



Lehrinhalte:

- Programmierung von Mikrocontrollern ohne Betriebssystem
- Erstellen von Binärcodes (Cross-Kompilieren) für ARM-Prozessoren
- Ausführen von Programmen auf dem BeagleBone ohne Betriebssystem
- Einlesen von Tastern, Ansteuerung von LEDs
- Aufbau eines GPIO-Treibers anhand von Datenblättern
- Echtzeitfähige Sensor- und Aktorsteuerung durch Interrupts und Polling
- Ansteuern von Servo und DC-Motoren
- Einlesen von analogen Signalen eines Helligkeitssensors und eines Joysticks
- Vervollständigung eines I²C-Treibers
- Kommunizieren mit einem Port-Expander über I²C
- Sensorbasierte Regelung eines Roboters

Zu erwerbende Kompetenzen:

- Funktionsweise von Mikrocontrollern
- Funktionsweise von Registern
- Hardwarenahes Programmieren
- Erstellung und Nutzung von Treiberdateien
- Echtzeitsysteme und Echtzeitfähigkeit
- Interaktion zwischen Hard- und Software
- Kommunikation zwischen elektronischen Komponenten
- Informationen aus Datenblättern in Programmcode umsetzen

Hinweis:

- Die im Anhang enthaltenen Datenblätter sind freundlicherweise zur Verfügung gestellt von Texas Instruments© .



Praktikum Echzeitfähige Geräte und Roboter

TECHNISCHE
UNIVERSITÄT
MÜNCHEN



Fakultät
Maschinenwesen
Medizinische Fakultät
Klinikum rechts des Isar / DHM

TUM MiMed

Lehrstuhl Mikrotechnik und Medizingerätetechnik
Univ.-Prof. Dr. Tim C. Lüth



PRAKTIKUM ECHTZEITFÄHIGE GERÄTE UND ROBOTER

Skript, Aufgaben und Dokumentation

v1.13

0. GRUNDLAGENSKRIPT	7
0.1 EINLEITUNG.....	7
0.2 ABLAUF DES PRAKTIKUMS	8
0.3 AUFBAU EINES MIKROCONTROLLERS	9
0.4 BIT-MANIPULATION	10
0.4.1 <i>Logische Operationen</i>	10
0.4.2 <i>Setzen von Bits</i>	12
0.4.3 <i>Löschen von Bits</i>	13
0.4.4 <i>Umschalten von Bits</i>	13
0.4.5 <i>Zusammenfassung</i>	13
0.5 HARDWARE IM PRAKTIKUM.....	14
0.6 DAS BEAGLEBONE	15
0.6.1 <i>Aufbau des AM3359</i>	16
0.6.2 <i>Register</i>	17
0.6.3 <i>Adressen</i>	17
0.6.4 <i>PINs</i>	17
0.6.5 <i>PIN-Muxing</i>	17
0.6.6 <i>Kompilieren und Booten</i>	18
0.7 EGR-CAPE	18
0.8 INSTALLATION DER ENTWICKLUNGSUMGEBUNG	19
0.8.1 <i>Virtualbox installieren</i>	19
0.8.2 <i>Extension Pack installieren</i>	19
0.8.3 <i>Appliance importieren und VM starten</i>	19
0.8.4 <i>Gemeinsamen Ordner einrichten</i>	20
0.8.5 <i>Entwicklungsumgebung erkunden</i>	21
0.9 PROGRAMMIERUNG MIT ECLIPSE	22
1. TERMIN: GRUNDLAGEN DER EIN- UND AUSGABE	27
1.1 GPIO - PINS.....	27
1.2 EINE LED EINSCHALTEN	27
1.2.1 <i>Pin-Muxing</i>	28
1.2.2 <i>Pin als Output konfigurieren</i>	30
1.2.3 <i>Pin einschalten</i>	30
1.3 EINEN TASTER EINLESEN	31
1.4 GPIO TREIBER	32
1.5 AUFGABEN TERMIN 1: GRUNDLAGEN DER EIN- UND AUSGABE.....	37
2. TERMIN: ANTRIEBE UND ECHTZEIT	41
2.1 EINLEITUNG.....	41
2.2 INTERRUPTS.....	41
2.2.1 <i>Unterschiede zum Polling</i>	41
2.2.2 <i>Interrupts beim BeagleBone</i>	42
2.2.3 <i>Interrupts des GPIO-Moduls</i>	43
2.3 TIMER.....	43
2.3.1 <i>Was ist ein Timer</i>	43
2.3.2 <i>Timer des BeagleBone</i>	44
2.4 MOTORENSTEUERUNG	47
2.4.1 <i>PWM – Puls Weiten Modulation</i>	47
2.4.2 <i>Der Servo-Antrieb</i>	48

2.4.3 DC-Motoren	49
2.5 AUSGABE AUF DEM PC-BILDSCHIRM.....	52
2.6 AUFGABEN TERMIN 2: ANTRIEBE UND ECHTZEIT	58
3. TERMIN: EIN- /AUSGABEGERÄTE UND SENSOREN	61
3.1 MODULE	61
3.1.1 LDR-Modul.....	61
3.1.2 Joystick-Modul	61
3.2 VERWENDUNG DES IM PROZESSOR INTEGRIERTEN ADC	61
3.2.1 Aktivieren des ADC-Moduls.....	61
3.2.2 Einstellen der ADC Parameter	62
3.3 STARTEN VON SD-KARTE	64
3.4 AUFGABEN TERMIN 3: EIN- AUSGABEGERÄTE UND SENSOREN	68
4. TERMIN: KOMMUNIKATIONSSCHNITTSTELLEN	71
4.1 I ² C-KOMMUNIKATION	71
4.1.1 Systemarchitektur	72
4.1.2 Protokoll.....	73
4.2 PROGRAMMIERUNG DER KOMMUNIKATION	74
4.2.1 Funktionsweise des Port-Expanders	75
4.2.2 Initialisierung.....	76
4.2.3 Daten senden	76
4.2.4 Daten empfangen.....	77
4.2.5 Sensoren auslesen	78
4.3 BINÄRE BILDSCHIRMAUSGABE	79
4.4 ÜBERPRÜFEN DER KOMMUNIKATION MITELS LOGIC-ANALYZER	79
4.5 AUFGABEN TERMIN 4: KOMMUNIKATIONSSCHNITTSTELLEN	86
5. TERMIN: OLYMPIADE	90
6. DATENBLATT DES EGR-CAPES	94
7. MODULDATENBLÄTTER.....	100
8. MEMORY MAP (BASISADRESSEN)	104
9. CONTROL MODULE (PINMUXING) REGISTER	114
10. GPIO REGISTER	122
11. INTERRUPTS	136
12. EHRPWM REGISTER.....	142
13. DM-TIMER REGISTER	150
14. TOUCHSCREEN CONTROLLER (ADC) REGISTER.....	160
15. I²C REGISTER	170
16. ANTWORTEN ZU DEN VERSTÄNDNISFRAGEN	188



Praktikum Echzeitfähige Geräte und Roboter

TECHNISCHE
UNIVERSITÄT
MÜNCHEN



Fakultät
Maschinenwesen
Medizinische Fakultät
Klinikum rechts des Isar / DHM

TUM MiMed

Lehrstuhl Mikrotechnik und Medizingerätetechnik
Univ.-Prof. Dr. Tim C. Lüth



Grundlagenskript

Scripthistorie:

*Ein großer Dank an Jonas Pfeiffer und Johannes Coy
für die ursprüngliche Konzeption des Skripts.*

*Danke auch an die zwei langjährigen HIWIs Tim Moser und Anton Robe
für die qualifizierten Beiträge bei der Skriptüberarbeitung.*

0. Grundlagenskript

0.1 Einleitung

In einer Vielzahl von Systemen befinden sich intelligente Steuerungssysteme, um die zunehmend komplizierter werdenden Prozesse in Systemen zu kontrollieren. Bei zeit- oder sicherheitskritischen Anwendungen steigen die Anforderungen an die Systemkomponenten und deren Zusammenwirken untereinander. In integrierten Systemen werden häufig Mikrocontroller eingesetzt. Diese bieten meist die Möglichkeit einer echtzeitfähigen Abarbeitung von Algorithmen mit geringer Komplexität, unter anderem deshalb, weil kein Betriebssystem benötigt wird. Mikrocontroller umgeben uns im täglichen Leben, auch wenn sie nicht immer sichtbar sind. Sie sind enthalten in industrieller Steuerungstechnik (Fertigungsüberwachung, Robotersteuerungen), in medizinischen Geräten (bildgebende medizinische Geräte, medizinischer Navigation, Geräte zur Überwachung der Lebensfunktionen), Telekommunikation (Telefone, TK-Anlagen, Mobiltelefone), Unterhaltungselektronik (CD/DVD-Spieler, Fernseher, Spielekonsolen), in der Fahrzeugtechnik (Einspritzregelung, elektronische Wegfahrsperre, Navigationssystem) und vielen anderen Bereichen.

Im Folgenden wird mit binären und hexadezimalen Zahlen gearbeitet. Um in C eine binäre Zahl zu definieren, wird ihr das Kürzel „0b“ vorangestellt (z.B. 0b01010011 = 83). Einer hexadezimalen Zahl wird ein „0x“ vorangestellt (0xFF = 0b11111111 = 255).

Die Nummerierung der Bits erfolgt von rechts nach links (0 bis 7), das bedeutet, die 1 in folgender Zahl: 0b00000010 steht an Stelle 1. Das Bit mit der höchsten Bitwertigkeit steht ganz links, das kleinste Bit steht rechts.

0.2 Ablauf des Praktikums

Das Praktikum wird jedes Semester sowohl als einwöchiges Blockpraktikum in der vorlesungsfreien Zeit, als auch semesterbegleitend angeboten. Es umfasst insgesamt fünf Termine (Kapitel 1 bis Kapitel 5), für deren Bearbeitung im Blockpraktikum jeweils ein Tag vorgesehen ist. Da das semesterbegleitende Praktikum nur einmal pro Woche halbtags stattfindet, sind dort jeweils zwei Termine pro Kapitel vorgesehen.

Zu Beginn jedes Termins wird die Theorie des aktuellen Kapitels jeweils in einem Kurzvortrag erklärt. Danach können Sie die Kapitel selbstständig bearbeiten und sich an der Lösung der jeweiligen Aufgaben versuchen. Bei Fragen helfen Ihnen die Tutoren jederzeit gerne weiter.

Sie müssen nicht zwingend alle Aufgaben eines Kapitels innerhalb eines Termins schaffen. Während des Praktikums ist ausreichend Puffer vorgesehen, sodass Sie Rückstände leicht aufholen können, denn nicht alle Kapitel sind gleich umfangreich. Genauso können Sie auch schon vorausarbeiten, wenn Sie ein Kapitel schneller lösen können, als in der vorgesehenen Zeit. Sie sollten lediglich versuchen, alle Aufgaben bis zum Ende des Praktikums fertig zu bekommen. Vergessen Sie außerdem nicht, sich die Aufgaben eines Kapitels jeweils von einem Tutor abnehmen zu lassen.

Die Benotung des Praktikums besteht einerseits aus der Abnahme Ihrer gelösten Aufgaben. Zusätzlich wird in vier kleinen schriftlichen Testaten (Dauer ca. 15 min) der Stoff des jeweiligen Kapitels abgefragt. Ihre Endnote setzt sich aus der Gesamtleistung aller Testate und der Abnahme der praktischen Aufgaben zusammen.

Am letzten Termin des Praktikums (Kapitel 5) findet eine Olympiade statt. Dort können Sie alles bisher gelernte anwenden, um Ihren selbst programmierten autonomen Roboter gegen andere Teilnehmer antreten zu lassen.

Verzweifeln Sie nicht, falls Sie an einer Stelle im Praktikum hängen, oder sich am Anfang noch schwer tun sollten. Sie werden sich wundern, in wie kurzer Zeit Sie vom ersten Anschalten einer LED bis hin zur vollautomatischen Steuerung eines Roboters kommen werden.

Wir wünschen Ihnen viel Spaß in unserem Praktikum!

0.3 Aufbau eines Mikrocontrollers

Ein Mikrocontroller ist im Grunde ein vollständiger Rechner mit Prozessor, Ein-/Ausgabeeinheit, Datenbus und Speicher (Abbildung 1). Im Gegensatz zum PC befinden sich alle Komponenten auf einem Chip.

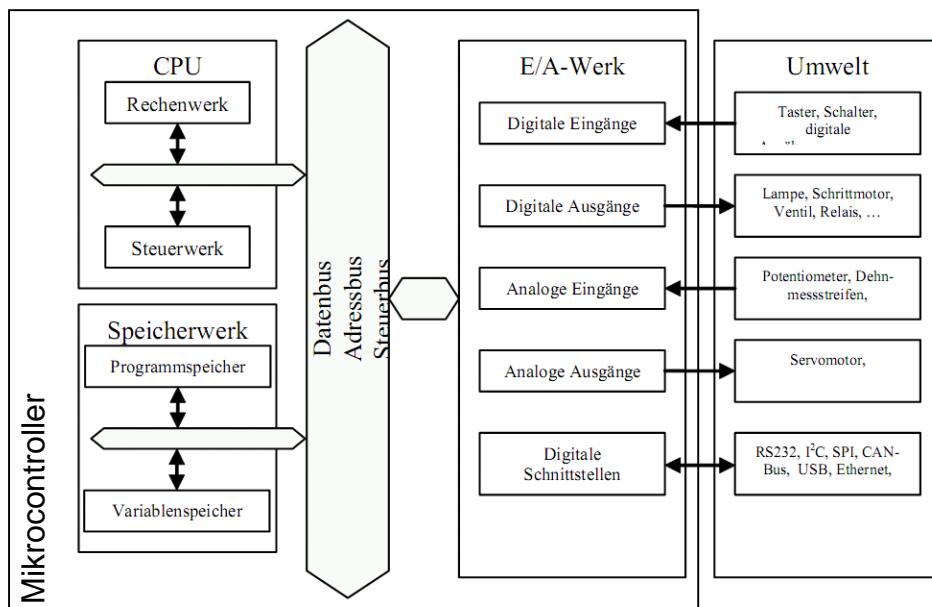


Abbildung 1: Prinzipieller Aufbau eines Mikrocontrollers (CPU, Speicherwerk, E/A-Werk, Bus).

Die CPU (Prozessor) führt logische und arithmetische Operationen aus, welche im Befehlssatz definiert sind. Das E/A-Werk stellt Ein- und Ausgabeschnittstellen zur Verbindung mit Peripheriegeräten her, deren Schnittstellendefinition einem entsprechenden Datenblatt entnommen werden kann. Man unterscheidet zwischen digitalen und analogen Ein- und Ausgängen. An digitalen Eingängen können beispielsweise Taster oder andere binäre Sensoren angeschlossen werden. Sie besitzen genau zwei Zustände: 0 oder 1. Digitale Ausgänge können Schrittmotoren, Ventile, Relais und viele weitere Komponenten ansteuern. Mit den analogen Eingängen können kontinuierliche Spannungssignale eingelesen und zeit- sowie wertdiskret weiterverarbeitet werden. Beispielsweise können damit Potentiometer, Dehnungsmessstreifen oder lichtabhängige Widerstände ausgewertet werden. Digitale Schnittstellen dienen zur Kommunikation mit anderen Geräten unter anderem über RS232, SPI, I²C® oder CAN-Bus.

0.4 Bit-Manipulation

Um Mikrocontroller zu programmieren, müssen häufig einzelne Bits in einem Register verändert werden, ohne dass der Inhalt der restlichen Bits verloren geht, oder es soll nur ein Wert ausgelesen werden. Am übersichtlichsten ist dies mit logischen Operatoren möglich. Die folgenden Beispiele zeigen aus Gründen der Übersicht nur 8Bit Variablen. In unserem Praktikum werden immer 32 Bit Register beschrieben. Die Vorgehensweise bleibt aber genau gleich.

0.4.1 Logische Operationen

Zum Manipulieren von Bits stehen verschiedene Operatoren zur Verfügung. Im Folgenden werden die grundlegendsten Operatoren vorgestellt.

AND-Operator (&):

Der AND-Operator verknüpft zwei Zahlen miteinander. Diese Verknüpfung findet bitweise statt und liefert für jedes Paar von Bits 1, wenn beide Bits 1 sind und sonst 0.

X	1 0 1 1 0 0 1 1
Y	0 1 1 0 1 0 0 1
X AND Y	0 0 1 0 0 0 0 1

```
unsigned int a = 60;           // 60 = 0b00111100
unsigned int b = 13;           // 13 = 0b00001101
unsigned int c = 0;

c = a & b;                  // 12 = 0b00001100
```

OR-Operator (|):

Auch beim OR- Operator werden die Zahlen bitweise miteinander verknüpft. Allerdings wird das Ergebnis hier nur 0, wenn beide Bits 0 sind, sonst liefert der Operator das Ergebnis 1.

X	1 0 1 1 0 0 1 1
Y	0 1 1 0 1 0 0 1
X OR Y	1 1 1 1 1 0 1 1

```
unsigned int a = 60;           // 60 = 0b00111100
unsigned int b = 13;           // 13 = 0b00001101
unsigned int c = 0;

c = a | b;                  // 61 = 0b00111101
```

XOR-Operator (^):

Der XOR-Operator liefert beim bitweisen Vergleich als Ergebnis 0, wenn beide Bits gleich sind, sonst 1.

X	1 0 1 1 0 0 1 1
Y	0 1 1 0 1 0 0 1
X XOR Y	1 1 0 1 1 0 1 0

```
unsigned int a = 60;           // 60 = 0b00111100
unsigned int b = 13;           // 13 = 0b000001101
unsigned int c = 0;

c = a ^ b;                   // 49 = 0b00110001
```

Einerkomplement / Bitweise invertieren (~):

Beim Einerkomplement wird jedes Bit invertiert. Das heißt, aus einer 0 wird eine 1, und aus einer 1 wird eine 0.

X	1 0 1 1 0 0 1 1
~	0 1 0 0 1 1 0 0

```
unsigned int Value=4;          //    4 = 0b00000100
Value = ~ Value;              // 251 = 0b11111011
```

Linksschieben (<<) und Rechtsschieben (>>):

Beim Linksschieben werden die Bits einer Zahl um einen beliebigen Betrag nach links verschoben, beim Rechtsschieben analog nach rechts.

Die frei gewordenen Stellen werden mit dem Vorzeichen-Bit, d.h. Nullen (bei nicht vorzeichenbehafteten bzw. vorzeichenbehafteten, positiven Variablen) oder Einsen (bei vorzeichenbehafteten, negativen Variablen), aufgefüllt.

X	1	0	1	1	0	0	1	1
X<<1	0	1	1	0	0	1	1	0
X<<2	1	1	0	0	1	1	0	0
X<<3	1	0	0	1	1	0	0	0
X>>1	0	1	0	1	1	0	0	1
X>>2	0	0	1	0	1	1	0	0
X>>3	0	0	0	1	0	1	1	0

```
unsigned int Value=4;           //  4 = 0b00000100
unsigned int Shift=2;           //  2 = 0b00000010

Value = Value << Shift; /* 16 = 00010000, d.h. die 1 wird zwei
                           nach links geschoben */

Value <<= Shift; /* 64 = 01000000 , noch mal 2nach links*/
```

0.4.2 Setzen von Bits

Es werden zum setzen, löschen, umschalten (0 wird zu 1 und umgekehrt) einzelner Bits Bitmanipulationen durchgeführt, welche die oben erläuterten logischen Operationen nutzen.

Definition: BIT0 = 0, BIT1=1, BIT2=2, BIT3=3...

Anfangszustand von REG: REG=0b10011010

```
REG |= (1<<BIT2);

→ (1<<BIT2) = (0b00000001<<2) = 0b00000100;
→ REG = REG | (0b00000100) = 0b10011010 | 0b00000100 = 0b10011110;
```

Also behält REG durch das OR den aktuellen Zustand und das zweite Bit wird gesetzt (egal ob es bereits gesetzt war). Diese Operation kann auch für mehrere Bits gleichzeitig durchgeführt werden (siehe Beispiel im nächsten Kapitel). Hier werden die weiteren zu setzenden Bits mit „OR“ angehängt: REG |= (1<<BIT2) | (1<<BIT0) | (1<<BIT2)

0.4.3 Löschen von Bits

Definition: BIT0 = 0, BIT1=1, BIT2=2, BIT3=3...

Momentaner Zustand von REG: REG=0b10011010

Die Manipulationen können auch auf mehrere Bits gleichzeitig ausgeführt werden:

```
REG &= ~((1<<BIT0) | (1<<BIT2) | (1<<BIT3));  
  

→ 0b00000001<<0 = 0b00000001  

→ 0b00000001<<2 = 0b00000100  

→ 0b00000001<<3 = 0b00001000  

→ ((0b00000001) | (0b00000100) | (0b00001000)) = 0b00001101  

→ ~ (0b00001101) = 0b11110010;  

→ REG = REG & (0b11110010) = 0b10011010 & 0b11110010 = 0b1000010;
```

Hier werden in REG BIT0, BIT2 und BIT3 gleichzeitig gelöscht (äußere Klammer beachten!)

0.4.4 Umschalten von Bits

Definition: BIT0 = 0, BIT1=1, BIT2=2, BIT3=3...

Momentaner Zustand von REG: REG=0b10011010

```
REG ^= (1<<BIT0) | (1<<BIT3);  
  

→ (0b00000001<<0) = 0b00000001;  

→ (0b00000001<<3) = 0b00001000;  

→ (0b00000001) | (0b00001000) = 0b00001001;  

→ REG = 0b10011010 ^ (0b00001001) = 0b1000011;
```

0.4.5 Zusammenfassung

Setzen: REG |= (1<<BIT0) | (1<<BIT1) | (1<<BIT2) | (1<<BIT3);

Löschen: REG &= ~((1<<BIT0) | (1<<BIT1) | (1<<BIT2) | (1<<BIT3));

Umschalten: REG ^= (1<<BIT0) | (1<<BIT1) | (1<<BIT2) | (1<<BIT3);

0.5 Hardware im Praktikum

Im Praktikum EGR werden Sie lernen, wie Mikrocontroller im Allgemeinen programmiert werden und wie Sie aus Datenblättern die notwendigen Informationen entnehmen können, um hardwarenahe Anwendungen zu schreiben. Sie werden lernen, wie Sensoren ausgelesen und Aktoren angesteuert werden. Die vorhandene Hardware besteht aus einem BeagleBone, einer für dieses Praktikum entwickelten Erweiterungsplatine (EGR-Cape) und diverser Module wie LEDs oder Sensoren und einem Chassis mit Sensoren und Motoren. Das Chassis verfügt über einen Litium-Ionen-Akku, der mittels eines Schalters (on/off) auf der rechten Seite des Chassis zugeschaltet werden kann. Wenn der Roboter über das Netzteil mit Strom versorgt wird, ist es empfehlenswert, diesen Schalter auf ‚on‘ zu stellen, um den Akku zu laden.

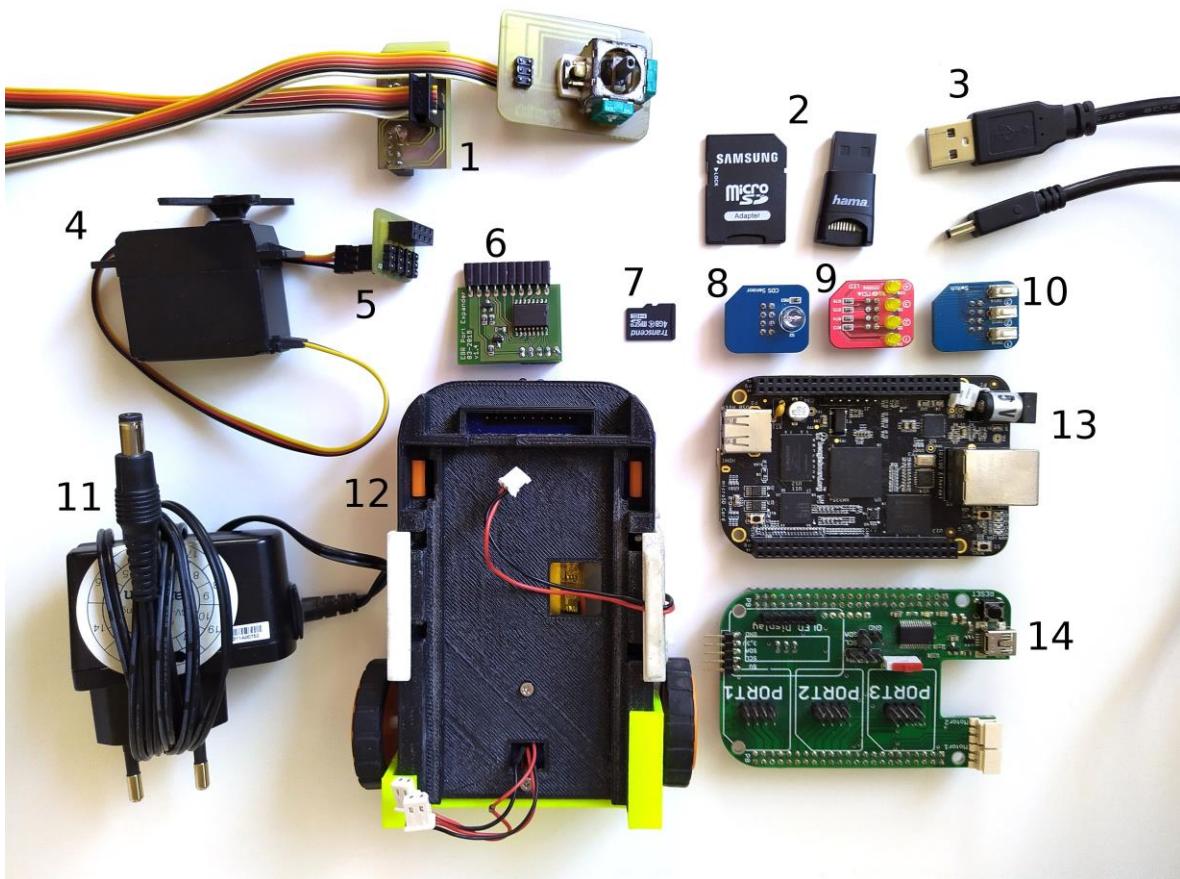


Abbildung 2: Die Hardware des Praktikums

In Abbildung 2 sehen Sie alle Komponenten, die ein Set enthält. Bei Erhalt sollten Sie dieses auf Vollständigkeit prüfen:

- | | | |
|----------------------|-------------------------------------|------------------|
| 1: Joystick | 2: jeweils ein SD- oder USB-Adapter | 3: USB_Kabel |
| 4: Servomotor | 5: Servo-Adapter | 6: Port-Expander |
| 8: Helligkeitssensor | 9: LED-Modul | 10: Taster-Modul |
| 12: Chassis mit Akku | 13: BeagleBone | 14: EGR-Cape |

0.6 Das BeagleBone

Das BeagleBone ist ein single-board computer (SBC), der neben einem Mikrocontroller noch zusätzliche Hardware besitzt. Er ist vergleichbar mit weiteren bekannten SBCs, wie z.B. dem Raspberry Pi.

SBCs beinhalten meist ganze Computersysteme samt Spannungsversorgung und programmierbarer Ausgänge, sowie gängige Schnittstellen, wie USB oder Netzwerkanschlüsse auf einer einzigen Platine. Sie finden typischerweise Verwendung in rechenintensiven Maschinensteuerungen (embedded systems) im Industrie- und Hobbybereich, sowie in der Robotik.

Aber auch einfachere bekannte Entwicklungsboards, wie Arduino oder Nucleo folgen einem ähnlichen Aufbau: In allen Fällen besteht ein solches Board aus einem Mikrocontroller und zusätzlicher Hardware zur einfachen Nutzung der Funktionen und zur Programmierung des Mikrocontrollers.

So sind beispielsweise Spannungsversorgung oder größere Speichermengen auf dem Beagle-Bone vorhanden. Außerdem bietet es physikalische Schnittstellen, an denen Hardware angeschlossen werden kann. So sind viele Kontakte des Mikrocontrollers direkt als „Pin“ in den Pin-Leisten herausgeführt, sodass Hardware, wie unser Cape, einfach angeschlossen werden kann.

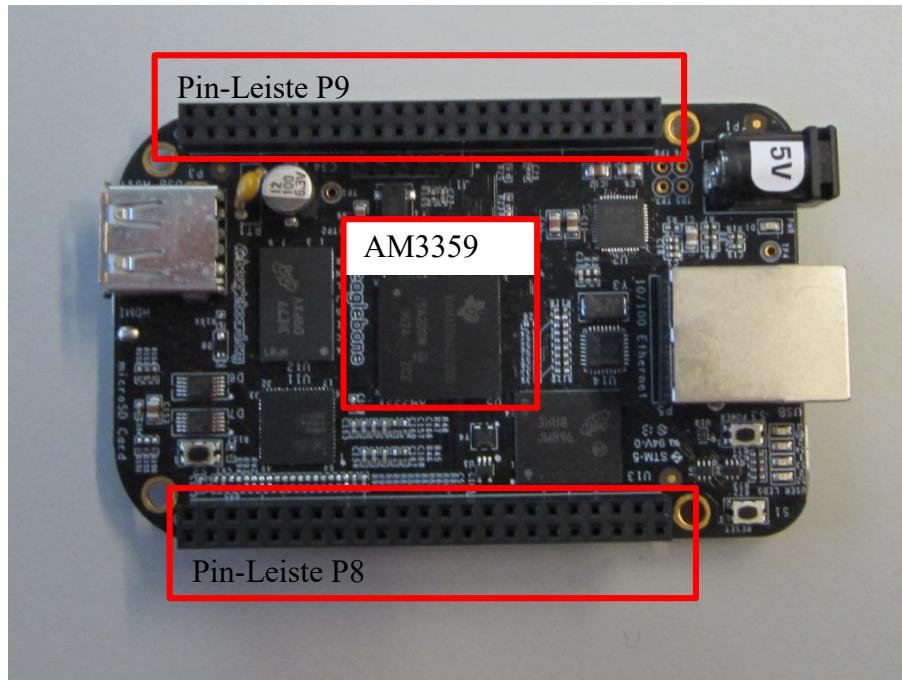


Abbildung 3: BeagleBone Black

Der Funktionsumfang eines single-board computers ist also hauptsächlich durch die Art des verbauten Mikrocontrollers und dessen Fähigkeiten definiert. Im Falle des BeagleBones ist es der AM3359 von Texas Instruments, einem ARM-basierten und damit weit leistungsfähigeren Chip als alle z.B. bei Arduino verbauten Atmel AVRs.

0.6.1 Aufbau des AM3359

Der Mikrocontroller AM3359 beinhaltet einen ARM Cortex-A8 Prozessor (Cortex-A Serie) und ist dadurch, im Gegensatz zur Cortex-M Serie, nicht nur für die Verwendung in eingebetteten Systemen geeignet, sondern auch in der Lage, performance-intensive Linux-Betriebssysteme, wie Android auszuführen. Deswegen findet dieser Prozessor auch Anwendung in mobilen Endgeräten, wie Smartphones oder Tablet-PCs. Der Prozessor ist nach ARM Standard aufgebaut. ARM definiert den Basisbefehlssatz (ARMv7), sodass unabhängig vom Hersteller die gleichen Compiler für alle ARM Prozessoren verwendet werden können.

Neben dem Prozessor beinhaltet der Mikrocontroller auch zusätzliche periphere Module, die in Abbildung 4 abgebildet sind. Diese Module stehen mit der CPU über das L3/L4 Interconnect in Verbindung und können vom Programm dynamisch aktiviert und deaktiviert werden. Sie bestehen aus speziellen Hardware-Schaltungen, die spezielle Aufgaben (z.B. Auswertung bestimmter Kommunikationsprotokolle, wie EtherCAT oder Ansteuerung eines Displays) unabhängig vom Prozessor erledigen können und somit die CPU entlasten.

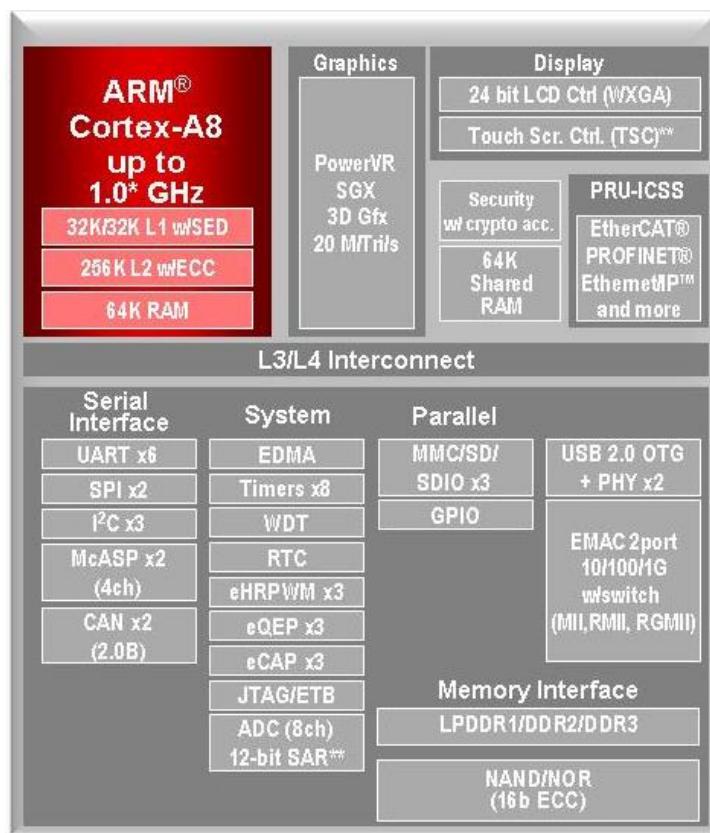


Abbildung 4: Aufbau eines AM335x Mikrocontrollers (www.ti.com)

In diesem Praktikum werden wir vor allem die GPIO-Module (Termin 1), die eHRPWM Module (Termin 2), die ADCs (Termin 3) und das I²C Modul (Termin 4) verwenden. Sie alle sind Teil des E/A-Werk des Mikrocontrollers. Eine genaue Beschreibung der Komponenten und wie sie zu benutzen sind, ist im **Technical Reference Manual** von TI nachlesbar. Dieses Dokument ist die Grundlage dieses Praktikums und sowohl das gesamte Handbuch als auch Ausschnitte daraus werden Ihnen zur Verfügung gestellt.

0.6.2 Register

Der ARMv7 ist ein 32 Bit System, das bedeutet, dass er 32 Bit in einem Takt gleichzeitig verarbeiten kann. Passenderweise sind die Speichereinheiten daher 32 Bit lang. Eine solche Speichereinheit, die den Zustand des Systems beschreibt, bezeichnen wir als Register. In einem Register ist beispielsweise gespeichert, ob ein Modul angeschaltet ist oder nicht, indem ein Bit (0 oder 1) im Controlregister gesetzt ist. Durch Schreiben und Lesen von Registern wird beeinflusst, wie sich der Prozessor und seine Peripherie verhalten soll.

Um in ein Register in der Sprache C schreiben zu können, benötigt man die Adresse des Registers. Diese Adresse muss zweimal gecastet (umwandeln eins Datentyps in der Sprache C) werden. Zum besseren Verständnis und zur besseren Lesbarkeit ist dieser Befehl bereits zu dem Befehl **HardWare REGister „HWREG ()“** zusammengefasst.

`HWREG(Adresse)=Wert;`

Oder

`HWREG(Basisadresse+Offset)=Wert;`

0.6.3 Adressen

Die Adresse eines Registers beschreibt, wo sich das Register befindet und ist bei unserem Mikrocontroller 32 Bit lang. Wir schreiben sie meist in hexadezimaler Schreibweise z.B.: 0x44E10000. Adressen werden im Handbuch (TRM) als Basisadresse und Offset angegeben (Adresse = Basisadresse + Offset). So müssen zum Beispiel bei mehreren GPIO-Modulen nur die Basisadressen geändert werden, wenn man von Modul 1 auf 2 wechselt. Die Adressen der Module sind in der Memory Map des TRM aufgelistet (Ausschnitt TRM Memorymap) dieOffsets der Register können dann in den einzelnen Modulbeschreibungen nachgelesen werden.

0.6.4 PINs

Als Pin bezeichnen wir die physikalische Schnittstelle des Microcontrollers (Output oder Input). Zu Beachten ist, dass der Mikrocontroller deutlich über 200 Pins besitzt. Nicht alle dieser Pins sind dabei zu den Buchsenleisten des Beaglebones herausgeführt und nur ein kleiner Teil davon wird in diesem Praktikum verwendet.

0.6.5 PIN-Muxing

Fast Alle Pins des Prozessors können in verschiedenen Modi betrieben werden, das bedeutet, einem Pin können unterschiedliche Funktionen zugewiesen werden (allerdings immer nur eine Funktion gleichzeitig). Die Funktion eines Pins wird durch Setzen des entsprechenden Werts im Control-module-Register festgelegt. Dieser Vorgang wird als **Pin-Muxing** bezeichnet. In der Dokumentation des Prozessors (TI-TRM oder Ausschnitt PIN-MUXING) kann die Adresse des Konfigurationsregisters für jeden Ball/Pin nachgelesen werden. Vor den meisten Anwendungen muss der Pin erst entsprechend seiner Funktion gemuxt werden. Im Control-Register des Pins können außerdem andere Verschaltungen wie Pull-Ups eingestellt werden.

0.6.6 Kompilieren und Booten

(weitergehender Stoff, nicht Prüfungsrelevant)

Das Erstellen von Code für ARM-Prozessoren bedarf einer speziellen ARM-Toolchain, die den z.B. in C geschriebenen Source-Code in die maschinenlesbare Form des ARMv7 Prozessors übersetzt. Um einen Code auf einen Prozessor ausführen zu können, muss der Prozessor zunächst wissen, welchen Code er ausführen soll. Häufig ist es möglich durch das Überbrücken bestimmter Kontakte (z.B. durch Drücken eines Buttons) den Speicher auszuwählen, in welchem der Prozessor nach dem ausführbaren Code suchen soll. In unserem Fall ist der Prozessor so konfiguriert, dass er im Speicher des BeagleBones nach ausführbarem Code sucht. Hier wird ein Programm MLO gefunden, das ein weiteres Programm (u-boot) in den Speicher lädt und ausführt. Diese Programme heißen Bootloader. Normale PCs (Mac oder Win) arbeiten meist ebenfalls mit Bootloadern. Die Besonderheit beim Bootsetup des BeagleBones ist, dass der Bootloader u-boot nach einer Konfigurationsdatei (uenv.txt) sucht. Sie wird zunächst auf der SD-Karte gefunden. In dieser Datei ist festgelegt, dass der Maschinencode per serieller Übertragung an das BeagleBone geschickt wird (Termin 1) und dass es diesen ausführen soll. Wenn diese Datei geändert wird, kann man das Bootverhalten ändern, z.B. um bei jedem Reset immer das gleiche Programm von der SD-Karte zu laden. Diese Möglichkeit wird in Termin 5 benötigt.

0.7 EGR-Cape

Weitere Hardware des Praktikums ist das EGR-Cape. Es wird über die Pin-Leisten auf das BeagleBone aufgesteckt und ist so direkt mit den Pins des Mikroprozessors verbunden (Siehe Datenblatt Cape). Auf das Cape können die Hardwaremodule an 3 verschiedenen Steckplätzen angesteckt werden. Zusätzlich verfügt das Cape über eine Pin-Leiste zum Anschluss der Infrarot-Sensoren, ein Display und über zwei H-Brücken zur Ansteuerung der Motoren des Chassis. Zur Programmierung des BeagleBones und zur Stromversorgung ist außerdem ein USB-Anschluss mit virtuellem COM-Port vorhanden (FTDI-Chip). Die Installation des virtuellen COM-Ports findet im ersten Termin statt.



Abbildung 5: EGR-Cape aufgesteckt auf das BeagleBone mit LED-Modul.

Das Cape bietet zusätzlich noch die Möglichkeit, Signale zu debuggen. Das Mitlesen von Signalen mittels Logic Analyser oder Oszilloskop ist eine wichtige Methode um Fehler in Hardware und hardwarenaher Software zu finden. Das Messen der Spannungen kann aber je nach Leiterbahnbreite oder Komplexität des Boards sehr schwierig sein, daher sind die wichtigsten Signale direkt mit Debug-Pins verbunden, um das Messen zu erleichtern.

0.8 Installation der Entwicklungsumgebung

Um mit möglichst vielen unterschiedlichen Computern und Betriebssystemen kompatibel zu sein, haben wir uns entschieden, die Entwicklungsumgebung für das Praktikum EGR in einer virtuellen Maschine bereitzustellen. Sie erhalten von uns einen USB-Stick, der die Installationsdateien, sowie einige Datenblätter, die im Verlauf des Praktikums benötigt werden, beinhaltet. Kopieren Sie den gesamten Inhalt des Sticks auf Ihren Rechner.

0.8.1 Virtualbox installieren

Um eine mit aktuellen Treibern und Betriebssystemen kompatible Version von Virtualbox zu erhalten, laden Sie sich bitte unter www.virtualbox.org/wiki/ → **Downloads** die aktuellste Version herunter, die Ihrem aktuell genutzten Betriebssystem entspricht. Installieren Sie die Software.

0.8.2 Extension Pack installieren

Als nächsten Schritt laden Sie das **VirtualBox Extension Pack** herunter. Dieses enthält z.B. die USB-Unterstützung, die wir im Verlauf des Praktikums benötigen.

Falls die Installation per Doppelklick bei Ihnen nicht funktionieren sollte, können Sie das Extension Pack auch über „Datei“ → „Einstellungen“ → „Zusatzpakete“ manuell zur VirtualBox hinzufügen.

0.8.3 Appliance importieren und VM starten

Nach der vollständigen Installation kann die vorbereitete VirtualBox über die Schaltfläche „Datei“ → „Appliance importieren“ importiert werden. Im folgenden Dialog die Datei „**EGRX.X.active.ova**“ auswählen, die Sie sich zuvor vom USB-Stick kopiert haben.

Anschließend die Virtuelle Maschine über einen Doppelklick oder die Schaltfläche „Starten“ hochfahren.

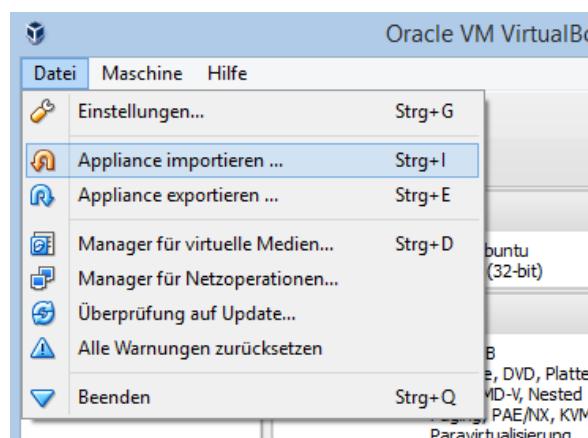


Abbildung 6: Screenshot VM Appliance importieren

0.8.4 Gemeinsamen Ordner einrichten

Um Dateien zwischen der virtuellen Maschine und dem Host-Betriebssystem auszutauschen, wird ein gemeinsamer Ordner benötigt. Der Inhalt dieses Ordners wird zwischen Ihrem Host-Betriebssystem und der virtuellen Maschine synchronisiert, damit Sie Dateien zwischen beiden Systemen austauschen können. Dies wird benötigt, um ab Kapitel 3 Programmdateien auf eine SD-Karte kopieren zu können, mit der Sie Ihren Roboter dann starten können, ohne das Programm per USB-Verbindung aufspielen zu müssen.

Erstellen Sie dazu an beliebiger Stelle einen Ordner in Ihrem Host-System und binden Sie ihn dann in der virtuellen Maschine ein. Im Oracle VM VirtualBox Manager geschieht dies über Rechtsklick auf die virtuelle Maschine → Ändern → Gemeinsame Ordner → neuer Ordner (Ordnersymbol mit grünem Plus). Wählen Sie über Ordner-Pfad → Ändern einen beliebigen Ordner und setzen Sie das Häkchen bei „Automatisch einbinden“ und „Permanent erzeugen“. Dieser Ordner erscheint dann (nach einem evtl. Neustart der VM) als „sf_Ordnername“ auf dem Desktop der VM.

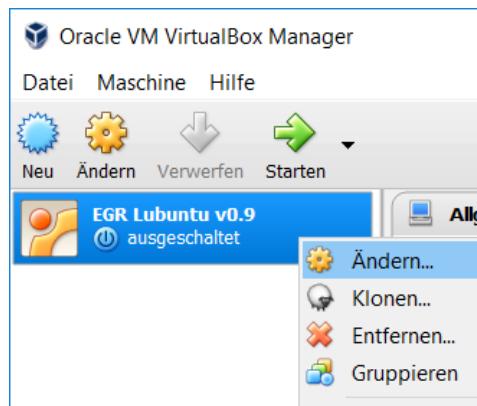


Abbildung 7: Screenshot Ändern der VM-Einstellungen

0.8.5 Entwicklungsumgebung erkunden

In der virtuellen Maschine läuft das Betriebssystem Lubuntu, ein Ableger der bekannten Linux-Variante Ubuntu, der mit einer besonders ressourcenschonenden Benutzeroberfläche ausgestattet ist (LXDE, Lightweight X11 Desktop Environment). Aus diesem Grund eignet sich Lubuntu besonders für ältere Rechner.

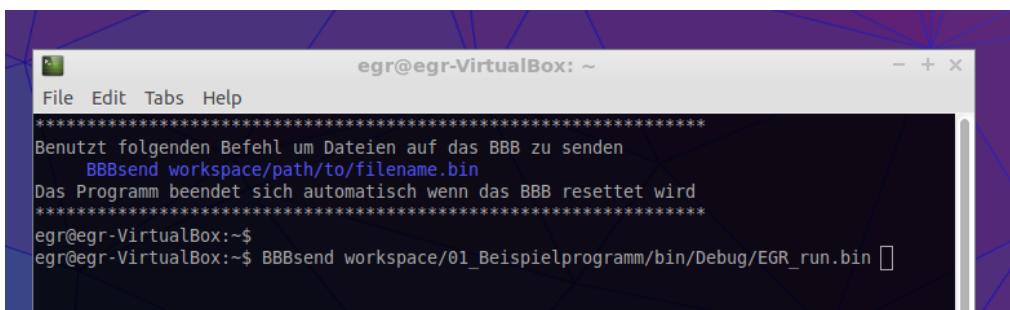
In der virtuellen Maschine sind zwei wesentliche Programme vorinstalliert:

1. Eclipse

In der Entwicklungsumgebung Eclipse wird der Quellcode erstellt und das Programm kompiliert (Siehe Kapitel 0.9).

2. BBBsend

Weil die erstellten Programme nicht auf Ihrem Rechner, sondern auf dem Roboter ausgeführt werden sollen, müssen die kompilierten Programmdateien über ein spezielles Skript per USB-Verbindung auf das BeagleBone übertragen werden. Dieses Skript wird über das Terminal aufgerufen und dient dazu, eine Verbindung zum BeagleBone herzustellen und Ihre Programme hochzuladen. Um ein Programm hochzuladen, müssen Sie den Befehl „BBBsend“, gefolgt vom Dateipfad der kompilierten Programmdatei (BIN-Datei) eingeben (siehe Abbildung 8).



```
egr@egr-VirtualBox: ~
*****
Benutzt folgenden Befehl um Dateien auf das BBB zu senden
  BBBsend workspace/path/to/filename.bin
Das Programm beendet sich automatisch wenn das BBB resettet wird
*****
egr@egr-VirtualBox:~$ BBBsend workspace/01_Beispielprogramm/bin/Debug/EGR_run.bin
```

Abbildung 8: Programmfenster BBBsend

Die Ordnerstruktur mit sämtlichen Quelldateien für die Programmierung aller Aufgaben ist bereits angelegt. Sie finden diese im Dateiexplorer (blaues Ordnersymbol in der Taskleiste) im Home-Verzeichnis im Ordner „workspace“.

0.9 Programmierung mit Eclipse

Eclipse ist eine bekannte Entwicklungsumgebung für die Programmierung in verschiedenen Sprachen. In diesem Praktikum werden wir die Programmiersprache C verwenden, um Programme zu erstellen, die auf dem Mikrocontroller ausgeführt werden.

Softwareprojekte werden in Eclipse immer in einem sogenannten „Workspace“ gespeichert. Für das Praktikum müssen Sie kein neues Projekt anlegen. Optimalerweise ist der Workspace bereits in Eclipse eingebunden und dort im Project Explorer als Ordnerstruktur zu sehen. Falls nicht, können Sie das noch manuell erledigen:

File → Import → Existing Projects into Workspace → Next
Dann bei „Select root directory“ `/home/egr/workspace` auswählen.
Select All → Finish

Die Aufgaben des Praktikums sind dort nach Termin sortiert in einzelne Ordner gruppiert. In jedem dieser Ordner finden Sie den Ordner „src“ (source), der alle Dateien beinhaltet, die Sie zur Bearbeitung der Aufgaben benötigen. In den meisten dieser Dateien ist bereits Code vorhanden, der teilweise nur noch ergänzt werden muss. Erstellen oder verändern Sie keine anderen Dateien.

Die Dateien, die den C-Quellcode enthalten, haben die Endung „.c“. Öffnen Sie die Datei „01_Beispielprogramm.c“ (siehe Abbildung 9).

```

1 | 
2 | * Beispiel EGR
3| 
4| #include "delay_ms.h"
5| #include <hw_types.h> //einbinden des HWREG Makros
6| #include <uartstdio.h>
7| //#define HWREG(x)    (*((volatile unsigned int *) (x)))
8| 
9| int main() {
10|     UARTprintf("Hello World! Welcome to EGR Praktikum! \r\n  Sizeof(int): %d", sizeof(int));
11|     //Vorbereiten der pins
12|     HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
13|     HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
14| 
15|     HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
16|     HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
17| 
18|     HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
19|     HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
20| 
21|     HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
22| 
23|     HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
24|     HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
25| 
26|     HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
27| 
28|     HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
29|     HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
30| 
31|     HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
32|     HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
33| 
34|     HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
35|     HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
36| 
37|     HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
38| 
39|     HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
40|     HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
41| 
42|     HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
43|     HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
44| 
45|     HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
46|     HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
47| 
48|     HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
49| 
50|     HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
51|     HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
52| 
53|     HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
54|     HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
55| 
56|     HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
57|     HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
58| 
59|     HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
60| 
61|     HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
62|     HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
63| 
64|     HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
65|     HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
66| 
67|     HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
68|     HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
69| 
70|     HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
71| 
72|     HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
73|     HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
74| 
75|     HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
76|     HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
77| 
78|     HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
79|     HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
80| 
81|     HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
82| 
83|     HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
84|     HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
85| 
86|     HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
87|     HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
88| 
89|     HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
90|     HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
91| 
92|     HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
93| 
94|     HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
95|     HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
96| 
97|     HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
98|     HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
99| 
100|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
101|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
102| 
103|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
104| 
105|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
106|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
107| 
108|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
109|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
110| 
111|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
112|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
113| 
114|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
115| 
116|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
117|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
118| 
119|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
120|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
121| 
122|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
123|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
124| 
125|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
126| 
127|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
128|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
129| 
130|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
131|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
132| 
133|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
134|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
135| 
136|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
137| 
138|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
139|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
140| 
141|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
142|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
143| 
144|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
145|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
146| 
147|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
148| 
149|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
150|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
151| 
152|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
153|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
154| 
155|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
156|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
157| 
158|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
159| 
160|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
161|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
162| 
163|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
164|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
165| 
166|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
167|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
168| 
169|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
170| 
171|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
172|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
173| 
174|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
175|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
176| 
177|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
178|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
179| 
180|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
181| 
182|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
183|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
184| 
185|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
186|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
187| 
188|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
189|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
190| 
191|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
192| 
193|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
194|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
195| 
196|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
197|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
198| 
199|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
200|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
201| 
202|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
203| 
204|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
205|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
206| 
207|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
208|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
209| 
210|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
211|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
212| 
213|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
214| 
215|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
216|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
217| 
218|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
219|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
220| 
221|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
222|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
223| 
224|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
225| 
226|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
227|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
228| 
229|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
230|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
231| 
232|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
233|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
234| 
235|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
236| 
237|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
238|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
239| 
240|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
241|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
242| 
243|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
244|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
245| 
246|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
247| 
248|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
249|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
250| 
251|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
252|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
253| 
254|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
255|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
256| 
257|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
258| 
259|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
260|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
261| 
262|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
263|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
264| 
265|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
266|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
267| 
268|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
269| 
270|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
271|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
272| 
273|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
274|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
275| 
276|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
277|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
278| 
279|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
280| 
281|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
282|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
283| 
284|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
285|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
286| 
287|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
288|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
289| 
290|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
291| 
292|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
293|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
294| 
295|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
296|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
297| 
298|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
299|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
300| 
301|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
302| 
303|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
304|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
305| 
306|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
307|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
308| 
309|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
310|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
311| 
312|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
313| 
314|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
315|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
316| 
317|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
318|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
319| 
320|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
321|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
322| 
323|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
324| 
325|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
326|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
327| 
328|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
329|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
330| 
331|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
332|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
333| 
334|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
335| 
336|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
337|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
338| 
339|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
340|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
341| 
342|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
343|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
344| 
345|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
346| 
347|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
348|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
349| 
350|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
351|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
352| 
353|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
354|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
355| 
356|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen des vorherigen Modes
357| 
358|    HWREG(0x44e10000+0x854) &=~ (0x7); //löschen des vorherigen Modes
359|    HWREG(0x44e10000+0x854) |= 0x7; //setzen des neuen Modes
360| 
361|    HWREG(0x44e10000+0x858) &=~ (0x7); //löschen des vorherigen Modes
362|    HWREG(0x44e10000+0x858) |= 0x7; //setzen des neuen Modes
363| 
364|    HWREG(0x44e10000+0x85C) &=~ (0x7); //löschen des vorherigen Modes
365|    HWREG(0x44e10000+0x85C) |= 0x7; //setzen des neuen Modes
366| 
367|    HWREG(0x44e10000+0x860) &=~ (0x7); //löschen
```

Die jeweilige Hauptdatei des Programms beinhaltet immer die Main-Funktion. Da diese eine Funktion des Rückgabetyps Integer ist, endet sie mit der Zeile:

```
return 0;
```

Sobald der Programmablauf den Return-Befehl erreicht, wird die Funktion und damit das Programm beendet. Damit das nicht passiert und wir unseren Code in einer Dauerschleife ausführen können, befindet sich vor dem Return-Befehl eine While-Schleife, die immer wahr ist:

```
while(1)
{
    /*Place main loop code here */
}
```

Code, der dauerhaft ausgeführt werden soll, wird innerhalb der While-Schleife platziert. So können Sie beispielsweise zyklisch den Zustand eines Sensors auslesen und die Ausgänge beschalten.

Alles, was in Ihrem Code nur einmalig ausgeführt werden soll, wie beispielsweise das Initialisieren von Variablen oder die Konfiguration der Pins (Pin-Muxing), wird vor der While-Schleife eingefügt.

Sie können auch weitere Funktionen erstellen, die von der Main-Funktion aufgerufen werden. Achten Sie allerdings darauf, dass Sie keine Funktionen innerhalb anderer Funktionen erstellen können.

Dateien, die als Bibliothek eingebunden werden können (sog. Header), haben die Endung „.h“. Solche Header werden wir verwenden, um uns eigene Hardware-Treiber zu schreiben. Dort können Variablen, Konstanten und Funktionen definiert werden, die im Hauptprogramm aufgerufen werden, wodurch dieses strukturierter und deutlich besser lesbar wird.

Um eine Bibliothek verwenden zu können, müssen Sie diese am Anfang des Programms einbinden. Dies geschieht über den Include-Befehl. Beispielsweise ist die Bibliothek zur Verwendung von Funktionen zum direkten Beschreiben von Registern des Mikrocontrollers bereits eingebunden:

```
#include <hw_types.h>
```

Header, die im selben Verzeichnis liegen, wie die Hauptdatei, werden mit Anführungszeichen eingebunden, damit der Präprozessor die Datei an der richtigen Stelle sucht:

```
#include "delay_ms.h"
```

Weitere Informationen zur Erstellung und Verwendung von Headern erfahren Sie in Kapitel 1.4.

Wenn Sie Ihr Programm ausführen wollen, müssen Sie es zunächst kompilieren. Klicken Sie dazu auf das Hammer-Symbol in der Symbolleiste des Editors. Je nach Größe des Programms und Leistungsfähigkeit Ihres Rechners kann der Kompiliervorgang mehrere Sekunden in Anspruch nehmen.

Wenn während des Kompilierens keine Syntaxfehler auftreten, wird Ihnen die erfolgreiche Kompilierung in der Ausgabekonsole des Editors angezeigt („Build Finished“). Die kompilierte Programmdatei finden Sie jetzt in „bin/Debug/EGR_run.bin“ und kann nun auf das BeagleBone geladen werden.

Stecken Sie das BeagleBone an Ihren Rechner an. Verwenden Sie dazu unbedingt die USB-Buchse auf dem EGR-Cape. Wenn das BeagleBone eingeschaltet, resetet und vollständig gebootet ist (Links neben der Netzwerkbuchse sollten zwei blaue LEDs leuchten), können Sie das Programm hochladen.

Öffnen Sie das Terminal und geben Sie den Befehl: *BBBsend workspace/01_Beispielprogramm/bin/Debug/EGR_run.bin* ein (siehe Abbildung 8).

Wenn alles funktioniert hat, sollte das Programm direkt starten und auf dem BeagleBone blaue LEDs blinken.



Verständnisfragen

1. Ist der AM3359 ein Teil des BeagleBone oder das BeagleBone Teil des AM3359?
2. Welches Modul bestimmt die Funktion (den Mode) eines Pins?
3. Was ist eine HardwareAdresse?
4. Was ist ein Register?
5. Erklären sie den Unterschied zwischen Offset- und BasisAdresse?
6. Wo finden sie die BasisAdresse eines Moduls?
7. Wie setzen Sie die Bits 12,14 und 23 in einem 32Bit-Register?
8. Wie lege ich fest, welches Programm beim Booten ausgeführt wird?



Termin 1

- Skript:
Grundlagen der Ein- und Ausgabe

1. Termin: Grundlagen der Ein- und Ausgabe

1.1 GPIO - Pins

Als **General Purpose In- and Output-Pins** bezeichnet man Pins, die vom Mikrocontroller digital geschaltet und digital ausgelesen werden können. Bei unserem Mikrocontroller bedeutet das, dass sie mit 3,3V (high) oder 0V (low) beschaltet werden können, oder erkennen können, ob eine Spannung anliegt oder nicht (Ausgabe nur 1 oder 0, nicht die Höhe der Spannung).

GPIOs können einfach verwendet werden und sind vielseitig einsetzbar. Die meisten Funktionen die wir in den nächsten Terminen per Hardware implementieren werden, könnten auch per Software über GPIOs umgesetzt werden.

Um die GPIO-Pins im Praktikum verwenden zu können, sind einige davon an den drei Steckplätzen/Ports des Capes herausgeführt, an die die Hardwaremodule angesteckt werden können. Die Pinbelegung der Steckplätze finden Sie im Datenblatt des Capes (Kapitel 6 des Skripts).

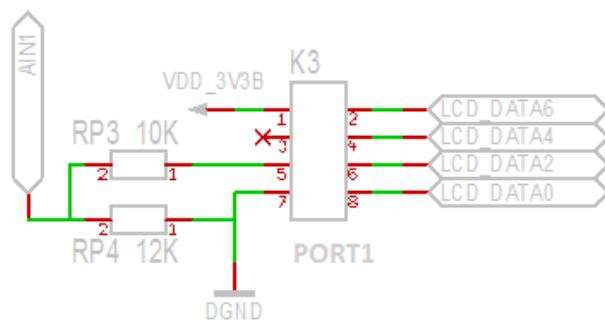


Abbildung 10: Pinbelegung von Steckplatz 1. Die GPIO-Pins (LCD_DATA) liegen an den Pins 2,4,6 und 8.

1.2 Eine LED einschalten

Die erste Aufgabe des Praktikums ist, eine LED leuchten zu lassen (Alle Aufgaben finden Sie am Ende der jeweiligen Kapitel). Stecken Sie dazu das LED-Modul an Steckplatz 1 des Capes. Der Aufbau des Moduls kann aus dem Moduldatenblatt entnommen werden (Kapitel 7).

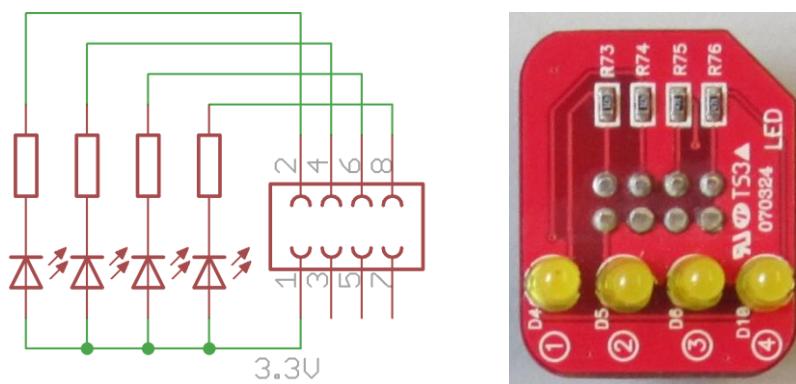


Abbildung 11: Links: Auszug aus dem Moduldatenblatt. Rechts: Das LED-Modul

Die LED 1 leuchtet, wenn Strom von Pin 1 nach Pin 2 fließt. In Verbindung mit dem Datenblatt des BeagleBoneCapes können wir erkennen, dass Pin 1 an VCC (3,3 V Spannungsversorgung) angeschlossen ist. Die LED leuchtet also, wenn wir Pin 2 auf 0 V schalten, sodass Strom von Pin 1 nach Pin 2 fließen kann. Wenn wir an Pin 2 3,3 V Spannung anlegen, ist die LED aus.

Um Pin 2 steuern zu können, müssen wir also Pin 2 von Steckplatz 1 als digitalen Output konfigurieren und auf 0V schalten. Dieser Pin ist mit dem GPIO-Pin mit dem Namen **LCD_DATA6** verbunden (siehe Abb. 10).

Um einen GPIO-Pin anzusteuern und damit eine LED einzuschalten, sind drei Schritte notwendig, die in den nachfolgenden Unterkapiteln erklärt werden: Pin-Muxing, Pin als Output konfigurieren, Pin einschalten.

Machen Sie sich mit der virtuellen Maschine und der Entwicklungsumgebung vertraut (Kapitel 0.9). Öffnen Sie in Eclipse die Quelldatei zur Bearbeitung von Aufgabe 1: *T1-A1.c*

1.2.1 Pin-Muxing

Die Ausgänge des verbauten ARM Mikrocontrollers sind Kugeln (Balls/Pins) auf der Unterseite des Prozessors. Fast jeder dieser Ausgänge kann in verschiedenen Modi betrieben werden. Das Einstellen eines Modus an einem Pin wird Pin-Muxing genannt. Im Datenblatt des Capes finden Sie eine Tabelle, in der jeder Pin der Steckplätze zusammen mit seinem Namen und allen einstellbaren Modi aufgelistet ist. Wir möchten den Pin als GPIO betreiben (Modus7).

Modul steckpl	Pin	PIN Name	SIGNAL NAME							
			MODE							
0	1	2	3	4	5	6	7			
Port1	2	LCD_DA TA6	Lcd_data6 c6	Gpmc_a _data_in c	Pr1_edio _index	eQEP2_i	Pr1_edio_	Pr1_pru1_p ru_r30_6	Pr1_pru1_p ru_r31_6	gpio2_12
Port1	4	LCD_DA TA4	Lcd_data4 TA4	Gpmc_a 4	Pr1_mii0 txd1	eQEP2A _in		Pr1_pru1_p ru_r31_4	Pr1_pru1_p ru_r31_4	gpio2_10
Port1	6	LCD_DA ---	Lcd_data2 ---	gpmc_a2		ehrpwm				gpio2_8

Abbildung 12: Auszug aus dem Datenblatt des Capes

Die Funktion eines Pins wird im sogenannten **Control-Module** festgelegt. Darin gibt es für jeden Pin ein **Konfigurationsregister**.

Table 9-10. CONTROL_MODULE REGISTERS (continued)			Table 2-2. L4_WKUP Peripheral Memory Map (continued)			
Offset	Acronym	Register Description	Region Name	Start Address (hex)	End Address (hex)	Size
878h	conf_gpmc_ben1		DMTIMER0	0x44E0_5000	0x44E0_5FFF	4KB
87Ch	conf_gpmc_csn0			0x44E0_6000	0x44E0_6FFF	4KB
880h	conf_gpmc_csn1		GPIO0	0x44E0_7000	0x44E0_7FFF	4KB
884h	conf_gpmc_csn2			0x44E0_8000	0x44E0_8FFF	4KB
888h	conf_gpmc_csn3		UART0	0x44E0_9000	0x44E0_9FFF	4KB
88Ch	conf_gpmc_csn4			0x44E0_A000	0x44E0_AFFF	4KB
890h	conf_gpmc_advn_ale		I2C0	0x44E0_B000	0x44E0_BFFF	4KB
894h	conf_gpmc_oen_ren			0x44E0_C000	0x44E0_CFFF	4KB
898h	conf_gpmc_wen		ADC_TSC	0x44E0_D000	0x44E0_EFFF	8KB
89Ch	conf_gpmc_ben0_cle			0x44E0_F000	0x44E0_FFFF	4KB
8A0h	conf_lod_data0		Control Module	0x44E1_0000	0x44E1_1FFF	128KB
8A4h	conf_lod_data1			0x44E1_2000	0x44E1_2FFF	4KB
8A8h	conf_lod_data2		DOR2IMODR PHY	0x44E1_3000	0x44E1_3FFF	4KB
8ACh	conf_lod_data3		Reserved	0x44E1_4000	0x44E1_4FFF	4KB
8B0h	conf_lod_data4		DMTIMER1_1MS (Accurate Trig timer)	0x44E1_5000	0x44E1_5FFF	4KB
8B4h	conf_lod_data5			0x44E1_6000	0x44E1_6FFF	4KB
8B8h	conf_lod_data6			0x44E1_7000	0x44E1_7FFF	4KB
8BCh	conf_lod_data7			0x44E1_8000	0x44E1_8FFF	4KB
8C0h	conf_lod_data8			0x44E1_9000	0x44E1_9FFF	4KB
8C4h	conf_lod_data9			0x44E1_A000	0x44E1_AFFF	4KB
8C8h	conf_lod_data10			0x44E1_B000	0x44E1_BFFF	4KB
8CCh	conf_lod_data11			0x44E1_C000	0x44E1_CFFF	4KB
8D0h	conf_lod_data12			0x44E1_D000	0x44E1_DFFF	12KB
8D4h	conf_lod_data13			0x44E1_E000	0x44E1_FFFF	4KB
8D8h	conf_lod_data14			0x44E3_F000	0x44E3_FFFF	4KB
8DCh	conf_lod_data15			0x44E4_0000	0x44E4_1FFF	256KB
8E0h	conf_lod_wsync			0x44E8_0000	0x44E8_1FFF	4KB
8E4h	conf_lod_hsync			0x44E8_2000	0x44E8_3FFF	4KB
8E8h	conf_lod_pk1			0x44E8_4000	0x44E8_5FFF	50KB
8ECh	conf_lod_ac_bias_en			0x44E8_6000	0x44E8_7FFF	1MB
8F0h	conf_mmco_dat3			0x44F0_0000	0x44F0_1FFF	4KB

Abbildung 13: Offsets der Pin Konfigurationsregister (links) und die Basisadresse des Control-Moduls (rechts)

In der Dokumentation des Prozessors (TI-TRM oder Kapitel 9 des Skripts) können die Adressen der Konfigurationsregister für jeden Ball/Pin nachgelesen werden. In unserem Fall benötigen wir die Adresse des Registers **conf_lcd_data6**: 0x8b8. Die Adressen sind immer nur als Offset angegeben, da diese Register Teil des Control-Modules sind. D.h. dass zur Programmierung die Basisadresse des Control-Modules zum Offset addiert werden muss um die vollständige Speicheradresse der Register zu erhalten. Die Basisadressen können in der Memory Map (Kapitel 8 des Skripts) herausgesucht werden. Für das Control-Modul ist sie 0x44E10000.

Der Aufbau eines Konfigurationsregisters ist ebenfalls in der TI-TRM oder im Ausschnitt zu finden (Abb. 14).

Figure 9-54. conf_<module>_<pin> Register

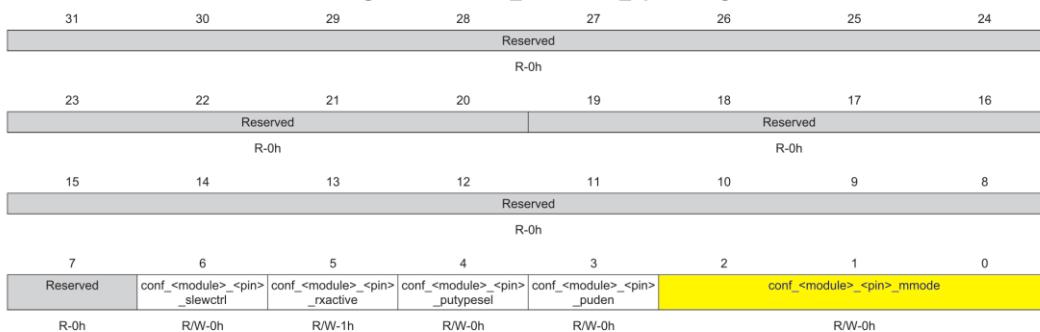


Table 9-61. conf_<module>_<pin> Register Field Descriptions

Bit	Field	Type	Reset	Description
31-20	Reserved	R	0h	
19-7	Reserved	R	0h	
6	conf_<module>_<pin>_slewctrl	R/W	X	Select between faster or slower slew rate 0: Fast 1: Slow Reset value is pad-dependent.
5	conf_<module>_<pin>_rxactive	R/W	1h	Input enable value for the PAD 0: Receiver disabled 1: Receiver enabled
4	conf_<module>_<pin>_pu typesel	R/W	X	Pad pullup/pulldown type selection 0: Pulldown selected 1: Pullup selected Reset value is pad-dependent.
3	conf_<module>_<pin>_pu den	R/W	X	Pad pullup/pulldown enable 0: Pullup/pulldown enabled 1: Pullup/pulldown disabled Reset value is pad-dependent.
2-0	conf_<module>_<pin>_m mode	R/W	X	Pad functional signal mux select. Reset value is pad-dependent.

Abbildung 14: Auszug aus der TI-TRM: Aufbau eines Konfigurationsregisters

Die Register des Mikrocontrollers sind 32bit groß. Im Konfigurationsregister sind die Bits 31 bis 7 reserviert und nicht beschreibbar. Die restlichen Bits können zur Konfiguration des Pins beschrieben werden. Der Modus, in dem der Pin betrieben werden soll, wird auf die Bits 2 bis 0 geschrieben. Jedem Pin kann also einer von maximal $2^3 = 8$ unterschiedlichen Modi zugewiesen werden.

Da wir nun wissen, dass wir im Control-Module in das Konfigurationsregister **conf_lcd_data6** auf die Bits 0 bis 2 eine 7 (GPIO-Modus) schreiben müssen, können wir mit der Programmierung beginnen.

Um direkt auf Register zugreifen zu können verwenden wir das Makro **HWREG**, das in der Datei **hw_types.h** definiert ist:

```
//Einbinden der benötigten Header
#include <hw_types.h>

//Pinmuxing
HWREG(0x44e10000+0xb8) &=~ ((1<<0)|(1<<1)|(1<<2)); //löschen des vorherigen Modes
HWREG(0x44e10000+0xb8) |= 0x7; //setzen des neuen Modes
```

1.2.2 Pin als Output konfigurieren

Da der Pin nun im GPIO-Modus gemuxt ist, ist er mit dem GPIO-Funktionsmodul verbunden, das die Register für alle GPIO-Funktionalitäten bereitstellt. Weil der AM3359 mehr als 32 GPIO-Pins besitzt, hat er vier GPIO-Module (GPIO0 - GPIO3). Sie besitzen alle den gleichen Registeraufbau, haben aber alle unterschiedliche Basisadressen.

Im Datenblatt des Capes ist der Signal Name unseres Pins in Mode 7 **gpio2_12** (siehe Abb. 12). Dies bedeutet es ist Pin Nr. 12 des GPIO-Moduls 2. Die Register eines GPIO-Moduls und deren Offsets sind in der TI-TRM bzw. in Kapitel 10 des Skripts beschrieben.

Eines dieser Register ist das Output-Enable (GPIO_OE) Register, in dem eingestellt werden kann, ob ein Pin als Output (0), oder als Input (1) betrieben wird. Das heißt, im Register GPIO_OE (Offset 0x134) muss Bit Nr. 12 auf 0 gesetzt werden um den Pin als Output zu konfigurieren:

```
//LED Pin als Output schalten
HWREG(0x481ac000 + 0x134) &= ~(1 << 12);
```

1.2.3 Pin einschalten

Unser Pin ist nun als Output konfiguriert und kann über das Register GPIO_DATAOUT (Offset 0x13C) ein- und ausgeschaltet werden. Um unseren Pin spannungsfrei zu schalten, muss Bit 12 auf 0 gesetzt werden:

```
//LED anschalten (Pin abschalten)
HWREG(0x481ac000 + 0x13c) &= ~(1 << 12);
```

Das erste Programm ist nun vollständig und kann kompiliert und ausgeführt werden (siehe Kapitel 0.9).

1.3 Einen Taster einlesen

Um einen Taster einlesen zu können, muss dieser an einen GPIO-Pin angeschlossen werden, der als Input konfiguriert ist. Der Zustand eines Pins, der als Eingang verwendet wird, kann aus dem GPIO_DATAIN Register gelesen werden und entweder eine logische "0" oder eine logische "1" annehmen. Je nach Mikrocontroller werden unterschiedliche Spannungsbereiche als "0" und "1" definiert (z.B. $[0 - 0,8V] \triangleq "0"$ und $[2,0 - 3,3V] \triangleq "1"$).

Bei Tastern und Schaltern tritt jedoch eine typische Problematik auf: Ist der Taster geöffnet, liegt am Input-Pin kein äußeres Signal an. Der Zustand der „offenen“ Leitung ist undefiniert und man weiß nicht, ob der Mikrocontroller eine 1 oder 0 detektiert (offene Leitung kann z.B. Antennenwirkung haben und Fremdsignale einfangen).

Um diesen undefinierten Zustand zu vermeiden, verwendet man sogenannte **Pull-Widerstände**. Diese ziehen die Leitung über einen hochohmigen Widerstand auf ein definiertes Potential (typischerweise $2 \dots 10k\Omega$ für externe Widerstände bzw. bis zu $100k\Omega$ für μ C-interne). Es werden hochohmige Widerstände verwendet, damit einerseits zwar der Potentialbezug hergestellt wird, gleichzeitig aber kein zu hoher Spannungsabfall durch den abfließenden Strom entsteht.

Ist der Taster gegen Ground verschaltet, muss man einen **Pull-Up-Widerstand** verwenden; ist er gegen das High-Potential geschaltet, einen **Pull-Down-Widerstand**.

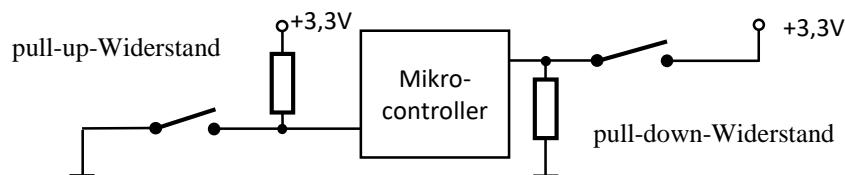


Abbildung 15: Verwendung von Pull-Widerständen für das Einlesen von Tastern

Am Beaglebone kann für jeden Pin ein im Chip verbauter Pull-Widerstand eingestellt werden (Bits 3 und 4 des Konfigurationsregisters).

1.4 GPIO Treiber

Das manuelle Recherchieren der einzelnen Registeradressen anhand der Datenblätter wird sehr schnell mühsam und der geschriebene Code wird noch dazu sehr unleserlich. Außerdem muss man denselben Code in seinen Programmen immer wieder verwenden. Darum verwendet man Treiber, um seine Hardware komfortabel anzusteuern ohne die einzelnen Adressen jedes Mal von Hand eingeben zu müssen. Treiber enthalten vorgefertigte Funktionen, denen man zum Beispiel nur mitteilen muss „setze Pin2 auf High“. In Termin 1 besteht nun eine Aufgabe darin einen solchen Treiber für die GPIO-Pins zu schreiben.

Für eine bessere Wiederverwendbarkeit und eine schönere Strukturierung verwendet man für Treiber eigene Dateien. Um neue Header- und Sourcecode-Dateien anzulegen reicht ein Rechtsklick auf das entsprechende Projekt → new → Header-File od. Source-File. In einem neuen Projekt muss man dann die jeweilige Header-Treiberdatei mittels

```
#include "abc.h"
```

Anweisung in seine main.c (bzw. Datei die die main()-Funktion enthält) einbinden, um den Treiber zu verwenden. Es ist nicht zu vergessen, selbstgeschriebene Funktionen in den Source-Files auch in den zugehörigen Header-Dateien zu deklarieren, um sie später verwenden zu können.

Um eine doppelte Einbindung eines Treibers und daraus resultierende Probleme bei der Komplierung des Programms zu vermeiden, beginnt jede Headerdatei mit

```
#ifndef ABC_H
#define ABC_H
```

und endet mit

```
#endif
```

Wenn man eigene Treiber erstellt, ist es wichtig seine Funktionalität sauber abzugrenzen. Es soll einen Treiber geben, der die Funktionen des GPIO-Moduls abbildet (GPIO.c). Dieser soll aber z.B. nichts mit dem Pin-Muxing zu tun haben. Das Pin-Muxing geschieht im zentralen Control-Modul für den gesamten Prozessor und soll daher einen eigenen Treiber erhalten (Control_mod.c). Beides hat wiederum nichts mit dem EGR-Cape zu tun, das auf dem Beaglebone steckt. Daher wird alles Cape-spezifische – z.B. #defines für die einzelnen Steckplätze – wiederum in eigene Datei ausgegliedert (EGR-Cape.h).

**Randnotiz:**

Es existieren zwei Formen der #include-Anweisung:

```
#include <Dateiname>
```

```
#include "Dateiname"
```

Wenn die Variante mit den Anführungszeichen gewählt wird, sucht der Präprozessor die angegebene Datei zuerst im Verzeichnis der Quelldatei. Kann er die betreffende Datei dort nicht finden, dann durchsucht er anschließend meist (aber nicht zwingend) all jene Verzeichnisse, die er normalerweise bei der #include-Anweisung mit spitzen Klammern berücksichtigt. Anführungszeichen um den Dateinamen finden im Allgemeinen dann Verwendung, wenn Header-Dateien einzufügen sind, die man selbst erstellt hat.

Hingegen sollten die Dateinamen in spitze Klammern gesetzt werden, wenn Sie sich auf Header-Dateien der Standardbibliothek beziehen. In diesem Fall erfolgt die Suche in extra angegebenen include-Verzeichnissen. Die Liste der Verzeichnisse, die dabei berücksichtigt werden, muss entweder über ein Settings-Menü bekanntgegeben werden oder in Form einer Umgebungsvariablen verfügbar sein.

Kochrezept – Termin 1

Schlagwörter: Pins Muxen, Funktionen von Pins konfigurieren

Im Control Module:

- Zusammenhänge der Schaltung erkennen
→ anzusprechende Pins über Schaltpläne identifizieren
- Modes (also die Funktion) der angeschlossenen Pins festlegen/muxen, d.h. den Pin mit einem Funktionsmodul (z.B. GPIO-Modul) verbinden
→ Control-Module als Basisadresse & entsprechende Pin-Adresse als Offset-Adresse
- Pull-Widerstände (pull-up/down - enable/disable) einstellen
→ Control-Module als Basisadresse & entsprechende Pin-Adresse als Offset-Adresse

Im GPIO-Modul:

- Nach dem Pinmuxing auf Mode 7 (GPIO), das entsprechende GPIO Modul konfigurieren
→ Basis-Adresse des dem Pin zugeordnete GPIO-Moduls suchen
→ Die zugehörigen Einstellungen in den Registern als Offset-Adresse

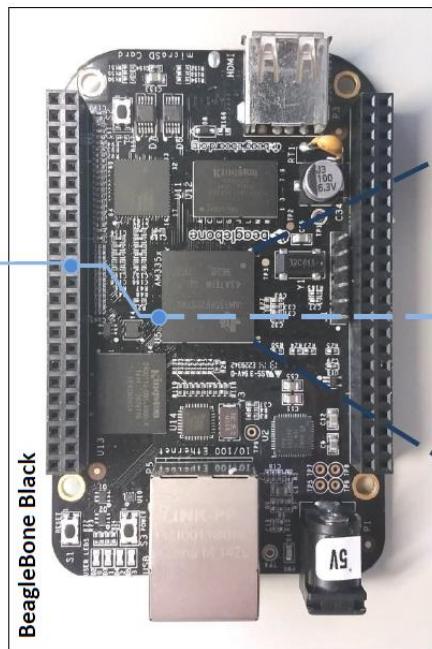
Verständnisfragen

1. Wie viele GPIO-Module hat der AM3359 und wie viele Digitale Pins kann man pro GPIO-Modul schalten?
2. Was bewirkt das Output_Enable Register?
3. Was ist der Unterschied zwischen dem Output_Enable und dem Data_Out Register?
4. Nennen Sie die drei notwendigen Schritte um eine LED anzuschalten?
5. Wofür brauche ich Pull-up und Pull-Down Widerstände?
6. Wozu benötigt man Hardwaretreiber?

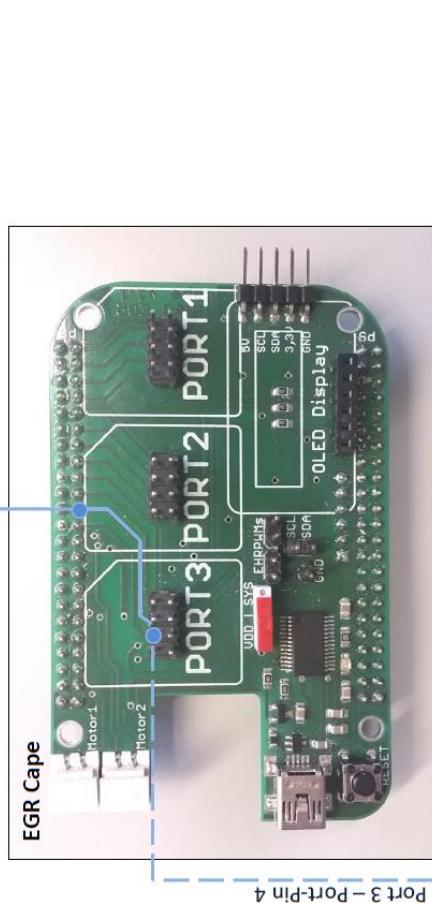
**HWREG(0x44E10000 + 0x800) |= ((1<<0) | (1<<1) | (1<<2));
HWREG(0x4804C000 + 0x134) &= ~ (1<<0);**

Übersicht: Konfiguration eines Pins als GPIO-Output

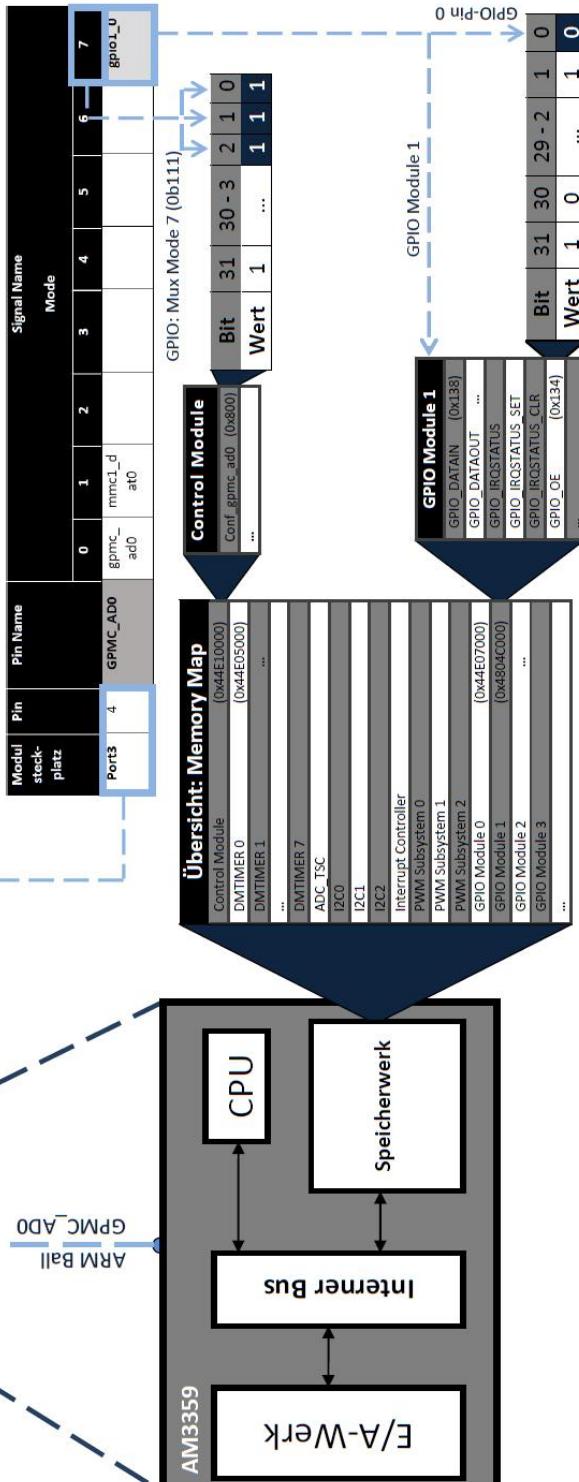
physische Schnittstelle



EGR Cape



**HWREG(0x44E10000 + 0x800) |= ((1<<0) | (1<<1) | (1<<2));
HWREG(0x4804C000 + 0x134) &= ~ (1<<0);**



Termin 1

- Aufgaben

1.5 Aufgaben Termin 1: Grundlagen der Ein- und Ausgabe

Aufgabe 1: Anschalten einer LED (wird vorgestellt)

Stecken Sie das LED-Modul in Steckplatz 1 und folgen Sie dem Skript zum Einschalten der ersten LED.

Inhalt: PinMUXen, Bitschiften GPIO

Aufgabe 2: Auslesen von Tastern

Nutzen Sie Steckplatz 1 zum Auslesen des Taster Moduls. Bei Betätigung von Taster 1 soll LED 1 des LED Moduls (an Steckplatz 2) leuchten. Bei Taster 2 soll LED 2 leuchten usw.

Tipp: Schalten sie an dem Taster-Pin die Pullups ein.

Inhalt: selbstständiges Lesen und Verstehen von Datenblättern.

Aufgabe 3: EGR Treiber

Vervollständigen Sie die für Sie bereitgestellten Treiberdateien EGR_Cape.h und GPIO.c.

Tipp 1: Orientieren sie sich für den GPIO.c Code an der Lösung von Aufgabe 2 um Fehler zu vermeiden.

Tipp 2: In der GPIO.c müssen sie die Funktionen des GPIO-Moduls ergänzen.

In der EGR_Cape.h müssen sie die Basisadressen der vom Cape verwendeten GPIO-Module und die Nummern der verwendeten Pins für die Steckplätze 2 und 3 ergänzen.

Die Treiberdateien Conf_mod.h und Conf_mod.c sind bereits fertig geschrieben und können zum Muxen der Pins und zur Einstellung des Pull-ups verwendet werden.

Inhalt: Umsetzen der Informationen aus den Datenblättern in wiederverwendbaren Code.

Aufgabe 4. Lauflicht

Schließen sie das LED-Modul am Steckplatz 2 an. Schreiben Sie – **unter Verwendung der zuvor angefertigten Treiber** – ein Programm, bei dem jede LED nacheinander eine halbe Sekunde leuchtet. Erweitern Sie das Programm um das Tastenmodul auf Steckplatz 1. Eine Taste soll das Lauflicht ein- und ausschalten. Die zweite Taste soll die Geschwindigkeit des Lauflichts erhöhen, solange die Taste gedrückt ist.

Tipp: Binden Sie die Datei delay_ms.h ein.

Inhalt: Delay, Anwenden der Treiber

Zusatzaufgabe 5. Notaus (Nicht notenrelevant)

Ändern Sie das Programm aus Aufgabe 4 so ab, dass das Lauflicht nach einmaligem Drücken einer Stopp-Taste augenblicklich stoppt und bei der zuletzt bekannten Position stehen bleibt.

Inhalt: Polling





Termin 2

- Skript:
Antriebe und Echtzeit

2. Termin: Antriebe und Echtzeit

2.1 Einleitung

Im ersten Termin des Praktikums haben wir gelernt, wie wir die Adressen zu bestimmten Registern einzeln suchen können, um daraus einen Treiber für die GPIO-Pins zu schreiben. Im weiteren Verlauf des Praktikums soll zusätzlich auf Bibliotheken von TI zurückgegriffen werden. In der Header Datei `soc_AM335x.h` sind z.B. alle Basisadressen der benötigten Register definiert (Nomenklatur: `SOC_Modulkürzel_Modulnummer_REGS`). So kann für die Basisadresse von GPIO1 zukünftig das Makro `soc_GPIO_1_REGS` verwendet werden. In seinem Projekt kann man Header-Dateien einfach im Code einbinden:

```
// Einbinden des Headerfiles für die Basisadressen
#include <soc_AM335x.h>      (Tipp: → Öffnen der Header-Datei über Strg+Mausclick oder F3)
```

Auch die Offset-Adressen der Modul-Register müssen nicht jedes Mal per Hand identifiziert werden. In den Includes im Projekt gibt es den Pfad [...]/include/hw. In diesem Pfad sind die Bibliotheken für die benötigten Module enthalten, wie z.B. `hw_gpio_v2.h` für die GPIOs, oder die `hw_dmtimer.h` für die in diesem Termin benötigten Timer.

```
// Einbinden der Headerfiles für GPIO-Modul und DMTimer-Modul
#include <hw_gpio_v2.h>
#include <hw_dmtimer.h>
```

2.2 Interrupts

Normalerweise arbeitet ein Prozessor ein Programm der Reihe nach ab. Mit Hilfe von Interrupts gibt es aber die Möglichkeit unabhängig vom normalen Programmfluss auf einige Ereignisse zu reagieren. Typische Interruptquellen sind der Überlauf eines Timers, das Ende einer AD Wandlung und ein empfangenes Zeichen an einer Schnittstelle.

Wenn ein Interrupt Signal vorliegt, pausiert das normale Programm, und die ISR (Interrupt Service Routine) wird stattdessen aufgerufen. Erst danach wird das normale Programm fortgesetzt. Mit den Interrupts kann die CPU mehrere Dinge quasi gleichzeitig erledigen. In der ISR (je nach CPU auch direkt durch die Hardware), wird der Zustand der Register gesichert und am Ende der ISR wieder hergestellt. Das Hauptprogramm bemerkt so wenig von der ISR, außer der unvermeidlichen Verzögerung und den gewünschten Effekten in der ISR.

2.2.1 Unterschiede zum Polling

Alternativ zur Verwendung von Interrupts lässt sich das Auftreten eines Ereignisses auch durch das regelmäßige Abfragen in der Programmschleife erkennen und abfangen. Diese Technik nennt sich Polling. Ein großer Nachteil beim Polling ist, dass man entweder auf eine Zustandsänderung wartet und in diesem Fall keine anderen Aufgaben erledigen kann, oder man führt andere Aufgaben aus und verpasst dann evtl. eine Zustandsänderung. Bei Interrupts wird sofort auf eine Zustandsänderung reagiert und danach gleich mit den anderen Aufgaben weitergemacht.

2.2.2 Interrupts beim BeagleBone

Die Funktionen, die für die Verwendung von Interrupts gebraucht werden, sind in der Bibliothek `interrupt.h` definiert. Diese Header-file muss daher zu Beginn eingebunden werden.

In der `main()` muss dem Prozessor zunächst mitgeteilt werden, dass man vor hat, überhaupt Interrupts in seinem Programm zu verwenden. Dies wird von der Funktion `IntMasterIRQEnable()` übernommen.

```
// Global enable Interrupt requests
    IntMasterIRQEnable();
```

Mit der Funktion `IntAINTCInit()` wird der Interrupt Controller des Prozessors initialisiert.

```
// Initialize the ARM interrupt controller
    IntAINTCInit();
```

Nun muss definiert werden, auf welche Interruptquellen der Prozessor reagiert. In der Dokumentation Ti-TRM des Prozessors sind über 100 unterschiedliche Quellen angegeben. Mit der Funktion `IntSystemEnable()` werden einzelne Quellen aktiviert. Die Quellen sind wiederum in der `interrupt.h` definiert. Interrupts für GPIO2A werden wie folgt enabled:

```
// Enable the system interrupt
    IntSystemEnable(SYS_INT_GPIOINT2A);
```

Der nächste Schritt ist dem Prozessor mitzuteilen, was er tun soll, wenn ein Interrupt-Ereignis eintritt. Dazu wird eine beliebige, eigene Funktion auf die entsprechende Interruptquelle registriert. Im folgenden Beispiel ist das die Funktion `myGPIO2Isr()`. Üblicherweise bezeichnet man die auszuführende Funktion als Interrupt Service Routine (ISR). Tritt in diesem Beispiel nun ein GPIO2A-Interrupt auf, wird das Hauptprogramm unterbrochen und springt sofort in die Funktion `myGPIO2Isr()`. Grundsätzlich gilt, dass man **ISRs so kurz wie möglich** halten sollte, um schnellstmöglich zum Hauptprogramm zurückzuspringen.

```
// Registering myGPIO2Isr
    IntRegister(SYS_INT_GPIOINT2A, myGPIO2Isr);
```

Benötigt man in seinem Programm mehrere, verschiedene Interruptquellen, muss man jeder einzelnen eine bestimmte Priorität zuweisen. Bei gleichzeitigem Auftreten wird immer der Interrupt mit der höheren Priorität zuerst ausgeführt. (Verwendet man nur eine Interruptquelle, braucht man die Priorität nicht explizit setzen.)

```
// Set the priority */
    IntPrioritySet(SYS_INT_GPIOINT2A, 0, AINTC_HOSTINT_ROUTE_IRQ);
```

Mit diesen Funktionen ist der Prozessor nun informiert, dass er auf bestimmte Interrupts hören soll. Alle Module, die Interrupts auslösen sollen, müssen nun ebenfalls konfiguriert werden.

2.2.3 Interrupts des GPIO-Moduls

Um mit einem GPIO-Pin einen Interrupt auslösen zu können, muss man Interrupts aktivieren und festlegen, welcher Zustand oder welche Zustandsänderung einen Interrupt auslösen soll (Interrupt-Trigger).

Die GPIO-Module des ARM-Prozessors können je zwei verschiedene Interrupts auslösen, z.B. GPIO2A und GPIO2B. Welcher Pin nun welchen Interrupt auslöst, setzen wir in dem `GPIO_IRQSTATUS_SET(x)` Register. Soll Interrupt A ausgelöst werden, dann setzen wir an der entsprechenden Stelle in `GPIO_IRQSTATUS_SET(0)` eine 1. (*Anmerkung:* Leider ist hier das TI-ReferenceManual wieder etwas umständlich. Interrupt A gehört zu `GPIO_IRQSTATUS_SET(0)` während Interrupt B zu `GPIO_IRQSTATUS_SET(1)` gehört)

Zusätzlich müssen wir definieren, **bei welchen Events der Interrupt ausgelöst werden soll**. Dazu schreiben wir in die Register `GPIO_LEVELDETECT(0)`, `GPIO_LEVELDETECT(1)`, `GPIO_RISINGDETECT` oder `GPIO_FALLINGDETECT` an die entsprechende Stelle des Pins eine 1.

Wird ein Interrupt ausgelöst springt das Programm in die zuvor registrierte ISR. Dort muss **das auslösende Interruptflag unbedingt gelöscht werden**, um in der Lage zu sein, später erneut einen Interrupt auszulösen. Im Register `GPIO_IRQSTATUS(x)` ist an der Stelle des auslösenden Pins eine 1. Gelöscht wird das Flag, indem nun erneut eine 1 an diese Stelle des Registers geschrieben wird:

```
void myGPIO2Isr(){
    // eigener ISR Code, z.B. LED an oder ausschalten
    // Löschen des Interrupt-Flags
    HWREG(SOC_GPIO_2_REGS + GPIO_IRQSTATUS(0)) |= (1<< --Mein-Interrupt-Pin--);
```

2.3 Timer

In einem Prozessor sind Timer eines der Hauptarbeitspferde. Mit Hilfe von Timern wird es möglich zu definierten Zeitpunkten bestimmte Aktionen auszuführen, Interrupts auszulösen und vieles mehr.

2.3.1 Was ist ein Timer

Ein Timer ist ein bestimmtes Register im Prozessor, das völlig ohne Zutun des Programms, also per Hardware, im Hintergrund hochgezählt wird. Um mit dem Timer zu interagieren kann man zum Beispiel Interrupts auslösen, wenn der Timer-Wert im Register einen bestimmten Wert erreicht hat. Ein solches Ereignis nennt man Compare-Match.

Wenn der Timer-Wert größer wird als der maximal im Register speicherbare Wert, dann gibt es einen sogenannten Overflow und der Benutzer muss entscheiden, ob der Timer wieder von vorne zählen oder andere Aktionen durchführen soll. Auch bei einem Overflow lassen sich Interrupts auslösen.

2.3.2 Timer des BeagleBone

Die verwendeten Timer im ARM Prozessor werden mit DMTimer bezeichnet. Der Prozessor stellt 8 solcher Timer zur Verfügung (DMTimer0-7), wobei DMTimer0 und DMTimer1 ein paar Besonderheiten aufweisen, die in diesem Praktikum nicht benötigt werden. Die Basisadressen der DMTimer Register sind in der Header `soc_AM335x.h` definiert und immer mit `soc_DMTIMER_x_REGS` bezeichnet. Die Adress-Offsets für die einzelnen Register sind in der Header `hw_dmtimer.h` definiert.

```
// Control Register des DMTimer 2 beschreiben
HWREG (SOC_DMTIMER_2_REGS + DMTIMER_TCLR) |= (1<<...NUR EIN BEISPIEL...);
```

Taktquelle

Die Timer des Prozessors können unterschiedliche Clock-Signale als Taktgeber verwenden. Entweder eine interne 24MHz Clock, eine interne 32,768kHz Clock oder eine externen Taktquelle. Per Default ist die 24MHz Clock ausgewählt, die für unsere Zwecke gut geeignet ist und deshalb verwendet werden kann.

Timer Modi

Im Timer Control Register (**TCLR**) kann der Modus des Timers festgelegt werden. Per Default befindet sich der Timer im One-shot Modus. In diesem stoppt der Timer sobald das Register das erste Mal überläuft. Im Praktikum wird stattdessen der sogenannte Auto-Reload (**AR**)-Modus benötigt, bei dem der Timer nach einem Überlauf automatisch neu beginnt. Dementsprechend muss das **AR**-Bit gesetzt werden, um diesen Modus zu verwenden (vgl. Abbildung 16, oder TI-TRM-Abschnitt-DMTimer). Darauf hinaus werden im Control Register (**TCLR**) Einstellungen für den Prescaler, den Compare Mode und den Start des Zählers getätigt.

Prescaler

Das Clock Signal kann über einen zusätzlichen Prescaler geteilt werden, um den Takt zu verlangsamen. Im Praktikum wird der Prescaler aber nicht benötigt und **muss daher disabled** werden. Dafür braucht man das Bit **PRE** im **TCLR** Register des DMTimer.

Timerwert und LOAD_VALUE

Im Timer Counter Register (**TCRR**) des DMTimer befindet sich der aktuelle Wert des Zählers. Das Register kann ausgelesen werden um den aktuellen Zählerwert zu erhalten. Darüber hinaus kann das Register jederzeit beschrieben und somit manuell ein beliebiger Timerwert gesetzt werden. (*Hinweis:* Initialwert setzen)

In das Register Timer Load Register (**TLDR**) wird der sogenannte `LOAD_VALUE` geschrieben. Wenn sich der Timer im Auto-reload Modus befindet, beginnt der Zähler nach einem Overflow von diesem Wert aus neu zu zählen. Dies ist wichtig, um den Zähler nicht im immer von 0 bis 0xFFFFFFF zählen zu lassen, was unter Umständen zu lange dauert.

Compare-Match Logik

Die Timer beinhalten eine Compare-Match Logik, die es erlaubt einen Interrupt auszulösen, sobald der Timerwert einen zuvor eingestellten Vergleichswert (**COMPARE_VALUE**) erreicht. Dieser **COMPARE_VALUE** wird in das 32 bit lange **TMatch Register (TMAR)** Register geschrieben.

```
// set a new compare value
HWREG(SOC_DMTIMER_2_REGS + DMTIMER_TMAR) = meineVariable;
```

Um den Compare Modus zu aktivieren muss das Bit **CE** im **TCLR** Register gesetzt werden.

Timer starten

Um den Timer zu starten, muss im letzten Schritt vor der Hauptprogramm-Schleife das Bit **ST** im **TCLR** Register des DMTimer gesetzt werden. Erst ab diesem Zeitpunkt beginnt der Timer zu zählen.

Table 20-19. TCLR Register Field Descriptions (continued)

Bit	Field	Type	Reset	Description
6	CE	R/W	0h	0x0 = Compare mode is disabled 0x1 = Compare mode is enabled
5	PRE	R/W	0h	Prescaler enable 0x0 = The TIMER clock input pin clocks the counter 0x1 = The divided input pin clocks the counter
4-2	PTV	R/W	0h	Pre-scale clock Timer value
1	AR	R/W	0h	0x0 = One shot timer 0x1 = Auto-reload timer
0	ST	R/W	0h	In the case of one-shot mode selected (AR = 0), this bit is automatically reset by internal logic when the counter is overflowed. 0x0(READ) = Stop timeOnly the counter is frozen 0x1 = Start timer

Abbildung 16: Ausschnitt des TCLR Registers

Timerinterrupts

Die Interrupts des DMTimer2 müssen zunächst wieder mit der Funktion `IntSystemEnable()` aktiviert und anschließend die eigene ISR-Funktion darauf registriert werden.

Im Register `IRQENABLE_SET` des DMTimer-Moduls werden die Auslöser für einen Interrupt bestimmt. Hier gibt es drei mögliche Auslöser: Ein Compare-Match-Interrupt, ein Overflow-Interrupt und ein Capture-Interrupt (Letzterer fürs Praktikum irrelevant). Für die folgenden Aufgaben wird ein Compare-Match und ein Overflow-Interrupt benötigt, dementsprechend müssen diese beiden Bits gesetzt werden (siehe Anhang 6 DMTimer).

Für jeden Interrupt des DMTimers wird die gleiche Funktion (ISR) aufgerufen. Daher muss zu Beginn dieser ISR zunächst geprüft werden, was die Ursache für den Interrupt war: Entweder ein „Timer Overflow“ oder ein „Compare Match“. Dies macht man über Abfrage des `IRQSTATUS` Registers. Ist das erledigt, muss wie zuvor beim GPIO-Modul, das Timer-Interrupt-Flag gelöscht werden, um erneut auf einen Interrupt reagieren zu können. Die Flags werden gelöscht, indem eine 1 an die entsprechende Stelle des `IRQSTATUS` Registers geschrieben wird:

```
#define OVF_IT_FLAG 1
#define MAT_IT_FLAG 0

myTimer2Isr(){
    // Abfrage der Interruptquelle
    if ((HWREG(SOC_DMTIMER_2_REGS + DMTIMER_IRQSTATUS) & (1<< OVF_IT_FLAG)) != 0)
    {
        // Löschen des Timer-interrupt-Flags
        HWREG(SOC_DMTIMER_2_REGS + DMTIMER_IRQSTATUS) = (1<< OVF_IT_FLAG);

        // hier: eigener Code für Timer Overflow
    }
    else if ((HWREG(SOC_DMTIMER_2_REGS + DMTIMER_IRQSTATUS) & (1<< MAT_IT_FLAG)) != 0)
    {
        // Löschen des Timer-interrupt-Flags
        HWREG(SOC_DMTIMER_2_REGS + DMTIMER_IRQSTATUS) = (1<< MAT_IT_FLAG);

        // hier: eigener Code für Compare Match
    }
}
}
```

Table 20-15. IRQSTATUS Register Field Descriptions

Bit	Field	Type	Reset	Description
31-3	Reserved	R	0h	
2	TCAR_IT_FLAG	R/W	0h	IRQ status for Capture 0x0(W) = No action 0x0(R) = No event pending 0x1(W) = Clear pending event, if any 0x1(R) = IRQ event pending
1	OVF_IT_FLAG	R/W	0h	IRQ status for Overflow 0x0(W) = No action 0x0(R) = No event pending 0x1(W) = Clear pending event, if any 0x1(R) = IRQ event pending
0	MAT_IT_FLAG	R/W	0h	IRQ status for Match 0x0(W) = No action 0x0(R) = No event pending 0x1(W) = Clear pending event, if any 0x1(R) = IRQ event pending

Abbildung 17: Interrupt Status Register des DMTimer

2.4 Motorensteuerung

2.4.1 PWM – Puls Weiten Modulation

Die sogenannte Pulsweltenmodulation ist eine Möglichkeit zur Steuerung von Energieumwandlung in technischen Systemen, insbesondere zur Ansteuerung von Motoren.

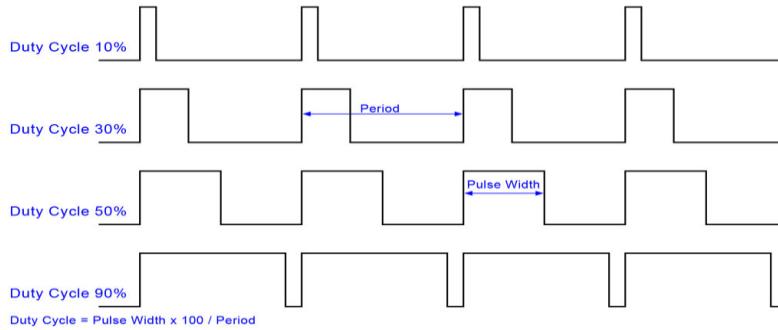


Abbildung 18: Pulsweltenmodulation

Ein PWM-Signal besteht aus einem Rechtecksignal mit fester Frequenz bei variabler Breite eines Rechteck-Pulses. Die Pulsbreite wird prozentual zur vollen Periodendauer angegeben und als „duty-cycle“ bezeichnet.

In einem Prozessor kann ein solches PWM-Signal mit Hilfe von Timern erzeugt werden. Dazu wird beim Zählerstartwert des Timers ein GPIO-Ausgang auf „HIGH“ gesetzt. Bei Eintreten eines Compare Match Interrupts wird der Ausgang zurück auf „LOW“ gesetzt. Tritt ein Overflow Ereignis ein und der Zähler beginnt von vorne, wird das Signal wieder „high“ gesetzt werden. Das Overflow-Ereignis tritt mit einer festen Frequenz auf, während durch Anpassen des Compare-Values die Breite des Pulses variiert werden kann.

2.4.2 Der Servo-Antrieb

Ein Servo-Antrieb wird über PWM gesteuert. Dazu werden 1-2ms Impulse im Abstand von 20ms gesendet. Die Pulsdauer ist direkt proportional zum Winkel des Motors. Also 1 ms-Impuls maximale Linksstellung, 2ms-Impuls maximale Rechtsstellung und 1,5ms Mittelstellung. Der Anschluss des Servos besteht aus den drei Leitungen, VCC, GND und die Leitung, auf der die Steuerimpulse gesendet werden.

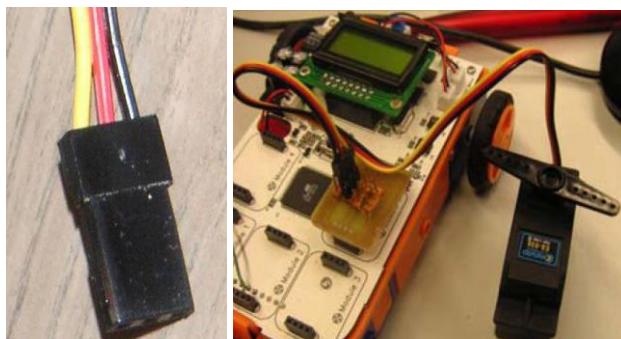


Abbildung 19: Elektrische und logische Anschlüsse des Servomotors

Schwarze Leitung	ist GND
Gelbe Leitung	ist Leitung für Steuerimpulse
Rohe Leitung	ist VCC (ca. 4-5V)

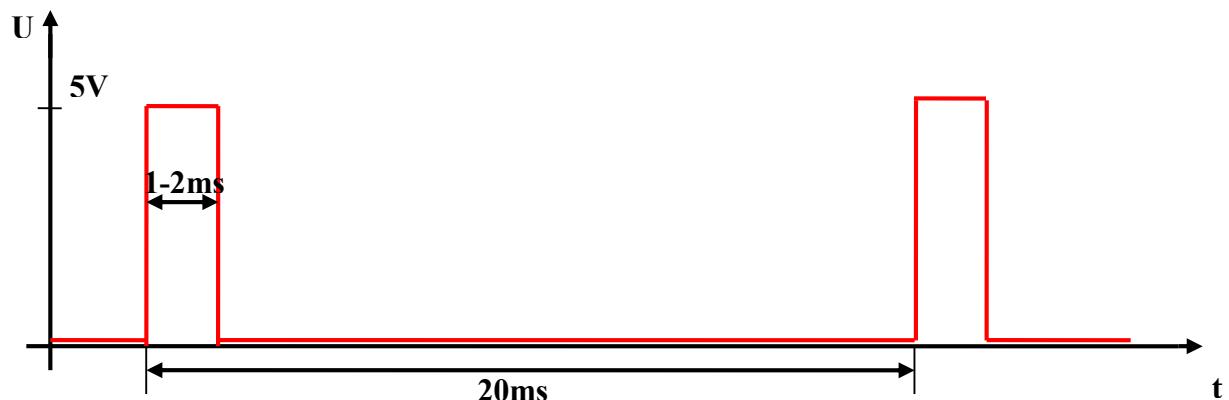


Abbildung 20: Steuerimpulstiming des Servomotors

Die Periodendauer von 20ms hat eine Toleranz von ca. +2ms ohne Beeinträchtigung des Verhaltens des Servos, aber die 1-2ms-Impulse müssen so genau wie möglich sein. D.h. bei einer Steuerung per µC wird hier taktgenau gearbeitet.

2.4.3 DC-Motoren

Motortreiber

Über zwei auf dem Cape verbaute H-Brücken-Bausteine lassen sich zwei separate DC-Motoren kontrollieren. Die Motoren 1 und 2 sind wie folgt angeschlossen:

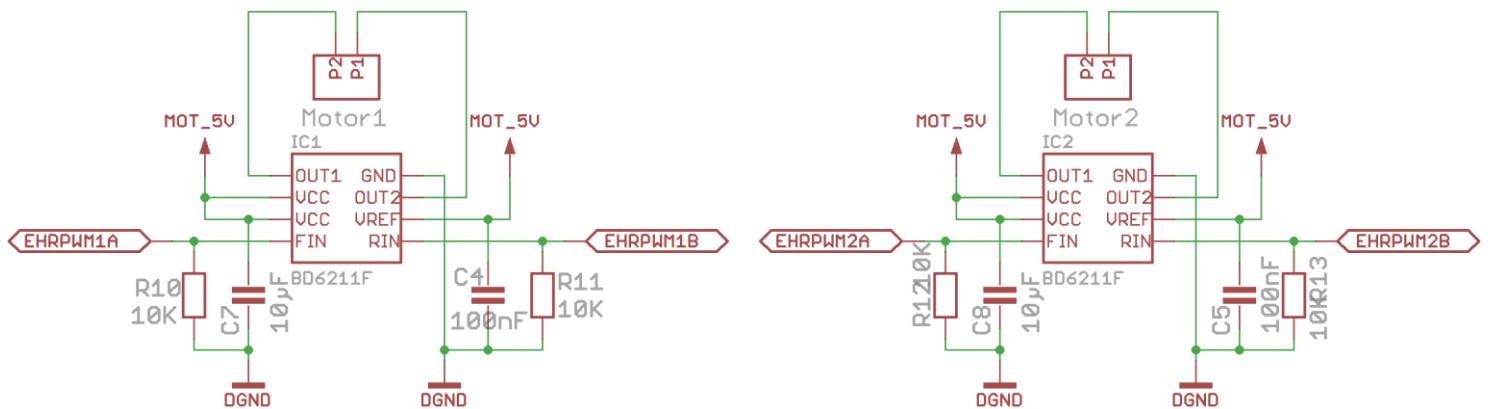


Abbildung 21: Verschaltung und Anschluss der H-Brücken (Motortreiber) auf dem Cape

OUT1 und OUT2 werden mit den beiden Anschlüsse des DC-Motors verbunden und verhalten sich wie folgt:

● Input / output truth table

FIN	RIN	OUT2	OUT1	Mode
H	L	L	H	Forward
L	H	H	L	Reverse
H	H	L	L	Brake
L	L	OPEN	OPEN	Standby

Abbildung 22: Wahrheitstabelle, die das Verhalten von OUT1 und OUT2 in Bezug auf die Eingänge wiedergibt.

Der Prozessor enthält neben den Timern bereits integrierte Hardware PWM-Module, welche ein direktes Erzeugen von PWM-Signalen ermöglichen. Die Steuerung der Motortreiber für die beiden DC-Motoren erfolgt über zwei dieser PWM-Module. Dafür sind die entsprechenden Ausgänge bereits mit den Motortreibern verbunden (siehe Abbildung 21). Um einen Motor zu drehen, muss an einem der beiden Eingänge FIN und RIN der Motortreiber ein PWM-Signal angelegt werden, während das andere auf „LOW“ (0V) gesetzt wird.

Modul steckplatz	Pin	Pin Name	Signal Name							
			Mode							
			0	1	2	3	4	5	6	7
MOTOR 1	1	GPMC _A2	gpmc_a 2	mii2_txd 3	rgmii2_t d3	mmc2_d at1	gpmc_a 18		ehr- pwm1A	gpio1_1 8
MOTOR 1	2	GPMC _A3	gpmc_a 3	mii2_txd 2	rgmii2_t d2	mmc2_d at2	gpmc_a 19		ehr- pwm1B	gpio1_1 9
MOTOR 2	1	GPMC _AD8	gpmc_a d8	lcd_data 23	mmc1_d at0	mmc2_d at4	ehr- pwm2A			gpio0_2 2
MOTOR 2	2	GPMC _AD9	gpmc_a d9	lcd_data 22	mmc1_d at1	mmc2_d at5	ehr- pwm2B			gpio0_2 3

Verwendung der EHRPWMs

Für die Verwendung der EHRPWMs (Enhanced High Resolution Pulse Width Modulation) des Prozessors werden im Praktikum bestimmte Funktionen zur Verfügung gestellt. Für die Verwendung der Funktionen, muss das Header-File `EGR_DCMotor.h` eingebunden werden. Es sind dort vier Funktionen definiert:

1. `EHRPWMinitForDCMotor()`: Initialisiert das EHRPWM-Modul und macht alle benötigten Einstellungen um EHRPWM1 und EHRPWM2 zu verwenden. Die Pins müssen allerdings separat gemuxt werden.
2. `EHRPWMsetDutyCycle (unsigned int baseAddr, unsigned short dutyCycle)`: Mit dieser API setzt man den duty-cycle des PWM-Signals, also die Pulsbreite in Prozent.
 - a. Parameter – `baseAddr`: Basisadresse des jeweiligen PWM-Moduls (zu finden in der `soc_AM335x.h` unter `soc_EPWM_1_REGS`)
 - b. Parameter – `dutyCycle`: Pulsbreite in Prozent von 0 bis 100

3. configEHRPWM_A (unsigned int baseAddr, unsigned int ActionOnZero, unsigned int ActionOnCompMatch): Mit dieser API wird das Verhalten der Ausgänge EHRPWMxA bei Erreichen bestimmter Zählerwerte eingestellt. Um ein PWM-Signal zu erzeugen muss ein Ausgang bei Zählerwert 0 (Zero) auf „HIGH“ und bei Erreichen des compareMatchWertes wieder auf „LOW“ gesetzt werden.
 - a. Parameter – baseAddr: Basisadresse des jeweiligen PWM-Moduls
 - b. Parameter – ActionOnZero: Verhalten des Ausgangspins bei Zählerstand = 0
 - c. Parameter – ActionOnCompMatch: Verhalten des Ausgangspins bei Zählerstand = CompareMatchWert
 - d. Mögliche Werte sind in EGR_DCMotor.h definiert:

```
#define EHRPWM_DO_NOTHING          0x0
#define EHRPWM_SET_OUTPUT_LOW        0x1
#define EHRPWM_SET_OUTPUT_HIGH       0x2
#define EHRPWM_OUTPUT_TOGGLE         0x3
```
4. configEHRPWM_B (unsigned int baseAddr, unsigned int ActionOnZero, unsigned int ActionOnCompMatch): Analog zu configEHRPWM_A für die Ausgänge EHRPWMxB

Das PIN Muxing um die EHPWM-Signale auf die entsprechenden Pins des BeagleBone zu muxen **muss separat dazu erfolgen**. Informationen zu den benötigten „modes“ sind im Datenblatt_Beaglebone_Cape.pdf zu finden.

Hinweis zu den Registeradressen des PWM Subsystems:

Die Basisadressen finden sich in der Memory-Map unter **ePWM1** bzw. **ePWM2**. Es existiert auch ein #define-Makro in der `soc_AM335x.h` Datei mit dem Namen `SOC_EPWM_1_REGS` bzw. `SOC_EPWM_2_REGS`.



2.5 Ausgabe auf dem PC-Bildschirm

Mit Hilfe der Funktionsbibliothek uartStdio.h lassen sich sehr einfach Zeichen an den COM-Port eines PC schicken, und dort auf einer Konsole ausgeben. Es muss lediglich die Headerdatei eingebunden werden:

```
#include <uartStdio.h> //Headerfile mit vorgefertigten Ausgabefunktionen
```

So kann man einfach debuggen und sehen welche Werte der ADC zurückliefert. Nach dem Einbinden des Headers, können die Funktionen im Programm aufgerufen werden, z.B.:

```
UARTprintf("Aktueller Analog-Wert: ");  
UARTPutNum(adcData);  
UARTprintf("\r\n");
```

Weitere nützliche Funktionen lassen sich durch Öffnen der uartStdio.h herausfinden.

Hinweis:

Achtung! Ausgaben über die serielle Schnittstelle können das Programm stark verlangsamen. Machen Sie sich bewusst, wie lange es dauert mit 115200 kBit/s Daten zu senden, im Verhältnis zur Taktzeit des Prozessors von bis zu 1GHz.



Kochrezept Termin 2

Konfigurieren der **Interrupts**:

- Allgemein Interrupts aktivieren
 - Enable über entsprechende *Int*-Funktion
- Allgemein den Interrupt Controller initialisieren
 - Initialisieren über entsprechende *Int*-Funktion
- Die vorgesehenen Interrupt-Quelle (also das auslösende Element) muss aktiviert werden
 - Enable über entsprechende *Int*-Funktion
- Der aktivierte Interrupt-Quelle muss eine Interrupt-Serviceroutine (ISR) zugewiesen werden, die ausgelöst wird
 - Registrieren der ISR über entsprechende *Int*-Funktion
- Schreiben der ISR
 - in der ISR muss das auslösende Interrupt-Flag gelöscht werden
 - Zugriff auf das Status-Flag: Interruptquelle als Basisadresse & IRQSATUS als Offset-Adresse

Speziell beim Auslösen über **GPIO**-Modul:

- Festlegen welcher Pin welchen Interrupt auslösen soll (GPIOxA oder GPIOxB)
 - Das entsprechende Bit (Bit = Pin-Nummer) im entsprechenden IRQ_STA-TUS_SET (A=0 ; B=1) setzen

Konfigurieren des **Timers**:

- Einen der Timer des BeagleBone auswählen
 - Acht Timer auf BeagleBone: DMTimer0 - DMTimer7
 - DMTimer2 bietet sich für das Praktium an!
 - Basis-Adresse wählen
- Modus des ausgewählten Timers festlegen (One-Shot/Auto-Reload)
 - TCLR Register
- Funktion des Prescalers definieren (disable/Prescale aktivieren)
 - TCLR Register
- Startwert für den ersten Timerstart festlegen
 - TCRR Register
- Reloadwert für den ersten Timer festlegen
 - TLDR Register
- Evtl. Compare-Match Logik (Compare Mode) für Timer aktivieren
 - TCLR Register
- Wert für Compare-Match des Timers festlegen
 - TMAR Register
- Timer starten
 - TCLR Register

Interrupts beim Timer:

- Timer Interrupts enablen
→ IRQENABLE_SET Register
- ISR Funktion registrieren
→ entsprechende *Int*-Funktion
- In der ISR Funktion die Interruptquelle abfragen
→ IRQSTATUS Register
- Interrupt-Flag löschen, damit Interrupt neu ausgelöst werden kann
→ IRQSTATUS Register
- Eine Prise eigenen Code hinzufügen

DC-Motoren ansteuern:

- Initialisieren des EHRPWM-Module
→ EHRPWMinitForDCMotor()
- PIN Muxing
- Gewünschten Duty-Cycle der EHRPWM-Module einstellen
→ EHRPWMsetDutyCycle
- H-Brücke wie in Abbildung 22 ansteuern, um die Drehrichtung einzustellen
→ configEHRPWM_A und configEHRPWM_B



Verständnisfragen

Echtzeit

1. Was bedeutet Echtzeit?
2. Welche Methoden gibt es um echtzeitfähige Programme zu schreiben?
3. Was ist Polling?
4. Was ist ein Interrupt?
5. Nennen sie drei Interrupt-quellen?
6. Warum dürfen in einer *Interrupt Service Routine* keine Delays verwendet werden?
7. Was ist ein Interrupt-Flag und warum muss es gelöscht werden?

Antriebe

1. Was ist ein Timer?
2. Was ist ein Comparematch Interrupt?
3. Was ist ein Overflow Interrupt?
4. Mit welchem Motor betreibe ich am besten einen PC-Lüfter?
5. Was ist der Unterschied zwischen einem PWM zur Leistungssteuerung und einem PWM zur Servo-Steuerung?
6. Was ist der Duty Cycle?
7. Wozu benötigt man eine H-Brücke?
8. Was ist der Vorteil eines Hardware-PWMs gegenüber einem PWM-Signal, das mit Delays erzeugt wird?



Termin 2

- Aufgaben



2.6 Aufgaben Termin 2: Antriebe und Echtzeit

Aufgabe 1: Interrupts (Notaus)

Erweitern sie ihr Lauflicht-Programm aus Termin 1 (Aufgabe 4) durch Interrupts. Lesen sie einen weiteren Taster aus mit GPIO-Interrupt, um eine Notaus-Funktion zu implementieren, die das Lauflicht sofort beendet.

Inhalt: Interrupts, GPIO-Interrupts

Aufgabe 2: Servo-Steuerung

Schließen sie das Servo-Modul am Steckplatz 3 an. Schreiben Sie ein Programm, das den Servomotor abwechselnd zwischen den beiden Maximalpositionen hin und her bewegt.

Schließen Sie das Tasten-Modul am Steckplatz 1 an. Nun soll der Servomotor um jeweils einen kleinen Schritt nach rechts bzw. links fahren, wenn der Taster eins bzw. zwei gedrückt wird. Der Taster drei bewirkt eine Nullstellung des Motors.

Inhalt: PWM, Time-Interrupt, Echtzeit, #define

Hinweis: benutzen Sie die DMTIMER2. Realisieren sie die Periode mithilfe des Overflows und des Reload-Values und die Länge des Pulses mit einem Compare-Match.

Aufgabe 3: DC-Motorsteuerung

Schreiben Sie ein Programm, das die Motoren mit maximaler Leistung antreibt und bei Tastendruck die Richtung ändert.

Inhalt: DC-Motor, H-Brücke

Aufgabe 4: DC-Geschwindigkeitssteuerung, Richtungssteuerung

Aufbauend auf Aufgabe 3 soll nun die Geschwindigkeit der Motoren variabel in einem Wertebereich von -100 bis 100 einstellbar sein. Dies soll über jeweils einen Taster zum Erhöhen bzw. Vermindern der Geschwindigkeit (geeignete Schrittweite) erfolgen. Anschließend sollen die Geschwindigkeitswerte an eine separate Funktion übergeben werden, welche die Motoren steuert. Binden Sie die gesamte Funktionalität in eine Headerdatei „Motorsteuerung.h“ ein.

Inhalt: Header, Hardware PWM

Hinweis: Nutzen sie zur Generierung des PWM-Signals das Modul EHRPWM.





Termin 3

- Skript:
Ein- /Ausgabegeräte und Sensoren

3. Termin: Ein- /Ausgabegeräte und Sensoren

3.1 Module

3.1.1 LDR-Modul

Das LDR-Modul (Abbildung 23) ist ein Sensormodul, welches auf Lichtstärke reagiert. Ein LDR (Light Dependent Resistor) ist ein lichtabhängiger elektrischer Widerstand. Er wird in einem Spannungsteiler verschaltet. Das Modul liefert eine analoge Spannung in Abhängigkeit von der Lichtstärke zurück. Den Schaltplan des LDR-Moduls finden Sie im Kapitel 7 – Moduldatenblätter.

Die Sensorspannung kann über einen Analog/Digital Converter (ADC) des Prozessors eingelesen und verarbeitet werden.



Abbildung 23: Foto des LDR-Moduls

3.1.2 Joystick-Modul

Das Joystickmodul besteht aus zwei Teilen: Dem Joystick selbst, welcher über zwei Potentiometer analoge Spannungen erzeugt, sowie aus der Anschlussplatine für das Beaglebone. Diese beiden Teile werden über ein 6-adriges Kabel verbunden. Den Schaltplan des Joystickmoduls finden Sie im Kapitel 7 – Moduldatenblätter.

3.2 Verwendung des im Prozessor integrierten ADC

Am Beispiel des LDR-Sensors und des Joysticks soll gezeigt werden, wie man mit dem Beaglebone analoge Sensorwerte einlesen kann.

3.2.1 Aktivieren des ADC-Moduls

Der AM335x-Prozessor des Beaglebone besitzt ein integriertes ADC-Modul. Dieses ADC-Modul ist intern mit einem Touchscreen-Controller-Modul zusammengefasst und taucht deshalb im Datenblatt unter Touchscreen-Controller auf (**siehe Kapitel 14 Touchscreen Controller (ADC) Register**).

Für die Abtastung der ADC-Eingänge benötigt man ein Clocksignal mit der entsprechenden Abtastrate. Für das Touchscreen-Controller-Sub-System (TSC_ADC_SS) muss daher ein Clock-Signal aktiviert werden. Dies geschieht im zentralen *Clock Module* (CM_WKUP)!

```
// Maskiert die letzten beiden Bits und setzt dort den Wert 0x2
HWREG(SOC_CM_WKUP_REGS + (0xbc)) &= ~(0x3);
HWREG(SOC_CM_WKUP_REGS + (0xbc)) |= 0x2;
```

3.2.2 Einstellen der ADC Parameter

Um den ADC zu verwenden, müssen zunächst einige Parameter voreingestellt werden. Hierzu stehen wieder verschiedene Kontroll- und Statusregister zu Verfügung (Makros definiert in hw_tsc_adc_ss.h)

```
// Einbinden des Headerfiles für den ADC
#include <hw_tsc_adc_ss.h>
```

Im Vergleich zu den meisten anderen Pins des Prozessors, dienen die Pins AIN0 – AIN7 nur als analoge Pins und haben keine weiteren Funktionen. Sie müssen daher nicht explizit gemuxt werden. Der Prozessor besitzt nur einen physikalischen 12 Bit A/D-Wandler, der immer nur einen Analogpin gleichzeitig auswerten kann. Mit Hilfe eines Multiplexers wird zwischen den 8 Eingangspins AIN0 – AIN7 gewählt.

Das ADC-Modul des Prozessors arbeitet mit einer konfigurierbaren, sequentiellen „State Machine“ um die ADC Aufgaben zu bewältigen und die verschiedenen Eingangspins auf den A/D-Wandler zu muxen. Diese State Machine durchläuft sequentiell bis zu 16 sogenannte STEPs. Für jeden dieser STEPs kann eine eigene ADC-Konfiguration gewählt werden. Zu dieser Konfiguration zählt welcher der Analogpins wann und wie abgetastet und welche Referenzspannung dafür gewählt werden soll.

Schreibschutz deaktivieren

Da die STEPs per default schreibgeschützt sind, muss im ersten Schritt der Schreibschutz aufgehoben werden. Hierfür muss das entsprechende Bit im TSC_ADC_SS_CTRL Register gesetzt werden.

Steps wählen und aktivieren

Im nächsten Schritt wählt man aus, welche und wie viele der 16 STEPs man verwenden will. Für jeden zu verwendenden Step muss das entsprechende Bit im STEP_ENABLE Register gesetzt werden.

```
HWREG(SOC_ADC_TSC_0_REGS + TSC_ADC_SS_STEPEnable) |= (1 << StepNr);
// bei StepNr die Nummer des zu aktivierenden Steps einsetzen
...
```

Um zwei analoge Spannungen an zwei unterschiedlichen Analogpins zu messen, empfiehlt es sich **zwei** der möglichen **16** Steps zu aktivieren.

Steps einstellen

Jeder Step hat ein eigenes STEP CONFIG Register, in welchem alle Einstellungen für den Step getätigter werden. Vorsicht ist bei der Verwendung der Defines von TI geboten: **TSC_ADC_SS_STEP CONFIG(1)** beschreibt den **Step Nr. 2** (also um 1 versetzt). Wichtige Einstellungen die hier gesetzt werden müssen, sind:

- Wahl des Modus (Bits 0 und 1): Ein Step unterstützt mehrere Modi. Per default ist der *One-Shot Mode ausgewählt*, bei dem nur ein einziger Analogwert aufgenommen wird. Um kontinuierlich Werte zu messen, muss in den Bits 0-1 als Mode *01 = SW enabled, continuous* gewählt werden.
- Wahl des Analogpins (Bits 19-22): Die drei Steckplätze/Ports des EGR-Capes sind mit den drei Pins AIN1 (Port 1), AIN3 (Port 2) und AIN5 (Port 3) verbunden. Lassen Sie sich im TI-TRM nicht von der Bezeichnung „Channel“ verwirren: AIN1 (Channel 2) wird über das Setzen einer 1 ausgewählt, AIN3 über eine 3 usw.
- Wahl der positiven Referenzspannung (Bits 12-14): Der ADC benötigt eine Referenzspannung um die analoge Spannung auswerten zu können. Hier wird die interne Referenz VDDA verwendet (ist schon per default gesetzt).
- Wahl des FIFO-Speichers (Bit 26 FIFO_select): Hat der AD-Wandler einen Wert erfasst, legt er ihn in einen von zwei FIFO-Speichern ab. In welchen FIFO er ihn ablegt, wird mit dem FIFO_select Bit ausgewählt.
- Ein letzter Punkt der eingestellt werden sollte ist das sogenannte „Averaging“ (Bit 2-4). Das bedeutet, der ADC nimmt bis zu 16 Werte auf und legt anschließend nur den Mittelwert daraus im FIFO ab.
Der Wert soll hier auf „16 samples average“ gesetzt werden, um hochfrequente Spannungsschwankungen auszumitteln.

AD-Wandler starten

Um die Analog-Digital-Wandlung zu starten muss zu guter Letzt das ENABLE Bit im TSC_ADC_SS_CTRL Register gesetzt werden.

Spannungen mit dem ADC auslesen

Die vom ADC gemessenen und im FIFO abgelegten Spannungswerte können auf unterschiedliche Weise ausgelesen werden. Die einfachste Variante ist der sogenannte Polling-Betrieb. Dabei wird bei jedem Schleifendurchlauf des Hauptprogramms der aktuelle AD-Wert in eine Variable gespeichert.

Der aktuelle AD-Wert kann aus dem FIFO0DATA oder FIFO1DATA Register ausgelesen werden. z.B.:

```
unsigned int meinAdcWert;
meinAdcWert = HWREG(SOC_ADC_TSC_0_REGS + TSC_ADC_SS_FIFODATA(0));
//Schreibt den aktuellen FIFO-Wert in meinAdcWert
```

3.3 Starten von SD-Karte

Das BeagleBone wird in mehreren Stufen gebootet. Zu Beginn wird im internen Flashspeicher ein *MLO* und ein *u-boot* Bootloader ausgeführt. Der *u-boot* Bootloader sucht in der SD-Karte nach der Datei „*uenv.txt*“. In ihr kann man angeben, an welcher Stelle das Programm liegt, das ausgeführt werden soll. Zum Benutzen des BeagleBones **ohne** serielle Verbindung muss die *uenv.txt* abgeändert werden in:

```
uenvcmd=mmcinfo;fatload mmc 0 0x80000000 EGR_run.bin; go 0x80000000
```

Das Kommando muss ohne Leerzeile hintereinander in eine Zeile geschrieben werden.

Legen Sie die komilierte Datei „EGR_run.bin“ neben der TXT-Datei in das Stammverzeichnis der SD-Karte. Übertragen Sie Ihr Programm dazu über den gemeinsamen Ordner (siehe Kapitel 0.8.4) in das Host-Betriebssystem und übertragen es von dort auf die SD-Karte.

Zum offline Booten des BeagleBones muss also das Programm nicht vom PC aus per Kermit in den Speicher geladen werden, sondern neben der veränderten *uenv.txt* auf der Karte unter dem Namen „*EGR_run.bin*“ gespeichert werden.

Stecken Sie die SD-Karte vorne in das BeagleBone. Nach einem Reset wird das Programm *EGR_run.bin* automatisch ausgeführt.

Kochrezept- Termin 3

ADC-Modul:

- Headerfile einbinden
→ include <hw_tsc_adc_ss.h>
- ADC-Modul aktivieren indem Clocksignal zugewiesen wird
→ CM WKUP Register
- Headerfile einbinden
→ include <hw_tsc_adc_ss.h>
- Schreibschutz für die STEPs der integrierten State Machine aufheben
→ CTRL Register
- Anzahl der STEPs wählen auf die zugegriffen werden soll
→ STEPENABLE Register
- Modus für STEPs einstellen (continuous mode)
→ STEPCONFIG Register
- Analogpins auswählen (Achtung: Heißen bei TI hier Channels!)
→ STEPCONFIG Register
- Referenzspannungen für die gewählten Pins festlegen
→ STEPCONFIG Register
- FIFO-Speicher auswählen, in welchen geschrieben werden soll
→ STEPCONFIG Register
- „Averaging“ für die Analogwertmessung festlegen
→ STEPCONFIG Register
- AD-Wandler starten
→ CTRL Register
- Auf die Werte im FIFO-Speicher zugreifen
→ FIFOXDATA Register

Verständnisfragen

1. Wofür steht die Abkürzung ADC?
2. Wofür wird ein ADC verwendet?
3. Wieviel ADCs hat der AM3359 und wie viele Analogeingänge stehen zur Verfügung?

Termin 3

- Aufgaben



3.4 Aufgaben Termin 3: Ein- Ausgabegeräte und Sensoren

Aufgabe 1: Helligkeitssensor

Schließen sie das LDR-Modul an Steckplatz 1 an. Schreiben Sie ein Programm, das die Lichtintensität einliest und auf der Konsole ausgibt.

Inhalt: AD-Wandler, uartStdio.h

Aufgabe 2: Joystickwert-Anzeige

Schließen Sie das Joystickmodul an Steckplatz 1 und 2 an. Schreiben Sie ein Programm, das die beiden Poti-Werte des Joysticks einliest und auf der Konsole ausgibt.

Inhalt: Poti

Aufgabe 3: Joysticksteuerung

Koppeln Sie die Joystickwert-Anzeige aus Aufgabe 2 mit Ihrer „Motorsteuerung.h“ und realisieren Sie damit eine Fernsteuerung für Ihren Roboter. Es soll eine Lenkung und eine Geschwindigkeitssteuerung möglich sein.

Aufgabe 4: Vorbereiten einer SD-Karte

Verändern Sie den Inhalt der SD-Karte, sodass Sie die Joysticksteuerung von der SD-Karte aus starten können, ohne dass der Roboter mit dem PC verbunden sein muss.

Inhalt: Bootloader.





Termin 4

- Skript:
Kommunikationsschnittstellen

4. Termin: Kommunikationsschnittstellen

4.1 I²C-Kommunikation

Mikrocontroller verfügen üblicherweise über eine Reihe peripherer Komponenten, die mit der CPU in Verbindung stehen. Sie können vom Programm dynamisch aktiviert und deaktiviert werden und sehr unterschiedliche Aufgaben erfüllen. Sie bestehen aus speziellen „in Hardware gegossenen“ Schaltungen, die mit dem Systemtakt versorgt werden und ihre Aufgabe fast unabhängig vom Prozessor erledigen. Dieser modulare Aufbau unseres Mikrocontrollers ist in Abbildung 24 zu erkennen.

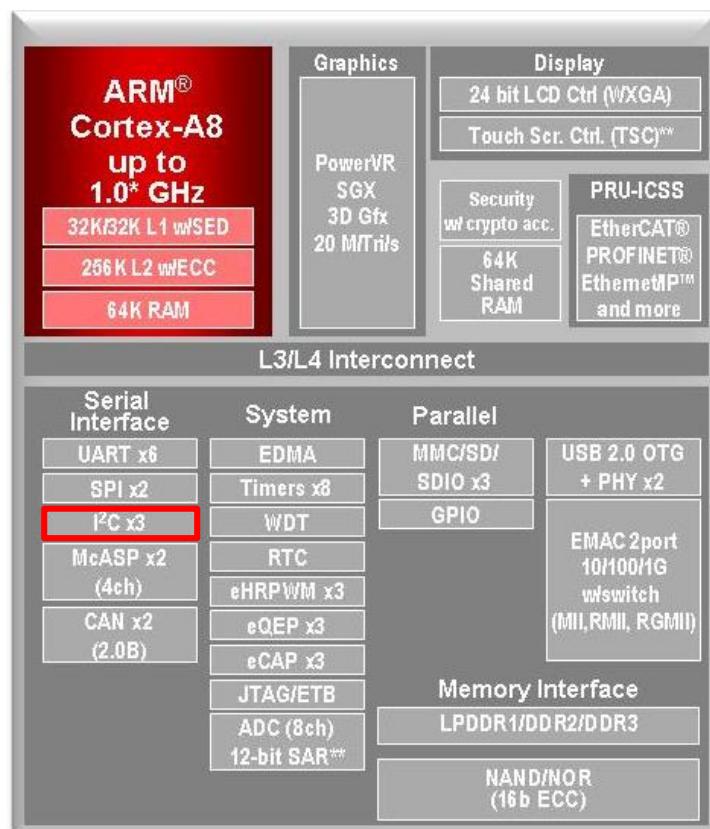


Abbildung 24: TI ARM Diagramm (www.ti.com)

Zwischen dem Prozessor und den peripheren Komponenten wird nur ein Minimum an Information ausgetauscht, beispielsweise zum An-/Abschalten und Konfigurieren der Komponente oder zum Datentransfer.

Gelernt haben wir das bereits in Kapitel 2 bei der Verwendung von Hardware-Interrupts oder Timern: Ein Software-Timer würde die CPU zu 100% in einer Warteschleife auslasten, während ein interrupt-gestützter, peripherer Counter nach Ablauf des gegebenen Zeitintervalls den Prozessor informiert, welcher in der Zwischenzeit anderen Aufgaben nachgehen kann. Ähnlich effizient funktioniert auch die Verwendung einer seriellen Kommunikations-Schnittstelle wie der UART oder I²C: Eine Software-Implementierung würde die CPU voll auslasten, die periphere Hardware belastet den Prozessor dagegen fast gar nicht.

In diesem Praktikum werden wir das **I²C-Interface** (Inter-Integrated Circuit, „I Square C“, „I-Quadrat C“) verwenden, um die Infrarotsensoren an der Unterseite des Roboters auszulesen und damit Markierungen auf dem Boden erkennen zu können. Dies wird uns bei Navigationsaufgaben, wie beispielsweise dem Folgen einer Linie behilflich sein.

4.1.1 Systemarchitektur

Das I²C-Interface wird auch TWI (Two Wire Interface) genannt, da der Datenbus mit insgesamt nur zwei Leitungen auskommt, einer Datenleitung (SDA) und einer Takteleitung (SCL). Deswegen werden für das sehr einfache Protokoll nur zwei I/O-Pins benötigt.

Ein weiterer Vorteil ist, dass man nur einen Mikrocontroller benötigt, um ein ganzes Netzwerk an integrierten Schaltungen zu kontrollieren. Denn das Konzept des Busses basiert auf einer Master-Slave Architektur, d. h. die Kontrolle der Kommunikation liegt in der Hand eines Masters, während die Slaves nur auf die Befehle reagieren. Ein Slave kann also nie selbstständig Daten senden. In einem I²C-Bus gibt es mindestens einen Master, Konfigurationen mit mehreren Mastern sind aber möglich (Multi-Master).

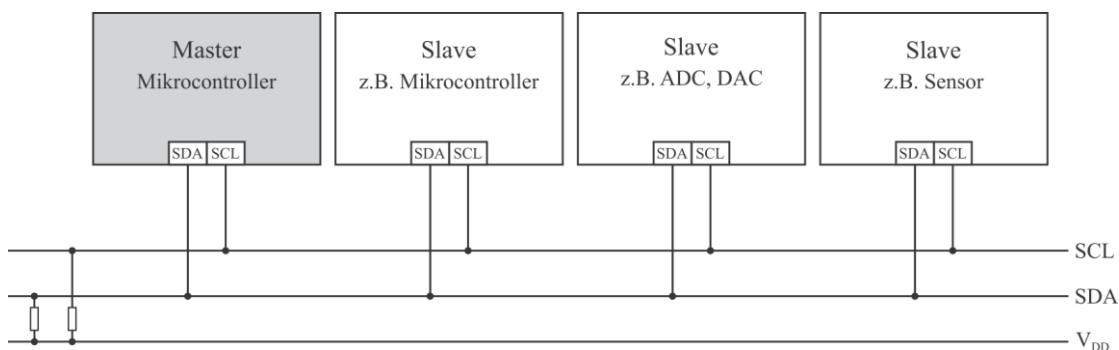
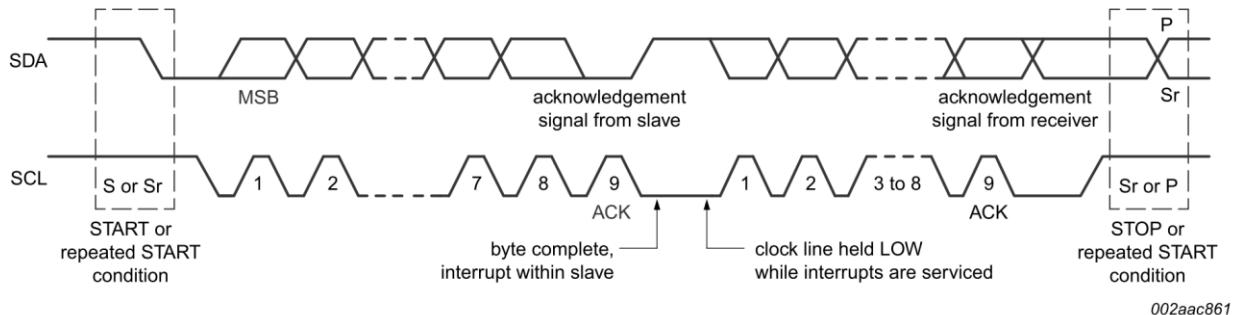


Abbildung 25: Master-Slave-Architektur des I²C-Busses

Der Hauptnachteil von I²C sind die geringen Übertragungsraten. Sie reichen von 100 kBit/s (Standard Mode) über 400 kBit/s (Fast Mode) bis hin zu maximal 3,4 Mbit/s (High-Speed Mode). Grund dafür ist die durch das Fehlen von Maßnahmen zur Übertragungssicherung bedingte Störanfälligkeit des Systems, was seine Verwendung auf störungsfreie Anwendungsgebiete einschränkt, d. h. Bereiche ohne Übersprechen (Crosstalk), Rauschen, EMV- und Kontaktprobleme (Stecker, Buchsen). Deswegen wird I²C typischerweise zur Kommunikation auf kurzen Strecken verwendet, z.B. zwischen zwei Mikrocontrollern auf einer Platine oder – wie in unserem Fall – zwischen Mikrocontroller und Sensor.

4.1.2 Protokoll



**Abbildung 26: Datentransfer auf dem I²C-Bus; UM 10204 I²C-bus specification and user manual.
NXP Semiconductor, 04 2014.**

Ein Datenaustausch über den Bus beginnt, indem der Master den Bus übernimmt. Dazu setzt er eine **Start-Bedingung** (Die Datenleitung wird auf LOW gezogen, während die Takteleitung auf dem HIGH-Pegel bleibt). Danach erzeugt er ein Rechteck-Signal auf der Takteleitung. Bei jeder steigenden Flanke der Takteleitung liest der Slave den Pegel der Datenleitung und erhält so ein Bit. Es handelt sich demnach um eine synchrone Datenübertragung.

Das erste gesendete Byte (1 Byte = 8 Bit) ist die **Adresse** des Slaves, wobei das achte Bit (**Read/Write-Bit**) den Lese- oder Schreibwunsch festlegt. Der Adressraum des I²C-Protokolls umfasst demnach 7 Bit. Dadurch wird die maximale Teilnehmeranzahl des Protokolls definiert: Abzüglich 16 reservierter Adressen, wie z.B. dem „General Call“ zum Ansprechen aller Teilnehmer, sind bis zu 112 Teilnehmer in einem Netzwerk erlaubt.

Das neunte Bit (**ACK = Acknowledge**), wird vom Slave gesetzt, indem er SDA auf LOW zieht. Dadurch bestätigt der Slave den Erhalt eines vollständigen Bytes. Falls der Master das neunte Bit nicht erhält (**NAK = Not Acknowledge**), muss er die Datenübertragung mit einer Stop-Bedingung beenden, oder mit einer Start-Bedingung neu starten.

Auch die weitere Datenübertragung erfolgt byteweise nach dem selben Schema. Abhängig vom Read/Write-Bit werden die Datenbytes entweder vom Master oder vom Slave auf den Bus geschrieben. Im Falle eines Lesevorgangs quittiert der Master die erhaltenen Bytes jeweils mit einem NAK.

Die Datenübertragung wird vom Master beendet, indem er eine **Stop-Bedingung** setzt (Steigende Flanke an SDA während SCL HIGH bleibt). Danach ist der Bus wieder als frei markiert und eine neue Datenübertragung kann von einem Master begonnen werden.

4.2 Programmierung der Kommunikation

Die Kommunikation über I_C realisieren wir über je eine Funktion für die **Initialisierung**, das **Senden** und das **Empfangen** von Daten, welche in der `i2c.c` bereits vorbereitet sind und lediglich vervollständigt werden müssen. Die Funktionsweise dieser Methoden wird in den nachfolgenden Unterkapiteln erläutert.

Die Logik des I²C-Protokolls muss dabei nicht selbst implementiert werden, da sich darum die Hardware des Mikrocontrollers kümmert, wie bereits zu Beginn von Kapitel 4.1 erklärt wurde.

Das Ziel ist, den Zustand der Sensoren an der Spitze des Roboters auszulesen. Dort befinden sich sechs Infrarotsensoren (3 auf der Unterseite, drei an der Stirn), welche aus je einer Leuchtdiode und einer Empfangsdiode bestehen. Wird das von der Leuchtdiode ausgesandte Licht ausreichend reflektiert (z.B. weiße Oberfläche), dann ergibt dies eine "1", wenn nicht (z.B. Abgrund, schwarze Linie usw.) dann eine "0".

Diese Infrarotsensoren sind nicht direkt mit dem Beaglebone verbunden, sondern hängen an einem eigenen Chip, dem **PCA9554A** von NXP Semiconductors, welcher das I²C-Datenprotokoll versteht. Dies ist in Abbildung 27 dargestellt. Wären die Sensoren direkt mit unserem Mikrocontroller verbunden, würden sie 12 GPIO-Pins belegen, was entsprechend viele Leitungen notwendig machen würde. Durch die Ansteuerung per I²C werden nur zwei Kommunikationsleitungen benötigt.

Das Beaglebone ist in unserem Fall der Master des Netzwerks. Der PCA9554A dient als Slave. Um die Sensoren auslesen zu können, ist eine Kommunikation der beiden Chips notwendig.

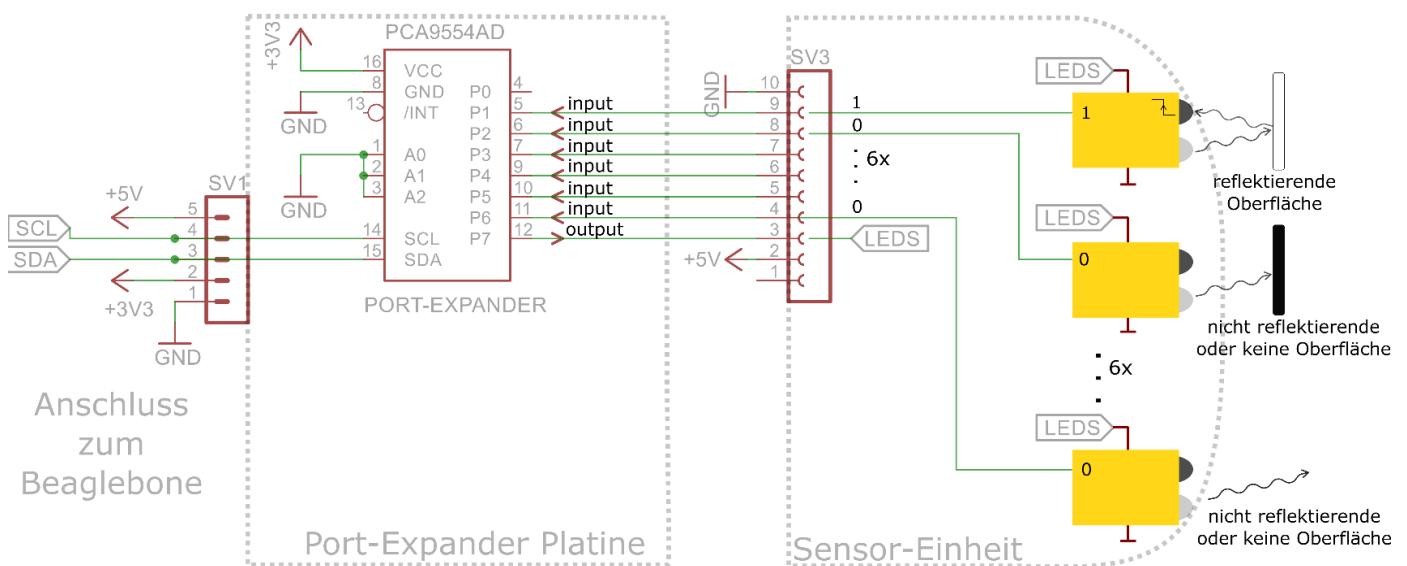


Abbildung 27: Skizze zu Anschluss und Funktionsweise der Infrarotsensoren

4.2.1 Funktionsweise des Port-Expanders

Der PCA9554A verfügt über eine interne Logik, welche das I²C-Protokoll versteht. Außerdem stellt er einen begrenzten Funktionsumfang zur Interaktion mit seinen 8 konfigurierbaren GPIO-Pins (P0 ... P7) zur Verfügung. Das Datenblatt des PCA9554A haben Sie beim ersten Termin zusammen mit weiteren Unterlagen auf dem USB-Stick erhalten. In Abschnitt 6 des Datenblatts ist die Funktionsweise des Chips erklärt.

Wie in Abbildung 27 dargestellt, sind die sechs Infrarotsensoren mit den GPIO-Pins des Port-Expanders verbunden: Die Empfangsdioden sind mit den Pins P1 bis P6 verbunden. Die LEDs sind alle mit Pin P7 verbunden.

Der Port-Expander muss nachfolgend also so konfiguriert werden, dass der Pin P7 als Output und die Pins P1 bis P6 als Input verwendet werden. Anschließend können die Zustände der Outputs gesetzt und die Zustände der Inputs gelesen werden. Dazu verfügt der Chip über einige Funktionsregister, deren Funktionsweise derer der GPIO-Module des Beaglebone stark ähnelt:

- | | |
|---|--|
| Register 3 – Configuration Register: | Konfiguration der Pins als Input oder Output |
| Register 1 – Output Port Register: | Definiert den Zustand der Outputs |
| Register 0 – Input Port Register: | Zeigt den Zustand der Inputs an |

Da der Chip nur über 8 GPIO-Pins verfügt, haben die Register auch nur eine Größe von jeweils 8 Bit. Dadurch sind sie ideal geeignet, um über I²C beschrieben zu werden.

In Abbildung 28 ist ein Schreibvorgang in ein solches Register dargestellt, welcher aus der Übertragung von drei Bytes besteht: address byte, command byte und data byte.

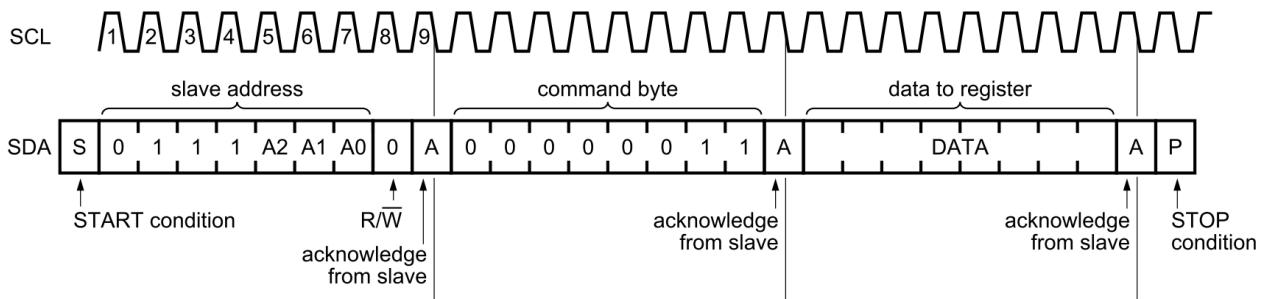


Abbildung 28: Schreibvorgang in das Konfigurationsregister; PCA9554_9554A product data sheet.
NXP Semiconductor, 03 2013.

Die I²C-**Adresse** des Slaves ergibt sich aus den Pins A0 bis A2. Durch die feste Verschaltung mit GND (siehe Abb. 27), ist die Adresse bei uns immer **0b0111000 = 56 = 0x38**.

Das **Command Byte** enthält die Adresse des Registers, in das geschrieben werden soll. Im Beispiel ist es das Konfigurationsregister, das die Adresse **3** hat.

Das **Data Byte** enthält den Inhalt, der in das Register geschrieben werden soll. In diesem Fall ist es die bitweise Konfiguration der Pins P0 bis P7, wobei eine 1 für die Konfiguration als Input und eine 0 für die Konfiguration als Output steht.

4.2.2 Initialisierung

Um das I²C-Modul auf dem Mikrocontroller zu aktivieren und zu konfigurieren, finden Sie in der `i2c.c` folgende Funktion:

```
void initI2C()
```

Zunächst werden die I²C-Interrupts aktiviert, um die entsprechenden Ereignisse abzufangen. Um I²C zu verwenden, müssen außerdem die beiden Signale (SDA und SCL) auf die richtigen Pins gemuxt werden (siehe Datenblatt Cape. Hinweis: `conf_spi0_d1` und `conf_spi0_cs0` Register).

Zur Verwendung muss dem internen I²C-Modul die Möglichkeit gegeben werden, die Clock zu generieren. Dies geschieht ähnlich wie bei den Timern und dem ADC-Modul im zentralen *Clock Module* (CM_PER).

```
HWREG(SOC_PRCM_REGS + CM_PER_I2C1_CLKCTRL) |=  
    CM_PER_I2C1_CLKCTRL_MODULEMODE_ENABLE;
```

Daraufhin muss das I²C-Modul zunächst deaktiviert werden, um daran Einstellungen vornehmen zu können. (siehe Datenblatt I2C_CON Register.)

Das Signal der Clock ist viel zu schnell für das I²C-Modul. Es muss daher ein Prescaler genutzt werden, der die Geschwindigkeit auf ca. 12MHz reduziert. Die verbundene Clock hat 48MHz, weshalb eine 0x3 in das Prescaler-Register geschrieben werden muss (entspricht einem Prescaler von 4 → siehe Register I2C_PSC).

Nun muss noch die Geschwindigkeit des Protokolls definiert werden. Jeweils die High Time und die Low Time werden in Pulsen der geprescalten Clock berechnet. Dazu müssen die Register im Modul gesetzt werden (siehe Datenblatt I2C). Wir nutzen in unserem Programm eine Bus-Geschwindigkeit von 400kHz und müssen dafür eine 8 in das I2C_SCLL und eine 10 in das I2C_SCLH Register schreiben (Es gibt in beiden Registern unterschiedliche Formeln, wie die Low- und High-Time berechnet werden).

Zuletzt muss das Modul wieder aktiviert werden. (siehe Datenblatt I2C Control-Register.)

4.2.3 Daten senden

Um Daten auf dem Bus zu senden, finden sie in der `i2c.c` folgende Funktion:

```
void writetoI2C(unsigned int sla_address, unsigned char*data, unsigned char count,  
char stp)
```

Als Eingabeparameter erhält die Funktion mehrere Werte:

- `sla_address` → Die Adresse des Slaves, an den gesendet wird
- `data` → Die Daten, die gesendet werden. Da mehrere Bytes gesendet werden können, erhält die Funktion einen Pointer auf ein Array des Typs `char` (1 `char` = 1 byte).

- `count` → Anzahl der zu sendenden Datenbytes. Da die Funktion nicht weiß, wie viele Bytes aus dem `data`-Array gesendet werden sollen, muss die Anzahl hier festgelegt werden. Das Adressbyte wird dabei nicht mitgezählt, da es nicht im `data`-Array steht.
- `stp` → Stop Condition. Für `stp = 1` wird die Stop Condition am Ende der Datenübertragung gesetzt, für `stp = 0` nicht.

Sehen Sie sich die Implementierung der Funktion an und versuchen Sie, diese zu verstehen.

Der Master beginnt die Kommunikation mit der Adresse des Slaves und schickt dann byteweise die Daten an den Empfänger. Zunächst muss die Slave Adresse in das I2C_SA Register geschrieben werden. danach wird die Länge der Nachricht in das I2C_CNT Register geschrieben (unsigned char). Die Count-Variable zählt die noch zu sendenden Bytes.

Um das Versenden zu starten, wird der Sendemodus aktiviert (I2C_CON siehe Datenblatt) und der Transmit-ready-interrupt aktiviert (I2C_IRQENABLE_SET Register).

Mit dem Setzen des I2C_CON_STT Startflags beginnen wir das Senden (I2C_CON Register).

Das Modul versendet nun immer die Daten, die im Register I2C_DATA abgelegt sind. Sobald dieses leer ist, wird der Transmit-ready-interrupt aufgerufen. Nun legen wir das nächste zu sendende Byte in das Register und die Kommunikation geht weiter.

Die Sende-Funktion wartet, bis der Count auf null heruntergezählt hat und beendet dann die Kommunikation mit einem Stop-Signal durch Setzen des STP-Flags im I2C_CON Register.

4.2.4 Daten empfangen

Neben der Funktion zum Senden der Daten beinhaltet die Datei `i2c.c` die folgende Funktion zum Empfangen von Daten:

```
void readfromi2c(unsigned int sla_address, char* data, unsigned char count, char stp)
```

Als Eingabeparameter erhält die Funktion mehrere Werte:

- `sla_address` → Adresse des Slaves von dem Daten empfangen werden sollen.
- `data` → Pointer auf ein Array des Typs `char`, in das die empfangenen Daten geschrieben werden sollen.
- `count` → Anzahl der vom Slave erwarteten Datenbytes. Da die Funktion nicht weiß, wie viele Bytes der Slave sendet, muss die Anzahl hier festgelegt werden. Das Adressbyte wird dabei nicht mitgezählt, da es nicht vom Slave gesendet wird.
- `stp` → Stop Condition. Für `stp = 1` wird die Stop Condition am Ende der Datenübertragung gesetzt, für `stp = 0` nicht.

Das Empfangen der Daten ist ähnlich dem Senden der Daten. Die Kommunikation beginnt wieder mit dem Senden der Slave-Adresse, nur dass die `readfromi2c()`-Funktion dieses mal mit dem Read/Write-Bit einen Lesewunsch festlegt. Die Implementierung der Funktion ist dementsprechend ähnlich der `writetoi2c()`:

Zunächst wird wieder die Adresse gesetzt, die Menge der zu lesenden Daten in das I2C_CNT Register gelegt und die globalen Variablen und Interrupts auf null gesetzt. Zum Aktivieren des Lesebetriebs muss, anders als beim Schreiben, kein extra Flag gesetzt werden. Es reicht aus, wenn man im I2C_CON Register das Modul als Master definiert und aktiviert. Zusätzlich muss ebenfalls der Interrupt aktiviert werden, der das Empfangen eines Bytes anzeigen (receive-Ready im I2C_IRQENABLE_SET).

Auch das Empfangen wird mit dem Start-Flag gestartet. Jedes Mal wenn ein ganzes Byte empfangen wurde, wird der **receive-Ready Interrupt** ausgelöst und die Daten im I2C_DATA Register können innerhalb der `I2CIsr()` Interrupt Service Routine in das datafromslave Array kopiert werden. Die Funktion `readfromi2c()` wartet, bis die erwartete Anzahl an Bytes gelesen ist, um dann den gelesenen Wert in ihren Ausgabearray zu kopieren.

4.2.5 Sensoren auslesen

Um die IR-Sensoren auszulesen, müssen zunächst die IR-LEDs eingeschaltet sein. Anschließend kann das Input-Port-Register des Port-Expanders ausgelesen werden. Das Auslesen eines Registers funktioniert ähnlich wie das Schreiben, besteht jedoch aus einem Schreib- und einem Lesebefehl. Denn bevor der Port-Expander einem Lesebefehl folgen kann, muss über einen Schreibbefehl das entsprechende Register ausgewählt worden sein (siehe Abb. 29).

Zuletzt müssen die LEDs wieder abgeschaltet werden, damit der Sensor genügend Kontrast bekommt. Leuchten die LEDs ununterbrochen, können die Sensoren nicht mehr unterscheiden, ob sie IR-Licht aus der Umwelt oder von der LED empfangen. **Warten Sie mindestens 1 ms**, bevor Sie die Sensoren erneut auslesen.

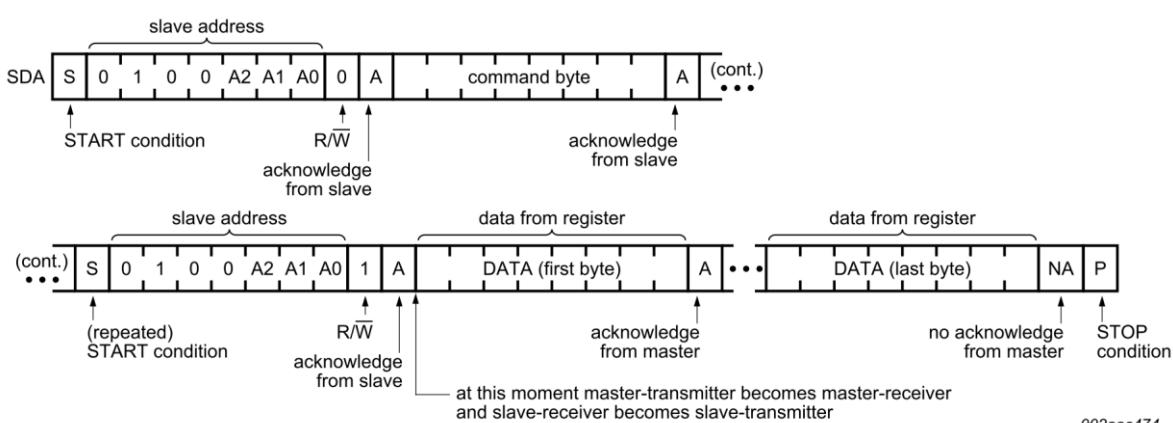


Abbildung 29: Lesevorgang eines Registers des Port-Expanders; PCA9554_9554A product data sheet. NXP Semiconductor, 03 2013.

4.3 Binäre Bildschirmausgabe

Mit Hilfe der beiden Makros BYTETOBINARYPATTERN und BYTETOBYTE(byte) (definiert in **i2c.h**) kann man sich 1 Byte Daten als binäre Werte über UARTprintf() ausgeben.

```
#define BYTETOBINARYPATTERN "%d%d%d%d%d%d%d"
#define BYTETOBYTE( byte ) \
    ( byte & 0x80 ? 1 : 0 ), \
    ( byte & 0x40 ? 1 : 0 ), \
    ( byte & 0x20 ? 1 : 0 ), \
    ( byte & 0x10 ? 1 : 0 ), \
    ( byte & 0x08 ? 1 : 0 ), \
    ( byte & 0x04 ? 1 : 0 ), \
    ( byte & 0x02 ? 1 : 0 ), \
    ( byte & 0x01 ? 1 : 0 )
```

Dadurch kann man sich z.B. das Sensorbyte, welches man über I²C empfangen hat, sehr schön leserlich auf dem Bildschirm ausgeben lassen:

```
UARTprintf (BYTETOBINARYPATTERN, BYTETOBYTE(byte));
```

4.4 Überprüfen der Kommunikation mittels Logic-Analyzer

Falls Ihre Kommunikation auch nach mehrmaligem Versuch nicht funktioniert oder Sie unerklärliche Werte erhalten und nicht wissen, woran es liegt, könnte es hilfreich sein, die tatsächlich auf dem Bus gesendeten Daten einmal live „mitzuhören“. Genau dafür gibt es sogenannte **Logic-Analyzer**, welche Sie sich bei den Tutoren abholen können.

Ein Logic-Analyzer ist ein Messgerät, dass den zeitlichen Verlauf elektrischer Signale aufzeichneten und bildlich darstellen kann. Im Gegensatz zu Oszilloskopen bietet er einen viel geringeren Funktionsumfang, ist jedoch perfekt für Test und Fehlersuche digitaler Kommunikation geeignet. Er versteht verschiedene Datenprotokolle und kann dadurch direkt die gesendeten Daten anzeigen.

Die beiden I²C-Kommunikationsleitungen sind auf der Oberseite des EGR-Capes einzeln hergeführt. Stecken Sie Channel 0 des Logic-Analyzers an SDA und Channel 1 an SCL. Verbinden Sie ihn per USB mit Ihrem Computer.

Um den Logic-Analyzer verwenden zu können, müssen Sie sich die zugehörige Software auf der Webseite des Herstellers herunterladen: www.saleae.com

In der Software finden Sie auf der linken Seite einen Button zum Starten der Aufzeichnung (siehe Abbildung 30). Sie können die Aufzeichnung aber auch beginnen lassen, indem Sie ein Trigger-Event hinzufügen: Drücken Sie dazu an Channel 0 (SDA) auf das Symbol mit der steigenden Flanke und wählen Sie eine fallende Flanke als Trigger aus. Wenn Sie die Aufzeichnung jetzt starten, wartet der Analyzer zunächst auf eine fallende Flanke an SDA, was der Startbedingung des I²C-Protokolls entspricht.

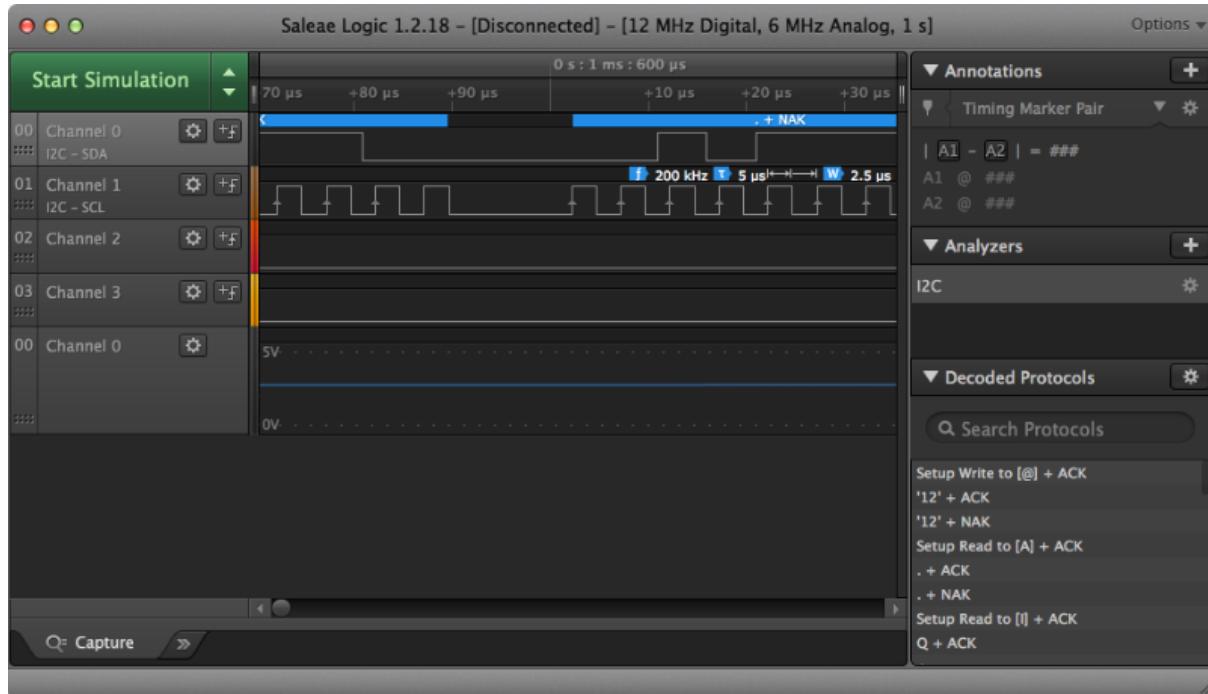


Abbildung 30: Screenshot der Auswertung einer I²C-Kommunikation mittels Logic-Analyzer

Wenn Sie erfolgreich eine Kommunikation aufzeichnen konnten, sehen Sie den zeitlichen Verlauf der digitalen Pegel im Programm. Um an eine bestimmte Stelle zu zoomen, können Sie diese im Spannungsverlauf doppelt anklicken.

Falls Sie das I²C-Protokoll verstehen, können Sie anhand der Spannungsverläufe bereits erkennen, welche Daten gesendet wurden. Um diese Aufgabe zu erleichtern, können Sie allerdings auch einen Analyzer hinzufügen, der Ihnen die Spannungspegel in konkrete Information übersetzt: Auf der rechten Seite des Programms können Sie unter „Analyzers“ einen I²C-Analyzer hinzufügen. Danach werden Ihnen die aufgezeichneten Bytes rechts unten im Programm angezeigt. Unter dem Menüpunkt „Options“ rechts oben können Sie zudem die Anzeige der Bytes von Ascii auf Binär (Bin) umschalten. Dadurch werden Ihnen die einzelnen Bits auch im Analyzer angezeigt.

Sehen Sie nach, ob die richtige Anzahl an Bytes gesendet wird. Untersuchen Sie, ob der Inhalt von Adress-, Command- und Data-Byte korrekt ist. Überprüfen Sie, ob der Slave die Kommunikation mittels ACK bestätigt.

Kochrezept – Termin 4

I²C Schnittstelle initialisieren:

- Initialisieren indem Clocksignal durchgeschaltet wird
→ CM_PER Register im *Clock Module*
- Aktivieren der Interrupts
→ über *Int*-Funktionen (siehe Termin 2!)
- Pin-Muxen der Signale SDA & SCL
→ conf_spi0_d1 & conf_spi0_cs0 Register
- I²C Modul deaktivieren, um Einstellungen vornehmen zu können
→ I2C_CON Register
- Prescaler für Clocksignal einstellen
→ I2C_PSC Register
- Geschwindigkeit des Protokolls definieren
→ I2C_SCLL & I2C_SCLH Register
- I²C Modul aktivieren
→ I2C_CON Register

Daten senden:

- Aufbau der Kommunikation: Erst Slave-Adresse, dann byte-weise die Daten!
- Slave-Adresse festlegen
→ I2C_SA Register
- Sendemodus aktivieren
→ I2C_CON Register
- Transmitter-Ready Interrupt aktivieren
→ I2C_IRQENABLE_SET Register
- Startflag setzen um das Senden zu starten
→ I2C_CON_STT Register
- Zu sendende Daten bereit stellen
→ I2C_DATA Register
- Stopflag setzen, um Senden zu beenden
→ I2C_CON Register



Daten empfangen:

- Slave-Adresse festlegen
→ I2C_SA Register
- Menge der zu lesenden Daten festlegen
→ I2C_CNT Register
- Empfängermodus aktivieren
→ I2C_CON Register
- Receive-Ready Interrupt aktivieren
→ I2C_IRQENABLE_SET Register
- Startflag setzen um die Kommunikation zu starten
→ I2C_CON_STT Register
- Empfangene Daten weiter verarbeiten
→ Mittels I2CISr () empfangenen Daten aus I2C_DATA Register auslesen & weiterverarbeiten.



Verständnisfragen

1. Was ist I²C?
2. Wofür kann I²C verwendet werden?
3. Welche Leitungen werden für I²C verwendet?
4. Woher weiß der Slave, ob er vom Bus lesen oder auf den Bus schreiben soll?
5. Wie viele Slaves kann man mit I²C maximal anschließen?
6. Wie kann ich meinem I²C Treiber signalisieren, dass ein Byte empfangen wurde?



Termin 4

- Aufgaben



4.5 Aufgaben Termin 4: Kommunikationsschnittstellen

Aufgabe 1: Senden mit I²C

Vervollständigen Sie die I²C Treiberdatei um die Initialisierungsroutine und konfigurieren Sie den Port-Expander so, dass Pin 7 als Ausgang und die restlichen Pins als Eingänge geschaltet sind.

Testen Sie eine erste Kommunikation mit dem Port-Expander, indem Sie die Infrarot-LEDs an der Spitze des Roboters blinken lassen.

(Hinweis: Da Infrarotlicht für das menschliche Auge unsichtbar ist, können Sie versuchen, die LEDs durch eine digitale Kamera zu betrachten. Achtung: einige hochwertige Smartphonekameras filtern Infrarotlicht)

Inhalt: Initialisieren von I²C und Verwendung von Port-Expander & Logic-Analyzer

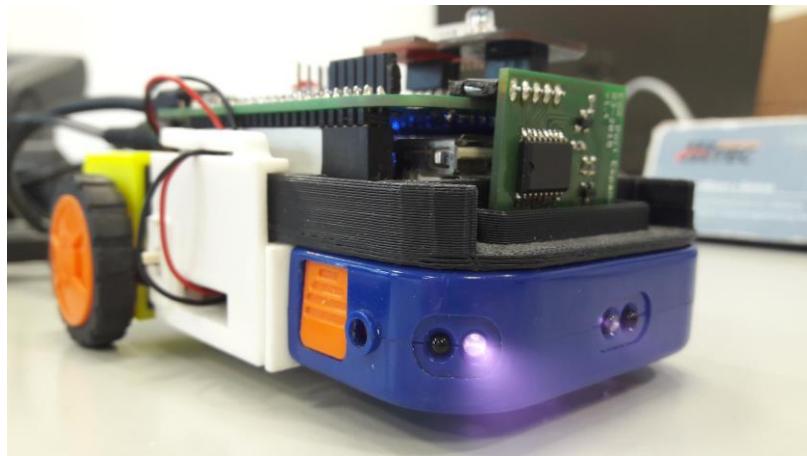


Abbildung 31: Durch Smartphonekamera sichtbare Infrarot-LEDs

Aufgabe 2: Empfangen mit I²C

Vervollständigen Sie die I²C Treiberdatei um die Empfangsroutine und lesen Sie die vom Port-Expander gesendeten Daten ein.

Geben Sie die gelesenen Daten binär auf der Konsole aus.

Inhalt: I²C lesen und Port-Expander

Aufgabe 3: Auslesen der IR-Sensoren

Schreiben Sie sich einen Treiber „Sensor.h“, der eine Funktion zum Auslesen der IR-Sensoren enthält. Gestalten Sie Ihren Treiber möglichst benutzerfreundlich, sodass Sie ihn zum Lesen der Sensoren in Kapitel 5 wieder einsetzen können.

Inhalt: IR-Sensoren





Termin 5

- Olympiade



5. Termin: Olympiade

In diesem Aufgabenblatt treten die Roboter in unterschiedlichen Wettkämpfen gegeneinander an. Die einzelnen Disziplinen sollen in getrennten Programmen implementiert werden.

Disziplin 1: Fahren entlang einer Linie

Ihr Programm sollte den Roboter möglichst elegant entlang einer leicht geschwungenen schwarzen Linie manövrieren.

Disziplin 2: Rennen

Auf einer eigens entworfenen Rennstrecke findet ein Rennen statt.

- Ein **Qualifying** legt das Teilnehmerfeld und die Startreihenfolge fest
 - Jede/r Teilnehmer/in hat drei Versuche, es wird jeweils die Zeit gestoppt, die für eine Runde auf der Rennstrecke benötigt wird.
 - Wird die Runde nicht zu Ende gefahren, gilt der Versuch als Fehlversuch.
 - Der schnellste der drei Versuche zählt.
- Das Verlassen der Strecke wird mit der Disqualifikation geahndet (Hierbei gilt der Grundsatz, dass der Roboter die Strecke verlassen hat, sobald er „vollständig“ über die schwarze Linie gefahren ist.).
- Als Sieger gilt, wer als erstes **4 Runden** hinter sich gebracht hat.
- Karosseriekontakt ist ausdrücklich erlaubt.
- Die Roboter beginnen auf Tastendruck nach einem Countdown von **5 Sekunden** mit der Fahrt (mit Timer implementieren).





Auszug Datenblätter

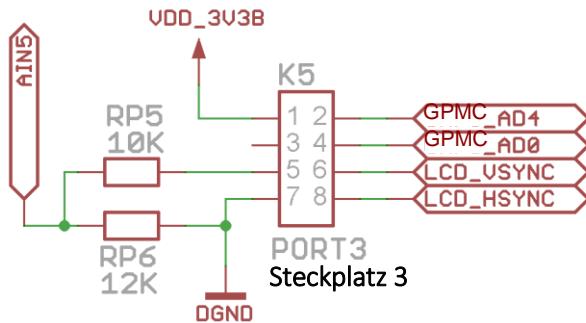
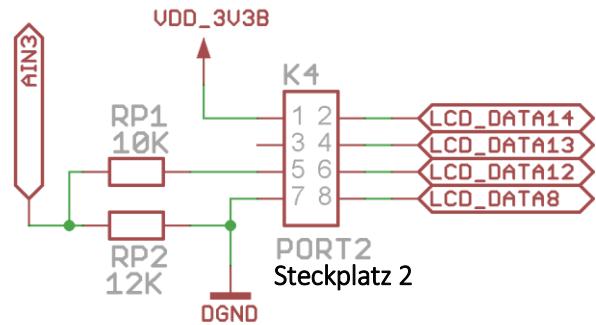
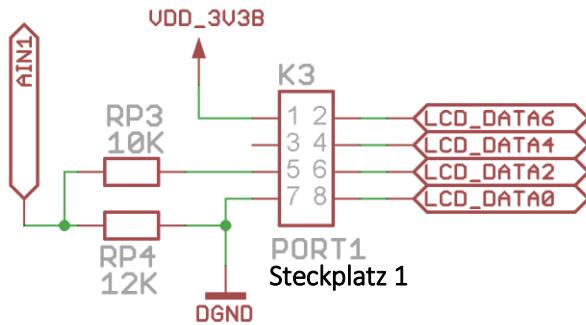
Hinweis:

- Nicht selbst erstellte Datenblätter sind freundlicherweise zur Verfügung gestellt von Texas Instruments.

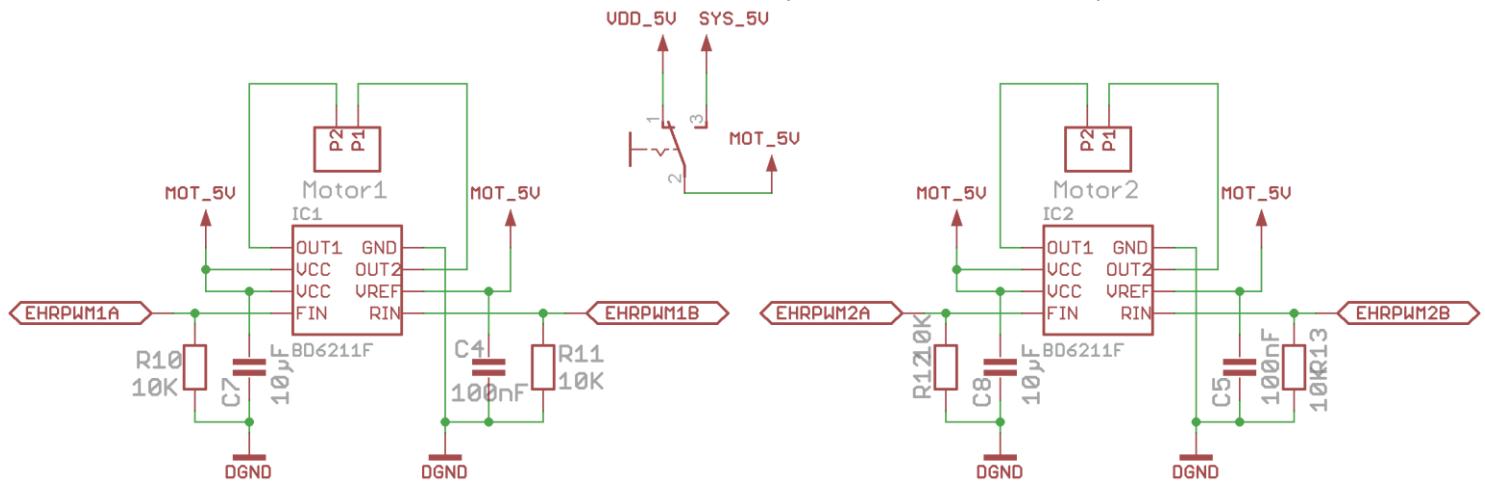


6. Datenblatt des EGR-Capes

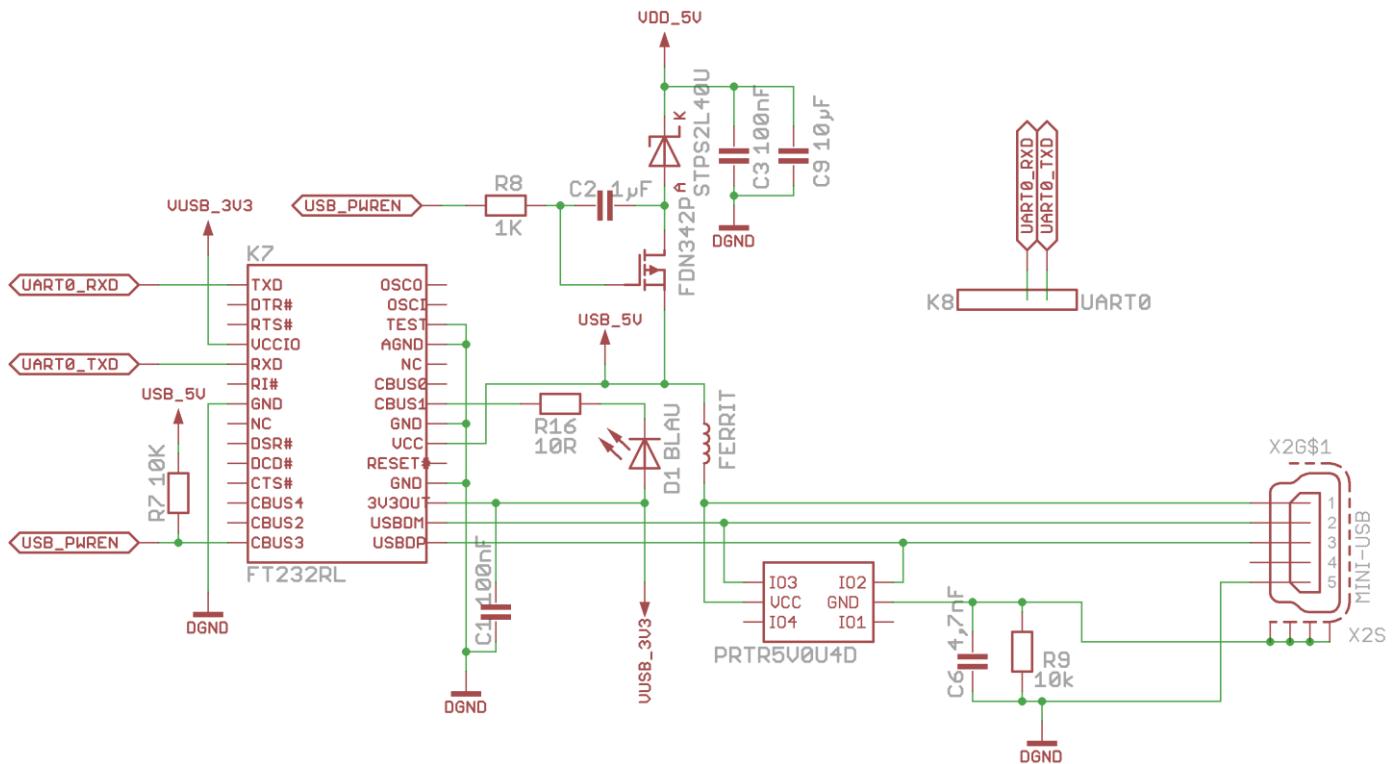
CAPE-STECKPLÄTZE



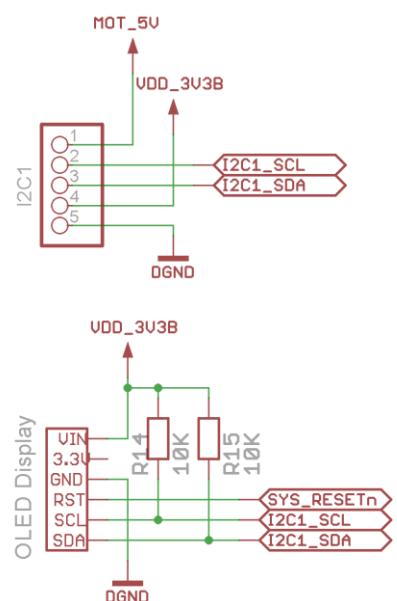
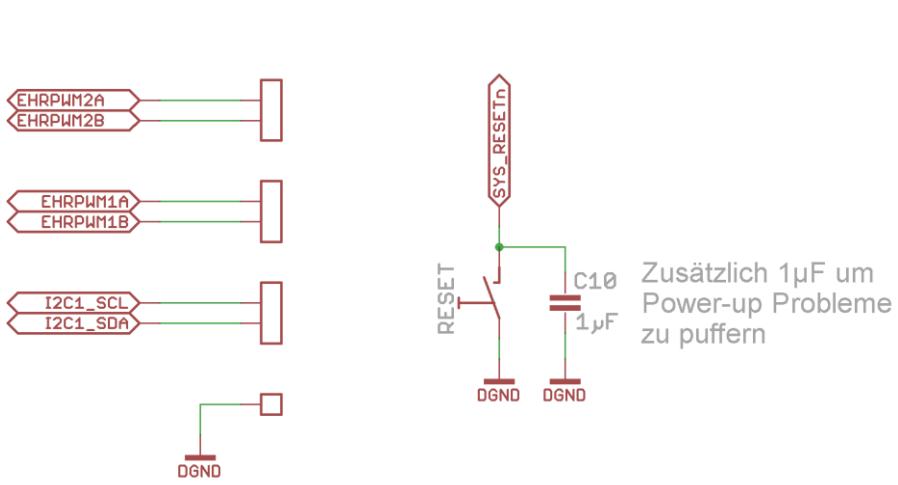
MOTOR-TREIBER (H-BRÜCKEN)



USB-FTDI-SCHALTUNG



OLED-DISPLAY / I²C / SYSTEM-RESET



Übersicht:

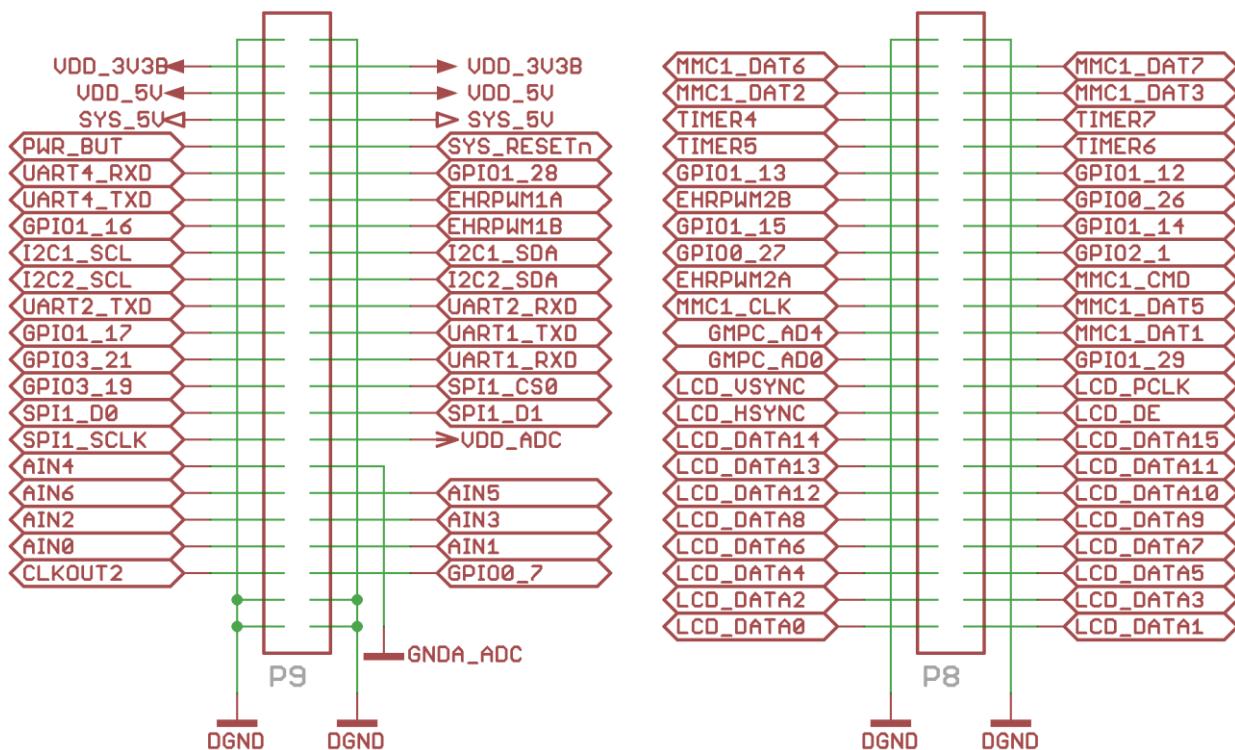
Verfügbare Beaglebone Pins

Modul steck- platz	Pin	Pin Name	Signal Name							
			Mux-Mode							
			0	1	2	3	4	5	6	7
Steck- platz 1	2	LCD_DATA6	lcd_da ta6	gpmc_ac 6	Pr1_edio _...	eQEP2_in- dex	Pr1_ed io...	Pr1_pru 1_...	Pr1_pru 1_...	gpio2_12
Steck- platz 1	4	LCD_DATA4	lcd_da ta4	gpmc_a4	Pr1_mii0 txd1	eQEP2A_i n		Pr1_pru 1_...	Pr1_pru 1_...	gpio2_10
Steck- platz 1	6	LCD_DATA2	lcd_da ta2	gpmc_a2		ehr- pwm2_...				gpio2_8
Steck- platz 1	8	LCD_DATA0	lcd_da ta0	gpmc_a0		ehr- pwm2A				gpio2_6
Steck- platz 2	2	LCD_DATA14	lcd_da ta14	gpmc_a1 8	eQEP1_i ndex	mcasp0_a xr1	uart5_ rx		uart5_ct sn	gpio0_10
Steck- platz 2	4	LCD_DATA13	lcd_da ta13	gpmc_a1 7	eQEP1B _in	mcasp0_fs r	mcasp 0_axr3		uart4_rt sn	gpio0_9
Steck- platz 2	6	LCD_DATA12	lcd_da ta12	gpmc_a1 6	eQEP1A _in	mcasp0_a clk	mcasp 0_axr2		uart4_ct sn	gpio0_8
Steck- platz 2	8	LCD_DATA8	lcd_da ta8	gpmc_a1 2	Ehr- pwm1_t ripzo...	mcasp0_a clkx	uart5_ tx		uart2_ct sn	gpio2_14
Steck- platz 3	2	GPMC_AD4	gpmc_ ad4	mmc1_d at4						gpio1_4
Steck- platz 3	4	GPMC_ADO	gpmc_ ad0	mmc1_d at0						gpio1_0
Steck- platz 3	6	LCD_VSYNC	lcd_vsy nc	gpmc_a8						gpio2_22
Steck- platz 3	8	LCD_HSYNC	lcd_hs ync	gpmc_a9						gpio2_23

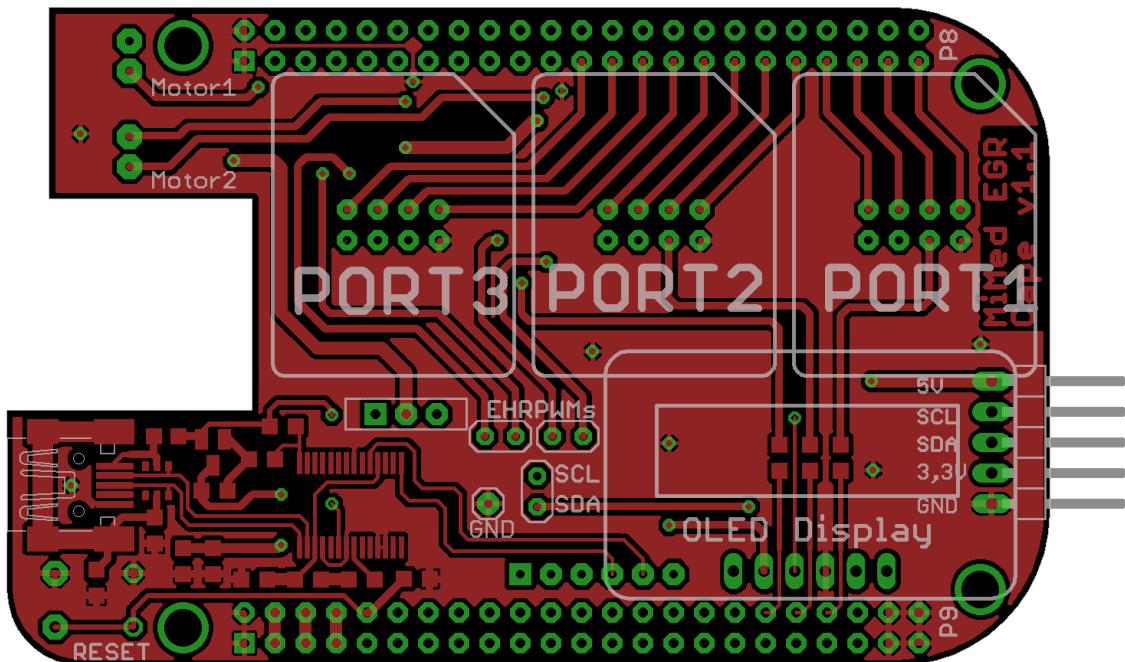
Modul steckplatz	Pin	Pin Name	Signal Name							
			Mux-Mode							
			0	1	2	3	4	5	6	7
MOTOR 1	1	GPMC_A2	gpmc_a2	mii2_txd3	rgmii2_td3	mmc2_dat1	gpmc_a18		ehr-pwm1A	gpio1_18
MOTOR 1	2	GPMC_A3	gpmc_a3	mii2_txd2	rgmii2_td2	mmc2_dat2	gpmc_a19		ehr-pwm1B	gpio1_19
MOTOR 2	1	GPMC_AD8	gpmc_ad8	lcd_data23	mmc1_dat0	mmc2_dat4	ehr-pwm2A			gpio0_22
MOTOR 2	2	GPMC_AD9	gpmc_ad9	lcd_data22	mmc1_dat2	mmc2_dat5	ehr-pwm2B			gpio0_23

Modul steckplatz	Pin Name	Signal Name								
		Mux-Mode								
		0	1	2	3	4	5	6	7	
I2C1_SCL		spi0_cs0	spi0_cs0	mmc2_sdwp	I2C1_SCL	ehr-pwm0_synci	pr1_uart0_txd			gpio0_5
I2C1_SDA		spi0_d1	spi0_d1	mmc1_sdwp	I2C1_SDA	ehr-pwm0_trigzone	pr1_uart0_rxd			gpio0_4

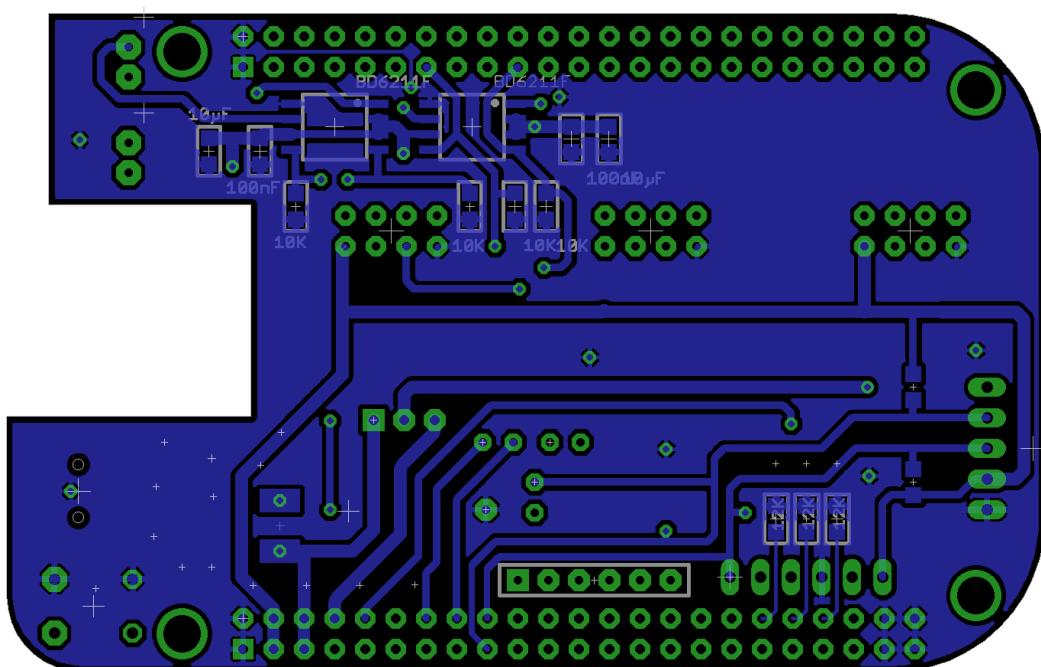
Pin Header P9 und P8



Cape Layout - Oberseite



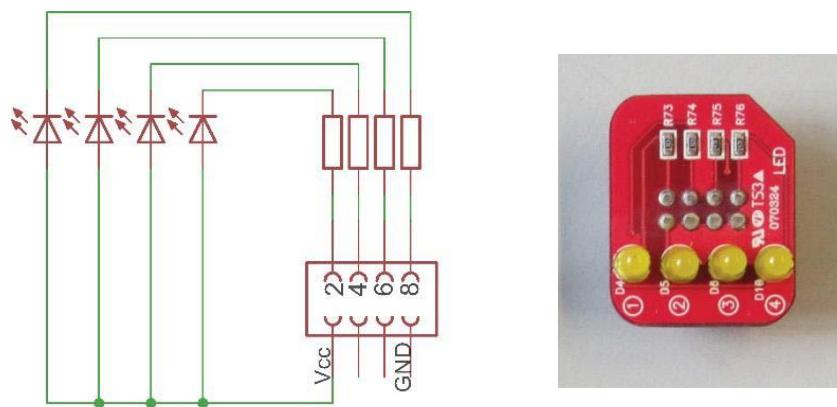
Cape Layout - Unterseite



7. Moduldatenblätter

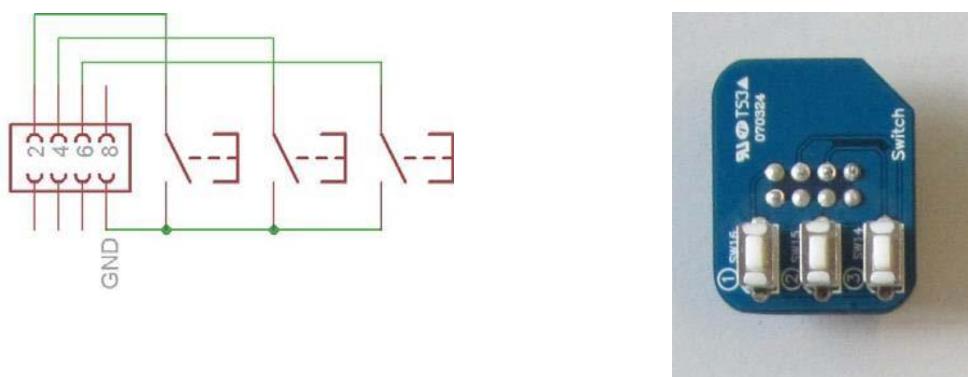
1. LED-Modul

Das LED-Modul ist ein optisches Ausgabemodul welches von Beaglebone gesteuert werden kann. Des Weiteren ist für jede LED ein Widerstand vorgesehen, der den Strom für die LED begrenzt.



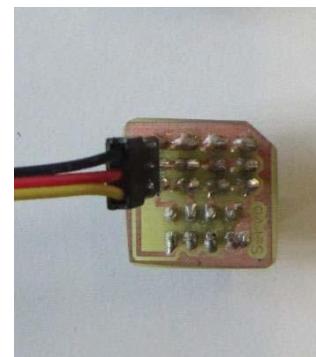
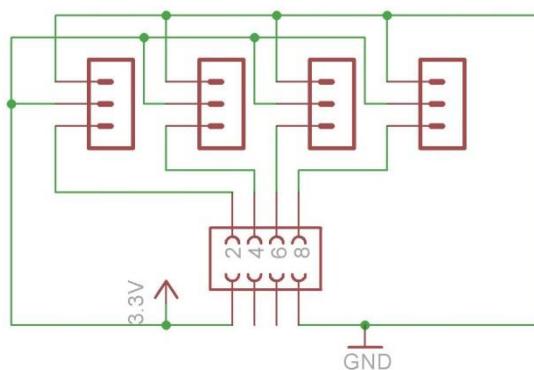
2. Taster-Modul

Auf dem Taster-Modul sind drei Taster aufgebracht. Diese sind als active-low Taster verschaltet. Über einen Pullup-Widerstand (damit bei Tastendruck kein Kurzschluss entsteht) liegt der HIGH-Pegel direkt am μ C an. Wird nun ein Taster betätigt wird der Strom über den Taster direkt auf Masse gezogen und ein LOW-Pegel liegt am μ C an.



3. Servomodul

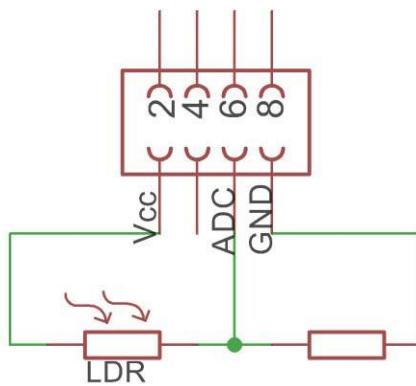
Ein Servo-Antrieb wird über PWM gesteuert. Dazu werden 1-2ms Impulse im Abstand von 20ms gesendet. Die Pulsdauer ist direkt proportional zum Winkel des Motors. Also 1ms-Impuls maximale Linksstellung, 2ms-Impuls maximale Rechtsstellung und 1,5ms Mittelstellung. Der Anschluss des Servos besteht aus den drei Leitungen, Vcc, GND und die Leitung, auf der die Steuerimpulse gesendet werden.



- | | |
|------------------|-------------------------------|
| Schwarze Leitung | ist GND |
| Gelbe Leitung | ist Leitung für Steuerimpulse |
| Rote Leitung | ist Vcc (ca. 3-5V) |

4. LDR-Modul

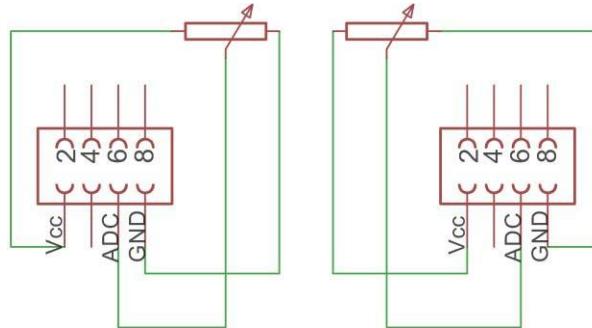
Das LDR-Modul ist ein Sensormodul, welches auf Lichtstärke reagiert. Ein LDR (Light Dependent Resistor) ist ein lichtabhängiger elektrischer Widerstand. Er wird in einem Spannungsteiler verschaltet. Das Modul liefert eine Spannung in Abhängigkeit von der Lichtstärke zurück.



Die Sensorspannung kann über einen Analog/Digital Converter (ADC) des Prozessors eingelesen und verarbeitet werden.

5. Joystick-Modul

Das Joystickmodul besteht aus zwei Teilen: Dem Joystick selbst, welcher über die Potentiometer analoge Spannungen erzeugt, sowie aus der Anschlussplatine für das Beaglebone. Diese beiden Teile werden über ein 6-adriges Kabel verbunden.



Der Joystick selber besteht im Grunde aus zwei Potentiometern, die jeweils am zugehörigen ADC- Eingang ausgelesen werden können.

8.

Memory Map (Basisadressen)

Memory Map

This section describes the memory map for the device.

2.1 ARM Cortex-A8 Memory Map

Table 2-1. L3 Memory Map

Block Name	Start_address (hex)	End_address (hex)	Size	Description
GPMC (Ex- ternal Memory)	0x0000_0000 ⁽¹⁾	0x1FFF_FFFF	512MB	8-/16-bit External Memory (Ex/R/W) ⁽²⁾
Reserved	0x2000_0000	0x3FFF_FFFF	512MB	Reserved
Boot ROM	0x4000_0000	0x4001_FFFF	128KB	
	0x4002_0000	0x4002_BFFF	48KB	32-bit Ex/R ⁽²⁾ – Public
Reserved	0x4002_C000	0x400F_FFFF	848KB	Reserved
Reserved	0x4010_0000	0x401F_FFFF	1MB	Reserved
Reserved	0x4020_0000	0x402E_FFFF	960KB	Reserved
Reserved	0x402F_0000	0x402F_03FF	64KB	Reserved
SRAM internal	0x402F_0400	0x402F_FFFF		32-bit Ex/R/W ⁽²⁾
L3 OCMC0	0x4030_0000	0x4030_FFFF	64KB	32-bit Ex/R/W ⁽²⁾ OCMC SRAM
Reserved	0x4031_0000	0x403F_FFFF	960KB	Reserved
Reserved	0x4040_0000	0x4041_FFFF	128KB	Reserved
Reserved	0x4042_0000	0x404F_FFFF	896KB	Reserved
Reserved	0x4050_0000	0x405F_FFFF	1MB	Reserved
Reserved	0x4060_0000	0x407F_FFFF	2MB	Reserved
Reserved	0x4080_0000	0x4083_FFFF	256KB	Reserved
Reserved	0x4084_0000	0x40DF_FFFF	5888KB	Reserved
Reserved	0x40E0_0000	0x40E0_7FFF	32KB	Reserved
Reserved	0x40E0_8000	0x40EF_FFFF	992KB	Reserved
Reserved	0x40F0_0000	0x40F0_7FFF	32KB	Reserved
Reserved	0x40F0_8000	0x40FF_FFFF	992KB	Reserved
Reserved	0x4100_0000	0x41FF_FFFF	16MB	Reserved
Reserved	0x4200_0000	0x43FF_FFFF	32MB	Reserved
L3F CFG Regs	0x4400_0000	0x443F_FFFF	4MB	L3Fast configuration registers
Reserved	0x4440_0000	0x447F_FFFF	4MB	Reserved
L3S CFG Regs	0x4480_0000	0x44BF_FFFF	4MB	L3Slow configuration registers
<u>L4_WKUP</u>	0x44C0_0000	0x44FF_FFFF	4MB	L4_WKUP
Reserved	0x4500_0000	0x45FF_FFFF	16MB	Reserved
<u>McASP0Data</u>	0x4600_0000	0x463F_FFFF	4MB	McASP0 Data Registers
<u>McASP1Data</u>	0x4640_0000	0x467F_FFFF	4MB	McASP1 Data Registers
Reserved	0x4680_0000	0x46FF_FFFF	8MB	Reserved
Reserved	0x4700_0000	0x473F_FFFF	4MB	Reserved

⁽¹⁾ The first 1MB of address space 0x0-0xFFFF is inaccessible externally.

⁽²⁾ Ex/R/W – Execute/Read/Write.

Table 2-1. L3 Memory Map (continued)

Block Name	Start_address (hex)	End_address (hex)	Size	Description
<u>USBSS</u>	0x4740_0000	0x4740_0FFF	20KB	USB Subsystem Registers
<u>USB0</u>	0x4740_1000	0x4740_12FF		USB0 Controller Registers
<u>USB0_PHY</u>	0x4740_1300	0x4740_13FF		USB0 PHY Registers
USB0 Core	0x4740_1400	0x4740_17FF		USB0 Core Registers
<u>USB1</u>	0x4740_1800	0x4740_1AFF		USB1 Controller Registers
<u>USB1_PHY</u>	0x4740_1B00	0x4740_1BFF		USB1 PHY Registers
USB1 Core	0x4740_1C00	0x4740_1FFF		USB1 Core Registers
<u>USB_CPII DMA Controller</u>	0x4740_2000	0x4740_2FFF		USB CPII DMA Controller Registers
<u>USB_CPII DMA Scheduler</u>	0x4740_3000	0x4740_3FFF		USB CPII DMA Scheduler Registers
<u>USB Queue Manager</u>	0x4740_4000	0x4740_4FFF		USB Queue Manager Registers
Reserved	0x4740_5000	0x477F_FFFF	4MB-20KB	Reserved
Reserved	0x4780_0000	0x4780_FFFF	64KB	Reserved
<u>MMCHS2</u>	0x4781_0000	0x4781_FFFF	64KB	MMCHS2
Reserved	0x4782_0000	0x47BF_FFFF	4MB-128KB	Reserved
Reserved	0x47C0_0000	0x47FF_FFFF	4MB	Reserved
<u>L4_PER</u>	0x4800_0000	0x48FF_FFFF	16MB	L4 Peripheral (see L4_PER table)
<u>TPCC</u> (EDMA3CC)	0x4900_0000	0x490F_FFFF	1MB	EDMA3 Channel Controller Registers
Reserved	0x4910_0000	0x497F_FFFF	7MB	Reserved
<u>PTPC0</u> (EDMA3TC0)	0x4980_0000	0x498F_FFFF	1MB	EDMA3 Transfer Controller 0 Registers
<u>PTPC1</u> (EDMA3TC1)	0x4990_0000	0x499F_FFFF	1MB	EDMA3 Transfer Controller 1 Registers
<u>PTPC2</u> (EDMA3TC2)	0x49A0_0000	0x49AF_FFFF	1MB	EDMA3 Transfer Controller 2 Registers
Reserved	0x49B0_0000	0x49BF_FFFF	1MB	Reserved
Reserved	0x49C0_0000	0x49FF_FFFF	4MB	Reserved
<u>L4_FAST</u>	0x4A00_0000	0x4AFF_FFFF	16MB	L4_FAST
<u>DebugSS</u>	0x4B00_0000	0x4BFF_FFFF	16MB	Debug Subsystem region
<u>EMIFO</u>	0x4C00_0000	0x4CFF_FFFF	16MB	EMIFO Configuration registers
Reserved	0x4D00_0000	0x4dff_FFFF	16MB	Reserved
Reserved	0x4E00_0000	0x4FFF_FFFF	32MB	Reserved
<u>GPMC</u>	0x5000_0000	0x50FF_FFFF	16MB	GPMC Configuration registers
Reserved	0x5100_0000	0x52FF_FFFF	32MB	Reserved
<u>Reserved</u>	0x5300_0000	0x530F_FFFF	1MB	Reserved
	0x5310_0000	0x531F_FFFF	1MB	Reserved
Reserved	0x5320_0000	0x533F_FFFF	2MB	Reserved
<u>Reserved</u>	0x5340_0000	0x534F_FFFF	1MB	Reserved
	0x5350_0000	0x535F_FFFF	1MB	Reserved
Reserved	0x5360_0000	0x54BF_FFFF	22MB	Reserved
<u>ADC_TSC DMA</u>	0x54C0_0000	0x54FF_FFFF	4MB	ADC_TSC DMA Port
Reserved	0x5500_0000	0x55FF_FFFF	16MB	Reserved
<u>SGX530</u>	0x5600_0000	0x56FF_FFFF	16MB	SGX530 Slave Port
Reserved	0x5700_0000	0x57FF_FFFF	16MB	Reserved
Reserved	0x5800_0000	0x58FF_FFFF	16MB	Reserved
Reserved	0x5900_0000	0x59FF_FFFF	16MB	Reserved
Reserved	0x5A00_0000	0x5AFF_FFFF	16MB	Reserved

Table 2-1. L3 Memory Map (continued)

Block Name	Start_address (hex)	End_address (hex)	Size	Description
Reserved	0x5B00_0000	0x5BFF_FFFF	16MB	Reserved
Reserved	0x5C00_0000	0x5DFF_FFFF	32MB	Reserved
Reserved	0x5E00_0000	0x5FFF_FFFF	32MB	Reserved
Reserved	0x6000_0000	0x7FFF_FFFF	512MB	Reserved
EMIF0 SDRAM	0x8000_0000	0xBFFF_FFFF	1GB	8-/16-bit External Memory (Ex/RW) ⁽³⁾
Reserved	0xC000_0000	0xFFFF_FFFF	1GB	Reserved

⁽³⁾ Ex/R/W – Execute/Read/Write.

Table 2-2. L4_WKUP Peripheral Memory Map

Region Name	Start Address (hex)	End Address (hex)	Size	Description
L4_WKUP configuration	0x44C0_0000	0x44C0_07FF	2KB	Address/Protection (AP)
	0x44C0_0800	0x44C0_0FFF	2KB	Link Agent (LA)
	0x44C0_1000	0x44C0_13FF	1KB	Initiator Port (IP0)
	0x44C0_1400	0x44C0_17FF	1KB	Initiator Port (IP1)
Reserved	0x44C0_1800	0x44C0_1FFF	2KB	Reserved (IP2 – IP3)
Reserved	0x44C0_2000	0x44CF_FFFF	1MB-8KB	Reserved
Reserved	0x44D0_0000	0x44D0_3FFF	16KB	Reserved
	0x44D0_4000	0x44D0_4FFF	4KB	Reserved
Reserved	0x44D0_5000	0x44D7_FFFF	492KB	Reserved
Reserved	0x44D8_0000	0x44D8_1FFF	8KB	Reserved
	0x44D8_2000	0x44D8_2FFF	4KB	Reserved
Reserved	0x44D8_3000	0x44DF_FFFF	500KB	Reserved
<u>CM_PER</u>	0x44E0_0000	0x44E0_3FFF	1KB	Clock Module Peripheral Registers
<u>CM_WKUP</u>	0x44E0_0400	0x44E0_04FF	256 Bytes	Clock Module Wakeup Registers
<u>CM_DPLL</u>	0x44E0_0500	0x44E0_05FF	256 Bytes	Clock Module PLL Registers
<u>CM_MPU</u>	0x44E0_0600	0x44E0_06FF	256 Bytes	Clock Module MPU Registers
<u>CM_DEVICE</u>	0x44E0_0700	0x44E0_07FF	256 Bytes	Clock Module Device Registers
<u>CM_RTC</u>	0x44E0_0800	0x44E0_08FF	256 Bytes	Clock Module RTC Registers
<u>CM_GFX</u>	0x44E0_0900	0x44E0_09FF	256 Bytes	Clock Module Graphics Controller Registers
<u>CM_CEFUSE</u>	0x44E0_0A00	0x44E0_0AFF	256 Bytes	Clock Module Efuse Registers
<u>PRM_IRQ</u>	0x44E0_0B00	0x44E0_0BFF	256 Bytes	Power Reset Module Interrupt Registers
<u>PRM_PER</u>	0x44E0_0C00	0x44E0_0CFF	256 Bytes	Power Reset Module Peripheral Registers
<u>PRM_WKUP</u>	0x44E0_0D00	0x44E0_0DFF	256 Bytes	Power Reset Module Wakeup Registers
<u>PRM_MPU</u>	0x44E0_0E00	0x44E0_0EFF	256 Bytes	Power Reset Module MPU Registers
<u>PRM_DEVICE</u>	0x44E0_0F00	0x44E0_0FFF	256 Bytes	Power Reset Module Device Registers
<u>PRM_RTC</u>	0x44E0_1000	0x44E0_10FF	256 Bytes	Power Reset Module RTC Registers
<u>PRM_GFX</u>	0x44E0_1100	0x44E0_11FF	256 Bytes	Power Reset Module Graphics Controller Registers
<u>PRM_CEFUSE</u>	0x44E0_1200	0x44E0_12FF	256 Bytes	Power Reset Module Efuse Registers
Reserved	0x44E0_3000	0x44E0_3FFF	4KB	Reserved
	0x44E0_4000	0x44E0_4FFF	4KB	Reserved

Table 2-2. L4_WKUP Peripheral Memory Map (continued)

Region Name	Start Address (hex)	End Address (hex)	Size	Description
<u>DMTMR0</u>	0x44E0_5000	0x44E0_5FFF	4KB	DMTimer0 Registers
	0x44E0_6000	0x44E0_6FFF	4KB	Reserved
<u>GPIO0</u>	0x44E0_7000	0x44E0_7FFF	4KB	GPIO Registers
	0x44E0_8000	0x44E0_8FFF	4KB	Reserved
<u>UART0</u>	0x44E0_9000	0x44E0_9FFF	4KB	UART Registers
	0x44E0_A000	0x44E0_AFFF	4KB	Reserved
<u>I2C0</u>	0x44E0_B000	0x44E0_BFFF	4KB	I2C Registers
	0x44E0_C000	0x44E0_CFFF	4KB	Reserved
<u>ADC_TSC</u>	0x44E0_D000	0x44E0_EFFF	8KB	ADC_TSC Registers
	0x44E0_F000	0x44E0_FFFF	4KB	Reserved
<u>ControlModule</u>	0x44E1_0000	0x44E1_1FFF	128KB	Control Module Registers
<u>DDR2/3/mDDR PHY</u>	0x44E1_2000	0x44E1_23FF		DDR2/3/mDDR PHY Registers
Reserved	0x44E1_2400	0x44E3_0FFF	4KB	Reserved
<u>DMTMR1_1MS</u> (Accurate 1ms timer)	0x44E3_1000	0x44E3_1FFF	4KB	DMTimer1 1ms Registers
	0x44E3_2000	0x44E3_2FFF	4KB	Reserved
Reserved	0x44E3_3000	0x44E3_3FFF	4KB	Reserved
	0x44E3_4000	0x44E3_4FFF	4KB	Reserved
<u>WDT1</u>	0x44E3_5000	0x44E3_5FFF	4KB	Watchdog Timer Registers
	0x44E3_6000	0x44E3_6FFF	4KB	Reserved
SmartReflex0	0x44E3_7000	0x44E3_7FFF	4KB	L3 Registers
	0x44E3_8000	0x44E3_8FFF	4KB	Reserved
SmartReflex1	0x44E3_9000	0x44E3_9FFF	4KB	L3 Registers
	0x44E3_A000	0x44E3_AFFF	4KB	Reserved
Reserved	0x44E3_B000	0x44E3_DFFF	12KB	Reserved
<u>RTCSS</u>	0x44E3_E000	0x44E3_EFFF	4KB	RTC Registers
	0x44E3_F000	0x44E3_FFFF	4KB	Reserved
DebugSS Instrumentation HWMaster1 Port	0x44E4_0000	0x44E7_FFFF	256KB	Debug Registers
	0x44E8_0000	0x44E8_0FFF	4KB	Reserved
Reserved	0x44E8_1000	0x44EF_FFFF	508KB	Reserved
Reserved	0x44F0_0000	0x44FF_FFFF	1MB	Reserved

Table 2-3. L4_PER Peripheral Memory Map

Device Name	Start_address (hex)	End_address (hex)	Size	Description
Reserved	0x4800_0000	0x4800_07FF	2KB	Reserved
	0x4800_0800	0x4800_0FFF	2KB	Reserved
	0x4800_1000	0x4800_13FF	1KB	Reserved
	0x4800_1400	0x4800_17FF	1KB	Reserved
	0x4800_1800	0x4800_1BFF	1KB	Reserved
	0x4800_1C00	0x4800_1FFF	1KB	Reserved
Reserved	0x4800_2000	0x4800_3FFF	8KB	Reserved
Reserved	0x4800_4000	0x4800_7FFF	16KB	Reserved
Reserved	0x4800_8000	0x4800_8FFF	4KB	Reserved
	0x4800_9000	0x4800_9FFF	4KB	Reserved
Reserved	0x4800_A000	0x4800_FFFF	24KB	Reserved
Reserved	0x4801_0000	0x4801_0FFF	4KB	Reserved

Table 2-3. L4_PER Peripheral Memory Map (continued)

Device Name	Start_address (hex)	End_address (hex)	Size	Description
	0x4801_1000	0x4801_1FFF	4KB	Reserved
Reserved	0x4801_2000	0x4801_3FFF	8KB	Reserved
Reserved	0x4801_4000	0x4801_FFFF	48KB	Reserved
Reserved	0x4802_0000	0x4802_0FFF	4KB	Reserved
	0x4802_1000	0x4802_1FFF	4KB	Reserved
<u>UART1</u>	0x4802_2000	0x4802_2FFF	4KB	UART1 Registers
	0x4802_3000	0x4802_3FFF	4KB	Reserved
<u>UART2</u>	0x4802_4000	0x4802_4FFF	4KB	UART2 Registers
	0x4802_5000	0x4802_5FFF	4KB	Reserved
Reserved	0x4802_6000	0x4802_7FFF	8KB	Reserved
Reserved	0x4802_8000	0x4802_8FFF	4KB	Reserved
	0x4802_9000	0x4802_9FFF	4KB	Reserved
<u>I2C1</u>	0x4802_A000	0x4802_AFFF	4KB	I2C1 Registers
	0x4802_B000	0x4802_BFFF	4KB	Reserved
Reserved	0x4802_C000	0x4802_CFFF	4KB	Reserved
	0x4802_D000	0x4802_DFFF	4KB	Reserved
Reserved	0x4802_E000	0x4802_EFFF	4KB	Reserved
	0x4802_F000	0x4802_FFFF	4KB	Reserved
<u>McSPI0</u>	0x4803_0000	0x4803_0FFF	4KB	McSPI0 Registers
	0x4803_1000	0x4803_1FFF	4KB	Reserved
Reserved	0x4803_2000	0x4803_2FFF	4KB	Reserved
	0x4803_3000	0x4803_3FFF	4KB	Reserved
Reserved	0x4803_4000	0x4803_4FFF	4KB	Reserved
	0x4803_5000	0x4803_5FFF	4KB	Reserved
Reserved	0x4803_6000	0x4803_6FFF	4KB	Reserved
	0x4803_7000	0x4803_7FFF	4KB	Reserved
<u>McASP0CFG</u>	0x4803_8000	0x4803_9FFF	8KB	McASP0 CFG Registers
	0x4803_A000	0x4803_AFFF	4KB	Reserved
Reserved	0x4803_B000	0x4803_BFFF	4KB	Reserved
<u>McASP1CFG</u>	0x4803_C000	0x4803_DFFF	8KB	McASP1 CFG Registers
	0x4803_E000	0x4803_EFFF	4KB	Reserved
Reserved	0x4803_F000	0x4803_FFFF	4KB	Reserved
<u>DMTIMER2</u>	0x4804_0000	0x4804_0FFF	4KB	DMTimer2 Registers
	0x4804_1000	0x4804_1FFF	4KB	Reserved
<u>DMTIMER3</u>	0x4804_2000	0x4804_2FFF	4KB	DMTimer3 Registers
	0x4804_3000	0x4804_3FFF	4KB	Reserved
<u>DMTIMER4</u>	0x4804_4000	0x4804_4FFF	4KB	DMTimer4 Registers
	0x4804_5000	0x4804_5FFF	4KB	Reserved
<u>DMTIMER5</u>	0x4804_6000	0x4804_6FFF	4KB	DMTimer5 Registers
	0x4804_7000	0x4804_7FFF	4KB	Reserved
<u>DMTIMER6</u>	0x4804_8000	0x4804_8FFF	4KB	DMTimer6 Registers
	0x4804_9000	0x4804_9FFF	4KB	L4 Interconnect
<u>DMTIMER7</u>	0x4804_A000	0x4804_AFFF	4KB	DMTimer7 Registers
	0x4804_B000	0x4804_BFFF	4KB	Reserved
<u>GPIO1</u>	0x4804_C000	0x4804_CFFF	4KB	GPIO1 Registers
	0x4804_D000	0x4804_DFFF	4KB	Reserved
Reserved	0x4804_E000	0x4804_FFFF	8KB	Reserved

Table 2-3. L4_PER Peripheral Memory Map (continued)

Device Name	Start_address (hex)	End_address (hex)	Size	Description
Reserved	0x4805_0000	0x4805_FFFF	64KB	Reserved
<u>MMCHS0</u>	0x4806_0000	0x4806_0FFF	4KB	MMCHS0 Registers
	0x4806_1000	0x4806_1FFF	4KB	Reserved
Reserved	0x4806_2000	0x4807_FFFF	120KB	Reserved
<u>ELM</u>	0x4808_0000	0x4808_FFFF	64KB	ELM Registers
	0x4809_0000	0x4809_0FFF	4KB	Reserved
Reserved	0x4809_1000	0x4809_FFFF	60KB	Reserved
Reserved	0x480A_0000	0x480A_FFFF	64KB	Reserved
	0x480B_0000	0x480B_0FFF	4KB	Reserved
Reserved	0x480B_1000	0x480B_FFFF	60KB	Reserved
Reserved	0x480C_0000	0x480C_0FFF	4KB	Reserved
	0x480C_1000	0x480C_1FFF	4KB	Reserved
Reserved	0x480C_2000	0x480C_2FFF	4KB	Reserved
	0x480C_3000	0x480C_3FFF	4KB	Reserved
Reserved	0x480C_4000	0x480C_7FFF	16KB	Reserved
<u>Mailbox0</u>	0x480C_8000	0x480C_8FFF	4KB	Mailbox Registers
	0x480C_9000	0x480C_9FFF	4KB	Reserved
<u>Spinlock</u>	0x480C_A000	0x480C_AFFF	4KB	Spinlock Registers
	0x480C_B000	0x480C_BFFF	4KB	Reserved
Reserved	0x480C_C000	0x480F_FFFF	208KB	Reserved
Reserved	0x4810_0000	0x4811_FFFF	128KB	Reserved
	0x4812_0000	0x4812_0FFF	4KB	Reserved
Reserved	0x4812_1000	0x4812_1FFF	4KB	Reserved
Reserved	0x4812_2000	0x4812_2FFF	4KB	Reserved
	0x4812_3000	0x4812_3FFF	4KB	Reserved
Reserved	0x4812_4000	0x4813_FFFF	112KB	Reserved
Reserved	0x4814_0000	0x4815_FFFF	128KB	Reserved
	0x4816_0000	0x4816_0FFF	4K	Reserved
Reserved	0x4816_1000	0x4817_FFFF	124KB	Reserved
Reserved	0x4818_0000	0x4818_2FFF	12KB	Reserved
	0x4818_3000	0x4818_3FFF	4KB	Reserved
Reserved	0x4818_4000	0x4818_7FFF	16KB	Reserved
Reserved	0x4818_8000	0x4818_8FFF	4KB	Reserved
	0x4818_9000	0x4818_9FFF	4KB	Reserved
Reserved	0x4818_A000	0x4818_AFFF	4KB	Reserved
	0x4818_B000	0x4818_BFFF	4KB	Reserved
OCP Watchpoint	0x4818_C000	0x4818_CFFF	4KB	OCP Watchpoint Registers
	0x4818_D000	0x4818_DFFF	4KB	Reserved
Reserved	0x4818_E000	0x4818_EFFF	4KB	Reserved
	0x4818_F000	0x4818_FFFF	4KB	Reserved
Reserved	0x4819_0000	0x4819_0FFF	4KB	Reserved
	0x4819_1000	0x4819_1FFF	4KB	Reserved
Reserved	0x4819_2000	0x4819_2FFF	4KB	Reserved
	0x4819_3000	0x4819_3FFF	4KB	Reserved
Reserved	0x4819_4000	0x4819_BFFF	32KB	Reserved
<u>I2C2</u>	0x4819_C000	0x4819_CFFF	4KB	I2C2 Registers
	0x4819_D000	0x4819_DFFF	4KB	Reserved

Table 2-3. L4_PER Peripheral Memory Map (continued)

Device Name	Start_address (hex)	End_address (hex)	Size	Description
Reserved	0x4819_E000	0x4819_EFFF	4KB	Reserved
	0x4819_F000	0x4819_FFFF	4KB	Reserved
<u>McSPI1</u>	0x481A_0000	0x481A_0FFF	4KB	McSPI1 Registers
	0x481A_1000	0x481A_1FFF	4KB	Reserved
Reserved	0x481A_2000	0x481A_5FFF	16KB	Reserved
<u>UART3</u>	0x481A_6000	0x481A_6FFF	4KB	UART3 Registers
	0x481A_7000	0x481A_7FFF	4KB	Reserved
<u>UART4</u>	0x481A_8000	0x481A_8FFF	4KB	UART4 Registers
	0x481A_9000	0x481A_9FFF	4KB	Reserved
<u>UART5</u>	0x481A_A000	0x481A_AFFF	4KB	UART5 Registers
	0x481A_B000	0x481A_BFFF	4KB	Reserved
<u>GPIO2</u>	0x481A_C000	0x481A_CFFF	4KB	GPIO2 Registers
	0x481A_D000	0x481A_DFFF	4KB	Reserved
<u>GPIO3</u>	0x481A_E000	0x481A_EFFF	4KB	GPIO3 Registers
	0x481A_F000	0x481A_FFFF	4KB	Reserved
Reserved	0x481B_0000	0x481B_FFFF	64KB	Reserved
	0x481C_0000	0x481C_0FFF	4KB	Reserved
Reserved	0x481C_1000	0x481C_1FFF	4KB	Reserved
	0x481C_2000	0x481C_2FFF	4KB	Reserved
Reserved	0x481C_3000	0x481C_9FFF	28KB	Reserved
Reserved	0x481C_A000	0x481C_AFFF	4KB	Reserved
	0x481C_B000	0x481C_BFFF	4KB	Reserved
<u>DCAN0</u>	0x481C_C000	0x481C_DFFF	8KB	DCAN0 Registers
	0x481C_E000	0x481C_FFFF	8KB	Reserved
<u>DCAN1</u>	0x481D_0000	0x481D_1FFF	8KB	DCAN1 Registers
	0x481D_2000	0x481D_3FFF	8KB	Reserved
Reserved	0x481D_4000	0x481D_4FFF	4KB	Reserved
	0x481D_5000	0x481D_5FFF	4KB	Reserved
Reserved	0x481D_6000	0x481D_6FFF	4KB	Reserved
	0x481D_7000	0x481D_7FFF	4KB	Reserved
<u>MMC1</u>	0x481D_8000	0x481D_8FFF	4KB	MMC1 Registers
	0x481D_9000	0x481D_9FFF	4KB	Reserved
Reserved	0x481D_A000	0x481F_FFFF	152KB	Reserved
<u>Interrupt controller (INTCP5)</u>	0x4820_0000	0x4820_0FFF	4KB	Interrupt Controller Registers
Reserved	0x4820_1000	0x4823_FFFF	252KB	Reserved
MPUSS config register	0x4824_0000	0x4824_0FFF	4KB	Host ARM non-shared device mapping
Reserved	0x4824_1000	0x4827_FFFF	252KB	Reserved
Reserved	0x4828_0000	0x4828_0FFF	4KB	Reserved
Reserved	0x4828_1000	0x482F_FFFF	508KB	Reserved
<u>PWMSSubsystem0</u>	0x4830_0000	0x4830_00FF	4KB	PWMSS0 Configuration Registers
<u>eCAP0</u>	0x4830_0100	0x4830_017F		PWMSS eCAP0 Registers
<u>eQEP0</u>	0x4830_0180	0x4830_01FF		PWMSS eQEP0 Registers
<u>ePWM0</u>	0x4830_0200	0x4830_025F		PWMSS ePWM0 Registers
	0x4830_0260	0x4830_1FFF	4KB	Reserved

Table 2-3. L4_PER Peripheral Memory Map (continued)

Device Name	Start_address (hex)	End_address (hex)	Size	Description
<u>PWMSubsystem1</u>	0x4830_2000	0x4830_20FF	4KB	PWMSS1 Configuration Registers
<u>eCAP1</u>	0x4830_2100	0x4830_217F		PWMSS eCAP1 Registers
<u>eQEP1</u>	0x4830_2180	0x4830_21FF		PWMSS eQEP1 Registers
<u>ePWM1</u>	0x4830_2200	0x4830_225F		PWMSS ePWM1 Registers
	0x4830_2260	0x4830_3FFF		Reserved
<u>PWMSubsystem2</u>	0x4830_4000	0x4830_40FF	4KB	PWMSS2 Configuration Registers
<u>eCAP2</u>	0x4830_4100	0x4830_417F		PWMSS eCAP2 Registers
<u>eQEP2</u>	0x4830_4180	0x4830_41FF		PWMSS eQEP2 Registers
<u>ePWM2</u>	0x4830_4200	0x4830_425F		PWMSS ePWM2 Registers
	0x4830_4260	0x4830_5FFF		Reserved
Reserved	0x4830_6000	0x4830_DFFF	32KB	Reserved
<u>LCDController</u>	0x4830_E000	0x4830_EFFF	4KB	LCD Registers
	0x4830_F000	0x4830_FFFF	4KB	Reserved
Reserved	0x4831_0000	0x4831_1FFF	8KB	Reserved
	0x4831_2000	0x4831_2FFF	4KB	Reserved
Reserved	0x4831_3000	0x4831_7FFF	20KB	Reserved
Reserved	0x4831_8000	0x4831_BFFF	16KB	Reserved
	0x4831_C000	0x4831_CFFF	4KB	Reserved
Reserved	0x4831_D000	0x4831_FFFF	12KB	Reserved
Reserved	0x4832_0000	0x4832_5FFF	16KB	Reserved
Reserved	0x4832_6000	0x48FF_FFFF	13MB-152KB	Reserved

Table 2-4. L4 Fast Peripheral Memory Map

Device Name	Start_address (hex)	End_address (hex)	Size	Description
<u>L4_Fast configuration</u>	0x4A00_0000	0x4A00_07FF	2KB	Address/Protection (AP)
	0x4A00_0800	0x4A00_0FFF	2KB	Link Agent (LA)
	0x4A00_1000	0x4A00_13FF	1KB	Initiator Port (IP0)
	0x4A00_1400	0x4A00_17FF	1KB	Reserved
	0x4A00_1800	0x4A00_1FFF	2KB	Reserved (IP2 - IP3)
Reserved	0x4A00_2000	0x4A07_FFFF	504KB	Reserved
Reserved	0x4A08_0000	0x4A09_FFFF	128KB	Reserved
	0x4A0A_0000	0x4A0A_0FFF	4KB	Reserved
Reserved	0x4A0A_1000	0x4A0F_FFFF	380KB	Reserved
<u>CPSW_SS</u>	0x4A10_0000	0x4A10_7FFF	32KB	Ethernet Switch Subsystem
<u>CPSW_PORT</u>	0x4A10_0100	0x4A10_07FF		Ethernet Switch Port Control
<u>CPSW_CPDMA</u>	0x4A10_0800	0x4A10_08FF		CPPI DMA Controller Module
<u>CPSW_STATS</u>	0x4A10_0900	0x4A10_09FF		Ethernet Statistics
<u>CPSW_STATERAM</u>	0x4A10_0A00	0x4A10_0BFF		CPPI DMA State RAM
<u>CPSW_CPTS</u>	0x4A10_0C00	0x4A10_0CFF		Ethernet Time Sync Module
<u>CPSW_ALE</u>	0x4A10_0D00	0x4A10_0D7F		Ethernet Address Lookup Engine
<u>CPSW_SL1</u>	0x4A10_0D80	0x4A10_0DBF		Ethernet Sliver for Port 1
<u>CPSW_SL2</u>	0x4A10_0DC0	0x4A10_0DFF		Ethernet Sliver for Port 2
Reserved	0x4A10_0E00	0x4A10_0FFF		Reserved

Table 2-4. L4 Fast Peripheral Memory Map (continued)

Device Name	Start_address (hex)	End_address (hex)	Size	Description
<u>MDIO</u>	0x4A10_1000	0x4A10_10FF		Ethernet MDIO Controller
Reserved	0x4A10_1100	0x4A10_11FF		Reserved
<u>CPSW_WR</u>	0x4A10_1200	0x4A10_1FFF		Ethernet Subsystem Wrapper for RMII/RGMII
CPPI_RAM	0x4A10_2000	0x4A10_3FFF		Communications Port Programming Interface RAM
Reserved	0x4A10_9000	0x4A13_FFFF	220KB	Reserved
Reserved	0x4A14_0000	0x4A14_FFFF	64KB	Reserved
	0x4A15_0000	0x4A15_0FFF	4KB	Reserved
Reserved	0x4A15_1000	0x4A17_FFFF	188KB	Reserved
Reserved	0x4A18_0000	0x4A1A_1FFF	136KB	Reserved
Reserved	0x4A1A_2000	0x4A1A_3FFF	8KB	Reserved
	0x4A1A_4000	0x4A1A_4FFF	4KB	Reserved
Reserved	0x4A1A_5000	0x4A1A_5FFF	4KB	Reserved
	0x4A1A_6000	0x4A1A_6FFF	4KB	Reserved
Reserved	0x4A1A_7000	0x4A1A_7FFF	4KB	Reserved
Reserved	0x4A1A_8000	0x4A1A_9FFF	8KB	Reserved
	0x4A1A_A000	0x4A1A_AFFF	4KB	Reserved
Reserved	0x4A1A_B000	0x4A1A_BFFF	4KB	Reserved
	0x4A1A_C000	0x4A1A_CFFF	4KB	Reserved
Reserved	0x4A1A_D000	0x4A1A_DFFF	4KB	Reserved
Reserved	0x4A1A_E000	0x4A1A_FFFF	8KB	Reserved
	0x4A1B_0000	0x4A1B_0FFF	4KB	Reserved
Reserved	0x4A1B_1000	0x4A1B_1FFF	4KB	Reserved
	0x4A1B_2000	0x4A1B_2FFF	4KB	Reserved
Reserved	0x4A1B_3000	0x4A1B_3FFF	4KB	Reserved
	0x4A1B_4000	0x4A1B_4FFF	4KB	Reserved
Reserved	0x4A1B_5000	0x4A1B_5FFF	4KB	Reserved
	0x4A1B_6000	0x4A1B_6FFF	4KB	Reserved
Reserved	0x4A1B_4000	0x4A1F_FFFF	304KB	Reserved
Reserved	0x4A20_0000	0x4A2F_FFFF	1MB	Reserved
PRU_ICSS	0x4A30_0000	0x4A37_FFFF	512KB	PRU-ICSS Instruction/Data/Control Space
	0x4A38_0000	0x4A38_0FFF	4KB	Reserved
Reserved	0x4A38_1000	0x4A3F_FFFF	508KB	Reserved
Reserved	0x4A40_0000	0x4AFF_FFFF	12MB	Reserved

9.

Control Module (Pinmuxing) Register

9.3 CONTROL_MODULE Registers

[Table 9-10](#) lists the memory-mapped registers for the CONTROL_MODULE. All other register offset addresses not listed in [Table 9-10](#) should be considered as reserved locations and the register contents should not be modified.

Table 9-10. CONTROL_MODULE REGISTERS

Offset	Acronym	Register Description	Section
0h	control_revision		Section 9.3.1
4h	control_hwinfo		Section 9.3.2
10h	control_sysconfig		Section 9.3.3
40h	control_status		Section 9.3.4
110h	control_emif_sdram_config		Section 9.3.5
41Ch	cortex_vbbldo_ctrl		Section 9.3.5
428h	core_sldo_ctrl		Section 9.3.6
42Ch	mpu_sldo_ctrl		Section 9.3.7
444h	clk32kdivratio_ctrl		Section 9.3.8
448h	bandgap_ctrl		Section 9.3.9
44Ch	bandgap_trim		Section 9.3.10
458h	pll_clkinpulow_ctrl		Section 9.3.11
468h	mosc_ctrl		Section 9.3.12
470h	deepsleep_ctrl		Section 9.3.13
50Ch	dpll_pwr_sw_status		Section 9.3.14
600h	device_id		Section 9.3.15
604h	dev_feature		Section 9.3.16
608h	init_priority_0		Section 9.3.17
60Ch	init_priority_1		Section 9.3.18
610h	mmu_cfg		Section 9.3.19
614h	tptc_cfg		Section 9.3.20
620h	usb_ctrl0		Section 9.3.21
624h	usb_sts0		Section 9.3.22
628h	usb_ctrl1		Section 9.3.23
62Ch	usb_sts1		Section 9.3.24
630h	mac_id0_lo		Section 9.3.25
634h	mac_id0_hi		Section 9.3.26
638h	mac_id1_lo		Section 9.3.27
63Ch	mac_id1_hi		Section 9.3.28
644h	dcan_raminit		Section 9.3.29
648h	usb_wkup_ctrl		Section 9.3.30
650h	gmii_sel		Section 9.3.31
664h	pwmss_ctrl		Section 9.3.32
670h	mreqprio_0		Section 9.3.33
674h	mreqprio_1		Section 9.3.34
690h	hw_event_sel_grp1		Section 9.3.35

Table 9-10. CONTROL_MODULE REGISTERS (continued)

Offset	Acronym	Register Description	Section
694h	hw_event_sel_grp2		Section 9.3.36
698h	hw_event_sel_grp3		Section 9.3.37
69Ch	hw_event_sel_grp4		Section 9.3.38
6A0h	smrt_ctrl		Section 9.3.39
6A4h	mpuss_hw_debug_sel		Section 9.3.40
6A8h	mpuss_hw_dbg_info		Section 9.3.41
770h	vdd_mpu_opp_050		Section 9.3.42
774h	vdd_mpu_opp_100		Section 9.3.43
778h	vdd_mpu_opp_120		Section 9.3.44
77Ch	vdd_mpu_opp_turbo		Section 9.3.45
7B8h	vdd_core_opp_050		Section 9.3.46
7BCh	vdd_core_opp_100		Section 9.3.47
7D0h	bb_scale		Section 9.3.48
7F4h	usb_vid_pid		Section 9.3.49
7FC _h	efuse_sma		Section 9.3.50
800h	conf_gpmc_ad0	See the device datasheet for information on default pin mux configurations. Note that the device ROM may change the default pin mux for certain pins based on the SYS-BOOT mode settings.	Section 9.3.51
804h	conf_gpmc_ad1		Section 9.3.51
808h	conf_gpmc_ad2		Section 9.3.51
80Ch	conf_gpmc_ad3		Section 9.3.51
810h	conf_gpmc_ad4		Section 9.3.51
814h	conf_gpmc_ad5		Section 9.3.51
818h	conf_gpmc_ad6		Section 9.3.51
81Ch	conf_gpmc_ad7		Section 9.3.51
820h	conf_gpmc_ad8		Section 9.3.51
824h	conf_gpmc_ad9		Section 9.3.51
828h	conf_gpmc_ad10		Section 9.3.51
82Ch	conf_gpmc_ad11		Section 9.3.51
830h	conf_gpmc_ad12		Section 9.3.51
834h	conf_gpmc_ad13		Section 9.3.51
838h	conf_gpmc_ad14		Section 9.3.51
83Ch	conf_gpmc_ad15		Section 9.3.51
840h	conf_gpmc_a0		Section 9.3.51
844h	conf_gpmc_a1		Section 9.3.51
848h	conf_gpmc_a2		Section 9.3.51
84Ch	conf_gpmc_a3		Section 9.3.51
850h	conf_gpmc_a4		Section 9.3.51
854h	conf_gpmc_a5		Section 9.3.51
858h	conf_gpmc_a6		Section 9.3.51
85Ch	conf_gpmc_a7		Section 9.3.51
860h	conf_gpmc_a8		Section 9.3.51
864h	conf_gpmc_a9		Section 9.3.51
868h	conf_gpmc_a10		Section 9.3.51
86Ch	conf_gpmc_a11		Section 9.3.51
870h	conf_gpmc_wait0		Section 9.3.51
874h	conf_gpmc_wpn		Section 9.3.51

Table 9-10. CONTROL_MODULE REGISTERS (continued)

Offset	Acronym	Register Description	Section
878h	conf_gpmc_ben1		Section 9.3.51
87Ch	conf_gpmc_csn0		Section 9.3.51
880h	conf_gpmc_csn1		Section 9.3.51
884h	conf_gpmc_csn2		Section 9.3.51
888h	conf_gpmc_csn3		Section 9.3.51
88Ch	conf_gpmc_clk		Section 9.3.51
890h	conf_gpmc_advn_ale		Section 9.3.51
894h	conf_gpmc_oen_ren		Section 9.3.51
898h	conf_gpmc_wen		Section 9.3.51
89Ch	conf_gpmc_ben0_cle		Section 9.3.51
8A0h	conf_lcd_data0		Section 9.3.51
8A4h	conf_lcd_data1		Section 9.3.51
8A8h	conf_lcd_data2		Section 9.3.51
8ACh	conf_lcd_data3		Section 9.3.51
8B0h	conf_lcd_data4		Section 9.3.51
8B4h	conf_lcd_data5		Section 9.3.51
8B8h	conf_lcd_data6		Section 9.3.51
8BCh	conf_lcd_data7		Section 9.3.51
8C0h	conf_lcd_data8		Section 9.3.51
8C4h	conf_lcd_data9		Section 9.3.51
8C8h	conf_lcd_data10		Section 9.3.51
8CCh	conf_lcd_data11		Section 9.3.51
8D0h	conf_lcd_data12		Section 9.3.51
8D4h	conf_lcd_data13		Section 9.3.51
8D8h	conf_lcd_data14		Section 9.3.51
8DCh	conf_lcd_data15		Section 9.3.51
8E0h	conf_lcd_vsync		Section 9.3.51
8E4h	conf_lcd_hsync		Section 9.3.51
8E8h	conf_lcd_pcclk		Section 9.3.51
8ECh	conf_lcd_ac_bias_en		Section 9.3.51
8F0h	conf_mmc0_dat3		Section 9.3.51
8F4h	conf_mmc0_dat2		Section 9.3.51
8F8h	conf_mmc0_dat1		Section 9.3.51
8FCh	conf_mmc0_dat0		Section 9.3.51
900h	conf_mmc0_clk		Section 9.3.51
904h	conf_mmc0_cmd		Section 9.3.51
908h	conf_mii1_col		Section 9.3.51
90Ch	conf_mii1_crs		Section 9.3.51
910h	conf_mii1_rx_er		Section 9.3.51
914h	conf_mii1_tx_en		Section 9.3.51
918h	conf_mii1_rx_dv		Section 9.3.51
91Ch	conf_mii1_txd3		Section 9.3.51
920h	conf_mii1_txd2		Section 9.3.51
924h	conf_mii1_txd1		Section 9.3.51
928h	conf_mii1_txd0		Section 9.3.51
92Ch	conf_mii1_tx_clk		Section 9.3.51
930h	conf_mii1_rx_clk		Section 9.3.51

Table 9-10. CONTROL_MODULE REGISTERS (continued)

Offset	Acronym	Register Description	Section
934h	conf_mii1_rxrd3		Section 9.3.51
938h	conf_mii1_rxrd2		Section 9.3.51
93Ch	conf_mii1_rxrd1		Section 9.3.51
940h	conf_mii1_rxrd0		Section 9.3.51
944h	conf_rmii1_ref_clk		Section 9.3.51
948h	conf_mdio		Section 9.3.51
94Ch	conf_mdc		Section 9.3.51
950h	conf_spi0_sclk		Section 9.3.51
954h	conf_spi0_d0		Section 9.3.51
958h	conf_spi0_d1		Section 9.3.51
95Ch	conf_spi0_cs0		Section 9.3.51
960h	conf_spi0_cs1		Section 9.3.51
964h	conf_ecap0_in_pwm0_out		Section 9.3.51
968h	conf_uart0_ctsn		Section 9.3.51
96Ch	conf_uart0_rtsn		Section 9.3.51
970h	conf_uart0_rxd		Section 9.3.51
974h	conf_uart0_txd		Section 9.3.51
978h	conf_uart1_ctsn		Section 9.3.51
97Ch	conf_uart1_rtsn		Section 9.3.51
980h	conf_uart1_rxd		Section 9.3.51
984h	conf_uart1_txd		Section 9.3.51
988h	conf_i2c0_sda		Section 9.3.51
98Ch	conf_i2c0_scl		Section 9.3.51
990h	conf_mcasp0_aclkx		Section 9.3.51
994h	conf_mcasp0_fsx		Section 9.3.51
998h	conf_mcasp0_axr0		Section 9.3.51
99Ch	conf_mcasp0_ahclkx		Section 9.3.51
9A0h	conf_mcasp0_aclkx		Section 9.3.51
9A4h	conf_mcasp0_fsr		Section 9.3.51
9A8h	conf_mcasp0_axr1		Section 9.3.51
9ACh	conf_mcasp0_ahclkx		Section 9.3.51
9B0h	conf_xdma_event_intr0		Section 9.3.51
9B4h	conf_xdma_event_intr1		Section 9.3.51
9B8h	conf_warmrstn		Section 9.3.51
9C0h	conf_nnm		Section 9.3.51
9D0h	conf_tms		Section 9.3.51
9D4h	conf_tdi		Section 9.3.51
9D8h	conf_tdo		Section 9.3.51
9DCh	conf_tck		Section 9.3.51
9E0h	conf_trstn		Section 9.3.51
9E4h	conf_emu0		Section 9.3.51
9E8h	conf_emu1		Section 9.3.51
9F8h	conf_rtc_pwronrstn		Section 9.3.51
9FCh	conf_pmic_power_en		Section 9.3.51
A00h	conf_ext_wakeup		Section 9.3.51
A04h	conf_rtc_kaldo_enn		Section 9.3.51
A1Ch	conf_usb0_drvvbus		Section 9.3.51

Table 9-10. CONTROL_MODULE REGISTERS (continued)

Offset	Acronym	Register Description	Section
A34h	conf_usb1_drvvbus		Section 9.3.51
E00h	cqdetect_status		Section 9.3.52
E04h	ddr_io_ctrl		Section 9.3.53
E0Ch	vtp_ctrl		Section 9.3.54
E14h	vref_ctrl		Section 9.3.55
F90h	tpcc_evt_mux_0_3		Section 9.3.56
F94h	tpcc_evt_mux_4_7		Section 9.3.57
F98h	tpcc_evt_mux_8_11		Section 9.3.58
F9Ch	tpcc_evt_mux_12_15		Section 9.3.59
FA0h	tpcc_evt_mux_16_19		Section 9.3.60
FA4h	tpcc_evt_mux_20_23		Section 9.3.61
FA8h	tpcc_evt_mux_24_27		Section 9.3.62
FACh	tpcc_evt_mux_28_31		Section 9.3.63
FB0h	tpcc_evt_mux_32_35		Section 9.3.64
FB4h	tpcc_evt_mux_36_39		Section 9.3.65
FB8h	tpcc_evt_mux_40_43		Section 9.3.66
FBCh	tpcc_evt_mux_44_47		Section 9.3.67
FC0h	tpcc_evt_mux_48_51		Section 9.3.68
FC4h	tpcc_evt_mux_52_55		Section 9.3.69
FC8h	tpcc_evt_mux_56_59		Section 9.3.70
FCCh	tpcc_evt_mux_60_63		Section 9.3.71
FD0h	timer_evt_capt		Section 9.3.72
FD4h	ecap_evt_capt		Section 9.3.73
FD8h	adc_evt_capt		Section 9.3.74
1000h	reset_iso		Section 9.3.75
1318h	dpll_pwr_sw_ctrl		Section 9.3.76
131Ch	ddr_cke_ctrl		Section 9.3.77
1320h	sma2		Section 9.3.78
1324h	m3_txev_eoi		Section 9.3.79
1328h	ipc_msg_reg0		Section 9.3.80
132Ch	ipc_msg_reg1		Section 9.3.81
1330h	ipc_msg_reg2		Section 9.3.82
1334h	ipc_msg_reg3		Section 9.3.83
1338h	ipc_msg_reg4		Section 9.3.84
133Ch	ipc_msg_reg5		Section 9.3.85
1340h	ipc_msg_reg6		Section 9.3.86
1344h	ipc_msg_reg7		Section 9.3.87
1404h	ddr_cmd0_ioctl		Section 9.3.88
1408h	ddr_cmd1_ioctl		Section 9.3.89
140Ch	ddr_cmd2_ioctl		Section 9.3.90
1440h	ddr_data0_ioctl		Section 9.3.91
1444h	ddr_data1_ioctl		Section 9.3.92

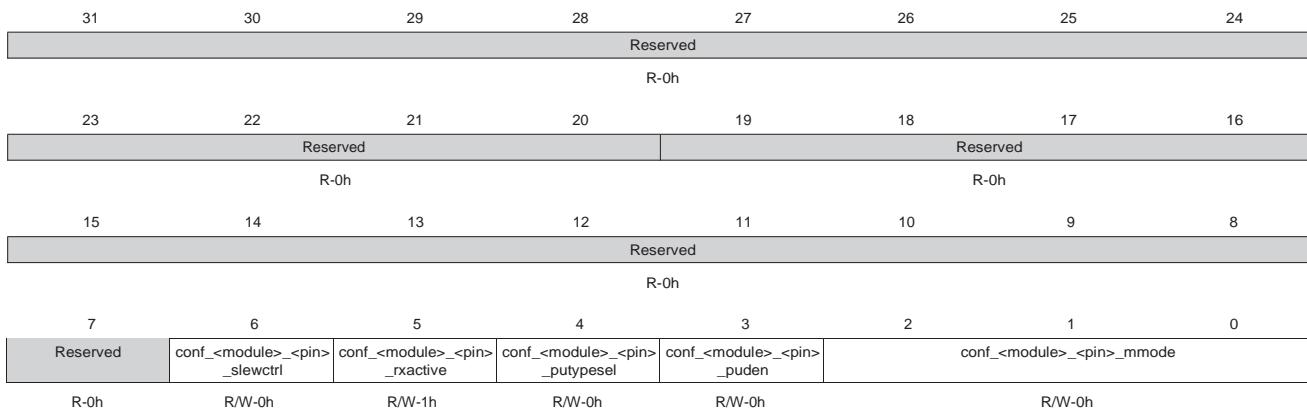
9.3.51 conf_<module>_<pin> Register (offset = 800h–A34h)

See the device datasheet for information on default pin mux configurations. Note that the device ROM may change the default pin mux for certain pins based on the SYSBOOT mode settings.

See [Table 9-10, Control Module Registers Table](#), for the full list of offsets for each module/pin configuration.

conf_<module>_<pin> is shown in [Figure 9-54](#) and described in [Table 9-61](#).

Figure 9-54. conf_<module>_<pin> Register



LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 9-61. conf_<module>_<pin> Register Field Descriptions

Bit	Field	Type	Reset	Description
31-20	Reserved	R	0h	
19-7	Reserved	R	0h	
6	conf_<module>_<pin>_slewctrl	R/W	X	Select between faster or slower slew rate 0: Fast 1: Slow Reset value is pad-dependent.
5	conf_<module>_<pin>_rxactive	R/W	1h	Input enable value for the PAD 0: Receiver disabled 1: Receiver enabled
4	conf_<module>_<pin>_putypesel	R/W	X	Pad pullup/pulldown type selection 0: Pulldown selected 1: Pullup selected Reset value is pad-dependent.
3	conf_<module>_<pin>_puuden	R/W	X	Pad pullup/pulldown enable 0: Pullup/pulldown enabled 1: Pullup/pulldown disabled Reset value is pad-dependent.
2-0	conf_<module>_<pin>_mmode	R/W	X	Pad functional signal mux select. Reset value is pad-dependent.

10. GPIO Register

25.4 GPIO Registers

25.4.1 GPIO Registers

Table 25-5 lists the memory-mapped registers for the GPIO. All register offset addresses not listed in Table 25-5 should be considered as reserved locations and the register contents should not be modified.

Table 25-5. GPIO REGISTERS

Offset	Acronym	Register Name	Section
0h	GPIO_REVISION		Section 25.4.1.1
10h	GPIO_SYSCONFIG		Section 25.4.1.2
20h	GPIO_EOI		Section 25.4.1.3
24h	GPIO_IRQSTATUS_RAW_0		Section 25.4.1.4
28h	GPIO_IRQSTATUS_RAW_1		Section 25.4.1.5
2Ch	GPIO_IRQSTATUS_0		Section 25.4.1.6
30h	GPIO_IRQSTATUS_1		Section 25.4.1.7
34h	GPIO_IRQSTATUS_SET_0		Section 25.4.1.8
38h	GPIO_IRQSTATUS_SET_1		Section 25.4.1.9
3Ch	GPIO_IRQSTATUS_CLR_0		Section 25.4.1.10
40h	GPIO_IRQSTATUS_CLR_1		Section 25.4.1.11
44h	GPIO_IRQWAKEN_0		Section 25.4.1.12
48h	GPIO_IRQWAKEN_1		Section 25.4.1.13
114h	GPIO_SYSSTATUS		Section 25.4.1.14
130h	GPIO_CTRL		Section 25.4.1.15
134h	GPIO_OE		Section 25.4.1.16
138h	GPIO_DATAIN		Section 25.4.1.17
13Ch	GPIO_DATAOUT		Section 25.4.1.18
140h	GPIO_LEVELDETECT0		Section 25.4.1.19
144h	GPIO_LEVELDETECT1		Section 25.4.1.20
148h	GPIO_RISINGDETECT		Section 25.4.1.21
14Ch	GPIO_FALLINGDETECT		Section 25.4.1.22
150h	GPIO_DEBOUNCENABLE		Section 25.4.1.23
154h	GPIO_DEBOUNCINGTIME		Section 25.4.1.24
190h	GPIO_CLEARDATAOUT		Section 25.4.1.25
194h	GPIO_SETDATAOUT		Section 25.4.1.26

25.4.1.6 GPIO_IRQSTATUS_0 Register (offset = 2Ch) [reset = 0h]

GPIO_IRQSTATUS_0 is shown in [Figure 25-12](#) and described in [Table 25-11](#).

The GPIO_IRQSTATUS_0 register provides core status information for the interrupt handling, showing all active events which have been enabled. The fields are read-write. Writing a 1 to a bit clears the bit to 0, that is, clears the IRQ. Writing a 0 has no effect, that is, the register value is not modified. Only enabled, active events trigger an actual interrupt request on the IRQ output line.

Figure 25-12. GPIO_IRQSTATUS_0 Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INTLINE[n]																															
R/W1C-0h																															

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 25-11. GPIO_IRQSTATUS_0 Register Field Descriptions

Bit	Field	Type	Reset	Description
31-0	INTLINE[n]	R/W1C	0h	Interrupt n status. 0x0(W) = No effect. 0x0(R) = IRQ is not triggered. 0x1(W) = Clears the IRQ. 0x1(R) = IRQ is triggered.

25.4.1.7 GPIO_IRQSTATUS_1 Register (offset = 30h) [reset = 0h]

GPIO_IRQSTATUS_1 is shown in [Figure 25-13](#) and described in [Table 25-12](#).

The GPIO_IRQSTATUS_1 register provides core status information for the interrupt handling, showing all active events which have been enabled. The fields are read-write. Writing a 1 to a bit clears the bit to 0, that is, clears the IRQ. Writing a 0 has no effect, that is, the register value is not modified. Only enabled, active events trigger an actual interrupt request on the IRQ output line.

Figure 25-13. GPIO_IRQSTATUS_1 Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INTLINE[n]																															
R/W1C-0h																															

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 25-12. GPIO_IRQSTATUS_1 Register Field Descriptions

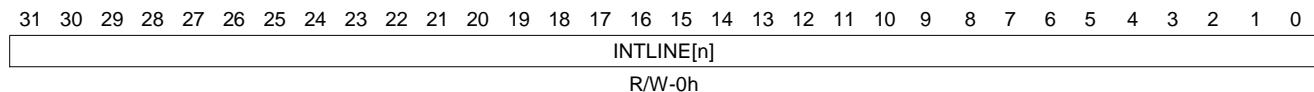
Bit	Field	Type	Reset	Description
31-0	INTLINE[n]	R/W1C	0h	Interrupt n status. 0x0(W) = No effect. 0x0(R) = IRQ is not triggered. 0x1(W) = Clears the IRQ. 0x1(R) = IRQ is triggered.

25.4.1.8 GPIO_IRQSTATUS_SET_0 Register (offset = 34h) [reset = 0h]

GPIO_IRQSTATUS_SET_0 is shown in [Figure 25-14](#) and described in [Table 25-13](#).

All 1-bit fields in the GPIO_IRQSTATUS_SET_0 register enable a specific interrupt event to trigger an interrupt request. Writing a 1 to a bit enables the interrupt field. Writing a 0 has no effect, that is, the register value is not modified.

Figure 25-14. GPIO_IRQSTATUS_SET_0 Register



LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 25-13. GPIO_IRQSTATUS_SET_0 Register Field Descriptions

Bit	Field	Type	Reset	Description
31-0	INTLINE[n]	R/W	0h	Interrupt n enable 0x0 = No effect. 0x1 = Enable IRQ generation.

25.4.1.9 GPIO_IRQSTATUS_SET_1 Register (offset = 38h) [reset = 0h]

GPIO_IRQSTATUS_SET_1 is shown in [Figure 25-15](#) and described in [Table 25-14](#).

All 1-bit fields in the GPIO_IRQSTATUS_SET_1 register enable a specific interrupt event to trigger an interrupt request. Writing a 1 to a bit enables the interrupt field. Writing a 0 has no effect, that is, the register value is not modified.

Figure 25-15. GPIO_IRQSTATUS_SET_1 Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INTLINE[n]																															
R/W-0h																															

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 25-14. GPIO_IRQSTATUS_SET_1 Register Field Descriptions

Bit	Field	Type	Reset	Description
31-0	INTLINE[n]	R/W	0h	Interrupt n enable 0x0 = No effect. 0x1 = Enable IRQ generation.

25.4.1.16 GPIO_OE Register (offset = 134h) [reset = FFFFFFFFh]

GPIO_OE is shown in [Figure 25-22](#) and described in [Table 25-21](#).

The GPIO_OE register is used to enable the pins output capabilities. At reset, all the GPIO related pins are configured as input and output capabilities are disabled. This register is not used within the module, its only function is to carry the pads configuration. When the application is using a pin as an output and does not want interrupt generation from this pin, the application can/has to properly configure the Interrupt Enable registers.

Figure 25-22. GPIO_OE Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OUTPUTEN[n]																															
R/W-FFFFFFFFFFh																															

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 25-21. GPIO_OE Register Field Descriptions

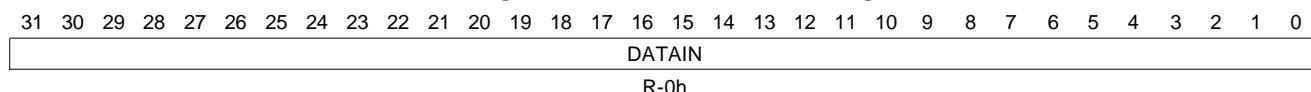
Bit	Field	Type	Reset	Description
31-0	OUTPUTEN[n]	R/W	FFFFFFFFFFh	Output Data Enable 0x0 = The corresponding GPIO port is configured as an output. 0x1 = The corresponding GPIO port is configured as an input.

25.4.1.17 GPIO_DATAIN Register (offset = 138h) [reset = 0h]

GPIO_DATAIN is shown in [Figure 25-23](#) and described in [Table 25-22](#).

The GPIO_DATAIN register is used to register the data that is read from the GPIO pins. The GPIO_DATAIN register is a read-only register. The input data is sampled synchronously with the interface clock and then captured in the GPIO_DATAIN register synchronously with the interface clock. So, after changing, GPIO pin levels are captured into this register after two interface clock cycles (the required cycles to synchronize and to write the data). When the AUTOIDLE bit in the system configuration register (GPIO_SYSConfig) is set, the GPIO_DATAIN read command has a 3 OCP cycle latency due to the data in sample gating mechanism. When the AUTOIDLE bit is not set, the GPIO_DATAIN read command has a 2 OCP cycle latency.

Figure 25-23. GPIO_DATAIN Register



LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 25-22. GPIO_DATAIN Register Field Descriptions

Bit	Field	Type	Reset	Description
31-0	DATAIN	R	0h	Sampled Input Data

25.4.1.18 GPIO_DATAOUT Register (offset = 13Ch) [reset = 0h]

GPIO_DATAOUT is shown in Figure 25-24 and described in Table 25-23.

The GPIO_DATAOUT register is used for setting the value of the GPIO output pins. Data is written to the GPIO_DATAOUT register synchronously with the interface clock. This register can be accessed with direct read/write operations or using the alternate Set/Clear feature. This feature enables to set or clear specific bits of this register with a single write access to the set data output register (GPIO_SETDATAOUT) or to the clear data output register (GPIO_CLEARDATAOUT) address.

Figure 25-24. GPIO_DATAOUT Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATAOUT																															
R/W-0h																															

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 25-23. GPIO_DATAOUT Register Field Descriptions

Bit	Field	Type	Reset	Description
31-0	DATAOUT	R/W	0h	Data to set on output pins

25.4.1.19 GPIO_LEVELDETECT0 Register (offset = 140h) [reset = 0h]

GPIO_LEVELDETECT0 is shown in [Figure 25-25](#) and described in [Table 25-24](#).

The GPIO_LEVELDETECT0 register is used to enable/disable for each input lines the low-level (0) detection to be used for the interrupt request generation. Enabling at the same time high-level detection and low-level detection for one given pin makes a constant interrupt generator.

Figure 25-25. GPIO_LEVELDETECT0 Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LEVELDETECT0[n]																															
R/W-0h																															

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 25-24. GPIO_LEVELDETECT0 Register Field Descriptions

Bit	Field	Type	Reset	Description
31-0	LEVELDETECT0[n]	R/W	0h	Low Level Interrupt Enable 0x0 = Disable the IRQ assertion on low-level detect. 0x1 = Enable the IRQ assertion on low-level detect.

25.4.1.20 GPIO_LEVELDETECT1 Register (offset = 144h) [reset = 0h]

GPIO_LEVELDETECT1 is shown in [Figure 25-26](#) and described in [Table 25-25](#).

The GPIO_LEVELDETECT1 register is used to enable/disable for each input lines the high-level (1) detection to be used for the interrupt request generation. Enabling at the same time high-level detection and low-level detection for one given pin makes a constant interrupt generator.

Figure 25-26. GPIO_LEVELDETECT1 Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LEVELDETECT1[n]																															
R/W-0h																															

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 25-25. GPIO_LEVELDETECT1 Register Field Descriptions

Bit	Field	Type	Reset	Description
31-0	LEVELDETECT1[n]	R/W	0h	High Level Interrupt Enable 0x0 = Disable the IRQ assertion on high-level detect. 0x1 = Enable the IRQ assertion on high-level detect.

25.4.1.21 GPIO_RISINGDETECT Register (offset = 148h) [reset = 0h]

GPIO_RISINGDETECT is shown in [Figure 25-27](#) and described in [Table 25-26](#).

The GPIO_RISINGDETECT register is used to enable/disable for each input lines the rising-edge (transition 0 to 1) detection to be used for the interrupt request generation.

Figure 25-27. GPIO_RISINGDETECT Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RISINGDETECT[n]																															
R/W-0h																															

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 25-26. GPIO_RISINGDETECT Register Field Descriptions

Bit	Field	Type	Reset	Description
31-0	RISINGDETECT[n]	R/W	0h	Rising Edge Interrupt Enable 0x0 = Disable IRQ on rising-edge detect. 0x1 = Enable IRQ on rising-edge detect.

25.4.1.22 GPIO_FALLINGDETECT Register (offset = 14Ch) [reset = 0h]

GPIO_FALLINGDETECT is shown in [Figure 25-28](#) and described in [Table 25-27](#).

The GPIO_FALLINGDETECT register is used to enable/disable for each input lines the falling-edge (transition 1 to 0) detection to be used for the interrupt request generation.

Figure 25-28. GPIO_FALLINGDETECT Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FALLINGDETECT[n]																															
R/W-0h																															

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 25-27. GPIO_FALLINGDETECT Register Field Descriptions

Bit	Field	Type	Reset	Description
31-0	FALLINGDETECT[n]	R/W	0h	Falling Edge Interrupt Enable 0x0 = Disable IRQ on falling-edge detect. 0x1 = Enable IRQ on falling-edge detect.

11. Interrupts

6.3 ARM Cortex-A8 Interrupts

Table 6-1. ARM Cortex-A8 Interrupts

Int Number	Acronym/name	Source	Signal Name
0	EMUINT	MPU Subsystem Internal	Emulation interrupt (EMUICINTR)
1	COMMTX	MPU Subsystem Internal	CortexA8 COMMTX
2	COMMRX	MPU Subsystem Internal	CortexA8 COMMRX
3	BENCH	MPU Subsystem Internal	CortexA8 NPMUIRQ
4	ELM_IRQ	ELM	Sinterrupt (Error location process completion)
5	Reserved		
6	Reserved		
7	NMI	External Pin (active low) ⁽¹⁾	nmi_int
8	Reserved		
9	L3DEBUG	L3	I3_FlagMux_top_FlagOut1
10	L3APPINT	L3	I3_FlagMux_top_FlagOut0
11	PRCMINT	PRCM	irq_mpu
12	EDMACOMPINT	TPCC (EDMA)	tpcc_int_pend_po0
13	EDMAMPERR	TPCC (EDMA)	tpcc_mpint_pend_po
14	EDMAERRINT	TPCC (EDMA)	tpcc_errint_pend_po
15	Reserved		
16	ADC_TSC_GENINT	ADC_TSC (Touchscreen Controller)	gen_intr_pend
17	USBSSINT	USBSS	usbss_intr_pend
18	USBINT0	USBSS	usb0_intr_pend
19	USBINT1	USBSS	usb1_intr_pend
20	PRU_ICSS_EVTOUT0	pr1_host[0] output/events exported from PRU-ICSS ⁽²⁾	pr1_host_intr0_intr_pend
21	PRU_ICSS_EVTOUT1	pr1_host[1] output/events exported from PRU-ICSS ⁽²⁾	pr1_host_intr1_intr_pend
22	PRU_ICSS_EVTOUT2	pr1_host[2] output/events exported from PRU-ICSS ⁽²⁾	pr1_host_intr2_intr_pend
23	PRU_ICSS_EVTOUT3	pr1_host[3] output/events exported from PRU-ICSS ⁽²⁾	pr1_host_intr3_intr_pend
24	PRU_ICSS_EVTOUT4	pr1_host[4] output/events exported from PRU-ICSS ⁽²⁾	pr1_host_intr4_intr_pend
25	PRU_ICSS_EVTOUT5	pr1_host[5] output/events exported from PRU-ICSS ⁽²⁾	pr1_host_intr5_intr_pend
26	PRU_ICSS_EVTOUT6	pr1_host[6] output/events exported from PRU-ICSS ⁽²⁾	pr1_host_intr6_intr_pend
27	PRU_ICSS_EVTOUT7	pr1_host[7] output/events exported from PRU-ICSS ⁽²⁾	pr1_host_intr7_intr_pend
28	MMCSD1INT	MMCSD1	SINTERRUPTN
29	MMCSD2INT	MMCSD2	SINTERRUPTN
30	I2C2INT	I2C2	POINTRPEND
31	eCAP0INT	eCAP0 event/interrupt	ecap_intr_intr_pend
32	GPIOINT2A	GPIO 2	POINTRPEND1
33	GPIOINT2B	GPIO 2	POINTRPEND2
34	USBWAKEUP	USBSS	slv0p_Swakeup
35	Reserved		

⁽¹⁾ For differences in operation based on AM335x silicon revisions, see [Section 1.2, Silicon Revision Functional Differences and Enhancements](#).

⁽²⁾ pr1_host_intr[0:7] corresponds to Host-2 to Host-9 of the PRU-ICSS interrupt controller.

Table 6-1. ARM Cortex-A8 Interrupts (continued)

Int Number	Acronym/name	Source	Signal Name
36	LCDCINT	LCDC	lcd_irq
37	GFXINT	SGX530	THALIAIRQ
38	Reserved		
39	ePWM2INT	eHRPWM2 (PWM Subsystem)	epwm_intr_intr_pend
40	3PGSWRXTHR0 (RX_THRESH_PULSE)	CPSW (Ethernet)	c0_rx_thresh_pend
41	3PGSWRXINT0 (RX_PULSE)	CPSW (Ethernet)	c0_rx_pend
42	3PGSWTXINT0 (TX_PULSE)	CPSW (Ethernet)	c0_tx_pend
43	3PGSWMISCO (MISC_PULSE)	CPSW (Ethernet)	c0_misc_pend
44	UART3INT	UART3	niq
45	UART4INT	UART4	niq
46	UART5INT	UART5	niq
47	eCAP1INT	eCAP1 (PWM Subsystem)	ecap_intr_intr_pend
48	Reserved		
49	Reserved		
50	Reserved		
51	Reserved		
52	DCAN0_INT0	DCAN0	dcan_intr0_intr_pend
53	DCAN0_INT1	DCAN0	dcan_intr1_intr_pend
54	DCAN0_PARITY	DCAN0	dcan_uerr_intr_pend
55	DCAN1_INT0	DCAN1	dcan_intr0_intr_pend
56	DCAN1_INT1	DCAN1	dcan_intr1_intr_pend
57	DCAN1_PARITY	DCAN1	dcan_uerr_intr_pend
58	ePWM0_TZINT	eHRPWM0 TZ interrupt (PWM Subsystem)	epwm_tz_intr_pend
59	ePWM1_TZINT	eHRPWM1 TZ interrupt (PWM Subsystem)	epwm_tz_intr_pend
60	ePWM2_TZINT	eHRPWM2 TZ interrupt (PWM Subsystem)	epwm_tz_intr_pend
61	eCAP2INT	eCAP2 (PWM Subsystem)	ecap_intr_intr_pend
62	GPIOINT3A	GPIO 3	POINTRPEND1
63	GPIOINT3B	GPIO 3	POINTRPEND2
64	MMCSD0INT	MMCSD0	SINTERRUPTN
65	McSPI0INT	McSPI0	SINTERRUPTN
66	TINT0	Timer0	POINTR_PEND
67	TINT1_1MS	DMTIMER_1ms	POINTR_PEND
68	TINT2	DMTIMER2	POINTR_PEND
69	TINT3	DMTIMER3	POINTR_PEND
70	I2C0INT	I2C0	POINTRPEND
71	I2C1INT	I2C1	POINTRPEND
72	UART0INT	UART0	niq
73	UART1INT	UART1	niq
74	UART2INT	UART2	niq
75	RTCINT	RTC	timer_intr_pend
76	RTCALARMINT	RTC	alarm_intr_pend
77	MBINT0	Mailbox0 (mail_u0_irq)	initiator_sinterrupt_q_n0
78	M3_TXEV	Wake M3 Subsystem	TXEV
79	eQEPOINT	eQEPO (PWM Subsystem)	eqep_intr_intr_pend
80	MCATXINT0	McASP0	mcasp_x_intr_pend

Table 6-1. ARM Cortex-A8 Interrupts (continued)

Int Number	Acronym/name	Source	Signal Name
81	MCARXINT0	McASP0	mcasp_r_intr_pend
82	MCATXINT1	McASP1	mcasp_x_intr_pend
83	MCARXINT1	McASP1	mcasp_r_intr_pend
84	Reserved		
85	Reserved		
86	ePWM0INT	eHRPWM0 (PWM Subsystem)	epwm_intr_intr_pend
87	ePWM1INT	eHRPWM1 (PWM Subsystem)	epwm_intr_intr_pend
88	eQEP1INT	eQEP1 (PWM Subsystem)	eqep_intr_intr_pend
89	eQEP2INT	eQEP2 (PWM Subsystem)	eqep_intr_intr_pend
90	DMA_INTR_PIN2	External DMA/Interrupt Pin2 (xdma_event_intr2)	pi_x_dma_event_intr2
91	WDT1INT (Public Watchdog)	WDTIMER1	PO_INT_PEND
92	TINT4	DMTIMER4	POINTR_PEND
93	TINT5	DMTIMER5	POINTR_PEND
94	TINT6	DMTIMER6	POINTR_PEND
95	TINT7	DMTIMER7	POINTR_PEND
96	GPIOINT0A	GPIO 0	POINTRPEND1
97	GPIOINT0B	GPIO 0	POINTRPEND2
98	GPIOINT1A	GPIO 1	POINTRPEND1
99	GPIOINT1B	GPIO 1	POINTRPEND2
100	GPMCINT	GPMC	gpmc_sinterrupt
101	DDRERR0	EMIF	sys_err_intr_pend
102	Reserved		
103	Reserved		
104	Reserved		
105	Reserved		
106	Reserved		
107	Reserved		
108	Reserved		
109	Reserved		
110	Reserved		
111	Reserved		
112	TCERRINT0	TPTC0	tptc_erint_pend_po
113	TCERRINT1	TPTC1	tptc_erint_pend_po
114	TCERRINT2	TPTC2	tptc_erint_pend_po
115	ADC_TSC_PENINT	ADC_TSC	pen_intr_pend
116	Reserved		
117	Reserved		
118	Reserved		
119	Reserved		
120	SMRFLX_MPU subsystem	Smart Reflex 0	intrpend
121	SMRFLX_Core	Smart Reflex 1	intrpend
122	Reserved		
123	DMA_INTR_PIN0	External DMA/Interrupt Pin0 (xdma_event_intr0)	pi_x_dma_event_intr0
124	DMA_INTR_PIN1	External DMA/Interrupt Pin1 (xdma_event_intr1)	pi_x_dma_event_intr1
125	McSPI1INT	McSPI1	SINTERRUPTN

Table 6-1. ARM Cortex-AS Interrupts (continued)

Int Number	Acronym/name	Source	Signal Name
126	Reserved		
127	Reserved		

12. EHRPWM Register

15.2.4.1 TBCTL Register (offset = 0h) [reset = 0h]

TBCTL is shown in Figure 15-71 and described in Table 15-59.

Figure 15-71. TBCTL Register

15	14	13	12	11	10	9	8
FREE_SOFT		PHSDIR		CLKDIV		HSPCLKDIV	
R/W-0h		R/W-0h		R/W-0h		R/W-0h	
7	6	5	4	3	2	1	0
HSPCLKDIV	SWFSYNC		SYNCOSEL	PRDLD	PHSEN	CTRMODE	
R/W-0h	R/W-0h		R/W-0h	R/W-0h	R/W-0h	R/W-0h	

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 15-59. TBCTL Register Field Descriptions

Bit	Field	Type	Reset	Description
15-14	FREE_SOFT	R/W	0h	Emulation Mode Bits. These bits select the behavior of the ePWM time-base counter during emulation suspend events. Emulation debug events can be set up in the Debug Subsystem. 0h (R/W) = Stop after the next time-base counter increment or decrement 1h (R/W) = Stop when counter completes a whole cycle. (a) Up-count mode: stop when the time-base counter = period (TBCNT = TBPRD). (b) Down-count mode: stop when the time-base counter = 0000 (TBCNT = 0000h). (c) Up-down-count mode: stop when the time-base counter = 0000 (TBCNT = 0000h). 2h (R/W) = Free run 3h (R/W) = Free run
13	PHSDIR	R/W	0h	Phase Direction Bit. This bit is only used when the time-base counter is configured in the up-down-count mode. The PHSDIR bit indicates the direction the time-base counter (TBCNT) will count after a synchronization event occurs and a new phase value is loaded from the phase (TBPHS) register. This is irrespective of the direction of the counter before the synchronization event. In the up-count and down-count modes this bit is ignored. 0h (R/W) = Count down after the synchronization event. 1h (R/W) = Count up after the synchronization event.
12-10	CLKDIV	R/W	0h	Time-base Clock Prescale Bits. These bits determine part of the time-base clock prescale value. TBCLK = SYCLKOUT/(HSPCLKDIV - CLKDIV) 0h (R/W) = /1 (default on reset) 1h (R/W) = /2 2h (R/W) = /4 3h (R/W) = /8 4h (R/W) = /16 5h (R/W) = /32 6h (R/W) = /64 7h (R/W) = /128

Table 15-59. TBCTL Register Field Descriptions (continued)

Bit	Field	Type	Reset	Description
9-7	HSPCLKDIV	R/W	0h	<p>High-Speed Time-base Clock Prescale Bits. These bits determine part of the time-base clock prescale value. $TBCLK = SYSLKOUT / (HSPCLKDIV - CLKDIV)$. This divisor emulates the HSPCLK in the TMS320x281x system as used on the Event Manager (EV) peripheral.</p> <p>0h (R/W) = /1 1h (R/W) = /2 (default on reset) 2h (R/W) = /4 3h (R/W) = /6 4h (R/W) = /8 5h (R/W) = /10 6h (R/W) = /12 7h (R/W) = /14</p>
6	SWFSYNC	R/W	0h	<p>Software Forced Synchronization Pulse. 0h (R/W) = Writing a 0 has no effect and reads always return a 0. 1h (R/W) = Writing a 1 forces a one-time synchronization pulse to be generated. This event is ORed with the EPWMxSYNCI input of the ePWM module. SWFSYNC is valid (operates) only when EPWMxSYNCI is selected by SYNCSEL = 00.</p>
5-4	SYNCSEL	R/W	0h	<p>Synchronization Output Select. These bits select the source of the EPWMxSYNCO signal. 0h (R/W) = EPWMxSYNC: 1h (R/W) = CTR = 0: Time-base counter equal to zero (TBCNT = 0000h) 2h (R/W) = CTR = CMPB : Time-base counter equal to counter-compare B (TBCNT = CMPB) 3h (R/W) = Disable EPWMxSYNCO signal</p>
3	PRDLD	R/W	0h	<p>Active Period Register Load From Shadow Register Select 0h (R/W) = The period register (TBPRD) is loaded from its shadow register when the time-base counter, TBCNT, is equal to zero. A write or read to the TBPRD register accesses the shadow register. 1h (R/W) = Load the TBPRD register immediately without using a shadow register. A write or read to the TBPRD register directly accesses the active register.</p>
2	PHSEN	R/W	0h	<p>Counter Register Load From Phase Register Enable 0h (R/W) = Do not load the time-base counter (TBCNT) from the time-base phase register (TBPHS) 1h (R/W) = Load the time-base counter with the phase register when an EPWMxSYNCI input signal occurs or when a software synchronization is forced by the SWFSYNC bit.</p>
1-0	CTRMODE	R/W	0h	<p>Counter Mode. The time-base counter mode is normally configured once and not changed during normal operation. If you change the mode of the counter, the change will take effect at the next TBCLK edge and the current counter value shall increment or decrement from the value before the mode change. These bits set the time-base counter mode of operation as follows: 0h (R/W) = Up-count mode 1h (R/W) = Down-count mode 2h (R/W) = Up-down-count mode 3h (R/W) = Stop-freeze counter operation (default on reset)</p>

15.2.4.6 TBPRD Register (offset = Ah) [reset = 0h]

TBPRD is shown in Figure 15-76 and described in Table 15-64.

Figure 15-76. TBPRD Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TBPRD								R/W-0h							

LEGEND: R/W = Read/Write; R = Read only; W1toCI = Write 1 to clear bit; -n = value after reset

Table 15-64. TBPRD Register Field Descriptions

Bit	Field	Type	Reset	Description
15-0	TBPRD	R/W	0h	<p>These bits determine the period of the time-base counter. This sets the PWM frequency. Shadowing of this register is enabled and disabled by the TBCTL[PRDLD] bit. By default this register is shadowed.</p> <ul style="list-style-type: none"> (a) If TBCTL[PRDLD] = 0, then the shadow is enabled and any write or read will automatically go to the shadow register. In this case, the active register will be loaded from the shadow register when the time-base counter equals zero. (b) If TBCTL[PRDLD] = 1, then the shadow is disabled and any write or read will go directly to the active register, that is the register actively controlling the hardware. (c) The active and shadow registers share the same memory map address.

15.2.4.11 AQCTLA Register (offset = 16h) [reset = 0h]

AQCTLA is shown in [Figure 15-81](#) and described in [Table 15-69](#).

Figure 15-81. AQCTLA Register

15	14	13	12	11	10	9	8
RESERVED			CBD			CBU	
R-0h			R/W-0h			R/W-0h	
7	6	5	4	3	2	1	0
CAD		CAU		PRD		ZRO	
R/W-0h		R/W-0h		R/W-0h		R/W-0h	

LEGEND: R/W = Read/Write; R = Read only; W1toCI = Write 1 to clear bit; -n = value after reset

Table 15-69. AQCTLA Register Field Descriptions

Bit	Field	Type	Reset	Description
15-12	RESERVED	R	0h	
11-10	CBD	R/W	0h	Action when the time-base counter equals the active CMPB register and the counter is decrementing. 0h (R/W) = Do nothing (action disabled) 1h (R/W)(Read) = Cleaforce EPWMxA output low. 2h (R/W) = Set: force EPWMxA output high. 3h (R/W) = Toggle EPWMxA output: low output signal will be forced high, and a high signal will be forced low.
9-8	CBU	R/W	0h	Action when the counter equals the active CMPB register and the counter is incrementing. 0h (R/W) = Do nothing (action disabled) 1h (R/W)(Read) = Cleaforce EPWMxA output low. 2h (R/W) = Set: force EPWMxA output high. 3h (R/W) = Toggle EPWMxA output: low output signal will be forced high, and a high signal will be forced low.
7-6	CAD	R/W	0h	Action when the counter equals the active CMPA register and the counter is decrementing. 0h (R/W) = Do nothing (action disabled) 1h (R/W)(Read) = Cleaforce EPWMxA output low. 2h (R/W) = Set: force EPWMxA output high. 3h (R/W) = Toggle EPWMxA output: low output signal will be forced high, and a high signal will be forced low.
5-4	CAU	R/W	0h	Action when the counter equals the active CMPA register and the counter is incrementing. 0h (R/W) = Do nothing (action disabled) 1h (R/W)(Read) = Cleaforce EPWMxA output low. 2h (R/W) = Set: force EPWMxA output high. 3h (R/W) = Toggle EPWMxA output: low output signal will be forced high, and a high signal will be forced low.
3-2	PRD	R/W	0h	Action when the counter equals the period. Note: By definition, in count up-down mode when the counter equals period the direction is defined as 0 or counting down. 0h (R/W) = Do nothing (action disabled) 1h (R/W)(Read) = Cleaforce EPWMxA output low. 2h (R/W) = Set: force EPWMxA output high. 3h (R/W) = Toggle EPWMxA output: low output signal will be forced high, and a high signal will be forced low.

Table 15-69. AQCTLA Register Field Descriptions (continued)

Bit	Field	Type	Reset	Description
1-0	ZRO	R/W	0h	Action when counter equals zero. Note: By definition, in count up-down mode when the counter equals 0 the direction is defined as 1 or counting up. 0h (R/W) = Do nothing (action disabled) 1h (R/W)(Read) = Clearforce EPWMxA output low. 2h (R/W) = Set: force EPWMxA output high. 3h (R/W) = Toggle EPWMxA output: low output signal will be forced high, and a high signal will be forced low.

15.2.4.9 CMPA Register (offset = 12h) [reset = 0h]

CMPA is shown in Figure 15-79 and described in Table 15-67.

Figure 15-79. CMPA Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CMPA								R/W-0h							

LEGEND: R/W = Read/Write; R = Read only; W1toCI = Write 1 to clear bit; -n = value after reset

Table 15-67. CMPA Register Field Descriptions

Bit	Field	Type	Reset	Description
15-0	CMPA	R/W	0h	<p>The value in the active CMPA register is continuously compared to the time-base counter (TBCNT). When the values are equal, the counter-compare module generates a "time-base counter equal to counter compare A" event. This event is sent to the action-qualifier where it is qualified and converted into one or more actions. These actions can be applied to either the EPWMxA or the EPWMxB output depending on the configuration of the AQCTLA and AQCTLB registers. The actions that can be defined in the AQCTLA and AQCTLB registers include the following.</p> <ul style="list-style-type: none"> (a) Do nothing the event is ignored. (b) Clear: Pull the EPWMxA and/or EPWMxB signal low. (c) Set: Pull the EPWMxA and/or EPWMxB signal high. (d) Toggle: Toggle the EPWMxA and/or EPWMxB signal. Shadowing of this register is enabled and disabled by the CMP-CTL[SHDWAMODE] bit. <p>By default this register is shadowed.</p> <ul style="list-style-type: none"> (a) If CMPCTL[SHDWAMODE] = 0, then the shadow is enabled and any write or read will automatically go to the shadow register. In this case, the CMPCTL[LOADAMODE] bit field determines which event will load the active register from the shadow register. (b) Before a write, the CMPCTL[SHDWAFULL] bit can be read to determine if the shadow register is currently full. (c) If CMPCTL[SHDWAMODE] = 1, then the shadow register is disabled and any write or read will go directly to the active register, that is the register actively controlling the hardware. (d) In either mode, the active and shadow registers share the same memory map address.

13. DM-Timer Register

20.1.4 Use Cases

20.1.5 TIMER Registers

[Table 20-10](#) lists the memory-mapped registers for the TIMER. All register offset addresses not listed in [Table 20-10](#) should be considered as reserved locations and the register contents should not be modified.

Table 20-10. TIMER REGISTERS

Offset	Acronym	Register Name	Section
00h	TIDR	Identification Register	Section 20.1.5.1
10h	TIOCP_CFG	Timer OCP Configuration Register	Section 20.1.5.2
20h	IRQ_EOI	Timer IRQ End-of-Interrupt Register	Section 20.1.5.3
24h	IRQSTATUS_RAW	Timer Status Raw Register	Section 20.1.5.4
28h	IRQSTATUS	Timer Status Register	Section 20.1.5.5
2Ch	IRQENABLE_SET	Timer Interrupt Enable Set Register	Section 20.1.5.6
30h	IRQENABLE_CLR	Timer Interrupt Enable Clear Register	Section 20.1.5.7
34h	IRQWAKEEN	Timer IRQ Wakeup Enable Register	Section 20.1.5.8
38h	TCLR	Timer Control Register	Section 20.1.5.9
3Ch	TCRR	Timer Counter Register	Section 20.1.5.10
40h	TLDR	Timer Load Register	Section 20.1.5.11
44h	TTGR	Timer Trigger Register	Section 20.1.5.12
48h	TWPS	Timer Write Posting Bits Register	Section 20.1.5.13
4Ch	TMAR	Timer Match Register	Section 20.1.5.14
50h	TCAR1	Timer Capture Register	Section 20.1.5.15
54h	TSICR	Timer Synchronous Interface Control Register	Section 20.1.5.16
58h	TCAR2	Timer Capture Register	Section 20.1.5.17

20.1.5.9 TCLR Register (offset = 38h) [reset = 0h]

TCLR is shown in [Figure 20-17](#) and described in [Table 20-19](#).

When the TCM field passed from (00) to any other combination then the TCAR_IT_FLAG and the edge detection logic are cleared. The ST bit of TCLR register may be updated from the OCP interface or reset due to one-shot overflow. The OCP interface update has the priority.

Figure 20-17. TCLR Register

31	30	29	28	27	26	25	24
Reserved							
R-0h							
23	22	21	20	19	18	17	16
Reserved							
R-0h							
15	14	13	12	11	10	9	8
Reserved	GPO_CFG	CAPT_MODE	PT	TRG	TCM		
R-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h		R/W-0h	
7	6	5	4	3	2	1	0
SCPWM	CE	PRE		PTV		AR	ST
R/W-0h	R/W-0h	R/W-0h		R/W-0h		R/W-0h	R/W-0h

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 20-19. TCLR Register Field Descriptions

Bit	Field	Type	Reset	Description
31-15	Reserved	R	0h	
14	GPO_CFG	R/W	0h	General purpose output this register drives directly the PORGPOCFG output pin 0x0 = PORGPOCFG drives 0 and configures the timer pin as an output. 0x1 = PORGPOCFG drives 1 and configures the timer pin as an input.
13	CAPT_MODE	R/W	0h	Capture mode. 0x0 = Single capture 0x1 = Capture on second event
12	PT	R/W	0h	Pulse or toggle mode on PORTIMERPWM output pin 0x0 = Pulse 0x1 = Toggle
11-10	TRG	R/W	0h	Trigger output mode on PORTIMERPWM output pin 0x0 = No trigger 0x1 = Trigger on overflow 0x2 = Trigger on overflow and match 0x3 = Reserved
9-8	TCM	R/W	0h	Transition Capture Mode on PIEVENTCPT input pin 0x0 = No capture 0x1 = Capture on low to high transition 0x2 = Capture on high to low transition 0x3 = Capture on both edge transition
7	SCPWM	R/W	0h	This bit should be set or clear while the timer is stopped or the trigger is off 0x0 = Clear the PORTIMERPWM output pin and select positive pulse for pulse mode 0x1 = Set the PORTIMERPWM output pin and select negative pulse for pulse mode

Table 20-19. TCLR Register Field Descriptions (continued)

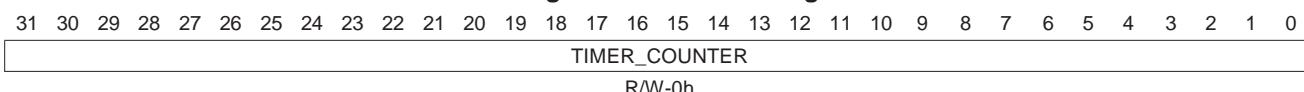
Bit	Field	Type	Reset	Description
6	CE	R/W	0h	0x0 = Compare mode is disabled 0x1 = Compare mode is enabled
5	PRE	R/W	0h	Prescaler enable 0x0 = The TIMER clock input pin clocks the counter 0x1 = The divided input pin clocks the counter
4-2	PTV	R/W	0h	Pre-scale clock Timer value
1	AR	R/W	0h	0x0 = One shot timer 0x1 = Auto-reload timer
0	ST	R/W	0h	In the case of one-shot mode selected (AR = 0), this bit is automatically reset by internal logic when the counter is overflowed. 0x0(READ) = Stop timeOnly the counter is frozen 0x1 = Start timer

20.1.5.10 TCRR Register (offset = 3Ch) [reset = 0h]

TCRR is shown in [Figure 20-18](#) and described in [Table 20-20](#).

The TCRR register is a 32-bit register, 16-bit addressable. The MCU can perform a 32-bit access or two 16-bit accesses to access the register. Note that since the OCP clock is completely asynchronous with the timer clock, some synchronization is done in order to make sure that the TCRR value is not read while it is being incremented. In 16-bit mode the following sequence must be followed to read the TCRR register properly: First, perform an OCP Read Transaction to Read the lower 16-bit of the TCRR register (offset = 28h). When the TCRR is read and synchronized, the lower 16-bit LSBs are driven onto the output OCP data bus and the upper 16-bit MSBs of the TCRR register are stored in a temporary register. Second, perform an OCP Read Transaction to read the upper 16-bit of the TCRR register (offset = 2Ah). During this Read, the value of the upper 16-bit MSBs that has been temporary register is forwarded onto the output OCP data bus. So, to read the value of TCRR correctly, the first OCP read access has to be to the lower 16-bit (offset = 28h), followed by OCP read access to the upper 16-bit (offset = 2Ah). As the TCRR is updated using more sources (shadow_in_tcrr, incremented value of tcrr, TLDR and 0), a priority order will be defined: The first priority is the OCP update. The second is the reload way (triggered through TTGR reg. or following an auto-reload overflow). The third is the one-shot overflow reset to 0. The last is the incremented value.

Figure 20-18. TCRR Register



LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 20-20. TCRR Register Field Descriptions

Bit	Field	Type	Reset	Description
31-0	TIMER_COUNTER	R/W	0h	Value of TIMER counter

20.1.5.11 TLDR Register (offset = 40h) [reset = 0h]

TLDR is shown in [Figure 20-19](#) and described in [Table 20-21](#).

LOAD_VALUE must be different than the timer overflow value (FFFF FFFFh).

Figure 20-19. TLDR Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LOAD_VALUE																															
R/W-0h																															

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 20-21. TLDR Register Field Descriptions

Bit	Field	Type	Reset	Description
31-0	LOAD_VALUE	R/W	0h	Timer counter value loaded on overflow in auto-reload mode or on TTGR write access

20.1.5.14 TMAR Register (offset = 4Ch) [reset = 0h]

TMAR is shown in [Figure 20-22](#) and described in [Table 20-24](#).

The compare logic consists of a 32-bit wide, read/write data TMAR register and logic to compare counter s current value with the value stored in the TMAR register.

Figure 20-22. TMAR Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COMPARE_VALUE																															
R/W-0h																															

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 20-24. TMAR Register Field Descriptions

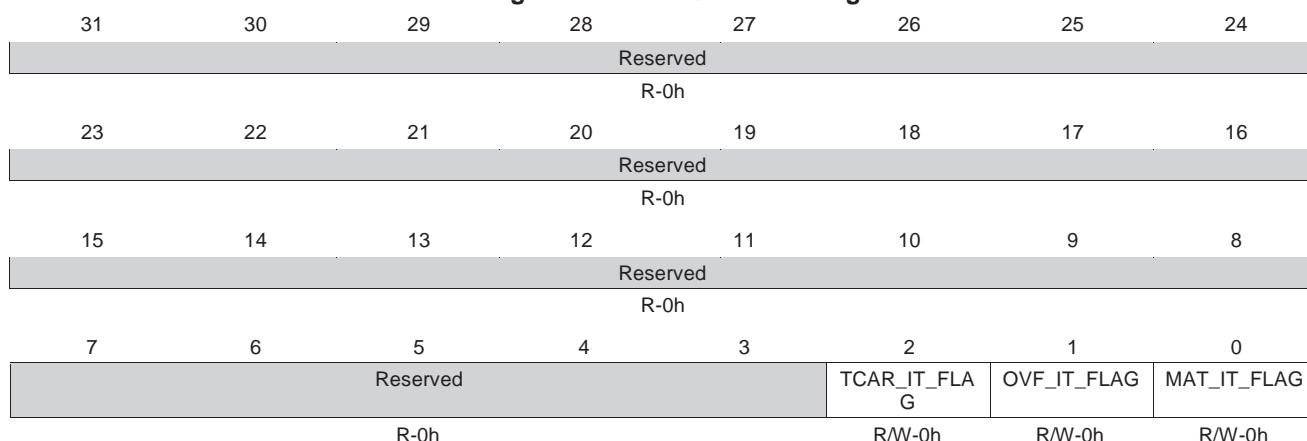
Bit	Field	Type	Reset	Description
31-0	COMPARE_VALUE	R/W	0h	Value to be compared to the timer counter

20.1.5.5 IRQSTATUS Register (offset = 28h) [reset = 0h]

IRQSTATUS is shown in [Figure 20-13](#) and described in [Table 20-15](#).

Component interrupt request status. Check the corresponding secondary status register. Enabled status is not set unless event is enabled. Write 1 to clear the status after interrupt has been serviced (raw status gets cleared, that is, even if not enabled).

Figure 20-13. IRQSTATUS Register



LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 20-15. IRQSTATUS Register Field Descriptions

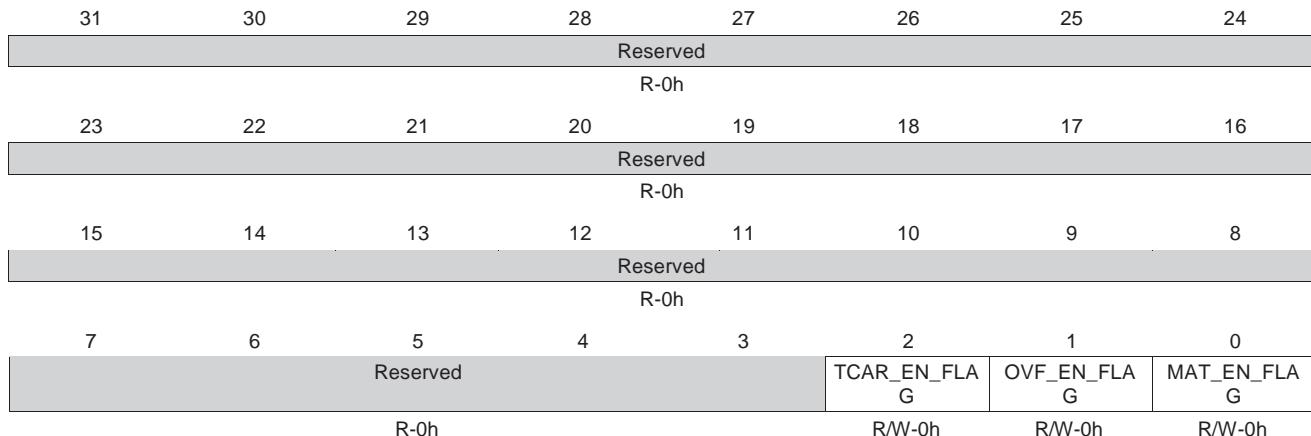
Bit	Field	Type	Reset	Description
31-3	Reserved	R	0h	
2	TCAR_IT_FLAG	R/W	0h	IRQ status for Capture 0x0(W) = No action 0x0(R) = No event pending 0x1(W) = Clear pending event, if any 0x1(R) = IRQ event pending
1	OVF_IT_FLAG	R/W	0h	IRQ status for Overflow 0x0(W) = No action 0x0(R) = No event pending 0x1(W) = Clear pending event, if any 0x1(R) = IRQ event pending
0	MAT_IT_FLAG	R/W	0h	IRQ status for Match 0x0(W) = No action 0x0(R) = No event pending 0x1(W) = Clear pending event, if any 0x1(R) = IRQ event pending

20.1.5.6 IRQENABLE_SET Register (offset = 2Ch) [reset = 0h]

IRQENABLE_SET is shown in [Figure 20-14](#) and described in [Table 20-16](#).

Component interrupt request enable. Write 1 to set (enable interrupt). Readout equal to corresponding _CLR register.

Figure 20-14. IRQENABLE_SET Register



LEGEND: R/W = Read/Write; R = Read only; W1toCI = Write 1 to clear bit; -n = value after reset

Table 20-16. IRQENABLE_SET Register Field Descriptions

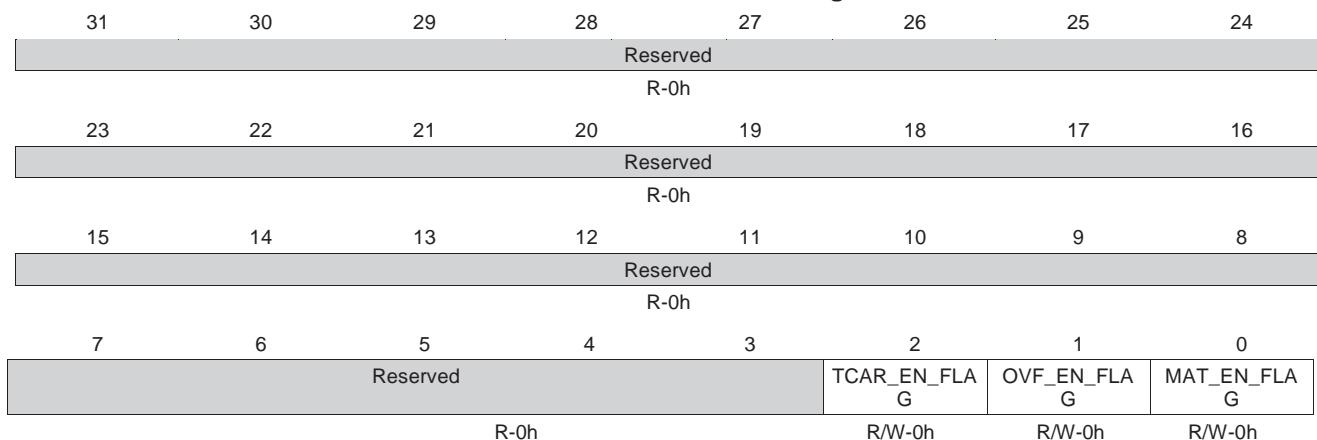
Bit	Field	Type	Reset	Description
31-3	Reserved	R	0h	
2	TCAR_EN_FLAG	R/W	0h	IRQ enable for Capture 0x0(W) = No action 0x0(R) = IRQ event is disabled 0x1(W) = Set IRQ enable 0x1(R) = IRQ event is enabled
1	OVF_EN_FLAG	R/W	0h	IRQ enable for Overflow 0x0(W) = No action 0x0(R) = IRQ event is disabled 0x1(W) = Set IRQ enable 0x1(R) = IRQ event is enabled
0	MAT_EN_FLAG	R/W	0h	IRQ enable for Match 0x0(W) = No action 0x0(R) = IRQ event is disabled 0x1(W) = Set IRQ enable 0x1(R) = IRQ event is enabled

*DMTimer***20.1.5.7 IRQENABLE_CLR Register (offset = 30h) [reset = 0h]**

IRQENABLE_CLR is shown in [Figure 20-15](#) and described in [Table 20-17](#).

Component interrupt request enable. Write 1 to clear (disable interrupt). Readout equal to corresponding _SET register.

**Figure 20-15.
IRQENABLE_CLR Register**



LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 20-17. IRQENABLE_CLR Register Field Descriptions

Bit	Field	Type	Reset	Description
31-3	Reserved	R	0h	
2	TCAR_EN_FLAG	R/W	0h	IRQ enable for Capture 0x0(W) = No action 0x0(R) = IRQ event is disabled 0x1(W) = Clear IRQ enable 0x1(R) = IRQ event is enabled
1	OVF_EN_FLAG	R/W	0h	IRQ enable for Overflow 0x0(W) = No action 0x0(R) = IRQ event is disabled 0x1(W) = Clear IRQ enable 0x1(R) = IRQ event is enabled
0	MAT_EN_FLAG	R/W	0h	IRQ enable for Match 0x0(W) = No action 0x0(R) = IRQ event is disabled 0x1(W) = Clear IRQ enable 0x1(R) = IRQ event is enabled

14.

Touchscreen Controller (ADC) Register

12.5 Touchscreen Controller Registers

12.5.1 TSC_ADC_SS Registers

Table 12-4 lists the memory-mapped registers for the TSC_ADC_SS. All register offset addresses not listed in Table 12-4 should be considered as reserved locations and the register contents should not be modified.

Table 12-4. TSC_ADC_SS REGISTERS

Offset	Acronym	Register Name	Section
0h	REVISION		Section 12.5.1.1
10h	SYSCONFIG		Section 12.5.1.2
24h	IRQSTATUS_RAW		Section 12.5.1.3
28h	IRQSTATUS		Section 12.5.1.4
2Ch	IRQENABLE_SET		Section 12.5.1.5
30h	IRQENABLE_CLR		Section 12.5.1.6
34h	IRQWAKEUP		Section 12.5.1.7
38h	DMAENABLE_SET		Section 12.5.1.8
3Ch	DMAENABLE_CLR		Section 12.5.1.9
40h	CTRL		Section 12.5.1.10
44h	ADCSTAT		Section 12.5.1.11
48h	ADCRANGE		Section 12.5.1.12
4Ch	ADC_CLKDIV		Section 12.5.1.13
50h	ADC_MISC		Section 12.5.1.14
54h	STEPENABLE		Section 12.5.1.15
58h	IDLECONFIG		Section 12.5.1.16
5Ch	TS_CHARGE_STEPCONFIG		Section 12.5.1.17
60h	TS_CHARGE_DELAY		Section 12.5.1.18
64h	STEPCONFIG1		Section 12.5.1.19
68h	STEPDELAY1		Section 12.5.1.20
6Ch	STEPCONFIG2		Section 12.5.1.21
70h	STEPDELAY2		Section 12.5.1.22
74h	STEPCONFIG3		Section 12.5.1.23
78h	STEPDELAY3		Section 12.5.1.24
7Ch	STEPCONFIG4		Section 12.5.1.25
80h	STEPDELAY4		Section 12.5.1.26
84h	STEPCONFIG5		Section 12.5.1.27
88h	STEPDELAY5		Section 12.5.1.28
8Ch	STEPCONFIG6		Section 12.5.1.29
90h	STEPDELAY6		Section 12.5.1.30
94h	STEPCONFIG7		Section 12.5.1.31
98h	STEPDELAY7		Section 12.5.1.32
9Ch	STEPCONFIG8		Section 12.5.1.33
A0h	STEPDELAY8		Section 12.5.1.34
A4h	STEPCONFIG9		Section 12.5.1.35
A8h	STEPDELAY9		Section 12.5.1.36
ACh	STEPCONFIG10		Section 12.5.1.37
B0h	STEPDELAY10		Section 12.5.1.38
B4h	STEPCONFIG11		Section 12.5.1.39
B8h	STEPDELAY11		Section 12.5.1.40
BCh	STEPCONFIG12		Section 12.5.1.41

Table 12-4. TSC_ADC_SS REGISTERS (continued)

Offset	Acronym	Register Name	Section
C0h	STEPDELAY12		Section 12.5.1.42
C4h	STEPCONFIG13		Section 12.5.1.43
C8h	STEPDELAY13		Section 12.5.1.44
CCh	STEPCONFIG14		Section 12.5.1.45
D0h	STEPDELAY14		Section 12.5.1.46
D4h	STEPCONFIG15		Section 12.5.1.47
D8h	STEPDELAY15		Section 12.5.1.48
DCh	STEPCONFIG16		Section 12.5.1.49
E0h	STEPDELAY16		Section 12.5.1.50
E4h	FIFO0COUNT		Section 12.5.1.51
E8h	FIFO0THRESHOLD		Section 12.5.1.52
EC _h	DMA0REQ		Section 12.5.1.53
F0h	FIFO1COUNT		Section 12.5.1.54
F4h	FIFO1THRESHOLD		Section 12.5.1.55
F8h	DMA1REQ		Section 12.5.1.56
100h	FIFO0DATA		Section 12.5.1.57
200h	FIFO1DATA		Section 12.5.1.58

12.5.1.10 CTRL Register (offset = 40h) [reset = 0h]

CTRL is shown in [Figure 12-14](#) and described in [Table 12-14](#).

@TSC_ADC_SS Control Register

Figure 12-14. CTRL Register

31	30	29	28	27	26	25	24
Reserved							
R-0h							
23	22	21	20	19	18	17	16
Reserved							
R-0h							
15	14	13	12	11	10	9	8
Reserved						HW_preempt	HW_event_mapping
R-0h							
7	6	5	4	3	2	1	0
Touch_Screen_Enable	AFE_Pen_Ctrl		Power_Down	ADC_Bias_Select	StepConfig_WriteProtect_n_active_low	Step_ID_tag	Enable
R/W-0h	R/W-0h		R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 12-14. CTRL Register Field Descriptions

Bit	Field	Type	Reset	Description
31-10	Reserved	R	0h	
9	HW_preempt	R/W	0h	0 = SW steps are not pre-empted by HW events. 1 = SW steps are pre-empted by HW events
8	HW_event_mapping	R/W	0h	0 = Map HW event to Pen touch irq (from AFE). 1 = Map HW event to HW event input.
7	Touch_Screen_Enable	R/W	0h	0 = Touchscreen transistors disabled. 1 = Touchscreen transistors enabled
6-5	AFE_Pen_Ctrl	R/W	0h	These two bits are sent directly to the AFE Pen Ctrl inputs. Bit 6 controls the Wiper touch (5 wire modes) Bit 5 controls the X+ touch (4 wire modes) User also needs to make sure the ground path is connected properly for pen interrupt to occur (using the StepConfig registers) Refer to section 4 interrupts for more information
4	Power_Down	R/W	0h	ADC Power Down control. 0 = AFE is powered up (default). 1 = Write 1 to power down AFE (the tsc_adc_ss enable (bit 0) should also be set to off)
3	ADC_Bias_Select	R/W	0h	Select Bias to AFE. 0 = Internal. 1 = Reserved.
2	StepConfig_WriteProtect_n_active_low	R/W	0h	0 = Step configuration registers are protected (not writable). 1 = Step configuration registers are not protected (writable).
1	Step_ID_tag	R/W	0h	Writing 1 to this bit will store the Step ID number with the captured ADC data in the FIFO. 0 = Write zeroes. 1 = Store the channel ID tag.
0	Enable	R/W	0h	TSC_ADC_SS module enable bit. After programming all the steps and configuration registers, write a 1 to this bit to turn on TSC_ADC_SS. Writing a 0 will disable the module (after the current conversion).

12.5.1.15 STEPENABLE Register (offset = 54h) [reset = 0h]

STEPENABLE is shown in Figure 12-19 and described in Table 12-19. Step Enable

Figure 12-19. STEPENABLE Register

31	30	29	28	27	26	25	24
Reserved							
R-0h							
23	22	21	20	19	18	17	16
Reserved							
R-0h							
R/W-0h							
15	14	13	12	11	10	9	8
STEP15	STEP14	STEP13	STEP12	STEP11	STEP10	STEP9	STEP8
R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h
7	6	5	4	3	2	1	0
STEP7	STEP6	STEP5	STEP4	STEP3	STEP2	STEP1	TS_Charge
R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 12-19. STEPENABLE Register Field Descriptions

Bit	Field	Type	Reset	Description
31-17	Reserved	R	0h	RESERVED.
16	STEP16	R/W	0h	Enable step 16
15	STEP15	R/W	0h	Enable step 15
14	STEP14	R/W	0h	Enable step 14
13	STEP13	R/W	0h	Enable step 13
12	STEP12	R/W	0h	Enable step 12
11	STEP11	R/W	0h	Enable step 11
10	STEP10	R/W	0h	Enable step 10
9	STEP9	R/W	0h	Enable step 9
8	STEP8	R/W	0h	Enable step 8
7	STEP7	R/W	0h	Enable step 7
6	STEP6	R/W	0h	Enable step 6
5	STEP5	R/W	0h	Enable step 5
4	STEP4	R/W	0h	Enable step 4
3	STEP3	R/W	0h	Enable step 3
2	STEP2	R/W	0h	Enable step 2
1	STEP1	R/W	0h	Enable step 1
0	TS_Charge	R/W	0h	Enable TS Charge step

12.5.1.19 STEPCONFIG1 Register (offset = 64h) [reset = 0h]

STEPCONFIG1 is shown in [Figure 12-23](#) and described in [Table 12-23](#). Step Configuration 1

Figure 12-23. STEPCONFIG1 Register

31	30	29	28	27	26	25	24
		Reserved		Range_check	FIFO_select	Diff_CNTRL	SEL_RFIM_SW_C_1_0
		R/W-0h		R/W-0h	R/W-0h	R/W-0h	R/W-0h
23	22	21	20	19	18	17	16
SEL_RFIM_SW_C_1_0		SEL_INP_SWC_3_0			SEL_INM_SWC_3_0		
R/W-0h		R/W-0h			R/W-0h		
15	14	13	12	11	10	9	8
SEL_INM_SW_C_3_0		SEL_RFP_SWC_2_0		WPNSW_SWC	YPNSW_SWC	XNPSW_SWC	YNNSW_SWC
R/W-0h		R/W-0h		R/W-0h	R/W-0h	R/W-0h	R/W-0h
7	6	5	4	3	2	1	0
YPPSW_SWC	XNNSW_SWC	XPPSW_SWC		Averaging		Mode	
R/W-0h	R/W-0h	R/W-0h		R/W-0h		R/W-0h	

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 12-23. STEPCONFIG1 Register Field Descriptions

Bit	Field	Type	Reset	Description
31-28	Reserved	R/W	0h	
27	Range_check	R/W	0h	0 = Disable out-of-range check. 1 = Compare ADC data with range check register.
26	FIFO_select	R/W	0h	Sampled data will be stored in FIFO. 0 = FIFO. 1 = FIFO 1.
25	Diff_CNTRL	R/W	0h	Differential Control Pin
24-23	SEL_RFIM_SWC_1_0	R/W	0h	SEL_RFIM pins SW configuration. 00 = VSSA. 01 = XNUR. 10 = YNLR. 11 = VREFN.
22-19	SEL_INP_SWC_3_0	R/W	0h	SEL_INP pins SW configuration. 0000 = Channel 1. 0111 = Channel 8. 1xxx = VREFN.
18-15	SEL_INM_SWC_3_0	R/W	0h	SEL_INM pins for negative differential. 0000 = Channel 1. 0111 = Channel 8. 1xxx = VREFN.
14-12	SEL_RFP_SWC_2_0	R/W	0h	SEL_RFP pins SW configuration. 000 = VDDA. 001 = XPUL. 010 = YPLL. 011 = VREFP. 1xx = INTREF.
11	WPNSW_SWC	R/W	0h	WPNSW pin SW configuration
10	YPNSW_SWC	R/W	0h	YPNSW pin SW configuration
9	XNPSW_SWC	R/W	0h	XNPSW pin SW configuration
8	YNNSW_SWC	R/W	0h	YNNSW pin SW configuration

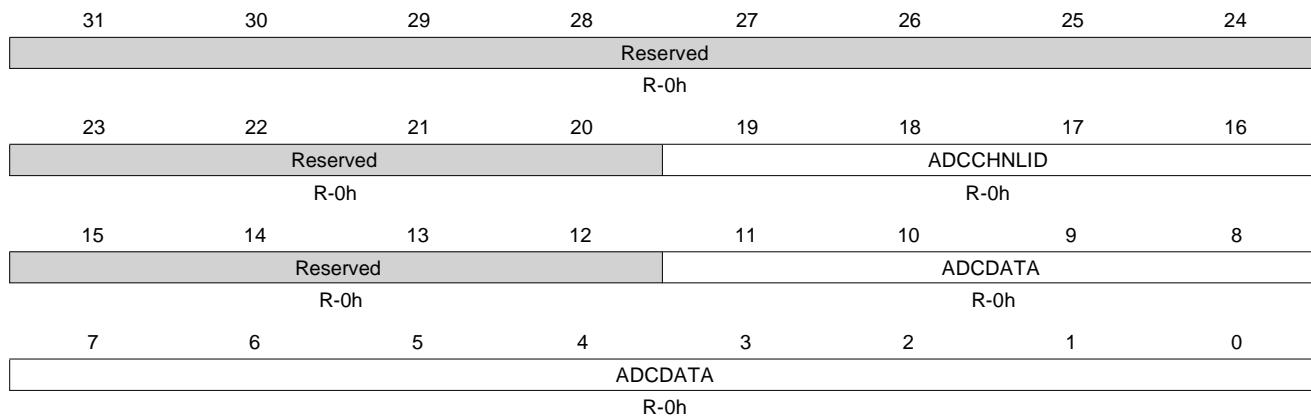
Table 12-23. STEPCONFIG1 Register Field Descriptions (continued)

Bit	Field	Type	Reset	Description
7	YPPSW_SWC	R/W	0h	YPPSW pin SW configuration
6	XNNSW_SWC	R/W	0h	XNNSW pin SW configuration
5	XPPSW_SWC	R/W	0h	XPPSW pin SW configuration
4-2	Averaging	R/W	0h	Number of samplings to average: 000 = No average. 001 = 2 samples average. 010 = 4 samples average. 011 = 8 samples average. 100 = 16 samples average.
1-0	Mode	R/W	0h	00 = SW enabled, one-shot. 01 = SW enabled, continuous. 10 = HW synchronized, one-shot. 11 = HW synchronized, continuous.

12.5.1.57 FIFO0DATA Register (offset = 100h) [reset = 0h]

FIFO0DATA is shown in [Figure 12-61](#) and described in [Table 12-61](#).
 ADC_FIFO0 _READ Data @TSC_ADC_SS_FIFO0 READ Register

Figure 12-61. FIFO0DATA Register



LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

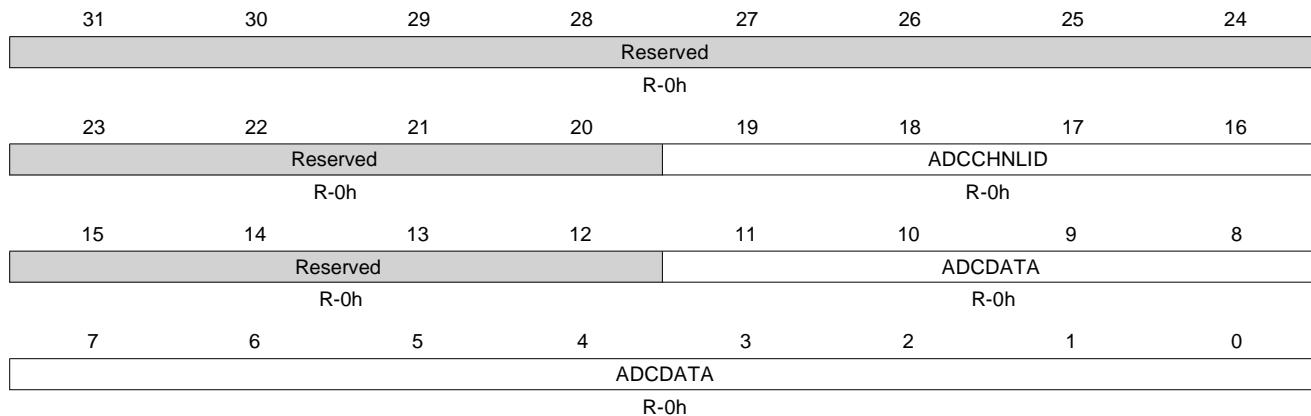
Table 12-61. FIFO0DATA Register Field Descriptions

Bit	Field	Type	Reset	Description
31-20	Reserved	R	0h	RESERVED.
19-16	ADCCHNLID	R	0h	Optional ID tag of channel that captured the data. If tag option is disabled, these bits will be 0.
15-12	Reserved	R	0h	
11-0	ADCDATA	R	0h	12 bit sampled ADC converted data value stored in FIFO 0.

12.5.1.58 FIFO1DATA Register (offset = 200h) [reset = 0h]

FIFO1DATA is shown in [Figure 12-62](#) and described in [Table 12-62](#).
 ADC FIFO1_READ Data@TSC_ADC_SS_FIFO1 READ Register

Figure 12-62. FIFO1DATA Register



LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 12-62. FIFO1DATA Register Field Descriptions

Bit	Field	Type	Reset	Description
31-20	Reserved	R	0h	RESERVED
19-16	ADCCHNLID	R	0h	Optional ID tag of channel that captured the data. If tag option is disabled, these bits will be 0.
15-12	Reserved	R	0h	RESERVED
11-0	ADCDATA	R	0h	12 bit sampled ADC converted data value stored in FIFO 1.

15. I²C Register

21.4 I2C Registers

NOTE: All bits defined as reserved must be written by software with 0s, for preserving future compatibility. When read, any reserved bit returns 0. Also, note that it is good software practice to use complete mask patterns for setting or testing individually bit fields within a register.

21.4.1 I2C Registers

Table 21-8 lists the memory-mapped registers for the I2C. All register offset addresses not listed in Table 21-8 should be considered as reserved locations and the register contents should not be modified.

Table 21-8. I2C REGISTERS

Offset	Acronym	Register Name	Section
00h	I2C_REVNB_LO		Section 21.4.1.1
04h	I2C_REVNB_HI		Section 21.4.1.2
10h	I2C_SYSC		Section 21.4.1.3
24h	I2C_IRQSTATUS_RAW		Section 21.4.1.4
28h	I2C_IRQSTATUS		Section 21.4.1.5
2Ch	I2C_IRQENABLE_SET		Section 21.4.1.6
30h	I2C_IRQENABLE_CLR		Section 21.4.1.7
34h	I2C_WE		Section 21.4.1.8
38h	I2C_DMARXENABLE_SET		Section 21.4.1.9
3Ch	I2C_DMATXENABLE_SET		Section 21.4.1.10
40h	I2C_DMARXENABLE_CLR		Section 21.4.1.11
44h	I2C_DMATXENABLE_CLR		Section 21.4.1.12
48h	I2C_DMARXWAKE_EN		Section 21.4.1.13
4Ch	I2C_DMATXWAKE_EN		Section 21.4.1.14
90h	I2C_SYSS		Section 21.4.1.15
94h	I2C_BUF		Section 21.4.1.16
98h	I2C_CNT		Section 21.4.1.17
9Ch	I2C_DATA		Section 21.4.1.18
A4h	I2C_CON		Section 21.4.1.19
A8h	I2C_OA		Section 21.4.1.20
ACh	I2C_SA		Section 21.4.1.21
B0h	I2C_PSC		Section 21.4.1.22
B4h	I2C_SCLL		Section 21.4.1.23
B8h	I2C_SCLH		Section 21.4.1.24
BCh	I2C_SYSTEST		Section 21.4.1.25
C0h	I2C_BUFSTAT		Section 21.4.1.26
C4h	I2C_OA1		Section 21.4.1.27
C8h	I2C_OA2		Section 21.4.1.28
CCh	I2C_OA3		Section 21.4.1.29
D0h	I2C_ACTOA		Section 21.4.1.30
D4h	I2C_SBLOCK		Section 21.4.1.31

21.4.1.5 I2C_IRQSTATUS Register (offset = 28h) [reset = 0h]

I2C_IRQSTATUS is shown in Figure 21-20 and described in Table 21-13.

This register provides core status information for interrupt handling, showing all active and enabled events and masking the others. The fields are read-write. Writing a 1 to a bit will clear it to 0, that is, clear the IRQ. Writing a 0 will have no effect, that is, the register value will not be modified. Only enabled, active events will trigger an actual interrupt request on the IRQ output line. For all the internal fields of the I2C_IRQSTATUS register, the descriptions given in the I2C_IRQSTATUS_RAW subsection are valid.

Figure 21-20. I2C_IRQSTATUS Register

31	30	29	28	27	26	25	24
Reserved							
R-0h							
Reserved							
R-0h							
15	14	13	12	11	10	9	8
Reserved	XDR	RDR	BB	ROVR	XUDF	AAS	BF
R-0h	R/W-0h						
7	6	5	4	3	2	1	0
AERR	STC	GC	XRDY	RRDY	ARDY	NACK	AL
R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 21-13. I2C_IRQSTATUS Register Field Descriptions

Bit	Field	Type	Reset	Description
31-15	Reserved	R	0h	
14	XDR	R/W	0h	Transmit draining IRQ enabled status. 0x0 = Transmit draining inactive 0x1 = Transmit draining enabled
13	RDR	R/W	0h	Receive draining IRQ enabled status. 0x0 = Receive draining inactive 0x1 = Receive draining enabled
12	BB	R/W	0h	Bus busy enabled status. Writing into this bit has no effect. 0x0 = Bus is free 0x1 = Bus is occupied
11	ROVR	R/W	0h	Receive overrun enabled status. Writing into this bit has no effect. 0x0 = Normal operation 0x1 = Receiver overrun
10	XUDF	R/W	0h	Transmit underflow enabled status. Writing into this bit has no effect. 0x0 = Normal operation 0x1 = Transmit underflow
9	AAS	R/W	0h	Address recognized as slave IRQ enabled status. 0x0 = No action 0x1 = Address recognized
8	BF	R/W	0h	Bus Free IRQ enabled status. 0x0 = No action 0x1 = Bus free

Table 21-13. I2C_IRQSTATUS Register Field Descriptions (continued)

Bit	Field	Type	Reset	Description
7	AERR	R/W	0h	Access Error IRQ enabled status. 0x0 = No action 0x1 = Access error
6	STC	R/W	0h	Start Condition IRQ enabled status. 0x0 = No action 0x1 = Start condition detected
5	GC	R/W	0h	General call IRQ enabled status. Set to '1' by core when General call address detected and interrupt signaled to MPUSS. Write '1' to clear. 0x0 = No general call detected 0x1 = General call address detected
4	XRDY	R/W	0h	Transmit data ready IRQ enabled status. Set to '1' by core when transmitter and when new data is requested. When set to '1' by core, an interrupt is signaled to MPUSS. Write '1' to clear. 0x0 = Transmission ongoing 0x1 = Transmit data ready
3	RRDY	R/W	0h	Receive data ready IRQ enabled status. Set to '1' by core when receiver mode, a new data is able to be read. When set to '1' by core, an interrupt is signaled to MPUSS. Write '1' to clear. 0x0 = No data available 0x1 = Receive data available
2	ARDY	R/W	0h	Register access ready IRQ enabled status. When set to '1' it indicates that previous access has been performed and registers are ready to be accessed again. An interrupt is signaled to MPUSS. Write '1' to clear. 0x0 = Module busy 0x1 = Access ready
1	NACK	R/W	0h	No acknowledgment IRQ enabled status. Bit is set when No Acknowledge has been received, an interrupt is signaled to MPUSS. Write '1' to clear this bit. 0x0 = Normal operation 0x1 = Not Acknowledge detected
0	AL	R/W	0h	Arbitration lost IRQ enabled status. This bit is automatically set by the hardware when it loses the Arbitration in master transmit mode, an interrupt is signaled to MPUSS. During reads, it always returns 0. 0x0 = Normal operation 0x1 = Arbitration lost detected

21.4.1.6 I2C_IRQENABLE_SET Register (offset = 2Ch) [reset = 0h]

I2C_IRQENABLE_SET is shown in [Figure 21-21](#) and described in [Table 21-14](#).

All 1-bit fields enable a specific interrupt event to trigger an interrupt request. Writing a 1 to a bit will enable the field. Writing a 0 will have no effect, that is, the register value will not be modified. For all the internal fields of the I2C_IRQENABLE_SET register, the descriptions given in the I2C_IRQSTATUS_RAW subsection are valid.

Figure 21-21. I2C_IRQENABLE_SET Register

31	30	29	28	27	26	25	24
Reserved							
R-0h							
23	22	21	20	19	18	17	16
Reserved							
R-0h							
15	14	13	12	11	10	9	8
Reserved	XDR_IE	RDR_IE	Reserved	ROVR	XUDF	AAS_IE	BF_IE
R-0h	R/W-0h	R/W-0h	R-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h
7	6	5	4	3	2	1	0
AERR_IE	STC_IE	GC_IE	XRDY_IE	RRDY_IE	ARDY_IE	NACK_IE	AL_IE
R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h	R/W-0h

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 21-14. I2C_IRQENABLE_SET Register Field Descriptions

Bit	Field	Type	Reset	Description
31-15	Reserved	R	0h	
14	XDR_IE	R/W	0h	Transmit draining interrupt enable set. Mask or unmask the interrupt signaled by bit in I2C_STAT[XDR]. 0x0 = Transmit draining interrupt disabled 0x1 = Transmit draining interrupt enabled
13	RDR_IE	R/W	0h	Receive draining interrupt enable set. Mask or unmask the interrupt signaled by bit in I2C_STAT[RDR]. 0x0 = Receive draining interrupt disabled 0x1 = Receive draining interrupt enabled
12	Reserved	R	0h	
11	ROVR	R/W	0h	Receive overrun enable set. 0x0 = Receive overrun interrupt disabled 0x1 = Receive draining interrupt enabled
10	XUDF	R/W	0h	Transmit underflow enable set. 0x0 = Transmit underflow interrupt disabled 0x1 = Transmit underflow interrupt enabled
9	AAS_IE	R/W	0h	Addressed as slave interrupt enable set. Mask or unmask the interrupt signaled by bit in I2C_STAT[AAS]. 0x0 = Addressed as slave interrupt disabled 0x1 = Addressed as slave interrupt enabled
8	BF_IE	R/W	0h	Bus free interrupt enable set. Mask or unmask the interrupt signaled by bit in I2C_STAT[BF]. 0x0 = Bus free interrupt disabled 0x1 = Bus free interrupt enabled
7	AERR_IE	R/W	0h	Access error interrupt enable set. Mask or unmask the interrupt signaled by bit in I2C_STAT[AERR]. 0x0 = Access error interrupt disabled 0x1 = Access error interrupt enabled

Table 21-14. I2C_IRQENABLE_SET Register Field Descriptions (continued)

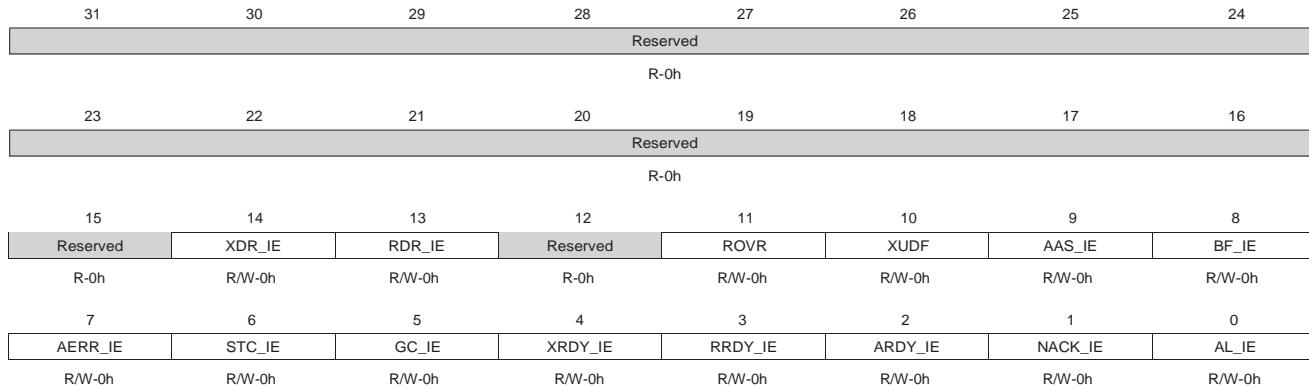
Bit	Field	Type	Reset	Description
6	STC_IE	R/W	0h	Start condition interrupt enable set. Mask or unmask the interrupt signaled by bit in I2C_STAT[STC]. 0x0 = Start condition interrupt disabled 0x1 = Start condition interrupt enabled
5	GC_IE	R/W	0h	General call interrupt enable set. Mask or unmask the interrupt signaled by bit in I2C_STAT[GC]. 0x0 = General call interrupt disabled 0x1 = General call interrupt enabled
4	XRDY_IE	R/W	0h	Transmit data ready interrupt enable set. Mask or unmask the interrupt signaled by bit in I2C_STAT[XRDY]. 0x0 = Transmit data ready interrupt disabled 0x1 = Transmit data ready interrupt enabled
3	RRDY_IE	R/W	0h	Receive data ready interrupt enable set. Mask or unmask the interrupt signaled by bit in I2C_STAT[RRDY]. 0x0 = Receive data ready interrupt disabled 0x1 = Receive data ready interrupt enabled
2	ARDY_IE	R/W	0h	Register access ready interrupt enable set. Mask or unmask the interrupt signaled by bit in I2C_STAT[ARDY]. 0x0 = Register access ready interrupt disabled 0x1 = Register access ready interrupt enabled
1	NACK_IE	R/W	0h	No acknowledgment interrupt enable set. Mask or unmask the interrupt signaled by bit in I2C_STAT[NACK]. 0x0 = Not Acknowledge interrupt disabled 0x1 = Not Acknowledge interrupt enabled
0	AL_IE	R/W	0h	Arbitration lost interrupt enable set. Mask or unmask the interrupt signaled by bit in I2C_STAT[AL]. 0x0 = Arbitration lost interrupt disabled 0x1 = Arbitration lost interrupt enabled

21.4.1.7 I2C_IRQENABLE_CLR Register (offset = 30h) [reset = 0h]

I2C_IRQENABLE_CLR is shown in [Figure 21-22](#) and described in [Table 21-15](#).

All 1-bit fields clear a specific interrupt event. Writing a 1 to a bit will disable the interrupt field. Writing a 0 will have no effect, that is, the register value will not be modified. For all the internal fields of the I2C_IRQENABLE_CLR register, the descriptions given in the I2C_IRQSTATUS_RAW subsection are valid.

Figure 21-22. I2C_IRQENABLE_CLR Register



LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 21-15. I2C_IRQENABLE_CLR Register Field Descriptions

Bit	Field	Type	Reset	Description
31-15	Reserved	R	0h	
14	XDR_IE	R/W	0h	Transmit draining interrupt enable clear. Mask or unmask the interrupt signaled by bit in I2C_STAT[XDR]. 0x0 = Transmit draining interrupt disabled 0x1 = Transmit draining interrupt enabled
13	RDR_IE	R/W	0h	Receive draining interrupt enable clear. Mask or unmask the interrupt signaled by bit in I2C_STAT[RDR]. 0x0 = Receive draining interrupt disabled 0x1 = Receive draining interrupt enabled
12	Reserved	R	0h	
11	ROVR	R/W	0h	Receive overrun enable clear. 0x0 = Receive overrun interrupt disabled 0x1 = Receive draining interrupt enabled
10	XUDF	R/W	0h	Transmit underflow enable clear. 0x0 = Transmit underflow interrupt disabled 0x1 = Transmit underflow interrupt enabled
9	AAS_IE	R/W	0h	Addressed as slave interrupt enable clear. Mask or unmask the interrupt signaled by bit in I2C_STAT[AAS]. 0x0 = Addressed as slave interrupt disabled 0x1 = Addressed as slave interrupt enabled
8	BF_IE	R/W	0h	Bus Free interrupt enable clear. Mask or unmask the interrupt signaled by bit in I2C_STAT[BF]. 0x0 = Bus free interrupt disabled 0x1 = Bus free interrupt enabled
7	AERR_IE	R/W	0h	Access error interrupt enable clear. Mask or unmask the interrupt signaled by bit in I2C_STAT[AERR]. 0x0 = Access error interrupt disabled 0x1 = Access error interrupt enabled

Table 21-15. I2C_IRQENABLE_CLR Register Field Descriptions (continued)

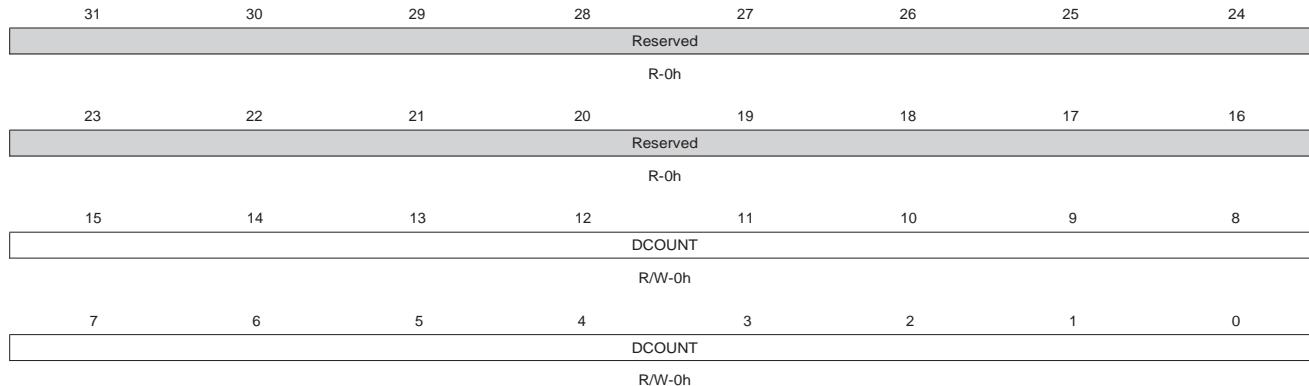
Bit	Field	Type	Reset	Description
6	STC_IE	R/W	0h	Start condition interrupt enable clear. Mask or unmask the interrupt signaled by bit in I2C_STAT[STC]. 0x0 = Start condition interrupt disabled 0x1 = Start condition interrupt enabled
5	GC_IE	R/W	0h	General call interrupt enable clear. Mask or unmask the interrupt signaled by bit in I2C_STAT[GC]. 0x0 = General call interrupt disabled 0x1 = General call interrupt enabled
4	XRDY_IE	R/W	0h	Transmit data ready interrupt enable clear. Mask or unmask the interrupt signaled by bit in I2C_STAT[XRDY]. 0x0 = Transmit data ready interrupt disabled 0x1 = Transmit data ready interrupt enabled
3	RRDY_IE	R/W	0h	Receive data ready interrupt enable set. Mask or unmask the interrupt signaled by bit in I2C_STAT[RRDY] 0x0 = Receive data ready interrupt disabled 0x1 = Receive data ready interrupt enabled
2	ARDY_IE	R/W	0h	Register access ready interrupt enable clear. Mask or unmask the interrupt signaled by bit in I2C_STAT[ARDY]. 0x0 = Register access ready interrupt disabled 0x1 = Register access ready interrupt enabled
1	NACK_IE	R/W	0h	No acknowledgment interrupt enable clear. Mask or unmask the interrupt signaled by bit in I2C_STAT[NACK]. 0x0 = Not Acknowledge interrupt disabled 0x1 = Not Acknowledge interrupt enabled
0	AL_IE	R/W	0h	Arbitration lost interrupt enable clear. Mask or unmask the interrupt signaled by bit in I2C_STAT[AL]. 0x0 = Arbitration lost interrupt disabled 0x1 = Arbitration lost interrupt enabled

21.4.1.17 I2C_CNT Register (offset = 98h) [reset = 0h]

I2C_CNT is shown in [Figure 21-32](#) and described in [Table 21-25](#).

CAUTION: During an active transfer phase (between STT having been set to 1 and reception of ARDY), no modification must be done in this register. Changing it may result in an unpredictable behavior. This read/write register is used to control the numbers of bytes in the I2C data payload.

Figure 21-32. I2C_CNT Register



LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 21-25. I2C_CNT Register Field Descriptions

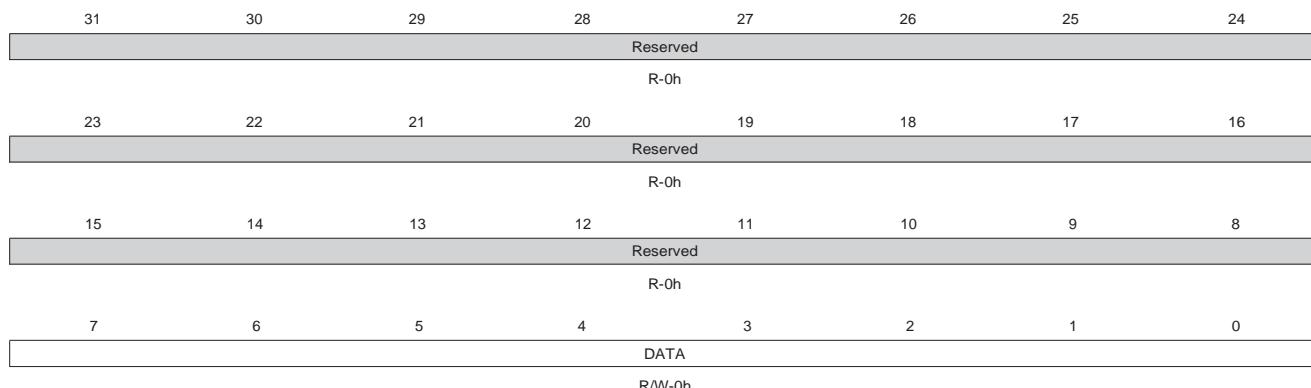
Bit	Field	Type	Reset	Description
31-16	Reserved	R	0h	
15-0	DCOUNT	R/W	0h	<p>Data count. I2C Master Mode only (receive or transmit F/S). This 16-bit countdown counter decrements by 1 for every byte received or sent through the I2C interface. A write initializes DCOUNT to a saved initial value. A read returns the number of bytes that are yet to be received or sent. A read into DCOUNT returns the initial value only before a start condition and after a stop condition. When DCOUNT reaches 0, the core generates a stop condition if a stop condition was specified (I2C_CON.STP = 1) and the ARDY status flag is set to 1 in the I2C_LRQSTATUS_RAW register. Note that DCOUNT must not be reconfigured after I2C_CON.STT was enabled and before ARDY is received. Note 1: In case of I2C mode of operation, if I2C_CON.STP = 0, then the I2C asserts SCL = 0 when DCOUNT reaches 0. The CPU can then reprogram DCOUNT to a new value and resume sending or receiving data with a new start condition (restart). This process repeats until the CPU sets to 1 the I2C_CON.STP bit. The ARDY flag is set each time DCOUNT reaches 0 and DCOUNT is reloaded to its initial value. Values after reset are low (all 16 bits). Note 2: Since for DCOUNT = 0, the transfer length is 65536, the module does not allow the possibility to initiate zero data bytes transfers. 0x0 = Data counter = 65536 bytes (216) 0x1 = Data counter = 1 bytes 0xFFFF = Data counter = 65535 bytes (216 - 1)</p>

21.4.1.18 I2C_DATA Register (offset = 9Ch) [reset = 0h]

I2C_DATA is shown in [Figure 21-33](#) and described in [Table 21-26](#).

This register is the entry point for the local host to read data from or write data to the FIFO buffer.

Figure 21-33. I2C_DATA Register



LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 21-26. I2C_DATA Register Field Descriptions

Bit	Field	Type	Reset	Description
31-8	Reserved	R	0h	
7-0	DATA	R/W	0h	<p>Transmit/Receive data FIFO endpoint. When read, this register contains the received I2C data. When written, this register contains the byte value to transmit over the I2C data. In SYSTEST loop back mode (I2C_SYSTEST: TMODE = 11), this register is also the entry/receive point for the data. Values after reset are unknown (all 8-bits).</p> <p>Note: A read access, when the buffer is empty, returns the previous read data value. A write access, when the buffer is full, is ignored. In both events, the FIFO pointers are not updated and an Access Error (AERR) Interrupt is generated.</p>

21.4.1.19 I2C_CON Register (offset = A4h) [reset = 0h]

I2C_CON is shown in [Figure 21-34](#) and described in [Table 21-27](#).

During an active transfer phase (between STT having been set to 1 and reception of ARDY), no modification must be done in this register (except STP enable). Changing it may result in an unpredictable behavior.

Figure 21-34. I2C_CON Register

31	30	29	28	27	26	25	24
Reserved							
R-0h							
23	22	21	20	19	18	17	16
Reserved							
R-0h							
15	14	13	12	11	10	9	8
I2C_EN	Reserved	OPMODE		STB	MST	TRX	XSA
R/W-0h	R-0h	R/W-0h		R/W-0h	R/W-0h	R/W-0h	R/W-0h
7	6	5	4	3	2	1	0
XOA0	XOA1	XOA2	XOA3	Reserved		STP	STT
R/W-0h	R/W-0h	R/W-0h	R/W-0h	R-0h		R/W-0h	R/W-0h

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 21-27. I2C_CON Register Field Descriptions

Bit	Field	Type	Reset	Description
31-16	Reserved	R	0h	
15	I2C_EN	R/W	0h	I2C module enable. When this bit is cleared to 0, the I2C controller is not enabled and reset. When 0, receive and transmit FIFOs are cleared and all status bits are set to their default values. All configuration registers (I2C_IRQENABLE_SET, I2C_IRQWAKE_SET, I2C_BUF, I2C_CNT, I2C_CON, I2C_OA, I2C_SA, I2C_PSC, I2C_SCLL and I2C_SCLH) are not reset, they keep their initial values and can be accessed. The CPU must set this bit to 1 for normal operation. Value after reset is low. 0x0 = Controller in reset. FIFO are cleared and status bits are set to their default value. 0x1 = Module enabled
14	Reserved	R	0h	
13-12	OPMODE	R/W	0h	Operation mode selection. These two bits select module operation mode. Value after reset is 00. 0x0 = I2C Fast/Standard mode 0x1 = Reserved 0x2 = Reserved 0x3 = Reserved
11	STB	R/W	0h	Start byte mode (I2C master mode only). The start byte mode bit is set to 1 by the CPU to configure the I2C in start byte mode (I2C_SA = 0000 0001). See the Philips I2C spec for more details [1]. Value after reset is low. 0x0 = Normal mode 0x1 = Start byte mode

Table 21-27. I2C_CON Register Field Descriptions (continued)

Bit	Field	Type	Reset	Description
10	MST	R/W	0h	<p>Master/slave mode (I2C mode only). When this bit is cleared, the I2C controller is in the slave mode and the serial clock (SCL) is received from the master device. When this bit is set, the I2C controller is in the master mode and generates the serial clock. Note: This bit is automatically cleared at the end of the transfer on a detected stop condition, in case of arbitration lost or when the module is configured as a master but addressed as a slave by an external master.</p> <p>Value after reset is low. 0x0 = Slave mode 0x1 = Master mode</p>
9	TRX	R/W	0h	<p>Transmitter/receiver mode (I2C master mode only). When this bit is cleared, the I2C controller is in the receiver mode and data on data line SDA is shifted into the receiver FIFO and can be read from I2C_DATA register. When this bit is set, the I2C controller is in the transmitter mode and the data written in the transmitter FIFO via I2C_DATA is shifted out on data line SDA. Value after reset is low. The operating modes are defined as follows: MST = 0, TRX = x, Operating Mode = Slave receiver. MST = 0, TRX = x, Operating Mode = Slave transmitter. MST = 1, TRX = 0, Operating Modes = Master receiver. MST = 1, TRX = 0, Operating Modes = Master transmitter. 0x0 = Receiver mode 0x1 = Transmitter mode</p>
8	XSA	R/W	0h	<p>Expand slave address. (I2C mode only). When set, this bit expands the slave address to 10-bit. Value after reset is low. 0x0 = 7-bit address mode 0x1 = 10-bit address mode</p>
7	XOA0	R/W	0h	<p>Expand own address 0. (I2C mode only). When set, this bit expands the base own address (OA0) to 10-bit. Value after reset is low. 0x0 = 7-bit address mode 0x1 = 10-bit address mode</p>
6	XOA1	R/W	0h	<p>Expand own address 1. (I2C mode only). When set, this bit expands the first alternative own address (OA1) to 10-bit. Value after reset is low. 0x0 = 7-bit address mode 0x1 = 10-bit address mode</p>
5	XOA2	R/W	0h	<p>Expand own address 2. (I2C mode only). When set, this bit expands the second alternative own address (OA2) to 10-bit. Value after reset is low. 0x0 = 7-bit address mode. (I2C mode only). 0x1 = 10-bit address mode</p>
4	XOA3	R/W	0h	<p>Expand own address 3. When set, this bit expands the third alternative own address (OA3) to 10-bit. Value after reset is low. 0x0 = 7-bit address mode 0x1 = 10-bit address mode</p>

Table 21-27. I2C_CON Register Field Descriptions (continued)

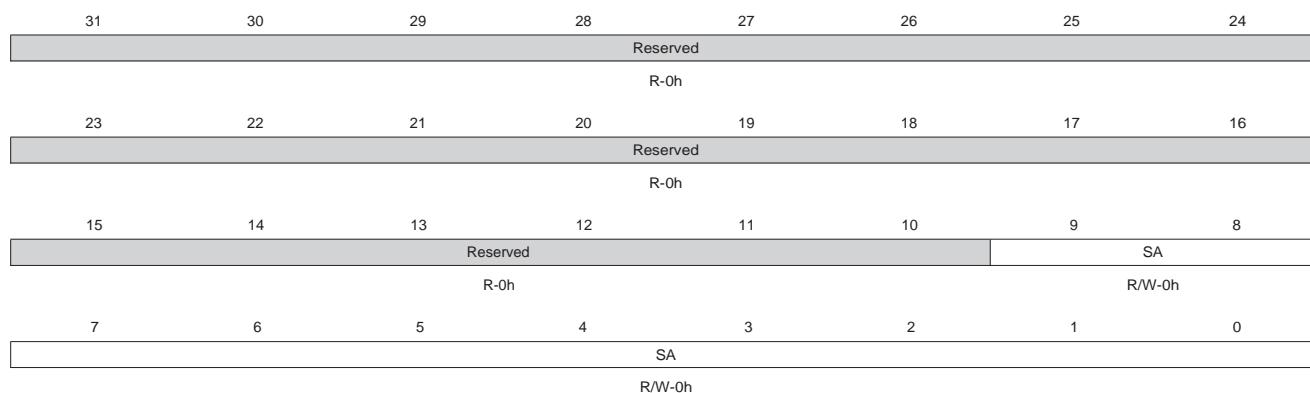
Bit	Field	Type	Reset	Description
3-2	Reserved	R	0h	
1	STP	R/W	0h	<p>Stop condition (I2C master mode only). This bit can be set to a 1 by the CPU to generate a stop condition. It is reset to 0 by the hardware after the stop condition has been generated.</p> <p>The stop condition is generated when DCOUNT passes 0. When this bit is not set to 1 before the end of the transfer (DCOUNT = 0), the stop condition is not generated and the SCL line is held to 0 by the master, which can re-start a new transfer by setting the STT bit to 1.</p> <p>Value after reset is low 0x0 = No action or stop condition detected 0x1 = Stop condition queried</p>
0	STT	R/W	0h	<p>Start condition (I2C master mode only). This bit can be set to a 1 by the CPU to generate a start condition. It is reset to 0 by the hardware after the start condition has been generated.</p> <p>The start/stop bits can be configured to generate different transfer formats.</p> <p>Value after reset is low. Note: DCOUNT is data count value in I2C_CNT register. STT = 1, STP = 0, Conditions = Start, Bus Activities = S-A-D. STT = 0, STP = 1, Conditions = Stop, Bus Activities = P. STT = 1, STP = 1, Conditions = Start-Stop (DCOUNT=n), Bus Activities = S-A-D..(n)..D-P. STT = 1, STP = 0, Conditions = Start (DCOUNT=n), Bus Activities = S-A-D..(n)..D. 0x0 = No action or start condition detected 0x1 = Start condition queried</p>

21.4.1.21 I2C_SA Register (offset = ACh) [reset = 0h]

I2C_SA is shown in Figure 21-36 and described in Table 21-29.

CAUTION: During an active transfer phase (between STT having been set to 1 and reception of ARDY), no modification must be done in this register. Changing it may result in an unpredictable behavior. This register is used to specify the addressed I2C module 7-bit or 10-bit address (slave address).

Figure 21-36. I2C_SA Register



LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 21-29. I2C_SA Register Field Descriptions

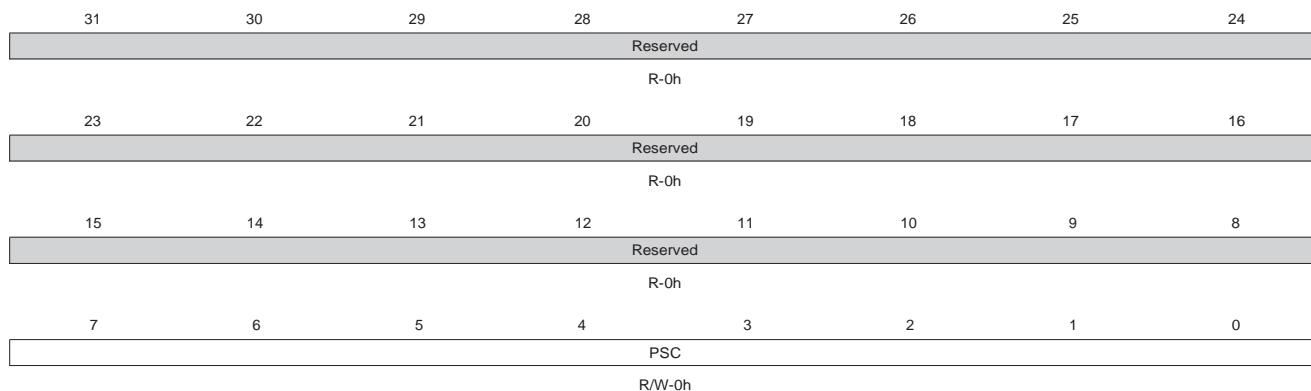
Bit	Field	Type	Reset	Description
31-10	Reserved	R	0h	
9-0	SA	R/W	0h	Slave address. This field specifies either: A 10-bit address coded on SA [9:0] when XSA (Expand Slave Address, I2C_CON[8]) is set to 1. or A 7-bit address coded on SA [6:0] when XSA (Expand Slave Address, I2C_CON[8]) is cleared to 0. In this case, SA [9:7] bits must be cleared to 000 by application software. Value after reset is low (all 10 bits).

21.4.1.22 I2C_PSC Register (offset = B0h) [reset = 0h]

I2C_PSC is shown in [Figure 21-37](#) and described in [Table 21-30](#).

CAUTION: During an active mode (I2C_EN bit in I2C_CON register is set to 1), no modification must be done in this register. Changing it may result in an unpredictable behavior. This register is used to specify the internal clocking of the I2C peripheral core.

Figure 21-37. I2C_PSC Register



LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 21-30. I2C_PSC Register Field Descriptions

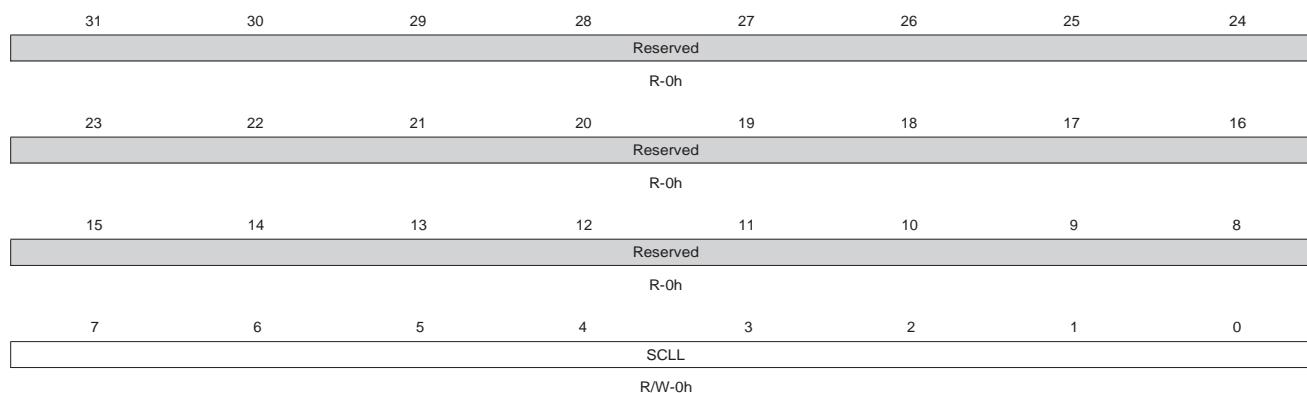
Bit	Field	Type	Reset	Description
31-8	Reserved	R	0h	
7-0	PSC	R/W	0h	Fast/Standard mode prescale sampling clock divider value. The core uses this 8-bit value to divide the system clock (SCLK) and generates its own internal sampling clock (ICLK) for Fast and Standard operation modes. The core logic is sampled at the clock rate of the system clock for the module divided by (PSC + 1). Value after reset is low (all 8 bits). 0x0 = Divide by 1 0x1 = Divide by 2 0xFF = Divide by 256

21.4.1.23 I2C_SCLL Register (offset = B4h) [reset = 0h]

I2C_SCLL is shown in [Figure 21-38](#) and described in [Table 21-31](#).

CAUTION: During an active mode (I2C_EN bit in I2C_CON register is set to 1), no modification must be done in this register. Changing it may result in an unpredictable behavior. This register is used to determine the SCL low time value when master.

Figure 21-38. I2C_SCLL Register



LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 21-31. I2C_SCLL Register Field Descriptions

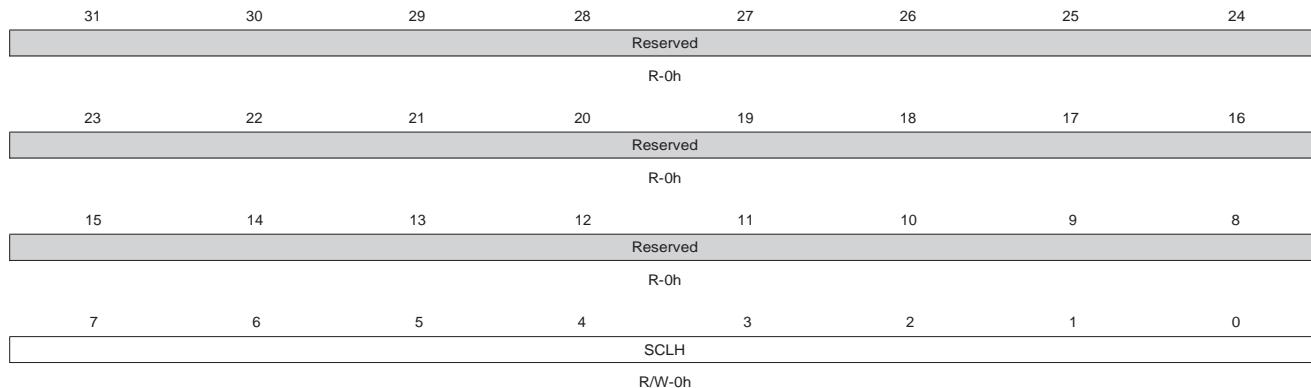
Bit	Field	Type	Reset	Description
31-8	Reserved	R	0h	
7-0	SCLL	R/W	0h	Fast/Standard mode SCL low time. I2C master mode only, (FS). This 8-bit value is used to generate the SCL low time value (tLOW) when the peripheral is operated in master mode. $tLOW = (SCLL + 7) * \text{ICLK}$ time period, Value after reset is low (all 8 bits).

21.4.1.24 I2C_SCLH Register (offset = B8h) [reset = 0h]

I2C_SCLH is shown in [Figure 21-39](#) and described in [Table 21-32](#).

CAUTION: During an active mode (I2C_EN bit in I2C_CON register is set to 1), no modification must be done in this register. Changing it may result in an unpredictable behavior. This register is used to determine the SCL high time value when master.

Figure 21-39. I2C_SCLH Register



LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 21-32. I2C_SCLH Register Field Descriptions

Bit	Field	Type	Reset	Description
31-8	Reserved	R	0h	
7-0	SCLH	R/W	0h	Fast/Standard mode SCL low time. I2C master mode only, (FS). This 8-bit value is used to generate the SCL high time value (tHIGH) when the peripheral is operated in master mode. - tHIGH = (SCLH + 5) * ICLK time period. Value after reset is low (all 8 bits).

16. Antworten zu den Verständnisfragen

Grundlagen 0

1. Teil des BeagleBone
2. Das CONTROL-Modul
3. Die durch die Hardware vorgegebene Adresse zu einem Speicherbereich
4. Ein Speicherbereich der direkt mit der Recheneinheit verbunden ist und den Zustand der Hardware definiert oder durch die Hardware definiert wird
5. Die Basisadresse zeigt zu einem Modul, die Offsetadresse zu den Registern innerhalb des Moduls
6. In der MemoryMap des Mikrocontrollers
7. `Reg |= ((1<<12)|(1<<14)|(1<<23));`
8. Durch den Bootloader

Termin 1

1. 4 Module 32 Pins
2. Es schaltet einen GPIO-Pin als Output oder Input
3. In OE wird die PIN-Richtung und in DataOut der PIN wert gesetzt
4. Pinmuxen, Pin als Output schalten (Output_Enable) und Wert setzen (Data_Out)
5. Vermeiden undefinierter Zustände an digitalen Eingängen bei geöffneten Tastern
6. Um die Hardwaresteuerung zu abstrahieren und mit wiederverwendbaren definierten Schnittstellen-Funktionen einfach ansprechen zu können

Termin 2

1. In einem Rechensystem bedeutet Echtzeit die Garantie, dass Ergebnisse zuverlässig innerhalb einer vorbestimmten Zeitspanne geliefert werden können. Dadurch lässt sich ein deterministisches Systemverhalten erreichen.
2. Polling und Interrupts
3. Polling bezeichnet das periodische Abfragen von Zuständen
4. Ein Interrupt ist eine vorübergehende Unterbrechung des laufenden Programms, z.B. um sehr schnell auf bestimmte Zustandsänderungen zu reagieren.
5. GPIO (Wertänderung an Pins), Timer, Kommunikationsschnittstellen, ADC, etc.
6. Interrupts signalisieren Zustandsänderungen. Werden in der Interrupt Service Routine Delays eingesetzt, wird das Hauptprogramm lange unterbrochen und man verpasst unter Umständen andere zeitkritische Anforderungen.
7. Interruptflag ist ein bestimmtes Bit, welches automatisch gesetzt wird, wenn der entsprechende Interrupt ausgelöst wird. Man muss es löschen, um diesen Interrupt wieder freizugeben und erneut darauf reagieren zu können.
8. Ein Zähler, der unabhängig vom restlichen Programm, mit einer einstellbaren Frequenz seinen Wert erhöht.
9. Ein Interrupt, der ausgelöst wird, wenn ein Timer einen vorher definierten Wert erreicht.
10. Ein Interrupt, der ausgelöst wird, wenn ein Timer-Register „voll“ ist (komplett mit Einsen belegt)

11. DC-Motor (mit PWM-Steuerung zur Drehzahl-Regelung)
12. Beim PWM zur Leistungssteuerung ist die Durchschnittsspannung des PWM-Signals relevant und kann direkt die Spannung für einen Motor einstellen.
Bei der Servo-Steuerung kodiert das PWM-Signal die Information über die Sollposition des Servomotors.
13. Der Duty Cycle gibt die Pulsbreite in Prozent und damit das Tastverhältnis des PWM-Signals an.
Also das Verhältnis von Einschaltzeit zu Periodendauer $\frac{t_{ein}}{t_{ein}+t_{aus}}$
14. Eine H-Brücke dient zur Steuerung von Motoren. Man kann damit sowohl die Drehrichtung als auch die Drehzahl per PWM steuern.
15. Es lässt sich automatisch ein Ausgangssignal an entsprechende Pins schalten.
Man muss das PWM Signal nicht mit eigenem Programmcode erzeugen.
Die Signalerzeugung läuft automatisch im Hintergrund und ist wesentlich genauer.

Termin 3

1. Analog-Digital-Converter
2. Um analoge (wertkontinuierliche) Signale zu diskretisieren und einzulesen.
3. 1 ADC und 8 Analogeingänge

Termin 4

1. Ein serieller, synchroner Zweidraht-Kommunikationsbus.
2. Zum Anschluss von externen Bausteinen, wie intelligenten Sensoren und ähnlichem
3. SDA = Serial Data (Datenleitung)
SCL = Serial Clock (Clockleitung)
4. Anhand des achten Adressbits (R/W-Bit)
5. 112
6. Mit Hilfe des Receive Ready Interrupts

