sizeOfStack, ThreadPriority priority)`
  Erstellt einen Thread dessen Entrypoint durch start gegben ist. Stack verweist auf den als Stack reservierten Bereich. Es wird ein absolutes Minimum von 64Bytes empfohlen. Priority gibt die Priorität des Threads an von T_PRIO_HIGHEST, T_PRIO_1, …, bis T_PRIO_14, T_PRIO_LOWEST. Liefert ATX_OK bei erfolg.

9.2 Synchronisierung

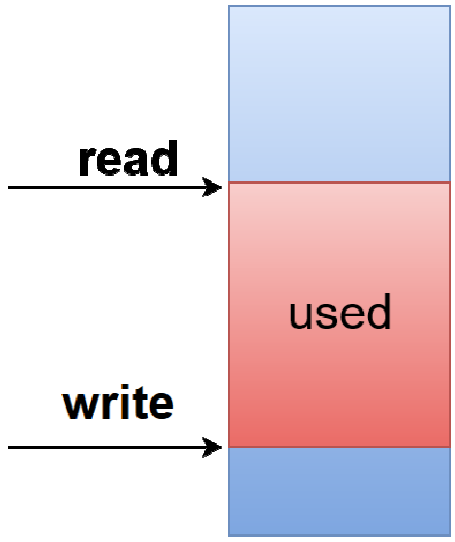
- `DEFINE_MUTEX(mutexname)`
  Makro. Definiert und initialisiert einen Mutex.
- `void MutexInit(MUTEX* m)`
  Initialisiert einen Mutex zur Laufzeit.
- `void MutexLock(MUTEX* m)`
  Lockt einen Mutex für den aktuellen Thread, Blockierender Aufruf.
- `void MutexUnlock(MUTEX* m)`
  Gibt einen Mutex wieder frei.
- `DEFINE_SEMAPHORE(semname, semasize)`
  Makro. Definiert und initialisiert eine Semaphore.
- `void SemaphoreInit(SEMAPHORE* s, BYTE size)`
  Initialisiert eine Semaphore zur Laufzeit.
- `void SemaphoreUp(SEMAPHORE* s)`
  Gibt einen platz im geschützten Bereich frei.
- `void SemaphoreDown(SEMAPHORE* s)`
  Beansprucht einen Platz im Geschützen Bereich, Blockierender Aufruf.

9.3 Zeit

- `DWORD CurrentTime()`
  Liefert die Zeit seit Systemstart in Millisekunden.
- `void Sleep(WORD s)`
  Legt den aufrufenden Thread für s Sekunden Schlafen.
- `void Msleep(WORD ms)`
  Legt den aufrufenden Thread für ms Millisekunden Schlafen.
- `void Usleep(WORD us)`
  Implementiert als Busy Loop! Nur für sehr kurze und unkritische Zeitdauern gedacht. Kann, falls nicht deaktiviert vom Scheduler unterbrochen werden.

10 BUFFERED UART

Um effizient über UART kommunizieren zu können wurde ein UART Treiber geschrieben welcher mit Interrupts und einem Sende / Empfangspuffer arbeitet.



Es existieren zwei Pointer für den Puffer. Ein Lese und ein Schreibpointer. Bei jeder Leseoperation wird der Pufferinhalt an der Leseoperation zurückgegeben und der Lesepointer inkrementiert, bei jedem Schreibvorgang dasselbe nur mit schreiben des Puffers.

Wenn der Schreib/Lesepointer über den Puffer hinauswächst wird er wieder auf Pufferanfang gesetzt.

Wenn der Schreibpointer gleich dem Lesepointer ist nach einer vorherigen Schreiboperation ist der Puffer voll und der empfang weiterer Bytes wird durch das deaktivieren eines Interrupts verhindert.

Leseoperationen reaktivieren Interrupts.

Wenn der Schreibpointer gleich dem Lesepointer ist nach einer vorherigen Leseoperation (und Interrupts aktiviert sind) ist der Puffer leer, falls weitere Bytes angefordert werden wird der aufrufende Thread schlafen gelegt bis wieder Daten anliegen.

Auf diese Art wurden sowohl ein Empfangs als auch ein Sendepuffer realisiert.

Vorteile	Nachteile
Kein Aktives Warten. Bei einer Baudrate von 9600 gehen nur 1,2Byte pro Millisekunde raus.	Es können nicht mehrere Threads auf dem UART lauschen, wenn ein Byte gelesen wurde ist es weg.

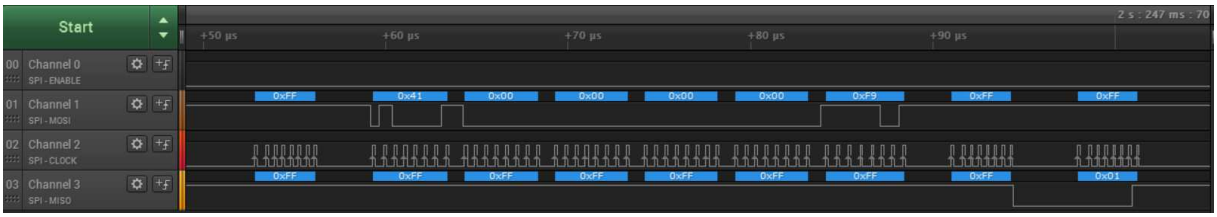
11 MMC/SD KARTEN ANSTEUERUNG ÜBER SPI

Ein Teil unseres Features war die Unterstützung von MMC/SD Karten über SPI. Die Karte wurden an GPIO Ports angeschlossen und von diesen gesteuert (bitbang).

Wie sich herausstellte war die uns zur Verfügung gestellte Karte eine MMC Karte (welche jedoch als SD Karte gelabelt wurde).

Da der SD Standard zum MMC Standard Abwärtskompatibel ist war das nicht weiter Schlimm, es gibt nur Unterschiede in der Initialisierung.

SD Karten sendet man über SPI einen 6 Byte großen Befehl, welcher aus einer Befehls-ID, 4 Bytes an Argumenten und einer Checksum besteht. Man erhält eine 1 Byte große Antwort zurück. Checksums sind im SPI Modus standartmäßig deaktiviert bis auf ein paar Ausnahmen in der Initialisierung.



Hier Beispielsweise senden wir den Befehl mit der ID 1 (0x41 letztendlich da das erste Bit 0 und das zweite 1 sein muss). Befehl 1 ist der Init Befel für MMC Karten, er hat keine Argumente. 0xF9 ist die Checksum. Wir erhalten als Antwort eine 1, was so viel heißt wie noch nicht bereit, versuche es später nochmal. Die Antwort 0 Bedeutet kein Fehler.

Einmal initialisiert kann man beliebig Blöcke Lesen und Schreiben. Die minimale Blockgröße beträgt 512 Byte. Es wurde lediglich ein single Block read und write implementiert da nur wenig Platz auf dem µC zur Verfügung ist.

Auf diesem Bild sieht man die Initialisierung einer MMC mit dem anschließendem auslesen des ersten Blocks. Man erkennt auf der MOSI gut wie erst CMD0 (reset) und dann CMD1 gesendet werden. Da CMD1 noch nicht bereit ist wird etwas gewartet und dann erneut CMD1 gesendet. Darauf folgt ein CMD17 (read single sector). Es wird auf MISO gewartet bis ein Starttoken empfangen wird, danach werden 512 Bytes Daten gelesen + 2 Bytes Checksum welche in unserer Implementierung verworfen werden. Es ist auch schön zu sehen wie jede Millisekunde geschedult wird (der Thread ist zum Testzeitpunkt der einzige gewesen). Das lesen eines 512 Byte Blockes dauert ungefähr 2ms, was einer Leserate von 256kb/s entspricht.

Vorteile	Nachteile
Da Ansteuerung über GPIO / Bitbang leicht portierbar. Einfache Implementierung des SPI Protokolls	Keine Unterstützung für „echte“ SD Karten, diese benötigen einen Komplexeren Initialisierungsprozess.

12 Basis Driver Functions

Alle Funktionen sind bereits im Source Dokumentiert. Hier eine Übersicht.

12.1 UART

- `void uart_initialize()`
  Initialisiert den Gepufferten UART.
- `char uart_getc()`
  Holt sich einen character aus dem UART. Blockierender Aufruf wenn Puffer voll.
- `void uart_putc(char c)`
  Schreibt einen character in den UART. Blockierender Aufruf wenn Puffer voll.

12.2 MMC/SD

- `BYTE mmc_initialize()`
  Initialisiert die mmc Karte. Liefert 0 wenn kein Fehler aufgetreten ist.
- `WORD mmc_writesector(DWORD sector, BYTE* buf)`
  Schreibt einen Sektor wobei sector die sektor-Nummer ist. Buf muss mindestens 512 Bytes fassen können. Liefert die Anzahl geschriebener Bytes.
- `WORD mmc_readsector(DWORD sector, BYTE* buf)`
  Liest einen Sektor wobei sector die sektor-Nummer ist. Buf muss mindestens 512 Bytes fassen können. Liefert die Anzahl gelesener Bytes.
- `WORD mmc_readsector_ex(DWORD sector, BYTE* buf, WORD offset, WORD n)`
  Liest einen Sektor startend bei offset für n Bytes wobei sector die sektor-Nummer ist. Buf muss mindestens n Bytes fassen können. Liefert die Anzahl gelesener Bytes.

13 DATEISYSTEM

Um die Daten auf der Karte verwalten zu können wurde sich ein möglichst minimalistisches Dateisystem Überlegt.

Zur Verwaltung des freien Speichers wurde sich für eine Bitmap entschieden. Jeder Sektor entspricht einem Bit in der Bitmap. 1 bedeutet belegt, 0 frei. Die Bitmap wird in die Ersten Sektoren der Karte geschrieben.

Es gibt zwei Typen von Einträgen. Ordner und Dateien. Der Erste freie Sektor nach der Bitmap wird der Root-Folder (vom Typ Ordner).

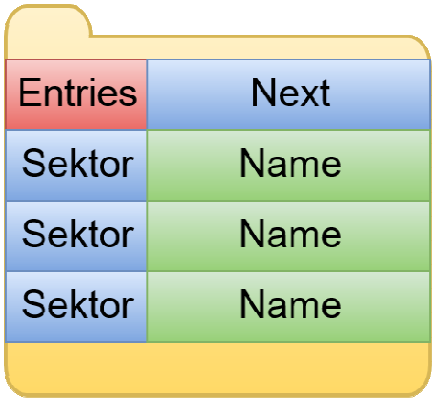
Es gibt zwei Typen von Dateien. Embedded (für sehr kleine Dateien) und Multiblock (für große).

Ein Eintrag (Ordner/Datei) wird durch die Sektornummer identifiziert in der er sich befindet.

Um Platz zu sparen wurde die Information um was für einen Eintragstyp es sich handelt und in welchen Sektor man ihn findet in 3 Bytes gepackt, einen sogenannten typesector. Das erste Bit repräsentiert hierbei den Typ (1 == Ordner), die restlichen die Sektornummer.

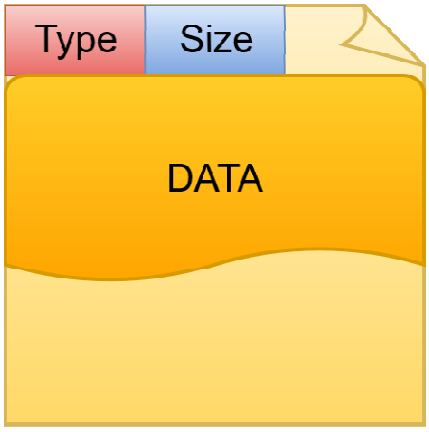
Es wurden Makros angelegt um die Benutzung der Typesektoren zu vereinfachen. Z.B. tscpy (kopieren), sfts (sektor extrahieren), tfts (typ extrahieren).

```
/*********************************************************************/
/* typefromtypesector. Returns the type of a typesector.             */
/*********************************************************************/
#define tfts(ts) ((ts.d[0] & 0x80) ? (et_dir) : (et_file))
```



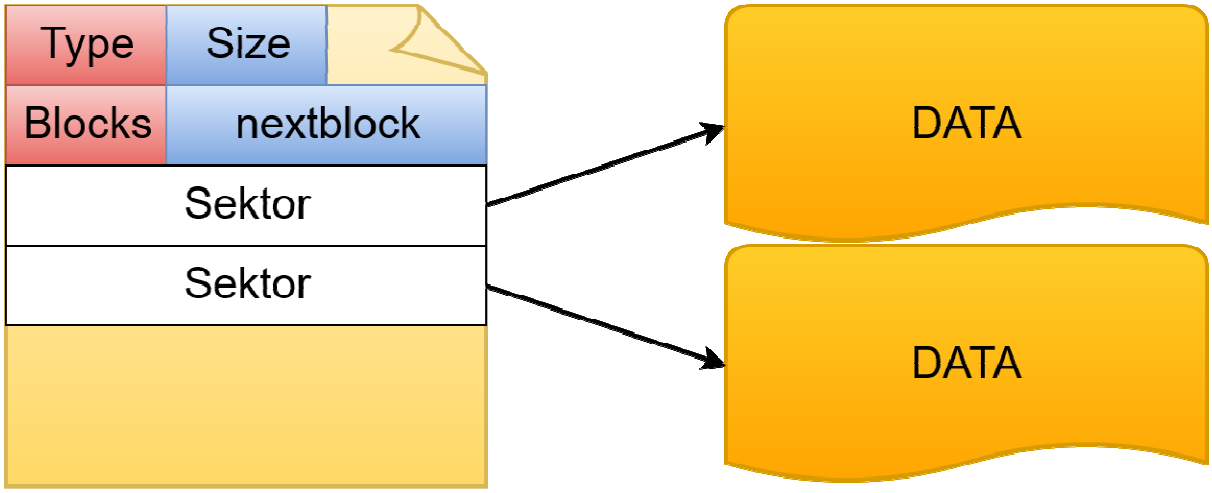
Ordnersektoren Enthalten einen 16 Byte Header welcher die Anzahl an Einträgen enthält sowie eine Sektornummer für einen Folgesektor falls kein Platz mehr für neue Einträge vorhanden ist. Die restlichen Bytes sind unbenutzt.

Der Restliche Platz im Block ist mit Filename zu typesektor Einträgen gefüllt. Ein Typesektor hat 3 Byte, ein Name kann maximal 13 Bytes lang sein, macht eine Größe von 16Bytes per Ordnereintrag und somit 31 Mögliche Einträge (abzgl. 16 Byte für Header).



Dateisektoren enthalten den Dateityp und die Dateigröße.

Im Beispiel links handelt es sich um eine Embedded Datei. Die Daten beginnen direkt unter dem Header. Da der Header lediglich 5 Byte groß ist, können Embedded Dateien bis zu 507 Byte groß werden ehe sie zu Multiblock Dateien transformiert werden müssen.



Multiblock Dateien haben den gleichen Header wie Embedded Dateien. Danach kommt jedoch eine Struktur ähnlich der eines Ordners. Blocks enthält die Anzahl der Blocks auf welche die Datei verteilt ist und nextblock eine Sektornummer falls der Platz für weitere Sektoren zu eng wird.

Die referenzierten Datenblöcke enthalten reine Daten. Mit einem einzigen Multiblock Sektor lässt sich eine Datei mit der Größe von ~83kb Organisieren ehe ein weiterer root-Dateiblock allokiert werden muss.

```
//a file contains a header and either an embedded file or multiblock data.
typedef struct __attribute__((__packed__)) utfs_file_s {
	utfs_file_header header;
	union {
		utfs_file_embedded embedded;
		utfs_file_multiblock multiblock;
	};
} utfs_file;
```

13.1 Abstraktion des Dateisystem

Für den Nutzer sollen letztendlich nur die allseits bekannten Methoden zum Bearbeiten von Dateien verfügbar sein. Dafür wurde ein Filedescriptor definiert. Er besteht aus einem Typesector (auch Ordner können durch Filedescriptoren beschrieben werden) und einem seek Wert für die aktuelle Position in einer Datei.

FS Methoden haben einen Rückgabewert mit UTFS_OK als Erfolgsrückmeldung.

Nur ein Thread kann gleichzeitig Dateioperationen durchführen (aufgrund des gemeinsam genutzten 512 Byte Puffers). Dies wird durch einen Mutex gewährleistet.

Beim Lesen/Schreiben einer Datei wird zuerst geprüft um was für einen Dateityp es sich handelt, dann werden die dazugehörigen FS-core Funktionen aufgerufen.

Es besteht die Möglichkeit der Enumeration eines Ordners. Dies geschieht mittels eines Callbacks.

Es gibt eine open Funktion welche einen absoluten Pfad annimmt. Zusätzlich gibt es eine open_ex Funktion welche es erlaubt einen Ursprungsordner für die path traversal anzugeben.

Im Folgenden die zur Verfügung stehenden Dateioperationen.

- `utfs_result utfs_open_ex(utfs_fd *fd, utfs_fd *root, char* path)`
  Öffnet die Datei / Ordner spezifiziert durch den Pfad path ausgehend vom Ursprungsordner root. Ergebnis wird in fd geschrieben. Liefert UTFS_OK bei Erfolg.
- `utfs_result utfs_open(utfs_fd *fd, char* path)`
  Wrapper für utfs_open_ex, root wird als das rootdirectory spezifiziert.
- `utfs_result utfs_close(utfs_fd *fd)`
  Schließt eine Datei.
- `WORD utfs_read(utfs_fd *fd, BYTE* data, WORD n)`
  Liest n Bytes aus fd beginnend bei seek nach data. Seek von fd wird um die Anzahl der Gelesenen Bytes erhöht. Liefert Anzahl gelesener Bytes oder EOF.
- `WORD utfs_write(utfs_fd *fd, BYTE* data, WORD n)`
  Schreibt n Bytes von data in fd an die aktuelle seek Position. Liefert Anzahl geschriebener Bytes. Seek wird um die Anzahl der geschriebenen Bytes erhöht.
- `utfs_result utfs_enum(utfs_fd *fd, BYTE(*callback)(char* n, utfs_entrytype t))`
  Iteriert durch alle Einträge eines Directories und ruft eine Callbackfunktion auf. N ist eine Referenz auf den Eintragsname, t ist der Typ des Eintrags (et_dir, et_file). Die Callbackfunktion muss, falls Dateioperationen vorgenommen wurden, 1 zurückliefern. Andernfalls 0. Bei Erfolg liefert utfs_enum UTFS_OK.
- `utfs_result utfs_seek(utfs_fd *fd, int pos, BYTE seektype)`
  Setzt den Seekwert einer Datei. Fd Beschreibt die Datei, pos die Position. Es gibt drei Mögliche Seektypen. SEEK_SET -> Absolut. SEEK_CUR -> relativ vom aktuellen Seekwert. SEEK_END -> Setzte Seek ans Ende Der Datei.
- `WORD utfs_ftell(utfs_fd *fd)`
  Liefert den Seekwert vom Filedescriptor fd.

14 Einfache Konsole

Um die Funktionalität aller Komponenten zu testen wurde eine Einfache Konsole geschrieben. Sie liefert Unterstützung für folgende simple Befehle und ist einfach erweiterbar:

- Ls
  Aktuelles Verzeichnis ausgeben, Bei Dateien die Dateigröße in Bytes.
- Cd
  Aktuelles Verzeichnis wechseln.
- Mkdir
  Neues Verzeichnis erstellen.
- Touch
  Neue Datei Erstellen.
- Format
  SD/MMC Karte Formatieren (löschen der Bitmap, schreiben eines neuen root Ordners).
- Cat
  Ausgeben des gesamten Dateiinhaltes.
- Write
  Schreiben in eine Datei.
- Uptime
  Liefert die Zeit zurück die das System bereits Läuft.
- Sleep
  Schläft für die angegebene Zeit an Sekunden.
- Echo
  Gibt den übergebenen Parameter auf der Konsole aus.

Befehle können meist ausgelagert werden.

Die Zuweisung von Befehlsname zu Befehlsfunktion erfolgt mittels einer Callbackfunktion.

```
//executes a command with arguments.
static void runcmd(char* cmd) {
        for(stringfunc* f = procs; f->string; f++) {
                if (!compareCmds(cmd, f->string)) {
                        f->func(getFirstArg(cmd));
                        return;
                }
        }

        if (*cmd) {
                writeString("unknown command ");
                writeString(cmd);
                writeString("\r\n");
        }
}
```

Relative Pfade werden unterstützt.

Die Konsole stellt folgende Befehle für Erweiterungen zur Verfügung:

- `void register_console()`
  Muss einmal aufgerufen werden, registriert die Konsole (erstellt einen Thread).
- `void writeString(char *str)`
  Schreibt einen String auf den UART.
- `utfs_fd *currentDir()`
  Liefert das Aktuelle Verzeichnis.

15 FAZIT

Viele Sachen sind gut gelungen, Es gibt keine Komponente die überhaupt nicht Läuft.

Am meisten Nachholbedarf gibt es beim Dateisystem. Die Aktuell implementierte Version ist bereits die zweite, da die erste Version zu viel Speicher gebraucht hat. Rückblickend betrachtet wäre es besser gewesen Dateien grundsätzlich in eine Art verketteten Liste mit Anhang zu schreiben (ähnlich wie bei einfachen Speicherverwaltungen). Durch den knappen Speicher muss jedes Mal das Root Directory gelesen werden um zu wissen wo es weitergeht, es wären Geschwindigkeitssteigerungen um 100% Möglich wenn wir stattdessen Verkettete Listen hätten (nur beim Lesen über Sektorgrenzen hinweg, große Nachteile treten auf wenn z.B. das Ende der Datei gelesen werden soll).

Auch gibt es noch keinen Schutz gegen das gleichzeitige Öffnen mehrerer Dateien von verschiedenen Threads. Es Wäre jedoch leicht diese Erweiterung mit Mutexen zu realisieren.

Das Timing-Modul zu Programmieren und die Fehler zu suchen war ein Krampf und dafür, dass unser Scheduler es nicht wirklich unterstützt, die Arbeit nicht wert. Eine Einbindung wäre jedoch auch zum jetzigen Zeitpunkt noch leicht möglich da keinerlei Abhängigkeiten zwischen Scheduler und Timing Modul bestehen.

Fehlerhandling fehlt an ein paar Stellen, die Wahrscheinlichkeit ist hoch, dass ein Thread Hängenbleibt falls die SD karte Versagen sollte (kam noch nie vor).

Es hätte, da unser OS ja auch I/O Operationen für Block und Character Devices unterstützt, ein Coolerer Scheduler gebaut werden können welcher I/O und CPU Times berücksichtigt.

Es ist nicht möglich alles was über Basis OS Funktionen hinausgeht zu portieren.

Ein paar Baustellen sind hier und da noch offen wie man sieht, aber man kann mit dem System gut arbeiten.