PES Zusammenfassung

ToC

1 Einführung / Keil RL-ARM GS Guide

ARM Cortex

Ist eine Standartarchitektur von Prozessoren, die für eine grosse Palette an Anwendungen benutzt werden kann. Es existieren drei Profile: A für High End Applikationen, R für Real Time Apps und M für günstige Ausführungen.

RTOS - Real Time Operating System

Bei RTOSs steht die Antwortzeit auf eingehende Events und deren Zuverlässigkeit im Fokus. Die RTOS Programmierung ist aufwendiger in der Performance und braucht mehr Speicherplatz als die herkömmliche Programmierung von Mikrokontrollern. Vor einigen Jahren war es noch nicht möglich ein RTOS auf einem Mikrokontroller zu realisieren.

Tasks

In RTOS arbeitet man nur mit Tasks. Diese sind ähnlich wie herkömmliche C-Prozesse, haben aber auch fundamentale Unterschiede.

loop anstatt return value
: Tasks laufen im unendlichen Loop

__task keyword
: Task Definitionen beginnen mit __task

Programmstart p21/22

Das Programm startet man in der main Routine. Der erste Task wird mit dem Befehl os_sys_init(task1) aufgerufen. Dieser muss nun andere Tasks starten. Sind mehrere Task gestartet, werden diese vom RTX Schduler verwaltet. Hat ein Task eine höhere Priorität als die anderen, wird nur diesem Prozessorzeit zugeteilt, bis er in den Zustand WAITING geht. Alle Tasks mit gleicher Priorität erhalten nach dem RoundRobin Prinzip Prozessorzeit.

Dem Task einen grösseren Stack mitgeben

Per default haben alle Tasks haben die gleiche Stackgrösse. Man kann aber mit "static U64 stk4 400/8" einen grossen Stack deklarieren und diesen dem Task mitgeben (Beachte *_user am Ende des Befehls):

```
tskID4 = os_tsk_create_user (Task4, priority, &stk4, sizeof (stk4));
```

Verwaltung von laufenden Tasks p24

Man kann die Prio von laufenden Tasks ändern.

```
OS_RESULT = os_tsk_prio (tskID2, priority);
OS_RESULT = os_tsk_prio_self (priority);
```

Oder man kann laufende Tasks löschen:

```
OS_RESULT = os_tsk_delete (tskID1);
OS_RESULT = os_tsk_delete_self ();
```

Oder man kann die Kontrolle dem nächsten "READY" Task gleicher Prio übergeben. Die wird benutzt, um den Scheduling Algo "co-operative" zu implementieren.

```
os_tsk_pass ();
```

Time Management p24

Clock/Time Einstellungen des RTOS

Es gib zwei Dateien mit wichtigen Einstellungen:

hal_system.c

In dieser Datei werden mit dem Befehl SysCtlClockSet(params..) der Taktteiler, Frequenz des Quartz, und, ob man die PLL benutzt, eingestellt.
Die PLL gibt immer um die 400 MHz aus. Mit dem Teiler 4, die dem Befehl mitgegeben werden, werden daraus 100 MHz. Dann gibt es intern zw. PLL und Takteiler einen weiteren /2 Teiler. Daraus resultiert also ein 50 MHz System Clock Frequenz.

RTX_Config.c

Dem System wird mitgeteilt, über welchen OS Clock es verfügt. Die 50 MHz aus der hal_system.c Datei werden hier mit "#define OS_CLOCK 50000000" eingetragen.
Ausserdem wird der OS_TICK definiert. Dieser Wert entscheidet mit dem OS_CLOCK, auf wie viel der System Tick Timer zählt. Die Formel geht so:

```
Reload-Wert = OS_TICK(zB 10ms)  * OS_CLOCK(zB 50MHz) (siehe P1 A3.2.5)
```

Das RTOS zählt den Reload Wert herunter, macht einen System Tick und reloaded den Reload Wert.

Timer Delay

Der Task wird in den WAITING State versetzt und der Scheduler schaltet zum nächsten READY Task.

```
void os_dly_wait (unsigned short delay_time_in_sysTicks)
```

Periodisches Ausführen eines Tasks

Man kann dem Task eine Zeit zuteilen, die er sich schlafen legen soll, bevor er sich wieder in den READY State begibt. Im Task:

```
void os_itv_set (unsigned short interval_time_in_sysTicks)
void os_itv_wait (void)
```

Wenn man nun im Task os_itv_wait() ausführt, wartet der Task bis das Intervall abgelaufen ist, bis er sich wieder in den READY Zustand begibt.

Virtual Timer p26

Ermöglicht es, eine Funktion aufzurufen, die nach einer bestimmten Zeit erst ausgelöst wird. Alle Virtual Timer Funktionen werden in einer os_tmr_call Funktion in der Datei RTX_Config.c definiert, und anhand eines case-switch unterschieden. Der Timer wird nach dem Aufruf os_tmr_create(switch-Wert) gesetzt.
Beim Praktikum 1 hat sich jedoch herausgestellt, dass sich der Virtual Timer nicht für sich wiederholende Aufrufe eignet, da man den Timer nach für jede Wiederholung erneut mit os_tmr_create Auslösen muss. Leider kann sich der Timer nach getaner Arbeit auch nicht selbst wieder auslösen (aus unbekannten Gründen).

Idle Demon p27 TODO

Inter Task Communication

Es gibt vier Möglichkeiten: Events, Semaphores, Mutexes und Mailboxes.

Events p28 TODO

RTOS Interrupt Handling p29

Es wird empfohlen, die Interrupts so selten wie möglich zu benutzen und statt dessen auf Events zu setzen.
Interrupts sind nicht ideal, weil der ARM7/9 alle General Purpose Pins blockiert bis man die ISR verlassen hat. Dies verzögert den SysTick und unterbricht den Kernel.
Beim Cortex-M ist das Verwenden des Interrupts weniger

proplematisch. Dem im Gegensatz zum ARM7/9 besitzt er nested Interrupts. Die INTs mit höherer Priorität können also andere INTs unterbrechen.
Trotzdem ist es ratsam, den Code in der ISR möglichst kurz zu halten.

```
void IRQ_Handler (void) __irq  {
 isr_evt_set (0x0001, tsk3);    // Signal Task 3 with an event
 EXTINT = 0x00000002;           // Clear the peripheral interrupt flag
 VICVectAddr = 0x00000000;      // Signal end of interrupt to the VIC
}
```

Die ISR setzt lediglich einen Event ab. Das eigentliche Handling spielt sich im Task ab ,der auf diesen Event wartet:

```
void Task3 (void)  {
 while (1)  {
  os_evt_wait_or (0x0001, 0xffff);  // Wait until ISR triggers an event
 // Handle the interrupt
 }                                  // Loop and go back to sleep
}
```

Semaphores p32 TODO
Mutex p38 TODO
Mailbox p39 TODO
RTX Kernel s7/p19

Besteht aus einem Scheduler, Zeit- und Speichermanagement Services. Es gibt drei Schedulung Algos:

- round-robin
- pre-emptive, and cooperative multitasking of program tasks

Ein zusätzliches RTOS Objekt ermöglicht Inner-task Kommunikation:

- Event triggering
- Semaphores
- Mutex
- Ein Mailbox System

2 Scheduling

Task Konzept

Jeder Task sollte eine bestimmte Funktion ausführen. Das Ziel ist, ein objektorientiertes und modulares Design und daraus die oft zitierten Vorteile zu erhalten: Die Task in anderen Projekten wiederverwendet werden. Bei Fehlfunktionen kann das Problem schnell auf eine Task zurückgeführt werden. Die Task können isoliert, oder in einer Simulationsumgebung getestet werden.

Multitasking

Die Task werden parallel ausgeführt. Die hat wiederum viele Vorteile:

- Prozessor IDLE vermeiden
  Während der Prozessor zB auf ein Laufwerk wartet können andere Tasks erledigt werden.
- Mehrere Benutzer an einem System
- Klarere Programmstrukturierung
  Embedded System mit vielen Teilsystemen: Abbildung der realen Aussenwelt.

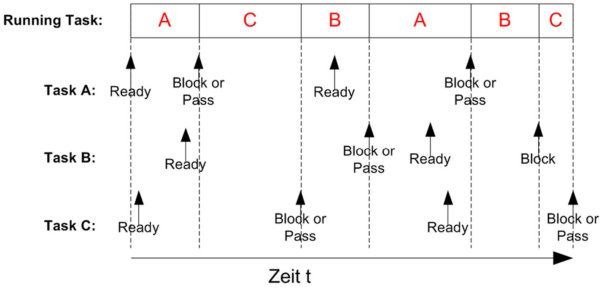
Scheduler und Scheduling Optionen

Der Scheduler teilt den Tasks anhand von zu definierenden Scheduling Algos Prozessorzeit zu und veranlasst unter gegebenen Bedingungen einen Taskwechsel. Es gibt diese Scheduling Algos:

- Round Robin
  - Jede Task hat gleiche Priorität

- Task läuft bis sie blockiert (os_wait..)
- Oder, falls sie nicht blockiert, bis der Round Robin Timeout eintritt
- Bei RL ARM lässt sich sich die Anzahl Timeout Ticks einstellen
  In der Datei rtx_config.c

- Preemptive Scheduling
  Wie RR, aber die Tasks können solche mit niedriger Priorität unterbrechen.
  Problem: Priority Inversion
  Bsb Prios Task 1-3: pT1 > pT2 > pT1
  T1 delegiert an T3, und wartet auf dessen Antwort. T3 arbeitet, wird dann aber von T2 unterbrochen. Nun hat also im übertragenen Sinne T2 T1 unterbrochen, trotz niedriger Priorität.
- Co-Operative Scheduling
  Der Taskwechsel findet nur statt, wenn der aktuelle Task blockiert.



RL-ARM Scheduling Tabelle

RR gewählt	Task Prioritäten	Scheduling
Nein	Unterschiedlich	Pre-emptive
Ja	Gleich	RR
Nein	Gleich	Co-operative
Ja	Unterschiedlich	Pre-emptive und RR

Probleme Prozess Synchronisation

- Mutual Exclusion
  Wenn ein Task kritische Daten bearbeitet, darf er von einem nebenläufigen Task, der die gleichen Daten bearbeitet, nicht unterbrochen werden.
- Deadlock
  T1 wartet auf Informationen von T2. T2 kann aber nicht weiterarbeiten, da er darauf wartet, bis T1 fertig ist.
- Synchronisation
  Ein Prozess kann erst weiterlaufen, wenn er Informationen von einem anderen Prozess hat.

Task Kreieren und Prioritäten setzen (Keil GS p23 / s14)

Am besten benutzt man den ersten Task um die eigentlichen Tasks zu starten. Dies geschieht mit dem Befehlt os_tsk_create(). Dem neuen Task können auch Priorität und Parameter mitgegeben werden.

```
tskID3 = os_tsk_create_ex (Task3, priority, parameter);
```

Priority nimmt Werte von 1-255. Die Priorität 0 ist für den IDLE Task reserviert.
Hat man alle Task erstellt, sollte man den ersten Task mit dem Befehl os_tsk_delete_self() töten.

Taskwechsel

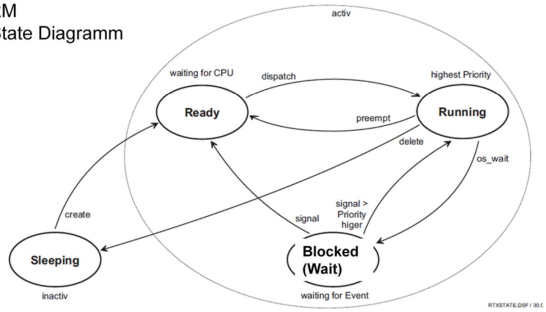
Zuerst einmal braucht der Scheduler einen Timer, um die

Zeit, zu messen. In unserem Fall ist dies der SysTick Timer des (Cortex-M-)Prozessors.
Bei jedem Taskwechsel müssen alle Informationen des abzulösenden Tasks gespeichert werden. Man unterscheidet zwischen "State" und "Runtime" Informationen. Ersteres wird im Task Stack und letzteres im Task Control Block gespeichert. Jede Task hat seinen eigenen Task Stack, in den sie bei einem Wechsel ihre Infos speichert. der Task Control Block wird vom Kernel benutzt und verwaltet.
Eine wichtige Kenngrösse ist die "Context Switch Time", die den Zeitbedarf eines Taskwechsels angibt. Sie hängt vom Kernel und vom Hardware Design ab.

Taskzustände s8

Können 4 verschiedene Zustände haben: Ready, Running, Blocked (Wait) und Sleeping. Die meiste Zeit verbringt der Task in den Zuständen Blocked und Ready.

RL-ARM
Task State Diagramm



Der Scheduler kann alle Tasks, die im Zustand Ready sind aktivieren und somit in den Zustand Running versetzen.
Auf p21 des Getting Started Buchs sind alle Zustände und deren Bedeutung tabelliert.

RUNNING	The currently running TASK
READY	TASKS ready to run
WAIT DELAY	TASKS halted with a time DELAY
WAIT INT	TASKS scheduled to run periodically
WAIT OR	TASKS waiting an event flag to be set
WAIT AND	TASKS waiting for a group event flag to be set
WAIT SEM	TASKS waiting for a SEMAPHORE
WAIT MUT	TASKS waiting for a SEMAPHORE MUTEX
WAIT MBX	TASKS waiting for a MAILBOX MESSAGE
INACTIVE	A TASK not started or detected

Events

Task 1 wartet auf einen Event
Task 2 schickt Event an Task 2

Semaphores

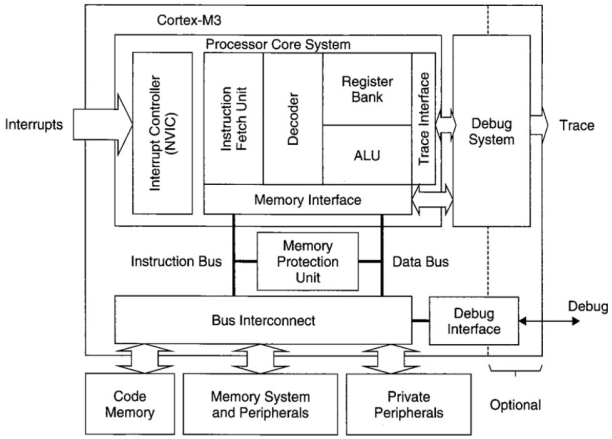
Semaphores step by step
Weiteres im Buch (getting stated) s35
Barrier Turnstile
Alle Tasks warten aufeinander bis sie den Entry Turnstile erreichen, Alle verlassen den Turnstile geleichzeitig.

3 ARM Architektur

Geschichte

Cortex-M3 Architektur

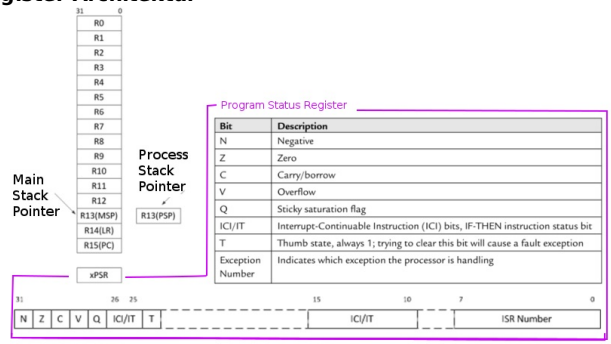
Cortex-M3 benutzt eine (Modified) Harvard Architektur



Instruction Pipeline

Die Pipeline hat 3 Stufen: Fetch, Decode und Execute. Der Vorteil der Pipeline ist, das Befehle parallel ausgeführt werden können. Der Nachteil ist, das bei einem Branch. die ganze Pipeline geflusht werden muss. Allerdings wird versucht diesem Nachteil mit Branch Prediction entgegenzutreten.

Register Architektur



R0-7 low registers
: Können von allen (auch 16-bit Thumb) Befehlen verwendet werden.

R8-12 high registers
: Können von Thumb-2 aber nicht von 16 Bit Thumb verwendet werden.

R13 Main/Process Stack Pointer
: Der MSP ist der default Stack Pointer. Es wird benutzt vom OS kernel, Exception Handlers und allen Programmen, die priviligierten Zugriff haben. Der PSP wird von "normalen" Programmen benutzt, solange sie nicht einen Exception Handler ausführen, benutzt. Wenn ein Prgromm läuft, kann es nur anderen Stack zugreifen, in dem die Spezialbefehle MSR oder MRS ausgeführt werden.

R14 Link Register
: Stores PC on subroutine calls. If several layers of subroutines are called, R14 is stored on Stack.

PSR Program Status Register

Control Register (Nicht auf dem Bild)

Hat lediglich 2 Bits.

Ctrl 0

Entscheidet ob es nach einer Exception im User oder Priviledged Thread Mode weitergeht. Ist nur im priviledged Mode beschreibbar.

Ctrl 1

Gibt an, welcher Stack benutzt wird (MSP oder PSP). Im Handler Mode immer 0. Es kann nur im Privilidged Thread mode beschrieben werden. Eine andere Methode dieses Bit zu überschreiben wäre das Bit-2 vom LR zu ändern. Dann wird beim Zurückkehren aus einer Exception dieses Bit angepasst.

Bedingte Befehle

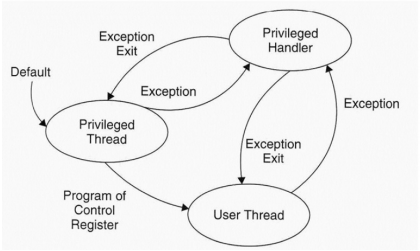
If/else Konditionen verursachen normalerweise Pipeline Branches. Dank der Conditional Execution im ARM Befehlssatz können Branches in diesen Situationen umgangen werden. Dafür werden die Assembler Befehle mit Suffixen erweitert.
Die ersten 4 Bits (31-28) im PSR geben Auskunft über den letzten ausgeführten Befehl. Sie werden gesetzt bei Vergleichsoperationen wie zB CMP automatisch gesetzt. Möchte man diese Bits auch bei anderen Operationen setzen, wird ein 'S' hinter den Assembler Befehl angefügt. Die vier Bits (siehe Tabelle) entscheiden nun zusammen mit den Condition Codes am Ende des nächsten Befehls, ob dieser ausgeführt oder mit einem NOOP zum nächsten übergegangen wird.
Condition Codes (zB EQ, CC, ...) werden wie das 'S' an die mnemonics des Assemblerbefehls angefügt.

Assembler Kürzel	Negativ **N**	Zero **Z**	Carry Over **C**	Over flow **V**	**Bedeutung**
EQ	x	1	x	x	Equal
NE	x	0	x	x	Not Equal
CS	x	x	1	x	Carry bit set
CC	x	x	0	x	Carry bit cloear
MI	1	x	x	x	Minus (Negativ)
PL	0	x	x	x	Plus (Positiv)
VS	x	x	x	1	Overflow bit set
VC	x	x	x	0	Overflow bit clear
HI	x	0	1	x	Higher Than (unsigned)
LS	x	1	0	x	Lower Same (signed)
GE	0 1	X x	X x	0 1	Greater or equal
LT	1 0	X x	X x	0 1	Less than
GT	Z = 0 UND (N = V) ….siehe ARM Reference Manual				Greater than
LE	Z = 1 OR (N != V) …. siehe ARM Reference Manual				Less than or equal
AL	x	x	x	x	Allways

Betriebsarten

Operation Modes	Privileged	User
Exception	Handle Mode	
Main Program	Thread Mode	Thread Mode

Privilege Levels



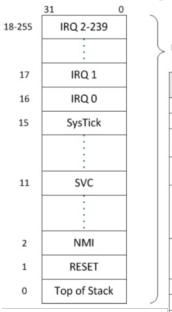
Das Programm startet zuerst im Priviledged Thread Modus und wechselt später in den User Thread Modus. Von da aus kann nur über eine Exception zum Priviledged Thread Modus gewechselt werden. Wenn man eine Exception aufruft, kommt man in den Priviledged Handler. Hier muss man nun das Controll Register Bit-0 auf 0 für Privileged Thread oder auf 1 für User Thread Mode setzen.

4 NVIC - Nested Vectored Interrupt Controller

Was passiert als erstes bei einem Interrupt?

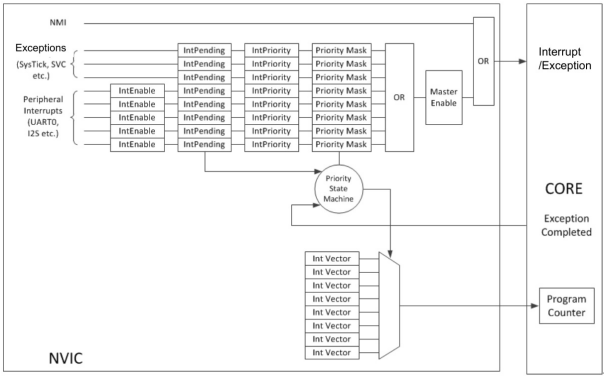
prgrm counter ändert Wert zu demjenigen des Interrupts aus der Interrupt Vektor Table

Cortex-M3 Interrupts und Exceptions



Exception Number	Exception Type	Priority	Description
1	Reset	−3 (Highest)	Reset
2	NMI	−2	Nonmaskable interrupt (external NMI input)
3	Hard Fault	−1	All fault conditions, if the corresponding fault handler is not enabled
4	MemManage Fault	Programmable	Memory management fault; MPU violation or access to illegal locations
5	Bus Fault	Programmable	Bus error; occurs when AHB interface receives an error response from a bus slave (also called *prefetch abort* if it is an instruction fetch or *data abort* if it is a data access)
6	Usage Fault	Programmable	Exceptions due to program error or trying to access coprocessor (the Cortex-M3 does not support a coprocessor)
7–10	Reserved	NA	–
11	SVCall	Programmable	System Service call
12	Debug Monitor	Programmable	Debug monitor (breakpoints, watchpoints, or external debug requests)
13	Reserved	NA	–
14	PendSV	Programmable	Pendable request for system device
15	SYSTICK	Programmable	System Tick Timer

NVIC

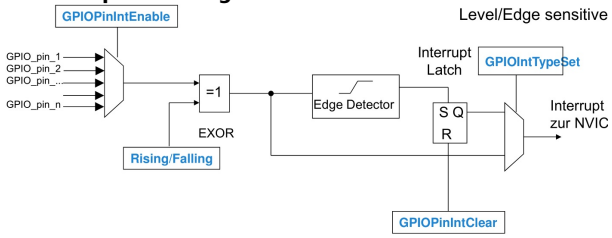


Anstehende Interrupts kommen zuerst ins IntPending Register. Stehen mehr als ein Interrupt an, entscheidet die Priority State Machine in welcher Reihenfolge die Interrupts ausgeführt werden.

Priotity Grouping

Priority	IntPriorityGroupingSet			
IntpriioritySet	3	2	1	0
0x0	0	0	0	0
0x20	1	0	0	0
0x40	2	1	0	0
0x60	3	1	0	0
0x80	4	2	1	0
0xa0	5	2	1	0
0xc0	6	3	1	0
0xe0	7	3	1	0

GPIO Interrupt Handling



Cortex Stack
Nested Interrupts

5 Cortex-M Stackpointer & ARM/Thumb Befehlssätze

Cortex Stack

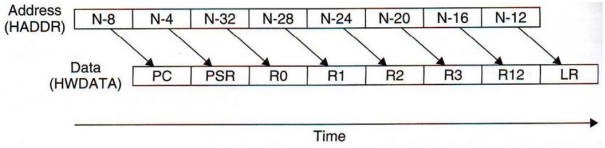
Es gibt eigentlich zwei Stack Pointers, nacheinander im RAM angelegt: Der Main Stack Pointer für das System (Handler Mode) und der Process Stack Pointer für User

Space Anwendungen (Privileged Thread Mode).

Interrupt Prozesswechsel

Bevor eine ISR ausgeführt werden kann, müssen alle Register auf den Stack geladen werden. Das kostet 12 Clock Zyklen. Das zurückladen in die Register kostet auch wieder 12 Zyklen.

Dank der Harvard Architektur kann das Register in den Stack geladen und gleichzeitig das Programm gefetched werden. Deshalb wird bei einem Wechsel zuerst der PC auf den Stack geladen. Somit kann das Programm gleich anschliessend das neue Programm laden, während es den Rest der Register auf den Stack bläst.



Address	Data	Push Order
Old SP (N) ->	(Previously pushed data)	-
(N-4)	PSR	2
(N-8)	PC	1
(N-12)	LR	8
(N-16)	R12	7
(N-20)	R3	6
(N-24)	R2	5
(N-28)	R1	4
New SP (N-32) ->	R0	3

Bei jedem Interrupt werden 8x4 Bytes gestacked.

Interrupt Situationen

Ein Zurückkehren zum Hauptprogramm nach einem Interrupt kostet jedesmal 12 Clock Zyklen. In den drei folgenden Situationen kehrt der Cortex-M nicht in das Hauptprogramm zurück, sondern springt direkt zum nächsten Interrupt. Deshalb muss es nicht die Register vom Hauptprogramm laden und spart so wertvolle Zyklen. Das wechseln zwischen zwei Interrupts, das sogenannte Tail Chaining, kostet 6 Zyklen.

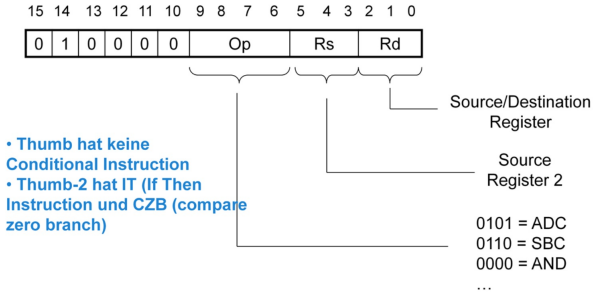
Nested Interrupts
: Ein Interrupt unterbricht einen Int niedrigerer Priorität.

Tail Chaining Interrupts
: Ein Interrupt gleicher oder niedrigerer Priorität folgt gleich anschliessend an einen anderen Interrupt.

Late Arrival Interrupts
: Ein Interrupt wird während ein Interrupt niedrigerer Priorität initiiert wird, noch vor dessen Ausführung vorgeschoben.

Befehlssätze

ARM 32 Bit
Thumb 16 Bit



Thumb 16 hat nur 3 Bits für die Adressierung der Register, es können also nur 8 Register direkt angesprochen werden.

Thumb2 16 Bit mit 32 Bit Befehlen

6 Filesystem folien/HPES15_6_filesystem.pdf

Es gibt 4 Drives:

- **F: Parallel Flash**
- **S: Serial (SPI) EEPROM**
- **R: Parallel SRAM (internes RAM, lab6.1-6.7)**
- **M: MM/SD Card (ab lab6.8)**

Die Einstellungen der Drives werden in der Datei File_config.c gemacht.

Inistialisierung

finit("M:") Iniitalisiert das Filesystem

File Stream

fopen() reserviert einen Buffer, diesen nennt man auch Stream. Um die CPU zu entlasten, wird der Stream wird nicht immer sofort in die Datei gesschrieben, sondern dann, wenn die CPU gerade nichts anderes zu tun hat oder bei fclose() und fflush().

File I/O Routines

Function	Description
fopen	Creates a new file or opens an existing file.
fclose	Writes buffered data to a file and then closes the file.
fflush	Writes buffered data to a file.

Functions for opened files

Function	Description
feof	Reports whether the end of the file stream has been reached.
ferror	Reports whether there is an error in the file steam.
fseek	Moves the file stream in pointer to a new location.
ftell	Gets the current location of the file pointer.
rewind	Moves the file stream in file pointer to the beginning of the file.

Functions for reading and writing to file streams

Function	Description
fwrite	Writes a number of bytes to the file stream.
fputc	Writes a character to the file stream.
fputs	Writes a string to the file stream.
fprintsf	Writes a formatted string to the file stream.
fread	Reads a number of bytes from the file stream.
fgetc	Reads a character from the file stream.
fgets	Reads a string from the file stream.
fscanf	Reads a formatted string from the file stream.

fopen() options

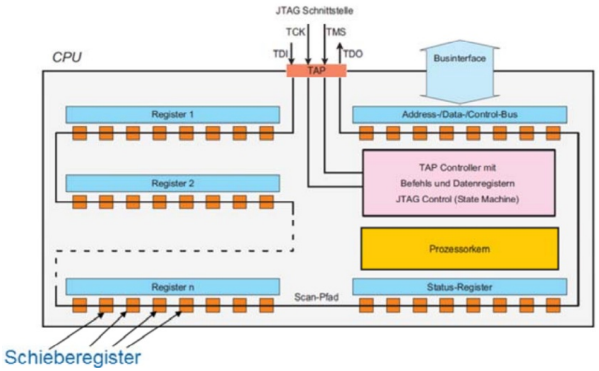
Mode	Description
"r"	Read mode. Opens a file for reading.
"w"	Write mode. Opens an empty file for writing. If the file exists, the content is destroyed. If the file does not exist, an empty file is opened for writing.
"a"	Append mode. Opens a file for appending text. If the file already exists, data is appended. If the file does not exist, an empty file is opened for writing. Opening a file with append mode causes all subsequent writes to be placed at the then current end-of-file, regardless of interfering calls to fseek().
"b"	Binary mode. Can be appended to any character above, but has no effect. The character is allowed for ISO C standard conformance.
"+"	Update mode. Can be used with any character above as a second or third character. When a file is opened with update mode, both reading and writing can be performed. Programmers must ensure that reading is not directly followed by writing without an interfering call to `fflush()` or to a file positioning function (`fseek()` or `rewind()`). Also, writing should not be followed directly by reading without an interfering call to a file positioning function, unless the writing operation encounters EOF.

7 Debuggin folien/HPES15_7_Debugging.pdf

Debug Monitor

In Circuit Emulator (ICE)

JTAG Schnittstelle

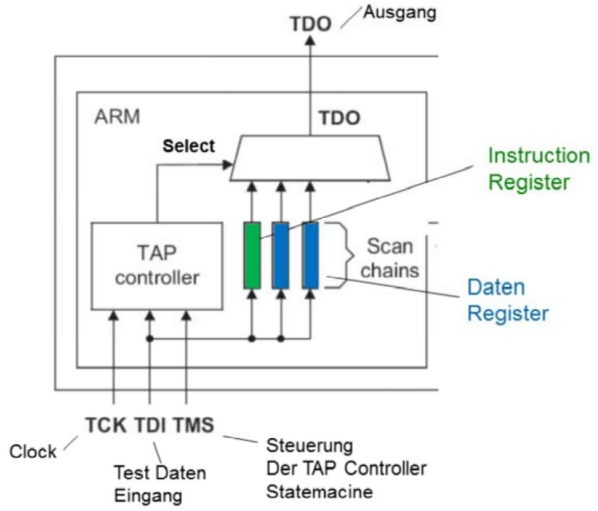


Zusätzliche Ffs werden an den Pineingängen platziert, um Verbindungen zu testen. Diese kann über eine serielle Schnittstelle abfragen und modifizieren. Ursprünglich wurde JTAG für das Überprüfen von Lötverbunden eingesetzt, heute kann man damit auch Debuggen.
JTAG benutzt zwei Scan Chains. Eines für Instructions und eine für Daten.

TAB Controller und Pins TDI, TCK, TMS & TDO

Table 14–1. IEEE Std. 1149.1 Pin Descriptions

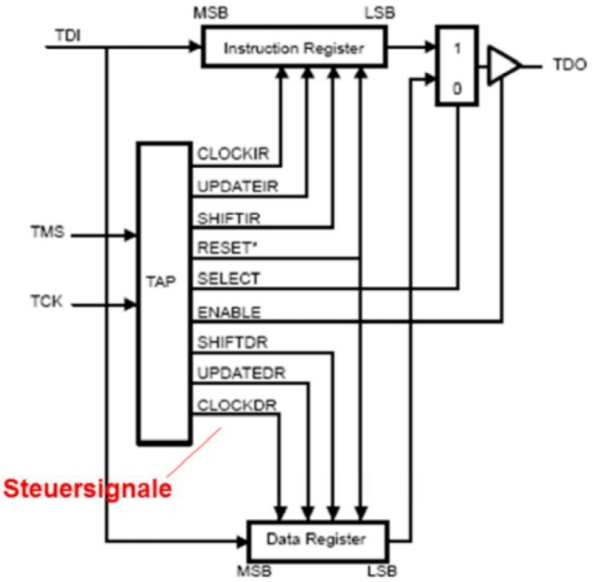
Pin	Description	Function
TDI	Test data input	Serial input pin for instructions as well as test and programming data. Signal applied to TDI is expected to change state at the falling edge of TCK. Data is shifted in on the rising edge of TCK.
TDO	Test data output	Serial data output pin for instructions as well as test and programming data. Data is shifted out on the falling edge of TCK. The pin is tri-stated if data is not being shifted out of the device.
TMS	Test mode select	Input pin that provides the control signal to determine the transitions of the TAP controller state machine. Transitions within the state machine occur at the rising edge of TCK. Therefore, TMS must be set up before the rising edge of TCK. TMS is evaluated on the rising edge of TCK. During non-JTAG operation, TMS is recommended to be driven high.
TCK	Test clock input	The clock input to the BST circuitry. Some operations occur at the rising edge, while others occur at the falling edge. The clock input waveform should have a 50% duty cycle.



Scan Chains

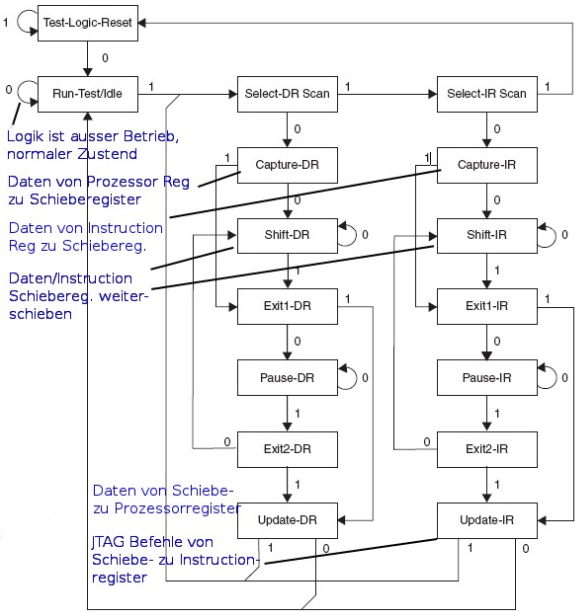
Es können mehrere Daten- oder Befehlsregister definiert werden.

TAB Controller signals



- Enable
  Turns on the TDO output
- Select
  switches between Instruction and Data Registers
- Shift
  shifts the contents of the scan chain
- Update
  Updates the register assoziated to the scan chain. The register is updated with the contents of the scan chain
- ClockIR
  latches contents of Register into scan chain

TAB Controller State Machine



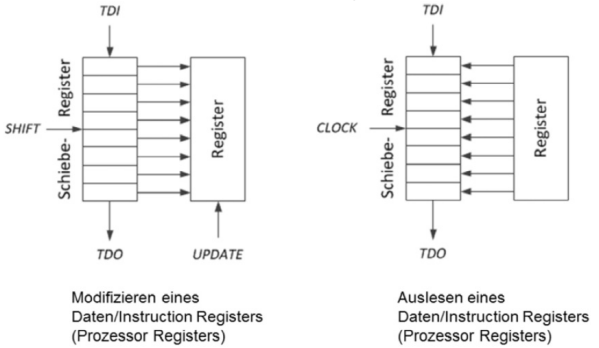
Test-Login-Reset

Ist der Zustand von TMS 5 steigenden Flanken des TCK hintereinander auf high, geht der TAP controller in den Zustand "Test-Logig-Reset".

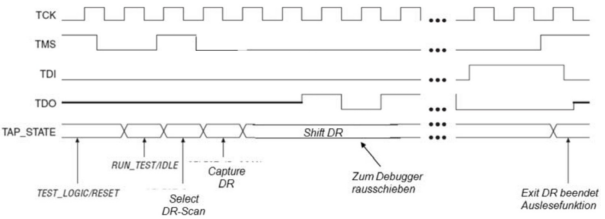
Shift-DR

Daten werden auf TDO geschoben.

Lesen und Schreiben der Befehl/Datenregister



BSP



ARM Debugging