Mikrocontroller Programmierung in “C”

S. Fath / 2009

1. Juli 2009

Inhaltsverzeichnis

Einleitung

Aufgabe diese Scrites ist es einem interessiertem Studierenden den Einstieg in die Welt der MicroController Programmierung zu geben. Als Sprache wird hier "C", genauer "ANSI C", da diese diesem Bereich sehr verbreitet ist, und zudem noch die Hochsprache ist, welche ein relativ große nähe zur Hardware zuläßt. Wichtig ist aber, daß die Leser diese Scriptes sich nicht nur mit der reinen Theorie zufrieden geben, sondern die Informationen, welche in diesem Script stehen, auch praktisch umsetzten in eigenen Programmen. Als Zielplattform wird hier der Microkontroller der Firma Texas Instruments, der MSP430, benutzt. Dieser bietet sehr viele Funktionen, die wiederum recht einfach und dennoch sehr vielseitig einsetzbar sind. Zudem wird dieser Controller immer mehr von der Industrie eingesetzt.

Dieses Skript ist nach einem bestimmten Muster aufgebaut. Zunächt wird der verwendete Controller betrachtet, wie dieser aufgebaut ist und welche Funktionsblöcke er enthält. Danach werden die Grundlagen der Programmiersprache "C" angesprochen, wie Operatoren, Srukturen , Funktioen, Variablen, ... Nach diesem Einstieg in die Programmiersprache, wird an Hand eines relativ einfachen Programmes das dort gezeigte umgesetzt. Dieses Programm wird dann immer weiter ausgebaut, Schritt für Schritt mit weiteren Funktionen. Wichtig ist dabei, daß das Programm immer modular bleibt. Im Laufe der Zeit werden in dieses Programm Funktionen, wie Port, LCD, Timer, ADC, Komperator und serielle Kommunikation eingebracht. Ein zentraler Bestandteil der Microkontroller Programmierung ist die Verwendung von Interrupts. Dieser Umstand wird hier Rechnung getragen und die Programmierung dahin gehen ausgelegt. Auch soll hier in diesem Script deutlich gezeigt werden, wie "C-Programme" nach Laufzeit und Codegröße hin optiemiert werden können, mit doch reicht einfachen Mitteln. Als nächsten Schritt soll eine Vermaschung zwischen "C" und "Assembler" aufgezeit werden, der sogenannten "Mixed-Language Programmierung". Dabei werden dann gezielt Funktionen, bzw. Routinen in Assembler gemacht und in den "C-Code" eingesetzt. Durch solche Maßnahmen wird der Code weiter optimiert. Damit dies geschehen kann, werden die Grundlagen der Assembler Sprache und der Umgang mit ihr besprochen.

Damit das gelesene und im Rahmen der Vorlesung erworbene Wissen überprüft werden kann, sind am Ende eines jeden Kapitels Fragen zusammen gestellt, welche diese Kapitel und der vorhergehenden Kapitel betreffen. Auch befinden sich dort teilweise Lückentexte, welche vom Studierenden vervollständigt werden sollen. Eine Aufgabensammlung rundet dieser Script ab. Die dort gestellten Aufgaben sind alle mit dem Wissen aus diesem Script und der Vorlesung lösbar.

1 Aufbau des MSP430

1.1 Die Familien des MSP430

Wenn wir uns den MSP430 betrachten, dann sehen wir zunächst nur ein schrwazes Etwas aus einem Kunststoff, aus dem an allen vier Seiten ein große Anzahl von Bein'chen (auch Pins genannt) herauskommen. Oben auf dem schwarzen Kunststoff sehen wir, dass dort einiges steht. Wichtig ist hier eine ganz bestimmte Zeile, nämlich die, wo der Typ des MSP430 daraus zu erkennen ist.

Es gibt mittelerweile eine große Anzahl von unterschiedlichen MSP430 Derivaten. Unter Derivaten versehen wir Typen der gleichen Familie, aber mit unterschiedlichen Inhalten. Bei der MSP430 Familie unterscheiden wir in zur Zeit 4 unterschidlichen Hauptgruppen. Diese sind:

Hauptgruppe	Charakteristik	Haupteinsatzgebiet
MSP430x1xxx	ohne LCD	Motor-Control
MSP430x2xxx	ohne LCD	Medizintechnik
MSP430x3xxx	mit LCD	veraltet
MSP430x4xxx	mit LCD	Messgeräte allg.
MSP430x5000	ohne LCD	Messgeräte, Zigbee

Das erste "x" nach dem Familiennamen MSP430 ist ein Platzhalter für die Art, wie der Speicher des MSP430 aufgebaut ist.

Zeichen	Beschreibung
C	ROM
P	OTP
F	Flash

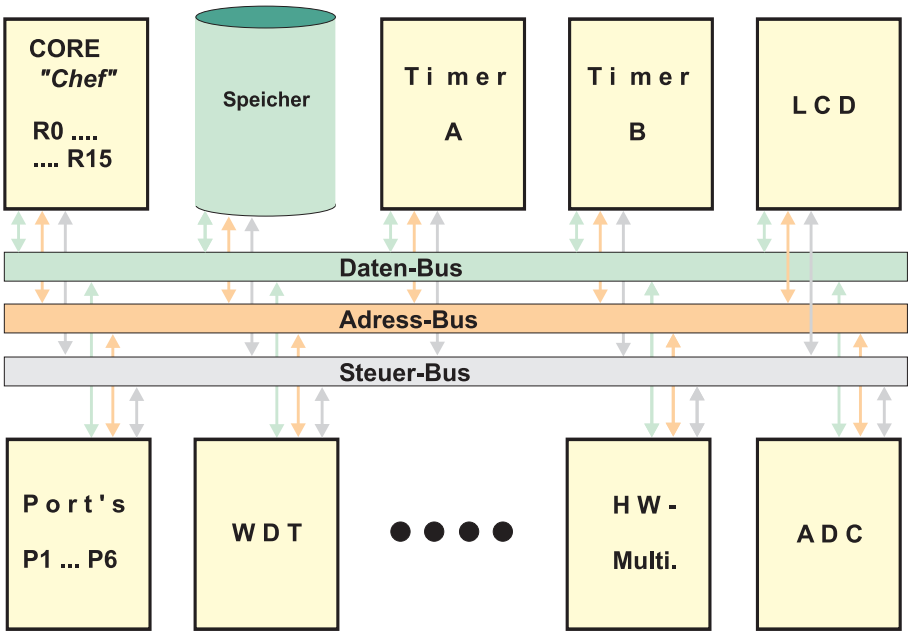

Abbildung 1.1: Blockdiagramm des inneren Aufbaus

1.2 Funktionsblöcke

Soweit diese Informationen als Einleitung. Innerhalb des schwarzen Gehäuses sind die eigentlichen Module, aus denen der MSP430 sich zusammen setzt, implementiert. Diese Module setzen sich aus der eigentlichen CPU, auch Core, genannt und den Peripherie Modulen, sowie dem Speicher zusammen. Streng genommen kann auch der Speicher als ein solches Modul betrachtet werden. All diese Module sind mit drei Bussystemen untereinander verbunden. Dies drei Busse haben unterschiedliche Aufgaben, die Aufgabe des jeweiligen Busses spiegelt sich in seinem Namen wieder. Da sind der Daten-Bus, der Adress-Bus und der Steuer-Bus. Über diese Busse kommunizieren die einzelnen Peripherie Modulen entweder untereinander, oder sie tauschen Daten über die CPU miteinander aus. Im unten abgebildeten Schema ist dier Ausbau stark vereinfacht dargestellt.

1.3 Die CPU

Die zentrale Rolle spielt, wie könnte es wohl auch anders sein, die CPU (Core = das Herz). In ihr werden alle Operationen zur Ausführung gebracht, egal ob es sich um eine Addition, oder um eine Kopierfunktion handelt, um hier nur zwei Beispiele zu nennen. Damit die CPU vernünftig arbeiten kann, hat sie zu diesem Zweck eigene Register, wo Werte und/oder Adressen maniluliert bzw. Operationen durch sie ausgeführt werden. Die CPU des MSP430 hat aber auch die Fähigkeit direkt, ohn Zuhilfenahme der Register, Werte im Speicher zu verändern. Diesen Umstand macht sie so mächtig und gleichemaßen

auch so einfach au zu programmieren, aber dazu später. Die Register der CPU sind alle gleich aufgebaut, mit einer Breite von jeweils 16Bit. Es gibt insgesamt 16 Register, welche die Bezeichnung R0 ... R15 haben. Während die Register R4 ... R15 als Universal-Register anzusehen sind, sind die übrigen 4 Register R0 ... R3 Spezialaufgaben vorbehalten. Die Verwendung der einzelnen Register sind in der Tabelle hier aufgeführt:

Register		Funktion
R0	PC	Programm-Counter
R1	SP	Stack-Pointer
R2	SR	Status-Register
R3	CG	Konstanten-Generator
R4	R4	Universal-Register
R5	R5	Universal-Register
R6	R6	Universal-Register
R7	R7	Universal-Register
R8	R8	Universal-Register
R9	R9	Universal-Register
R10	R10	Universal-Register
R11	R11	Universal-Register
R12	R12	Universal-Register
R13	R13	Universal-Register
R14	R14	Universal-Register
R15	R15	Universal-Register

Hier eine kurze Bschreibung der einzelnen Register, die für eine Spezialfunktion vorgeseghen sind:

1.3.1 Das Status-Register R2

Das Status-Register (SR) regelt Verzweigungen und Bedingungen, auf welche der Code reagieren kann. In Abhängigkeit von Operationen werden die verschiedensten Status-Bits gesetzt oder auch gelöscht. Dies Bits werden dann von verschiedenen Befehlen genutzt um den Programmfluss zu verändern, sei es durch einen Sprung oder durch automatische Korrekturmechanismen.

1.3.2 Der Konstanten-Generator R3

Dier sogennante Konstanten-Generator beschleunigt manche Operationen, so dass sie innerhalb eines Taktzykluses zur Ausführung gebracht werden kann. Gemeint sin hier Anweisungen wo Adresse oder Register mit einem sehr häufig anzutreffenden Wert besetzt werden soll. Solche Werte sind: 0, 1, 2, 4, 8. Der Programmierer bekommt von der Arbeit des Konstantengenerator nicht mit, da die kodierte Bits vom Assembler bei der Umsetzung des Quellcodes in den eigentlichen Maschinen-Code herangezogen werden um die Anweisungen, wie z.B. MOV #00, 0x0200, zu dekodieren.

1.3.3 Die Universal-Register R4 ... R15

Die "Universal Register" sind wie der Name schon sagt, für alle Operationen gleimässig geeignet und können frei von Programmierer in den Applicationen eingesetzt werden. Das Operationen welche mit den Registern ausgeführt werden schneller abgearbeitet werden, als zum Beispiel Zugriffe direkt auf oder über den Speicher, sollten die Register für zeitkritische Operationen vorzugsweise eingesetzt werden.

1.4 Der Speicher

Der Speicher des MSP430 ist nach der klassischen "Von-Neumann" Architektur aufgebaut. Was nichts anderes heißt, dass sich Programm und Daten den gleichen Speicher teilen. Eines vorab, alle Adressen haben immer eine breite von 8 Bit, also 1 Byte. Dies ist für das weitere Verständnis sehr wichtig.

Der Speicher gliedert sich in zwei Hauptteile auf, das sind der Teil, wo immer nur ein bytebreiter Zugriff möglich ist und der Bereich wo in der Regel (Ausnahme RAM, der innerhalb dieses Bereiches liegt) ein 16-Bit breiter Zugriff statt findet. Man spricht bei einem 16-Bit (2Byte) Zugriff auch von einem Word-Zugriff. Der Adressbereich 00h ... 0FFh ist der Bereich wo der byteweite Zugriff nur möglich ist.

In diesem Bereich sind neben den sogenannten "Spezial-Function-Register" (SFR) auch die Register zu finden, mit denen die Peripherie Module initiiert, bzw. eingestellt, konfiguriert werden. Da jedoch der MSP430 mit seinen sehr vielen und sehr mächtigen Peripherie Modulen mehr Speicherstellen benötigt, als in diesm Bereich zur Verfügung stehen, sind hier noch im Adressberech 0100h ... 01FFh weitere Peripherie Module untergebracht. Was verstehen wir eigentlich unter dem Begriff "Spezial-Function-Register" (SFR)?

Mit diesem Begriff wurde in der Anfangszeit der Microcontroller der Speichebereich gemeint, wo die Steuerung des Microcontrollers abgelegt ist. Auch wurden in diesen Bereich die damals noch wenigen Steuerungsbits der Peripherie Module mit integriert. Der Bereich umfasste darmals wie auch heute noch 16 Adressen. Diese liegen beim MSP430 an den Adresse 00h ... 0Fh. Es wurde jedoch bald klar, dass dieserSFR-Bereich nicht mehr ausreicht, und so wurde ein sogenannter "moderne / erweiteter SFR" Bereich definiert. Dieser wurde auf eine Länge von 255 Adressen, mit einer Adressbreite von 8Bit (1Byte) festgelegt. Dieser Bereich ist hier im MSP430 der bereits weiter oben erwähnte Steicherbereich 00h ... 0FFh. An diese Auftreilung hat sich auch wie fast alle Chip Hersteller, Texas Instruments gehalten.

Dieser obere Bereich von 0100h ... 0FFFFh beinhaltet neben den zusätzlichen Peripherie Modulen, die im unteren Adressbereich keinen Platz mehr gefunden haben, Der eigentlichen Programmspeicher, die Interrupt-Vector-Tabelle, einem Boot-ROM, einem Extra-Speicher (mit besonderen Eigenschaften), sowie dem RAM. Bis auf den Speicherbereich RAM sind alle anderen Teile dieses Adressbereiches als Flash-Speicher realisiert. Diese können nur im Programmiermoduls beschrieben werden und sind somit im Arbeit-Modus (Programmlaufzeit) nicht veränderbar. Dies ist zwar nicht ganz korret,

denn es gibt schon Möglichkeiten während der Programmlaufzeit Daten in den Flash zu schreiben bzw. Daten darauf zu löschen, aber diese Funktion soll hier nicht besprochen werden. Vielleich kommt säter einmal ein eigenes Kapitel dazu heraus. Auch der Extra-Speicher wird hier nicht weiter beschrieben, nur soviel, dass es sich bei diesem Speicherbereich um eine Art Flash handelt, der aber byteweise unter bestimmten Vorraussetzungen während der Laufzeit manipuliert werden kann. Auch auf das integrierte ROM wird in diesem Script vorerst nicht weiter eingegangen. Einzigste Anmerkung hierzu, in diesem Bereich sind Programm-Sequenzen abgelegt, welches es dem User erlauben über die JTEG-Schnittstelle mit dem MSP430 zu kommunizieren. Auch sind dort Routinen abgelegt um Programmcode und auch Daten über die UART (Serielle Schnittstelle) auf den Speicher zu schreiben. Diese Routinen werden auch unter dem Begriff "Boot-Step-Loader" zusammengefasst. Übrig bleiben jetzt nur noch der RAM sowie der Breich des Programmcodes, sowie die Interrupt-Vector-Tabelle. Diese drei Bereichen werden wir uns etwas näher betrachten.

1.5 Der Programm-Speicher

Dieser im Flash liegende Speicherbereich beansprucht den meisten Platz im gesamten Speicher. In diesem Bereich wieder der vom Assebmler übersetzen Quellcode abgelegt, forausgesetzt der Programmierer will das so. Vorteil dieses Speicherbereiches ist, dass der Code auch nach abeschalten der Betriebsspannung weiterhin in diesen Speichestellen bleibt. Er wird also nicht duch Spannungsschwankungen während der Laufzeit beeinflusst. Demnach sollte der Programmierer auch darauf achten dass sein Code immer in diesem Bereich abgelegt ist. Durch eine Routine, welche im ROM abgelegt ist wird automatisch der Code zur Ausführung gebracht, der an der ersten geraden Adresse liegt, die als Programm-Code Bereich ausgewiesen ist.

Wo diese Bereiche beginnen und wie gross sie sind ist von Derivat zu Derivat unterschiedlich. Hier muss der Programmierer sich diese Information aus dem jeweiligen Datenblatt seines MSP430 Derivates holen. Es wäre viel zu umfangreich hier alle Drivate aufzulisten mit ihren Berschreibungen wo welcher Adressebereich bei welchem Derivat liegt.

1.6 Der RAM

RAM das steht für *Read Access Memory*, was soviel bedeutet wie Speicher zum lesen und schreiben, während der Programmlaufzeit. Dieser Speicherbereich ist uns vermutlich bereits bekannt unter dem Begriff des *Arbeitsspeichers* im PC. In diesem Speicherbereich werden temporäre Daten gehalten die während der Laufzeit des Programmes erfast, manipuliert oder sonst etwas mit ihnen gemacht wird. Sollte die Betriebsspannung unter einen Level abfallen, so ist nicht sichergestellt, dass diese Daten wenn die Betriebsspannung wieder auf dem normalen Level liegt, immer noch so sind wie vor dem Spannungseinbruch.

Aus unseren PC's kennen wir Arbeitsspeicher von Größen wie 512MB, 1GB ... . Hier in der Welt der Microcontroller sind dies RAM-Bereiche sehr viel kleiner. Es gibt Microcontroller die einen RAM von gerade mal 128 Byte haben aber es gint auch Typen, welche einen RAM von bis zu 64kB aufweisen können. Das Derivat, welches im Labor eingesetzt wird ist der MSP430FG4617 mit 8kB (8192 Byte). Er geghört schon zu den Derivaten, welche die meisten Module und den zweitgrößten RAM haben. Früher wurde der MSP430F449 benutzt, der mit seinen 2kB doche einen eher bescheidenen RAM hatte. Jedoch sollte der Studierende sich nicht von der Größe des RAM's beeindrucken lassen, da in der Welt der MicroController der Speicher wesentlich effektiver genutzt wir, als in der PC-Welt.

Auch hier wieder ist die Lage der RAM's von Derivat zu Derivat unterschiedlich. In einer Tabelle sind die wichtigsten Daten zusammengefasst.

Derivate	Speicher Art	Type	Größe	Start-Adr.
MSP430F449	Daten	RAM	2 kB	0x0200
	Programm	FLASH	60 kB	0x01100
MSP430F1611	Daten	RAM	10 kB	0x01100
	Programm	FLASH	60 kB	0x04000
MSP430FG4617	Daten	RAM	8 kB	0x01100
	Programm	FLASH	96 kB	0x03100
MSP430FG4618	Daten	RAM	8 kB	0x01100
	Programm	FLASH	120 kB	0x03100

1.7 Die Interrupt-Vector-Tabelle

Bei der "Interrup-Vektor-Tabelle" handelt es sich um einen Adressbereich, wo Sprungmarken (Pointer) abgelegt sind. Diese Tabelle kann man sich als einen Speicherort vorstellen, wo darin steht, wo die CPU die nächste Anweisung findet, die ausgeführt werden soll, wenn ein Interrupt auftritt. Diese Tabelle hat einen festen Platz im Speicher. Die Zuordnung Adresse - Interrupt ist vom System vorgegeben. Was nichts anderest bedeutet als dass diese Zuordnung in der Hardware realisiert ist und demnach vom Programmierer nicht verändert werden kann.

Wie genau der Interrupt arbeitet, was man sich darunter vorstellen soll, bzw. welche Auswirkungen ein Interrupt auf den momentanen Zustand der CPU hat, folgt später.

Zum Abschluss des Kapitels noch eine Grafik, wo alle relevanten Informationen des Speichers abgebildet sind. Diese Grafik mag zwar nicht alle Bereiche und alle Einzelheiten des Speichers wiedergeben, aber sie erlaubt einen eifachen und schnellen Überblick über alles was sich im Speicher wie und wo befindet.

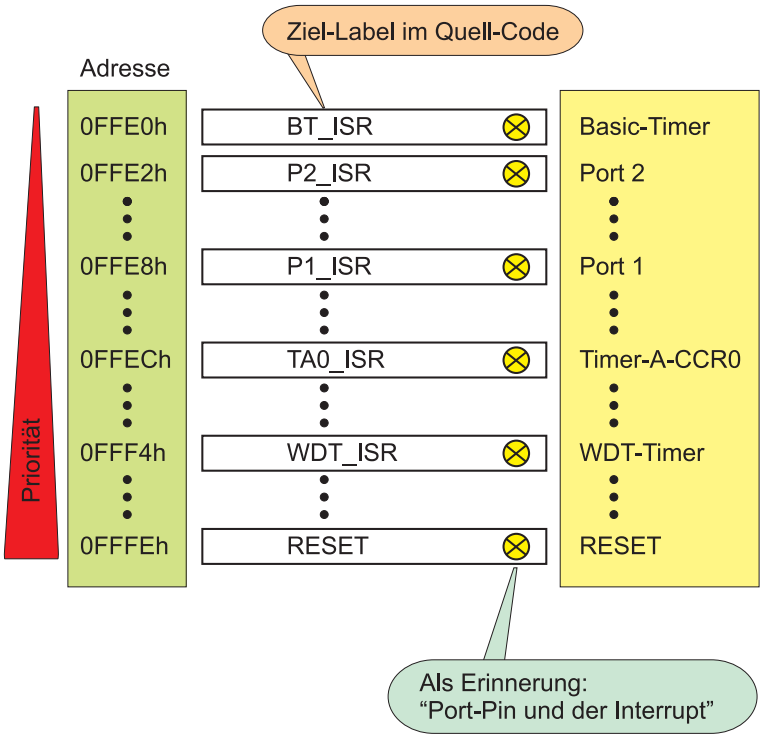

Abbildung 1.2: Interrupt -Vector - Tabelle

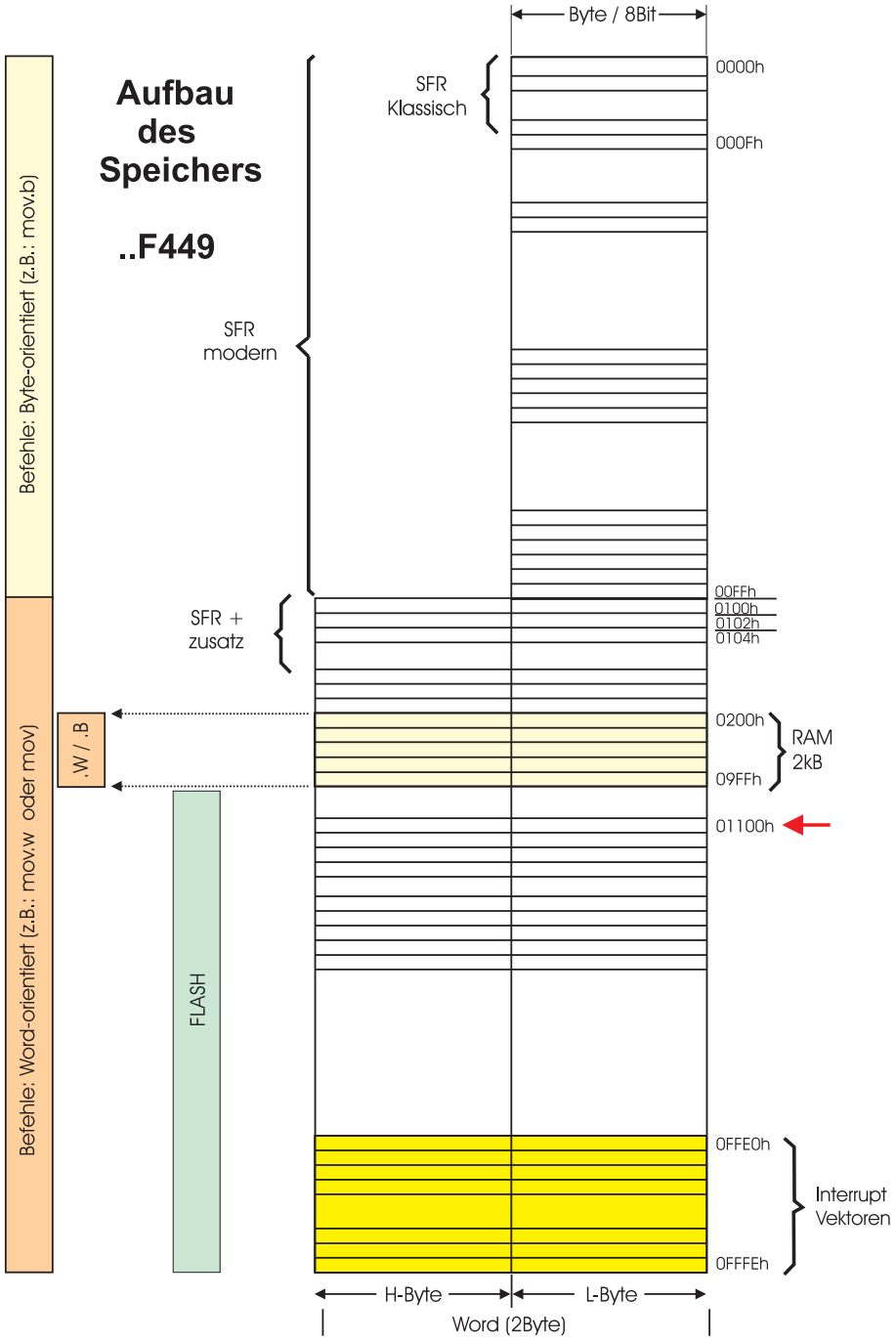

Abbildung 1.3: Aufbau des Speichers

2 Funktionsblöcke der MSP430

2.1 Allgemeines

Um mit dem Mikrocontroller arbeiten zu können, müssen neben der Kenntinss, wie die CPU, Speicher, ... aufgebaut sind, auch die Funktionsblöcke bekannt sein. Denn ohne dieses Wissen ist es nicht möglich diese Module anzusprechen bzw. zu konfigurieren. In diesen Script werden nicht alle der vorhandenen Module besprochen, sondern nur dies, welche auch in der Vorlesung behandelt werden. Das sind:

- Der Port
- Der Watch-Dog-Timer (WDT)
- Der TimerA
- Das LCD
- Der Analog-Digital-Wandler (ADC)
- der Komperator

Erst danach kann mit der eigentlichen Programmierung begonnen werden.

2.2 Der Port

2.2.1 Allgemeines zum Port

Der Port-Block, auch IO-Modul (IO = Input Output) genannt, ist zuständig für die Bereitstellung der Hardware, der Pins. Pins sind kleine Signalleiter und werden über den Port auf den Bus übertragen, worüber die verschiedenen Module wie zum Beispiel der Timer und Adressen im RAM zur Verfügung stehen. Acht Pins sind jeweils zu einem Port zusammengefasst. Das hat den Zweck, da 8 Pins auch 8 Bit ergeben, also eine Adresse im Speicher bilden. Unser Derivat, der MSP430F449, hat insgesamt 6 Ports mit jeweils 8 Bit breite. Die Ports 1 und 2 sind interruptfähig, während die anderen diese Funktionalität nicht implementiert haben.

In diesem Kapitel wird versucht die Funktionsweise des Ports an Hand eines einzelnen Pins zu erklären. Es wurde hier absichtich eine recht bildliche Art der Beschreibung gewählt. Da dies den Einsteiger die Funktionsweise am schnellsten verständlich macht.

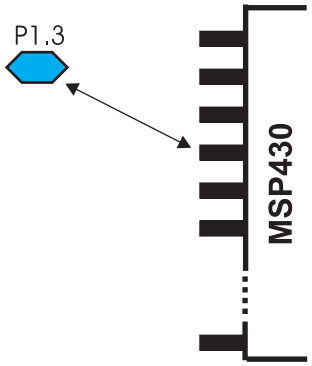

Abbildung 2.1: Funktionsweise eines IO Pins

Die Grafiken, welche diese Funktionsweise darstellen wurden so aufbereitet, dass ein Ablauf klar zu erkennen ist. Viele Studierende haben in der Vergangenheit mir dies bestätigt.

2.2.2 Die Funktionsweise eines Ports

Ausgehend von dem Beinchen am IC, welches ein Port-Pin sein kann, wollen wir hier die Funktionsweise schrittweise erklären. Bei einem Blick in das Datenblatt (Datashhet) des MSP430F449, stellen wir fest dass fast alle Beinchen des MSP430 mit mehreren Funktionen versehen sind. So auch die Beinchen (Pins) der Ports.

Diese Funktionen sind entweder “Binärer IO” oder z.B. “TA0-C-OUT” um nur eine Möglichkeit zu nennen. Wir betrachten den Port Pin 1.3, stellvertretend für alle Port-Pins. Sollten Unterschiede auftreten, bzw. Spezialfälle, die von dem hier beschriebenen Standard abweichen, so werden diese an den betreffenden Stellen sofort besprochen. Wir müssen somit dem Portpin zunächst mitteilen wie er in unserem Programm benutzt wird. Dies geschieht wie immer durch den Zustand eines Bits. Diese Auswahl, ob IO-Funktion, oder Spezial-Funktion wird im Ports spezifischen Register *PxSEL* (x = steht hier für 1 ... 6), eingestellt. In den Grafiken symbolisieren die Schalter die Zustände der Bits, also “0” oder “1”. Wenn wir den Port P1.3 als binärerer IO-Pin benutzen wollen, dann muß das Bit 3 im P1SEL-Register demnach auf “0” stehen. Wäre das nicht der Fall, dann wöder dieser Pin P1.3 einem anderem Modul gehören. Die Spezial-Funktionen können nicht vom Anwender frei vergeben werden, sondern sie sind fest vorgegeben, also in der Hardware fest implementiert.

Nachdem wir festgelegt haben, dass dieser Port-Pin als binärer IO benutzt werden soll, muss jetzt geklärt werden welche Datenrichtung gefordert ist. Eingang (Input) oder Ausgang (Output). Diese Funktion wird im Register *PxDIR* eingestellt. Der Bit-Wert “0” wird hier gesetzt, wenn dieser Port-Pin als Eingang benutzt werden soll. Dieses Funktion Input / Output wird auch bei der Funktionsweise "’Spezial-Funktion"’ benötigt. Da

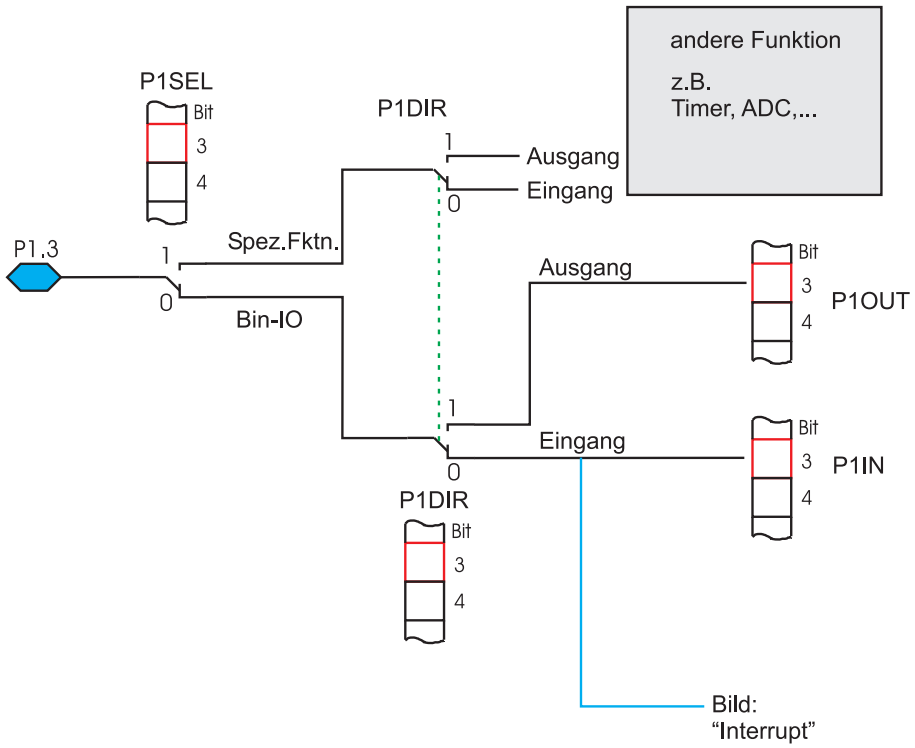

Abbildung 2.2: Port Pin IO-Funktion

es auch in diesem Betriebsfall vorkommen kann, dass wiederum zwei unterschiedliche Funktionalitäten dann an diesem Port-Pin zur Verfügung stehen.

Wenn wir in *P1DIR* das Bit3 auf Null setzen, so wird jetzt der Pin als Eingang geschaltet. Der Zustand der am Pin aussen anliegt (Logik-Pegel), liegt dann gleichzeitig auch am *P1IN* Register an. In der Grafik ist auch zu sehen, dass bei P1.3 = Bin-IO Input noch eine Verzweigung existiert. Diese wird weiter unten besprochen. Sollte das Bit3 im *P1DIR* Register auf “1” gesetzt werden, so wird der Zustand des Bit3 im *P1OUT*-Register an den Port-Pin gelegt. Wir wir sehen hat der MSP430 ein seperates *IN*- und *OUT*-Register. Diese Funktionalität haben nicht alle Microcontroller. Es ist somit möglich zunächst einen Zustand zu empfangen und dann am gleichen Port-Pin wieder einen Zustand auszugeben, wobei diese Zustände nicht unbedingt gleich sein müssen.

Nachdem diese erste Hürde genommen wurde, wollen wir uns der Sonderfunktion von Port 1 und Port 2 anschauen, die nur dann verfügbar ist, wenn der betreffende Port-Pin sich im Betriebszustand “binärer IO und Input” befindet.

Externer Interrupt an Port 1 und 2

Wie schon eingangs angesprochen sind die beiden 8-Bit breiten Ports 1 und 2, interrupt fähig. Das bedeutet, daß eine Änderung eines Pin Zustandes benutzt werden kann um den Programmfluss zu steuern. Zunächst muß entschieden werden ob dieser Port-

Pin überhaupt interruptfähig sein soll oder nicht. Dies wird im Register *P1IE* (= P1 Interrupt Enable) eingestellt. Als nächstes muss noch geklärt werden auf welche Art von Änderung zum auslösen des Interupts berücksichtigt wird. Die steigende- oder die fallende Flanke. Im fall der steigenden Flanke spricht man üblicherweise von der positiven Flanke und dementsprechend von der fallenden Flanke von der negativen Flanke. Dies wird im Register *P1IES* (= P1 Interrupt Eage Select) eingestellt. Wenn das betreffende Bit = "'0"' ist, dann wird die positive Flanke zur Erkennung eines Interruptes genutzt. Sobald ein solches Erreignis aufgetreten ist, wird ein Impuls generiert, der im *P1IFG* (= P1 Interrupt Flag) gespeichert wird. Alle Bits im *P1IFG*-Register teilen sich einen sogenannten Interrupt-Vektor, das bedeutet, dass wenn ein Interrupt vom Port P1 ausgelöst wurde, der Programmierer durch geeignete Abfragen selbst heraus finden muss, von welchem Port1-Pin der Interrupt kam. Hier ein Beispiel:

```
if ((P1IFG & 0x08) > 0) // Teste ob Bit 3 gesetzt ist
{
                        // wenn das zutrifft, dann
   P1OUT |=  0x080;  // Setze das Bit 7 auf "1"
   P1IFG &= ~(0x08); // lösche das Flag-Bit
}
else
{
                         // ansonsten, weiter ab hier
   P1OUT &= ~(0x080); // Setze das Bit 7 auf "0"
}
```

Damit dieses “Programm Fragment” funktionieren kann, müssen folgende Konfigurationen im Quellcode eingestellt werden:

- P1.3 und P1.7 müssen Bin. IO sein
- P1.3 muss als Eingang konfiguriert sein
- P1.7 muss als Ausgang konfiguriert sein
- P1.3 muss interruptfähig gemacht werden

Es empfiehlt sich das Flag-Register immer bei der Initialisierung zu löschen, damit sichergestellt ist, dass nicht nach dem globalen Interrupt Freigabe, aktiviert durch setzt des *GIE*-Bits (*GIE = Global Interrupt Enable*) im Statis Register sofort ein Interrupt kommt, obwohl keiner seit der Initialisierung aufgetreten ist.

Wieder ein neuer Begriff, bzw. eine neue Funktin in der Interrupt Verarbeitung, der GIE. Diese Funktion hilft uns zu bestimmen ab wann überhaupt Interrupts gewünscht sind, bzw. ab wann wir sagen, dass unser Programm soweit richtig konfigurtiert ist, dass es auch richtig reagiert, sobald ein Interrupt auftritt. Ein eingener (emulierter) Befehl zum setzten unf zum Rücksetzen des GIE ist im Befehlsumfang des MSP430 implementiert.

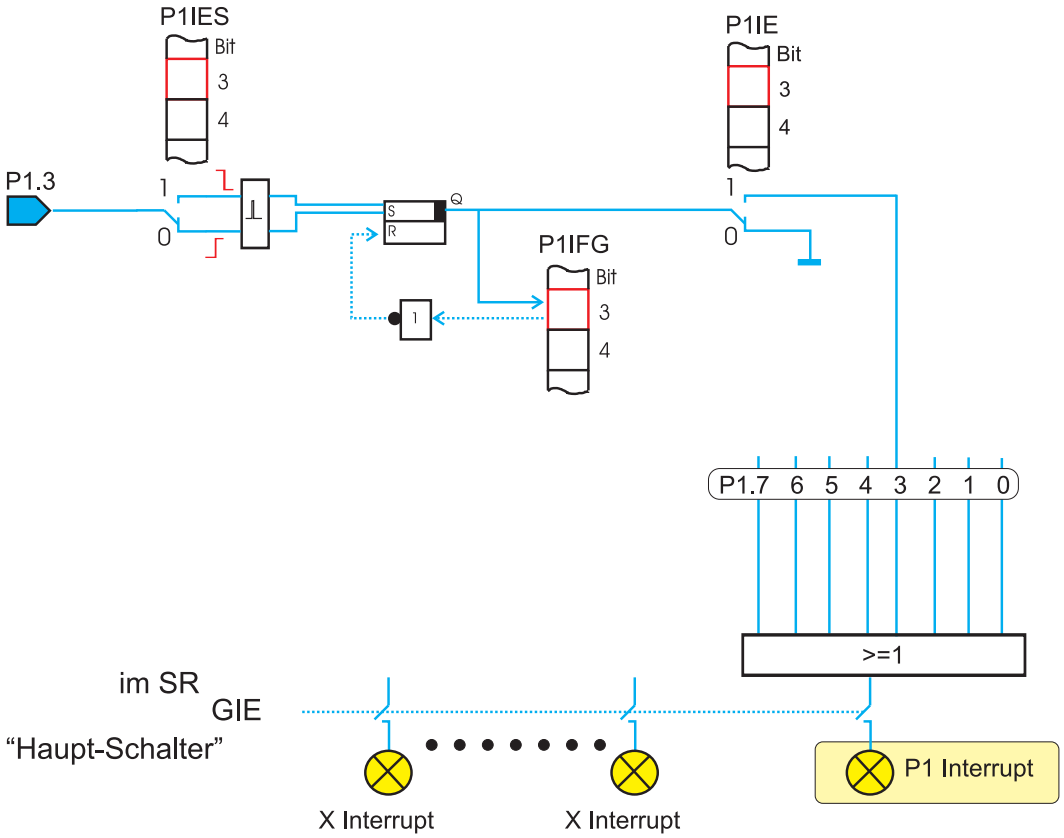

Abbildung 2.3: Interrupt am binären Input-Pin

Gesetzt, also Interrupts zugelassen, wird das GIE durch _EINT und rückgesetzt durch _DINT.

Zum Abschluss des Kapitels soll noch folgende Konfiguration als Beispiel durchgeführt werden:

- P1.0 und P1.1 interruptfähig
- P1.0 bei pos.- und P1.1 bei neg.Flanke
- P6.0 ... P6.3 als Spezial-Funktion (hier ADC)
- P6.4 ... P6.7 als Bin.Ausgang

```
P1SEL = 0x00;       // Port1 komplett als Bin.IO Port
P1IE  = 0x03;       // Bit0 u. Bit1 von Port1 interruptfähig
```

```
P1IES = 0x02;       // Bit1 von Port1 auf neg.Flanke stellen
P1IFG = 0x00;       // Alle Port1 Int.Flags löschen
P6SEL = 0x0F;       // Bit0 ... Bit3 von Port6 auf ADC12 setzen
P6DIR = 0xF0;       // Bit4 ... Bit7 von Port6 als Ausgang setzen
```

2.3 Der Watch-Dog-Timer (WDT)

Ein wichtiges Hardwaremodul des MSP430 ist der sogenannte Watchdog Timer (die gängige Abkürzung dafür ist WDT). Den Kern des Watchdog Timers bildet ein 16-Bit Zähler (WDCNT - Watchdog Counter). Auf den Zähler kann nicht mit Hilfe der Software zugegriffen werden. Die Steuerung des Zählers erfolgt über das WDT-Control-Register, das zur Beeinflussung mehrerer wichtiger Parameter des Moduls unabdingbar ist. Der WDT hat folgende Eigenschaften:

Vier softwareprogrammierbare Zeitintervalle
Watchdog - Modus
Timerfunktion mit Interruptmöglichkeit
Passwort geschützter Zugriff auf das WDT Control-Register
Unterschiedliche Takt Quellen

Der WDT ist darauf ausgelegt zwei wichtige für den Betrieb des Mikrocontrollers Funktionen zu erfüllen:

2.3.1 Die Überwachungsfunktion des Watchdog - Timers (WDT)

Die Überwachungsfunktion ist die primäre Funktion des *WDT*. Sie besteht in der ständigen Überwachung der Softwareprozesse. Sollte ein Fehler beim Ausführen der Software vorkommen, wird ein Neustart des Systems erzwungen. Ein System-Reset wird erst nach dem Ablauf eines bestimmten Zeitintervalls automatisch generiert. In der Praxis bedeutet dies, dass beim Überlauf bzw. Erreichen eines voreingestellten Wertes im Zähler ein entsprechendes Neustart-Signal erzeugt wird. Um ein Programm kontinuierlich ausführen zu können muß die Software vor dem Erreichen des kritischen Wertes den Zähler zurücksetzen. Das regelmäßige Zurücksetzen des Timers ist die Voraussetzung für die ordnungsgemäße Programmausführung und wird durch das Setzen des Counter-Clear-Bits im *WDT-Control-Register* erreicht. Wird der Zähler nicht zurückgesetzt, geht der Watchdog-Timer davon aus, dass das Programm nicht mehr reagiert oder nicht sachgemäß ausgeführt werden kann, z.B. wenn eine Endlosschleife im Programm vorkommt. Als Folge wird das ganze System neugestartet. Somit bietet die Überwachungsfunktion zusätzlich eine Methode zur Fehlererkennung.

Bit.Nr	15...8	7	6	5	4	3	2	1	0
Fktn.:	Passwort	WDT	WDT	WDT	WDT	WDT	WDT	WDT	WDT
		HOLD	NMIES	NMI	TMSEL	CNCTL	SSEL	IS1	IS0

Tabelle 2.1: Watch-Dog-Timer-Control Register (WDTCTL)

2.3.2 Die Timerfunktion mit Interruptmöglichkeit des Watchdog - Timers (WDT)

Falls die Überwachungsfunktion in der Anwendung nicht nötig ist, kann der *WDT* auch als ganz normaler Timer vom Anwender in seinen Applikationen eingesetzt werden. In dieser Betriebsart stehen vier verschiedene Timerintervalle zur Verfügung. Wie bei allen anderen Timern (siehe oben) kann auch hier ein Interrupt generiert werden. Solch ein Interrupt signalisiert das Ende eines Timerintervalls. Mit Hilfe der Timerfunktion kann das Modul dazu konfiguriert werden, nach bestimmten Zeitintervallen Interrupts zu generieren. Der Programmierer hat die Möglichkeit, die Dauer der Zeitabstände selbst zu bestimmen. Um den *WDT* in den Timer-Modus zu versetzen muss das *WDTTMSEL*-Bit (Watchdog Timer Timer Mode Select) gesetzt werden.

2.3.3 Das Controlregister des Watchdog - Timers (WDTCTL)

WDCTL (Watchdog Control) ist ein 16-Bit passwortgeschütztes Register, worauf sowohl lesend als auch schreibend zugegriffen werden kann. Für alle Zugriffe auf das *WDCTL-Register* müssen Wordbefehle benutzt werden. Für schreibende Zugriffe wird das Schreibpasswort 05Ah benötigt, das im höheren Byte eingetragen werden muss. Beim Lesen wird der Inhalt des höheren Bytes als 069h ausgegeben. Solcher Mechanismus verursacht den Neustart des Systems und gewährleistet somit die Sicherheit bei unbefugten Schreibzugriffen. Wenn nicht gebraucht, kann der *WDT* ausgeschaltet werden. Zu diesem Zweck wird bereits vor Beginn der Programmausführung das *WDCTL-Register* entsprechend beschrieben:

Da in unseren Anwendungen der *WDT* keine Überwachungsfunktion hat, wird er gleich zu Beginn des Hauptprogramms (Hauptprogramm dient zur Initialisierung des Systems) ausgeschaltet. Der MSP430 gehört zu den Microcontrollern, wo der *WDT* standardmässig eingeschaltet ist. Damit diese Funktion unser Programm in keinster weise beeinflussen kann, wird der Watchdog Timer (*WDT*) gleich zu Beginn des Hauptprogramms deaktiviert. Dies geschieht mit dem Befehl:

```
WDTCTL = WDTPW + WDTHOLD;  // WatchDogTimer dealtivieren
```

2.4 Der Timer A (TA)

In diesem Script wird nur der "TimerA" beschrieben, denn die beschriebenen Funktionen sind genauso auch beim Timer B implementiert. Neben den Funktionen des TimerA sind beim TimerB einige weitere Funktionen zusätzlich implementiert (verschiedene Capture - Compare - Funktionen). Diese werden aber nicht weiter behandelt, da dies den Rahmen eines Skriptes sprengen würde. TimerA stellt eine Baugruppe dar, die eine Vielfalt von zeitbezogenen Abläufen im Mikrocontroller steuert. In den verschiedenen Derivaten des MSP430 ist diese Baugruppe unterschiedlich implementiert. Man hält z.B. TimerA3 und TimerA5 auseinander. Der Unterschied zwischen den beiden liegt vor allem in der Anzahl der Capture / Compare-Kanäle. Allen Varianten des Timers ist die Tatsache gemeinsam, dass sie als eine Art Uhr mit zusätzlichen Funktionen, z.B. als Stoppuhr oder Wecker, fungieren. Außerdem ist es mit Hilfe von TimerA möglich, eine Pulsweitenmodulation (PWM) zu realisieren sowie zahlreiche Interruptereignisse zu generieren.

2.4.1 Aufbau des Timer A (TA)

Das Periphrie Modul Timer ist in sich wieder in verschiedene Module unterteilt. Diese Module sind:

- Taktquelle
- Taktteiler
- der eingentliche Zähler
- Ausgabe Einheit

Das erste Modul was hier betrachtet wird ist das Modul Takteingang. Dieses Modul ist zuständig für die Auswahl der Taktquelle. Als Quelle hate der Timer vier verschiedene Möglichkeiten. Diese sind ein externer Takt (*TACLK*) dann der Auxillary-Clock (*ACLK*), der Sub-Main-Clock (*SMCLK*) und dann nochmals der inverse *TACLK*. Während der externe Tackt (Clock) über einen speziellem, dem Timer zugeordnetem Port-Pin (hardware spezifisch) zugeführt wird, werden die beiden anderen Takt innerhalb des MSP430 mehr oder weniger erzeugt. Da ist zunächst der *ACLK*, er leitet sich im Normalfall vom Quarz1 direkt ab. Das bedeutet die Frequenz vom Quarz1 ist auch die gleiche wie die Frequenz des ACLKs. In unserem Fall ist der Quarz1 mit den sogennaten Uhren-Quarz bestückt und hat eine Frequenz von 32,768 kHz. Welcher der vier Taktquellen benutzt wird, wird über zwei Bit im Control-Register des Timers (*TACTL*) eingestellt. Diese Bits haben das Label *TASSEL0* und *TASSEL1* von Texas Instruments erhalten. Alle diese symbolischen Namen sind in der Headerdatei auch so hinterlegt (dies hier nur nochmals am Rande bemerkt). Der Programmierer braucht sich somit nit die Bitpositionen in den betreffenden Control- oder Konfigurations-Registern zu merken, sonder muß nur die doch recht aussagekräftigen Labels in den Befehlen anzugeben.

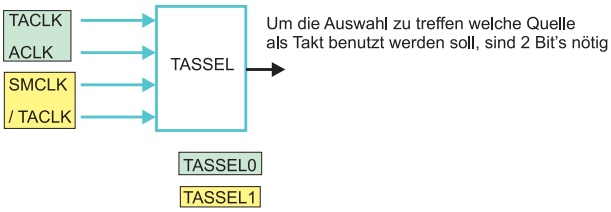

Abbildung 2.4: Taktquellen von Timer (A / B)

Nachdem die Auswahl getroffen wurde welcher Takt benutzt wird steht dem Programmierer eine weitere Funktion zur Verfügung um den Takt zu manipulieren. Das ist der Taktteiler (Input-Dividierer). Dieser Baustein ermöglicht es den Takt nochmals zu reduzieren. In dem er vier Teiler Möglichkeiten anbietet. Diese sind 1/1, 2/1, 4/1 und 8/1.

Was müssen wir uns darunter vorstellen. In einer Tabelle wird es vermutlich verständlich. An einem Beispiel, bei 2/1, hier die Beschreibung. Wenn auf der Eingangsseite zwei Takte von dem Quell-Baustein her kommen, wird nur ein Takt weitergegeben. Da mit jedem Takt der Zähler im eigentlichem Timer um 1 erhöht wird, bis er seinen Grenzwert erreicht hat, bedeutet, werden also doppelt so viele Takte auf seitens der Taktquelle benötigt.

n Eingangstakte	Einstellung	ID1	ID0	ID_x	Ausgangstakt
1 Takt	1 / 1	0	0	ID_0	1 Takt
2 Takte	2 / 1	0	1	ID_1	1 Takt
4 Takte	4 / 1	1	0	ID_2	1 Takt
8 Takte	8 / 1	1	1	ID_3	1 Takt

Beispiel: wir haben als Taktquelle den *ACLK* mit einer Frequenz von 32,768 kHz ausgewählt. Das bedeutet, dass in einer Sekunde 32768 Takte erzeugt werden. Diese Zahl entspricht genau dem Zählerstand von 08000h. Da unser Zähler ein 16Bit Zähler ist liegt sein maximaler Wert bei 0x0FFFF. Was nichts anderes ist als 65535, als genau das Doppelte. Bei einer Einstellung des Input-Dividierers von 1/1 braucht der Zähler von 0 bis zum erreichen seines maximalen Wertes genau 2s. Wenn wir jetzt den Teiler auf 2/1 einstellen, so braucht der Timer Zähler 4s um auf seinen Endwert zu kommen. Bei 4/1 dann schon 8s und schliesslich bei 8/1 volle 16s. Die Bits mit denen der Taktteiler eingestellt wird heissen ID0 und ID1. Statt immer die einzelnen Bits zu setzten ist auch eine Kombination der Einzelbits bereits in der Headerdatei hinterlegt (gilt für alle Funktionen, welche mehr als ein Bit benötigen um die Einstellung vorzunehmen). Diese erden dann immer mit dem Label gefolgt von einen tielfgestelltem Strich und dem binären Wert, welche die Kombination aufweist dargestellt.

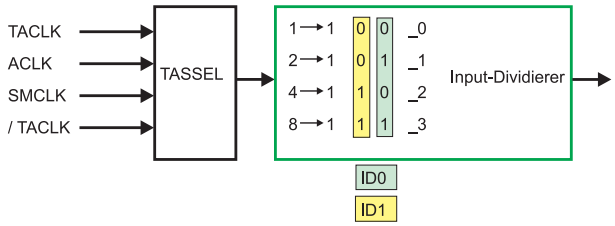

Abbildung 2.5: Input-Dividierer Timer(A / B)

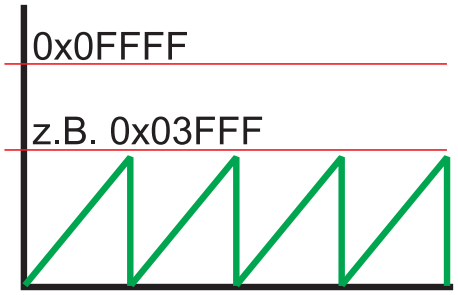

Abbildung 2.6: Timer Beriebsart: Up-Mode

Als nächsten Block, bzw. Funktionseinheit, welch im Blockschaltbild des Timers in den Block des eigentlichen Zählers integriert dargestellt ist. Ist der Betriebsarten Selector. Mit ihm wird eingestellt wie genau der Zähler arbeiten soll. Soll er überhaupt arbeiten? oder soll er immer nur bis zu einem vom Programmierer vorgegebenen Grenzwert zählen, oder immer bis zu seinem maximalen Wert? Oder soller als Hoch- und Runter-Zähler arbeiten. Diese Betriebsarten werden jetzt genauer unter die Lupe genommen.

2.4.2 Betriebsart: Stop-Mode

Mit dem Stop Modus kann der Timer angehalten werden. Dabei wird weder die Zählrichtung noch der Zählerstand verändert. Der Timer reagiert auf keinen weiteren Zählimpuls solange der Stop-Modus aktiv ist. Um den Timer wieder zu starten muss die Bitkombination des unterbrochenen Modus ins *TACTL*-Register eingetragen werden.

2.4.3 Betriebsart Up-Mode

Der Up-Modus bietet die Möglichkeit den Timer bis zu einem bestimmten Wert vorwärts zählen zu lassen. Dabei sieht der Algorithmus folgend aus:

Hier wird der Timer so eingestellt, dass er nicht bis zu seinem max. Wert von 0x0FFF zählt, sondern nur bis zu einem Wert der kleiner oder gleich diesem max. Wert ist. Dieser Wert muss im dem sogennaten *TACCR0*-Register (*TACCR0* = TimerA Capture-Compare-Register-0) eingetragen werden. Wenn jetzt diese Betriebsart gewählt wurde

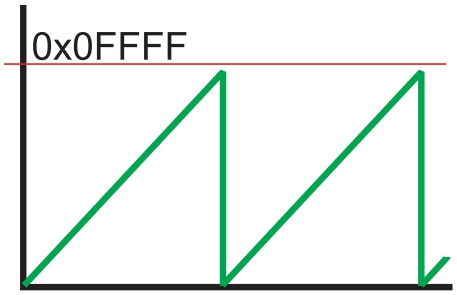

Abbildung 2.7: Timer Betriebsart: Continius-Mode

Abbildung 2.8: Timer Betriebsart: Up-Down-Mode

zählt der Timer immer von Null bis zu diesemn Wert und springt dann wieder zurüxck auf Null, sobald er diesen in *TACCR0* eingetragenem Wert erreicht hat. Er beginnt dann wieder von neuem zu zählen.

2.4.4 Betriebsart Continius-Mode

m Continius-Mode ist als oberer Grenzwert immer sein max. Wert von 0x0FFFF (65535) fest vorgegeben. Wenn er diesen Wert erreicht, so kann im TA-Controlregister (*TACTL*) eingestellt werden, dass bei erreichen dieses Wertes ein Interrupt ausgelöst wird. Dieser wird mit dem setzen des *TAIE* Bits im *TACTL* freigeschaltet.

2.4.5 Betriebsart Up-Down-Mode

Dieser Mode arbeitet wie auch schon der Up-Mode mit dem Grenzwert, der im *TACCR0* steht, fest zusammen. In diesem Mode wird jedoch nach erreichen dieses Grenzwertes nicht wieder auf Null zurückgestellt, sondern ab da an beginnter der Zähler Takt für Takt zurück bis auf Null zu zählen. Wenn wir also den Timer im Up-Mode so eingestellt hätten, dass er von volle 16s braucht , so könnte durch Umschalten vom Up-Mode in den Up-Down-Mode diese Zeit nochmals verdoppelt werden, auf 32s.

2.4.6 Timer Interrupt Möglichkeiten, allgemein

Jetzt kennen wir zwar den Timer als Baustein, aber wie wir einen Interrupt auslösen, der vom Timer initiiert wurde wissen wir noch nicht. Das soll sich jetzt aber ändern. Zunächt ist es von Bedeutung, dass es beim TimerA und -B mehrere Interrupt-Vektoren gibt. TimerA und -B haben jeweils 2 davon.

Timer	Vector-Adresse	gültig für
TA1	0x0FFEA	TA ..CC1, ..CC2, und TA
TA0	0x0FFEC	TA..CC0
TB1	0x0FFF8	TB ..CC1, ..CC2, und TB
TB0	0x0FFFA	TB..CC0

2.4.7 Timer Interrupt (TA / TB)

Wir müssen hier unterscheiden, ob ein Timer Interrupt ausgelöst wird, auf Grund eines Überlaufs der Timer-Counters (= Counterwert > Grenzwert / Vergleichswert) , oder auf Grund des Erreichens eines Vergleichs- oder Grenzwertes, welche von den Caprure-Compare-Einheiten des Timers vorgegeben werden. Betrachten wir zunächst den Fall, Überlauf des Timer-Counters *TAR* (beim TimerB wäre das *TBR*). Wir wissen, dass der Timer bei Null beginnt zu zählen und dann Schritt für Schritt, sich seinem max. Wert von 0x0FFFF (= 65535 dezimal), nähert. Sobald dieser Wert errreicht ist ist das 16-Bit breite Zählregister voll. Sollte jetzt +1 dazu addiert werden, so geschieht ein Überlauf, was nichts anderes bedeutet, als dass das Carry-Bit auf 1 gesetzt wird und der Inhalt des Zählregisters jetzt den Wert 0x0000 hat. Dieses Setzen des Carry Bits im Zusammenhang mit der Timerfunktion kann einen Interrupt auslösen, wenn wir das wollen. Diesen TimerA Interrupt "TA" können wir im TimerA Controlregister *TACTL* freigeben, durch setzen des Bits *TAIE*. Nochmals als Erinnerung: Dieser TA Interrupt kann nur dann kommen wenn wir es dem TA-Zählregister (*TAR*) erlauben seinen $TA_{max.Wert} + 1$ zu errreichen. Dies geschieht immer in der Betriebsart *"Continius-Mode"*. In den anderen Modis kann dieser Interrupt mit zur Auslöse gebreacht werden, muss aber nicht.

2.4.8 Timer Interrupt der CC-Module

Betrachten wir uns jetzt den "Up-Mode", beim "Up-Down-Mode" ist dies identisch. Hier beginnt das Zählregister wie immer bei dem Wert Null, aber der max. Wert kann unterhalb der Timer-Max-Wert von 0x0FFFF liegen. Diesen oberen Grenzwert, den wir vorgeben wollen müssen wir selbstverständlich in einem Register eintragen. Dieses Register ist das TimerA-Capture-Compare-Register-0 (*TACCR0*). Wie auch das *TAR* ist dieses Register 16 Bit breit und kann mit einem Wert von 0x0000 und 0x0FFFF geladen werden. Sobald der "UP-Mode" oder der "Up-Down-Mode" im *TACTL* aktiviert wird, vergleicht der Timmer immer, ob der momentane Zählwert noch kleiner ist als der Wert, der im *TACCR0* eingetragen ist, oder nicht. Jetzt ist zwar bekannt bis wohin der Timer-Zähler aufaddieren soll, aber wer stellt den Interrupt zur Verfügung.

Das *CCR0-Register* gehört zu den, dem Timer zugehöreneden, Capture-Compare-Modulen, demnach gibt es auch zu jedem dieser Module ein Control-Register. Dort wird dann der betreffende Interrupt frei geschaltet. Das Control-Register welches mit dem CCR0 zusammen arbeitet ist das *TACCTL0*. Das Bit-4 im *TACCTL0 Register* ist zuständig für die Interrupt Freigabe. TI hat diesem Bit in der Header-Datei *"MSP430F44x.h"* den Namen CCIE gegeben. Neben dem "*TACCR0 mit TACCTL0*" gibt es auch noch die beiden weiteren Gruppen, das "*TACCR1 mit TACCTL1*" und dem *"TACCR2 mit TACCTL2"*. Für was diese beiden Module eingesetzt werden können wird im nächsten Unterkapitel beschrieben. Doch bevor es soweit ist, soll hier zunächst ein Beispiel für die Konfiguration eines Timers aufgezeigt werden:

2.4.8.1 Aufgabe: Timer Konfiguration:

Realisieren Sie eine Funktion, welche im 0,5s Takt im Wechsel einen Ausgang-1 und einen Ausgang-2 auf logisch "1" setzt. Diese Funktion soll durch einen Übergang von $0 \rightarrow 1$ gestartet werden. Wenn an diesem Eingang der Pegel wieder von $1 \rightarrow 0$ wechselt, soll diese Fuktion abgeschaltet werden. Solange die Funktion nicht aktiv ist, sollen beide Ausgänge den logischen Wert "0" haben. Welcher von beiden Ausgänge als erstes gesetzt wird, bei Aktitvierung der Funktion, bleibt dem Programmierer überlassen. Bevor überhaupt eine Zeile Code geschrieben wird, sollte man sich klar machen was wird wie, wo und wann gebraucht.

2.4.8.2 Lösung Timer Konfiguration

Einen interruptfähigen Eingang, der bei pos. Flanke die Funktion ein- und bei neg. Flanke die Funktion wieder ausschaltet. Dann brauchen wir zwei binäre Ausgänge, welche zu Beginn beide den Wert Null (Low-Pegel) anstehen haben sollen. Dann benötigen wir zu guter letzt noch einen Timer. Wir wählen hier den Timer A (könnte auch Timer B sein), den wir mit dem Takt des $ACLK$ betreiben wollen. Da wir bei der Verwendung des $ACLK$ die Obergrenze nach 2s, 4s, 8s oder 16s erreichen, in Abhängigkeit des Inputdividierers, wir aber eine Peripdendauer von 0,5s wollen, wählen wir den Up-Mode. Den Wert den wir als oberer Zähler-Grenzwert brauchen rechnen wir wie folgt:

$$\tau_{0,5s} = \frac{0,5s}{2,0s} \cdot 65535$$

$$\tau_{0,5s} = 16384 \equiv 0x04000$$

Die Ein- und Ausgänge die benötigt werden können alle im Port_1 abgebildet werden. Wir legen fest: P1.0 als Eingang und P1.6 sowie P1.7 als Ausgang. Als Interrupt-Verktor benötigen wir den von TA0, da dieser bekannter massen mit dem *TACCTL0* zusammen arbeitet. Nachdem dies alles geklärt ist folgt jetzt der 1. Teil-Code den wir für die Initialisierung im sogenannten Hauptprogramm brauchen.

```
// ------- Init Port_1
P1OUT = 0x00;                // Port1 Output Reg. auf NULL
P1DIR = 0x0C0;               // Port1 Bit6 + Bit7 als Ausgang setzen
P1IE  = 0x01;                // Port1 Bit0 Int. freigeben
P1IFG = 0x00;                // Port1 löschen aller Flag-Bits
// ------- Init TA
TACCr0  = 0x04000;           // Setze Wert für 0,5s
TACCTL0 = CCIE;              // Aktiviere Int. bei CC0
TACTL   = TASSEL_1 + TACLR; // Wähle ACLK aus und TCounter Reset
_EINT()                      // Interrupts zulassen
```

Nachdem alle unsere Initialisierungen im Hauptprogramm gemacht wurden, folgen jetzt die beiden Funktionen, welche durch den jeweiligen Interrupt zur Ausführung gebracht werden, die

Port-1 Interrupt Service Routine (*P1_ ISR*)
TimerA Interrupt Service Routine (*TA_ ISR*)

```
// ------- Port_1 ISR -------
#pragma vector=PORT1_VECTOR
__interrupt void P1_ISR()
{
    if ((P1IFG & 0x01) > 0)         // Wurde Int. von P1.0 ausgelöst ?
    {                               // wenn JA, dann
        if ((P1IES & 0x01) > 0)     // Int. nach neg.Flanke
        {
            P1IES &= ~(0x01);       // setze wieder auf pos.Flanke
            P1OUT &= ~(0x0C0);      // Setze Bit6+7 auf "L"
            TACTL &= ~(MC0);        // Stoppe TimerA
            TACTL |= TACLR;         // Reset TAR auf NULL
        }
        else                        // wenn Int. nach pos.Flanke
        {
            P1IES |= 0x01;          // setze wieder auf pos.Flanke
            P1OUT |= 0x040;         // Setze Port1 Bit6 auf "H"
            TACTL |= TASSEL_1+MC0; // Aktiviere TimerA im Up-Mode
        }
        P1IFG &= ~(0x01);           // lösche das Bit0 im Flag-Reg.
    }
}

// ------- Port_1 ISR -------
#pragma vector=TIMERA0_VECTOR
__interrupt void TA0_ISR()
{
    P1OUT ^= 0x0C0;                 // Toggle Port1 Bit7 und Bit6
                                    // als Lebenszeichen
}
```

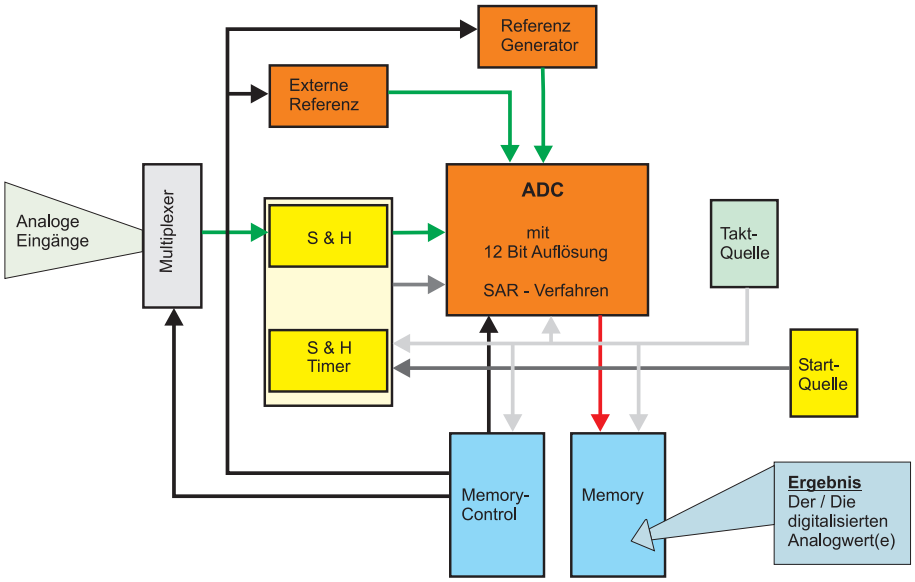

Abbildung 2.9: Bocküübersicht des ADC12

2.5 Der Analog Digital Wandler (ADC12)

2.5.1 Allgemeines, ADC12

Bei dem Analog / Digtal-Wandler des MSP430F449 handelt es sich mit um das umfangsreichste und vielseitigste Peripherie Modul innerhalb des MSP430. Der ADC12, der im MSP430F44x eingebaut ist, hat eine Auflösung von 12Bit, daher auch der Name ADC12 Auf der nächsten Seite ist der ADC12 mit seinen einzelnen Funktions Blöcken dargestellt. Man erkenn hier schon, daß diese sehr komplexe Struktur dem Entwickler, Programmierer, sehr viele Einstell- und Konfigurations-Möglichkeiten zuläßt. Zunächst werden die einzelnen Blöcke des ADC12 kurz beschrieben, so das der User eine Vorstellung erhält wie diese arbeiten. Bevor mit den Blöcken begonnen werden kann muß noch das Wandelprinzip des ADC12 kurz vorgestellt werden. Das ist notwndig um zu verstehen warum manche Komponenten im ADC12 vorhanden sind. Zudem ist es auch wichtig um die Verarbeitungszeit (Taktzyklen) die der ADC12 braucht um aus einem Analogen ein binäres Signal zu erzeugen. Der ADC12 ist ein sogenannter SAR-Wandler. SAR steht für “Sukzessive Approximation”. Neben diesem Wandelverfahren gibt es noch zahlreiche andere Verfahren, wie z.B. :Delta-Sigma-, Pipline- und Dual-Slope - Verfahren, um hier nur einige zu nennen. Diese anderen Verfahren werden hier in diesem Script jedoch nicht besprochen, da sie in diesem Derivat des MSP430 nicht zum Einsatz kommen.

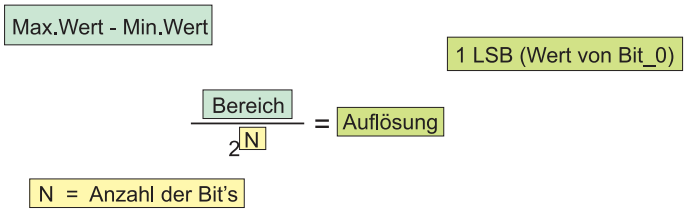

Abbildung 2.10: Auflösung / Quantisierung

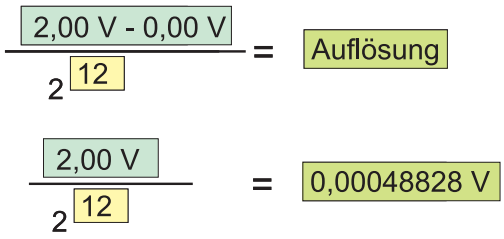

Abbildung 2.11: Bsp.Auflösung

2.5.2 Das SAR Verfahren, ADC12

Bei dem SAR-Verfahren wird der eingelesene Messwert (Analogwert) verglichen mit dem 1/2 Wert der Referenz. Analogwert - 1/2 Referenz-Wert = Ergebnis sollte das Ergebnis größer oder exakt gleich Null sein, dann wird das diesem Wandelschritt zugeordnete Bit gesetzt, ansonsten wird es auf Null gesetzt. Das Verfahren beginnt immer mit dem Bit, welches im binären Ergebnis die größte Wertigkeit hat und arbeitet sich dann Schritt für Schritt (Bit für Bit) runter bis zum Bit00, dem sogenannten LSB. Das Bit welches die größte Wertigkeit hat, nennt man auch das MSB Da wir hier einen 12-Bit-Wandler haben, brauchen wir somit auch 12 Takte um die eigendliche Wandlung durchzuführen. Wenn wir von der Auflösung eines ADC sprechen, dann bedeutet das:

gg

Der Max.Wert wird durch den Referenzwert nach oben hin begrenzt. Der Min.Wert wird durch den Wert V des Systems nach unten hin begrenzt. ss Im Normalfall wird der Min.Wert auf Null eingestellt, er kann aber auch davon abweichen.

Hier ein Beispiel dazu:

$V_{Ref+} = 2,00V$ (Max.Wert), $V_{Ref-} = 0,0V$ (Min.Wert), 12Bit Wandler

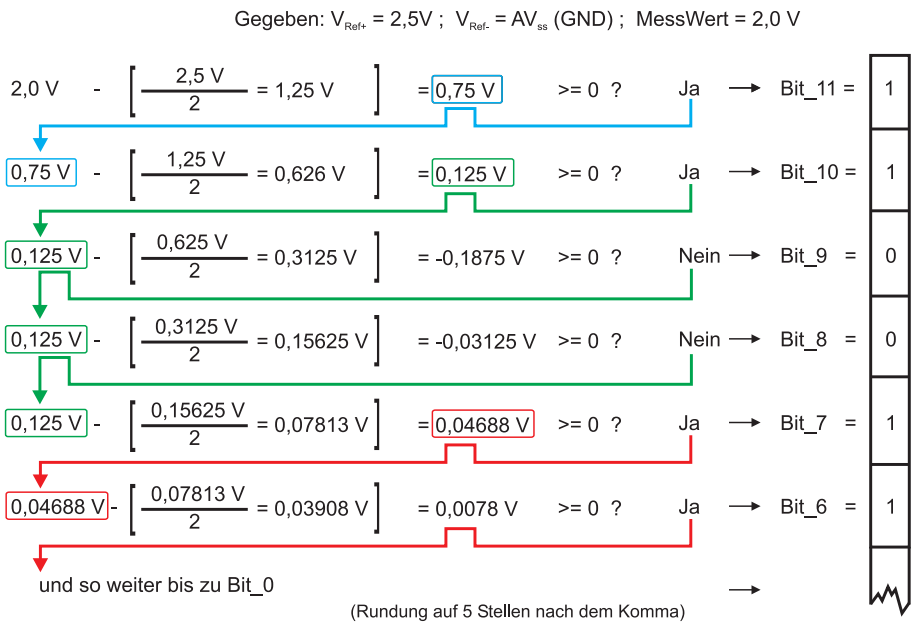

Abbildung 2.12: SAR Verfahren, ADC

Damit das Wiisen sich weiters festigt, nachfolgend einige Aufgaben.

V_{Ref+}	V_{Ref-}	ADC n-Bits		1 LSB
1,00V	0,00V	12	→	0,000244V
1,50V	0,00V	12	→	0,0.....
2,50V	0,00V	12	→	0,0.....
3,30V	0,00V	12	→	0,0.....
2,50V	0,00V	8	→	0,0.....
3,30V	0,00V	8	→	0,0.....
2,50V	0,00V	10	→	0,0.....
3,30V	0,00V	10	→	0,0.....

Hier nun das SAR Verfahren ausführlich dargestellt. Stellvertretend für alle Bits wurden hier die ersten x Bits beschrieben.

2.5.3 Signal-Quelle, ADC12

Da im MSP430 nur einen ADC12 hat, aber aus mehreren Signal-Quellen analoge Werte eingelesen und in binäre Signale umgewandelt werden sollen, müssen alle "Analogen Eingänge" über einen Multiplexer geschaltet werden, bevor sie weiter geleitet werden können. Insgesamt können bei dem hier implementierten Multiplexer 16 Kanäle angesteuert werden. Unter einem Multiplexer kann man sich einen Schalter vorstellen, der in

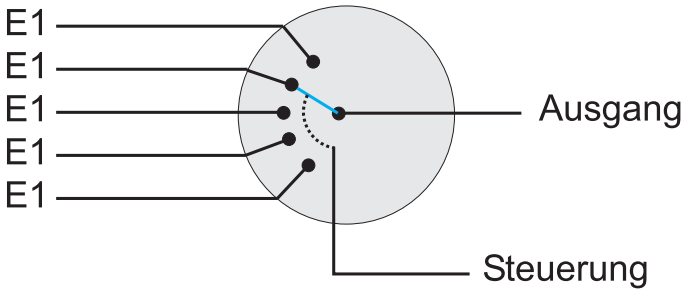

Abbildung 2.13: Multiplexer, ADC

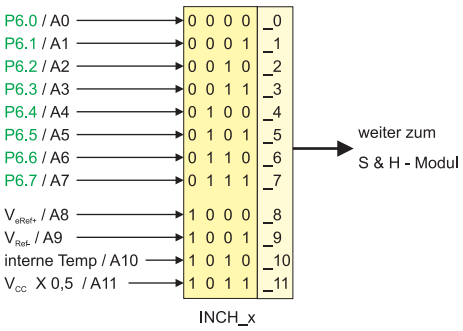

Abbildung 2.14: Struktur Eingangs-Multiplexer, ADC

Abhängigkeit der Schalterstellung einen Eingang an den Ausgang weitergibt.

Unser Derivat, der MSP430F449 hat insgesamt 12 unterschiedliche analoge Signalquellen implementiert. Der Port 6 des MSP430 ist so verschaltet, dass dieser als Port für bis zu 8 Analogsignale benutzt werden kann. Zu diesem Zweck muss das jeweilige Bit (welches dem jeweiligem Port-Pin zugewiesen ist) im Register *P6SEL* auf "1" gesetzt werden. Zudem ist sicherzustellen, dass das Direction-Register (*P6DIR*) auch auf Eingang (Input) steht. Diese Zuordnung kann jederzeit zur Laufzeit des Programmes erfolgen, bzw wieder entfernt werden. Die anderen vier Signalquellen direkt oder auch indirekt aus internen Quellen. Das sind V_{eRef+}, V_{eRef-}, eine Temp-Diode sowie Vcc/2.

2.5.4 Referenz-Spannung, ADC12

Der ADC12 hat einen eigenen Generator, der eine Referenzspannung aus der Versorgungsspannung generiert. Der Programmierer kann zwischen zwei Spannungen wählen, 0 ... 1,5 V oder 0 ... 2,5 V. Es besteht aber auch die Möglichkeit, wenn keiner der beiden "internen Ref. Spannungen" gewünscht werden, eine "externe Ref. Spannung" zu wählen. Zu diesem Zweck hat der MSP430 spezielle Pin's. Diese heißen "V_{eRef+}" und "V_{eRef-}".

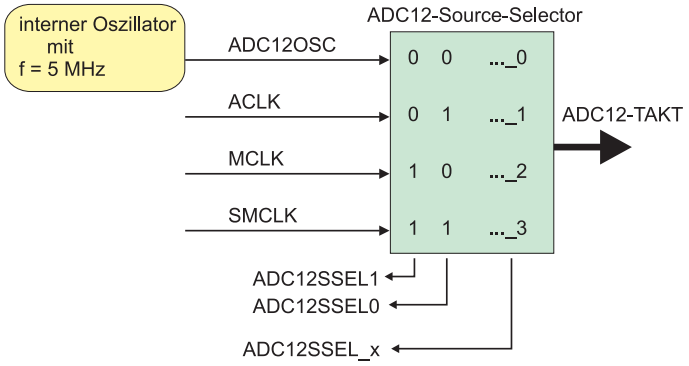

Abbildung 2.15: Takt-Block, ADC

Es ist darauf zu achten, dass die Spannungswerte nicht die Grenzen der Versorgungsspannung AV_{CC}Über, bzw. unterschreiten. Das bedeutet, dass in unserem Fall (Labor-Board) der Bereich von 0V ... 3,3V nicht verlassen werden darf. Die Funktion Ref. Spannung muß vom Programmierer eingeschaltet werden, wenn diese benutzt werden soll. Sollte diese nicht gebraucht werden, so muß sie auch nicht arbeiten und somit auch keinen Strom verbrauchen. Auch kann als Referenzspannung die Versorgungsspannung AV_{CC} des MSP430 benutzt werden.

2.5.5 Eingans-Takt, ADC12

Da der ADC12 ein taktabhängiges Modul ist, benötigt er zum Arbeiten einen Takt. Der Programmierer hat die Möglichkeit hier unter verschiedenen Takt-Quellen zu wählen:

Auf dem Bild sieht man, dass der ADC12 die Möglichkeit hat, seinen Arbeitstakt aus vier verschiedenen Quellen zu beziehen. Da sind:

ADC12OSC

Dies ist ein eigener Taktgenerator, welcher mit 5MHz atbeitet. Dieser Takt steht ausschliesslich dem ADC12 zur Verfügung.

ACLK

Dieser Takt ist im Normalfall der Takt des Quarz-1, der eine Frequenz von 32,768kHz hat. Was aber nicht immer der Fall sein muss. Siehe Kapitel "'Takt und Takterzeugung"'.

MCLK

Bei diesem Takt handeltes sich um den gleichen Takt, mit dem auch die CPU arbeitet. Wenn er nicht vom Anwender verändert wurde, beträgt dieser 1,048MHz.Siehe Kapitel "'Takt und Takterzeugung"'.

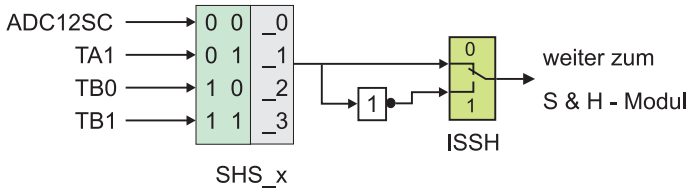

Abbildung 2.16: Start.Block, ADC

SMCLK

Dieser Takt ist vom MCLK abgeleitet, er kann gemauso gross sein wie der MCLK muss aber nicht. Auch ist es möglich diesen SMCLK von einem zweiten Quarz zu generieren, siehe Kapitel "'Takt und Takterzeugung"'. Im Standardfall ist die Frequenz des SMCLK und des MCLK gleich.

Um diese vier Taktquellen zu wählen, werden diese mit Hilfe der beiden Bit's *ADC12SSEL0* und *ADC12SSEL1* ausgewählt. Statt immer die beiden Bit's einzeln zu setzen (falls gewünscht), können diese aber auch über ein Equvivalent gemeinsam gesetzt werden. Diese hat die Bezeichnung *ADC12SSEL_x*. x steht hier für 0 ... 3.

Quelle	ADC12SSEL1	ADC12SSEL0	ADC12SSEL_x
ADC12OSC	0	0	_0
ACLK	0	1	_1
MCLK	1	0	_2
SMCLK	1	1	_3

Start / Aktivierung, ADC12

Um die Wandlung überhaupt starten zu können muß dem ADC12 ein Signal gesendet werden, damit er gezielt aktiviert werden kann. Er kann über vier verschieden Quelle aktiviert werden:

ADC12SC

Wenn diese Quelle ausgewähl wird, muß der Programmierer dafür sorgen, dass der ADC12 per Software gestartet wird. Dies geschieht dann duch das setzten von Bit_0 (*ADC12SC*) im *ADC12CTL0 Register* (ADC12 Control-Register-0) Dieses Bit im *ADC12CTL0* wird nur dann ausgewertet, beachtet, wenn auch die dementsprechende ADC Startart ausgewählt wurde. Sobald der ADC sich aktiviert hat, setzt er selbstständig dieses Bit zurück auf Null.

TA1

Hier dient als Start-Signalgeber der Timer A. Wenn der Wert im TA Capture-Compare Register 1 überschritten wird, kann der ADC gestartet werden. Wichtig ist dabei, dass die Betriebsart des TA Compare-Mode der OUT-OUT "SET/RESET" eingestellt ist. Denn nur in diesem *OUTMOD* kann der Flankenwechsel auch zur Ansteuerung des ADC benutzt werden.

TB0

Siehe TA1, nur daß hier als Quell Timer der Timer B benutzt wird und der Profgrammierer neben dem Capture-Compare-Register 0 (*TBCCTL1*) auch noch die von *TBCCTL1* benutzen kann.

TB1

Siehe TA1, nur daß hier als Quell Timer der Timer B benutzt wird und der Profgrammierer neben dem Capture-Compare-Register 0 (*TBCCTL1*) auch noch die von *TBCCTL1* benutzen kann.

Um eine dieser vier ADC12 Start's zu wählen, werden diese mit Hilfe der beiden Bit's *ADC12SHS0* und *ADC12SHS1* ausgewählt. Statt immer die beiden Bit's einzeln zu setzen (falls gewünscht), können diese aber auch über ein Equvivalent gemeinsam gesetzt werden. Diese hat die Bezeichnung *ADC12SHS_x*. x steht hier für 0... 3.

Quelle	ADC12SHS1	ADC12SHS0	ADC12SHS_x
ADC12SC	0	0	_0
TimerA.OUT1	0	1	_1
TimerB.OUT0	1	0	_2
TimerB.OUT1	1	1	_3

2.6 Sample & Hold, ADC12

Der Block "Sample & Hold" ist wohl mit der umfangreichste am ganzen ADC12. Zunächst was bedeutet Sample & Hold? Dieser Block ist zuständig um den Analogwert für die Umwandlung im eigendlichen ADC aufzunehmen und ihn dort zu halten bis die Umwandlung abgeschlossen ist. Wie lange die Aufnahme des Signales andauert wird entweder über den Timer im "S & H" festgelegt, oder über einen externen Timer.Abhängig ist die davon, ob im ADC12CTL1 das Bit "SHP" gesetzt ist oder nicht. Wenn dieses Bit = "1" ist, dann ist der "S & Hold - Timer " aktiv. Man kann sich den "S & H" Block wie ein RC-Glied vorstellen, das in der Eingangs- und Ausgangs-Seite je einen Schalter hat. Sobald ein "analoges Signal" aufgenommen werden soll, wird der Schalter auf der Eingangs-Seite geschlossen. Nach Ablauf der durch den Sample-Timer eingestellten Zeit (n - Taktzyklen), wird der Schalter auf der Eingangs-Seite geöffnet und dann der Schalter auf der Ausgangs-Seite geschlossen. Das so festgehaltene "Analog-Signal" ist jetzt abgekoppelt von der eingendlichen Signal-Quelle.

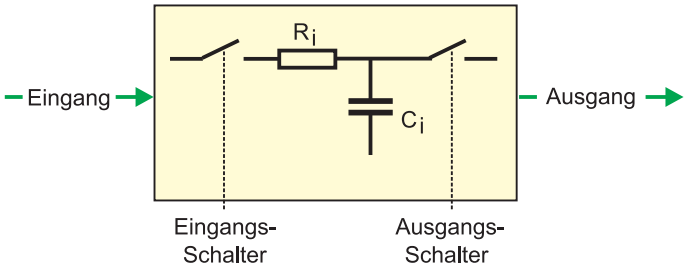

Abbildung 2.17: Sample & Hold, ADC

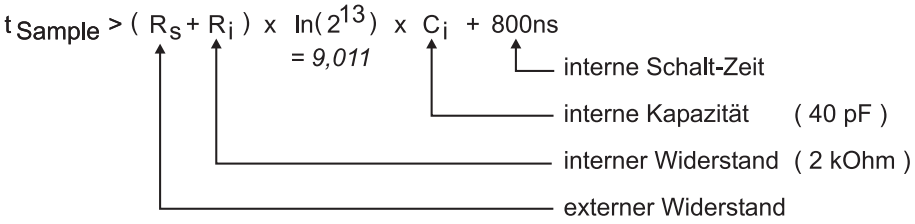

Abbildung 2.18: Gleichung: Sample-Zeit, ADC

Die Werte für Ri und Ci stehen im Datenblatt des MSP430. Wie groß die eigentliche Sample-Zeit ist, welche benötigt wird um ein Signal umzusetzen kann aus der nachfolgenden Gleichung berechnet werden:

Das Ersatzschaltbild sieht dann so aus:

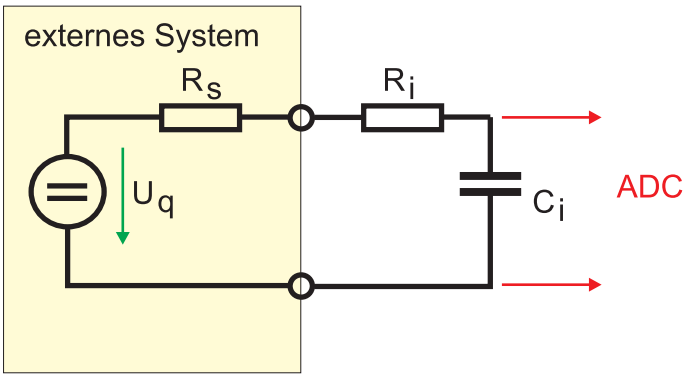

Abbildung 2.19: Eingangs-Ersatz-Schaltung, ADC

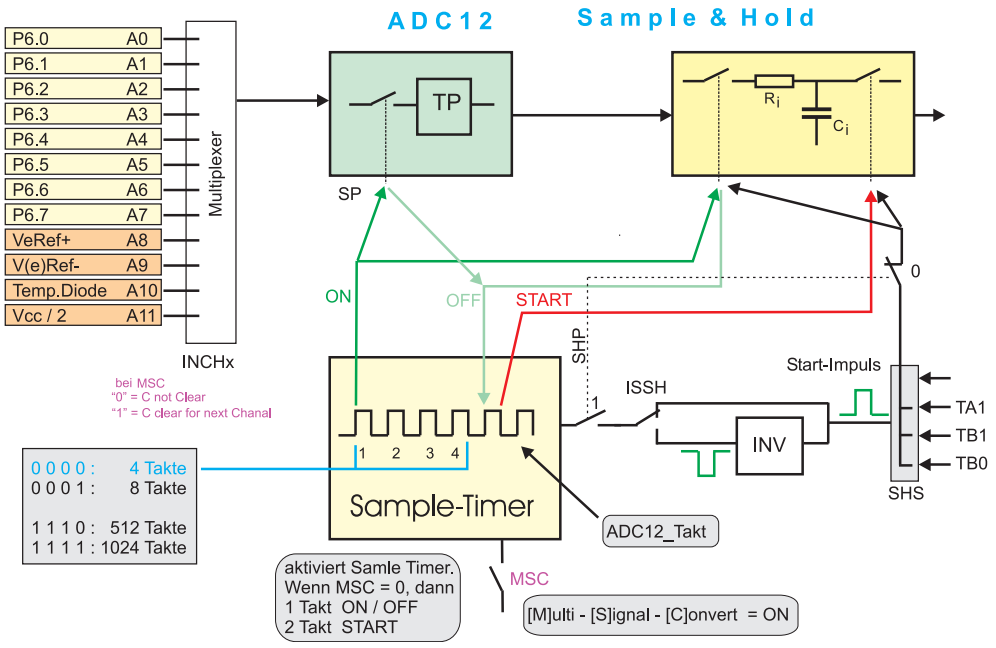

Abbildung 2.20: Funktionalität "S & H",ADC

Die minimale Sample -Zeit ist somit nicht nur vom System des ADC12 abhängig sondern auch vom EingangsWiderstand der Signalquelle. Wie groß ist die Sample-Zeit bei:

$R_S = 100\Omega \rightarrow t_{Sample} \geq$ ..... s
$R_S = 500\Omega \rightarrow t_{Sample} \geq$ ..... s
$R_S = 1k\Omega \rightarrow t_{Sample} \geq$ ..... s
$R_S = 2k\Omega \rightarrow t_{Sample} \geq$ ..... s
$R_S = 10k\Omega \rightarrow t_{Sample} \geq$ ..... s

ier jetzt der gesamte Aufbau des "Sample & Hold" Fragments.

Was bedeutez das genau für den Anwender? Warum sollte manchmal die Sample-Zeit länger , mal kürzer gewählt werden. Ich will versuchen diese Fragen in einigen Sätzen zu beantworten. Betrachten wir uns das nachfolgende Bild. Hier sehen wir einen typischen Verlauf einen "Analoge Signal's".

Wenn wir jetzt das "S \& H - Timer" so einstellen, daß er so schnell wie möglich die Werte erfasst, würde das dann so aussehen:

Wir wir sehen, bekommen wir alle Schwankungen des "Analogen Signal's" mit. Das

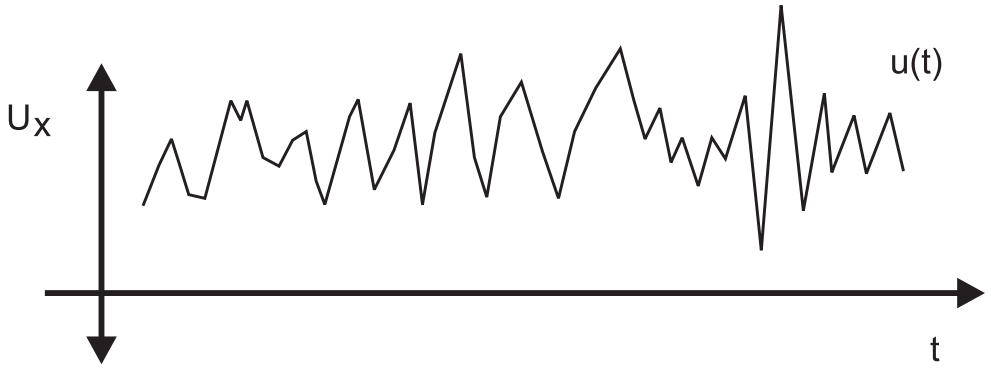

Abbildung 2.21: Analoges Signal Step1, Bsp.A

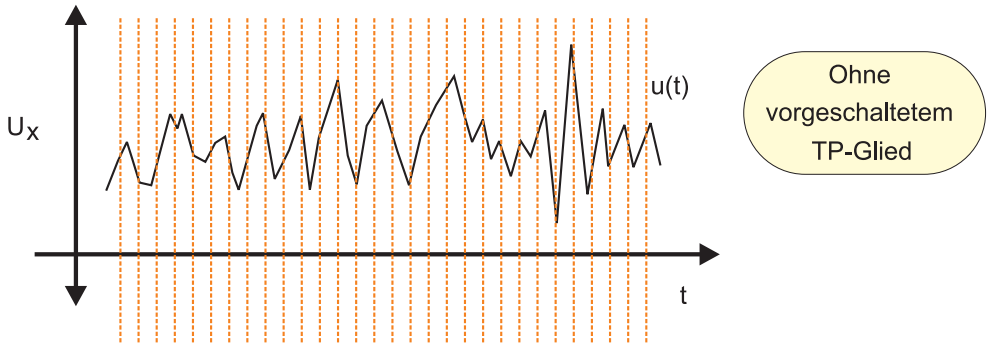

Abbildung 2.22: ADC mit kurzer Sample-Zeit

kann erwünscht sein, oder aber auch nicht. Denn oft ist es so, dass Oberwellen, oder Rauschen auf dem eigendlichem Signal liegen. Dieses Störungen wollen wir vielleicht nicht mit aufnehmen. Um solche Störungen zu unterdrücken lassen wir das Zeitfenster des "S & H - Gliedes" länger auf. Diese Verlängerung des Zeitfensters wirkt sich wie ein Mittelwert-Bildner aus. Schankungen werden somit gedämpft.

3 Grundlagen der Sprache "C"

3.1 Allgemeines

Nachdem der Mikrocontroller bekannt ist und auch die wichtigsten Module, kann endlich mit der eigendlichen Programmierung begonnen werden. Da stellt sich natürlich die Frage: "Warum gerade C?". Diese Programmiersprache bietet die Möglichkeit relativ hardwarenah und dennoch abstrakt die gewünschten Aufgaben umzusetzen. Diesen Umstand verdankte "C", daß sie als Standard Sprache in der Industrie einzug fand. In der Zwischenzeit gibt es viele unterschiedliche Arten der C-Sprache. Jedoch bauen alle auf den spgenanten ANSI-C auf, was als der urvater der Sprache heute gilt. Dieses ANSI-C wird auch in der Industrie meist eingesetzt, wenn es darum geht, Mikrocontroller, oder DSP's (Digitale Signal Prozessoren) zu programmieren. Aus diesem Grund wird auch in diesem Script ANSI-C verwendet. Zunächt werden die wichtigsten Komponenten aufgezeigt und auch an Beispielen erklärt. Danach wird das gelernte in kleinen Aufgaben eingebracht und analysiert. Übungsaufgaben sowie Verständnisfragen runden das Ganze dann noch ab Hier in diesem Script werden hauptsächlich die "C" Sprachinhalte besprochen, die auch in der Mikrocontroller Welt Verwendung finden. Auf die Möglichkeit der "String Verarbeitung" wird hier nicht weiter eingegangen, da diese Strings als eine Art von Zeichenfolge von Werte angesehen werden kann und somit diese dann über diesen Weg hin abgearbeitet, untersucht und auswerten läßt.

3.2 Variablen, Operatoren und Operanden

3.2.1 Variablen

In "C" wird alles über sogenannte Variablen realisiert. Diese nehmen die Daten (bzw. Werte) auf und werden in ihnen modifiziert bzw. weiter verarbeitet. Variablen, welche im "C" Programm durch den User definiert sind, werden immer im RAM abgelegt. Die Reihenfolge, wie sie dort stehen, hängt von der Reihenfolge der Initialisierung ab. Beschrieben wird der RAM von der kleinsten Adresse hin zur größten Adresse. Platzbedarf der unterschiedlichen Variablen (MSP430):

Bezeichnung	Platzbedarf	Werte Bereich (+)	Werte Bereich (±)
char	1 Byte	0 ... 255	-128 ... +127
short	2 Byte	0 ...65535	-32768 ... +32767
int	2 Byte	0 ... 65535	-32768 ... +32767
long	4 Byte	0 ... 2147483649	-2147483648 ... 2147483647
float	4 Byte		$-3{,}4 \cdot 10^{-38}$ ... $+3{,}4 \cdot 10^{38}$
double	8 Byte		$-1{,}7 \cdot 10^{-308}$ ... $1{,}7 \cdot 10^{308}$

Wenn nur Zahlen im positiven Bereich verwendet werden, dann spricht man in "C" von signed und wenn Zahel im ±Wertebereich benutzt werden sollen, dann spricht man von unsigned. Wenn vor dem variablen Type nichts vermerkt ist, wird automatisch der Type *signed* angenommen, einzigste Ausnahne hier ist der Variablen Typ "char". Hier ist als Standard *unsigned* implementiert.

Bei der Verwendung von Variablen kann eine Speicherklasse vergeben werden, dessen Gültigkeitsbereich und die Lebensdauer einer Variable beeinflußt. Dazu werden die Schlüsselwörter *static*, *extern*, *register*, *auto* für die Variablendefinition verwendet. Auch die Attribute *const* und *volatile* dienen dazu. Diese beiden Attribute werden sehr häufig eingesetzt. Variablen die mit diesen Attributen versehen sind werden vom Compiler nachdem sie einmal definiert sind an "Ort" und "Stelle" belassen.

Speicherklasse	Ort	Beschreibung
auto	RAM	Variable nur im Speicher, wenn sie benötigt wird
register	CPU	Variable wird als CPU Register gehalten
static	RAM	Variable behält ihren Wert
extern	?	Variable ist in einer anderen Datei definiert
const	FLASH	Variable ist fest,zur Laufzeit nicht veränder
volatile	RAM	Variable behält ihren Platz im RAM, immer

Bei dem Variablen Typ "*register*" muß darauf geachtet werden, wie größ denn die Register der CPU sind. Beim MSP430 haben diese eine Breite von 16Bit und können somit maximal Variablen vom Typ *int* oder *short* beinhalten.

```
// Beisiele von Variablen Definitionen
int iX;                // Ohne Init Wert, Daten: +/-
int iMaus = 0;         // Mit  Init Wert, Daten: +/-
unsigned long lY;      // Ohne Init Wert, Daten: 0 ... +
char cZeichn = '0';    // Mit  Init Wert, Daten: 0 ... +, Hier Wert = 30h
const fFaktor = 3.14; // Mit  Init Wert, Daten: +/-, Wert im Flash
```

3.2.2 Array's

Unter dem Begriff "Array's" versteht man Felder von Variablen, die alle unter einer Variable zusammen gefaßt sind. Die einzelnen Variablenwerte werden über Indizies an-

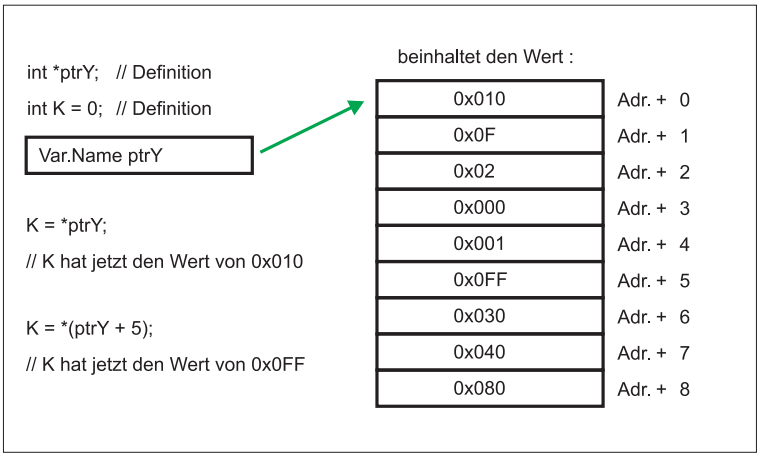

Abbildung 3.1: Arbeitsweise des C - Pointers

gesprochen.

```
// Beispiel von Variablen Feldern (Arrays)
int iX[10];           // Var mit 10 Interger Werten
unsigned int iZ[16];  // Var mit 10 Interger Werten, alle nur 0 ... +
char cSend[100];      // Var mit 100 Werten im Bereich 0 ... 255
```

3.2.3 Pointer / Zeiger

Unter dem Begriff "Pointer" bzw "Zeiger" versteht man eine Variable, welche nicht selbst den eingentlichen Wert beinhaltet, sondern die Adresse, wo der Wert steht.

Mit Hilfe dieser Art von Variablen kann sehr einhach und auch elegant auf Adressen direklt zugegriffen werden. Jedoch sollte man solange man ein reines "C-Programm" schreibt den absoluten, direkten Speicherzugriff dem Compiler überlassen. Denn diese verwantet ja bekanntermaßen den RAM für uns. Sollten wir also dort in sein Teritorium kommen, so kann es zu ungeahnten Reaktionen währen der Laufzeit des Programmes kommen. Deshalb sollte der Einsatz von Pointer in der Welt der Mikrocontroller nur gezielt und mit bedacht eingesetzt werden. Eine Ausnahme gibt es auch auch hier wiederum, wo der Pointer in der Praxis sehr häufig zum Einsatz kommt. Betrachten wir uns dazu das Code Fragment:

```
// Beispiel von Variablen Feldern (Arrays)
int iX[255];          // Zeile  1
int *ptrInput;        // Zeile  2
int y = 0;            // Zeile  3
```

```
unsigned int i = 0;  // Zeile  4

ptrInput = iX;       // Zeile  5

y = *ptrInput;       // Zeile  6
i = 5;               // Zeile  7
y = *(ptrInput + i); // Zeile  8
i = 7;               // Zeile  9
y = *(ptrInput + i); // Zeile 10
```

Betrachten wir uns nacheinander die einzelnen Zeilen des Quellcode's:

Zeile_1 Definition eines Feldes (Arrays) mit 255 Variablen vom Typ int.

Zeile_2 Definition eines Pointers der auf Adressen zeigt, welche Variablen / Werte enthält, die vom Typ int sind.

Zeile_3 Definiert eine Variable vom Typ int, die später unser Ergebnis aufnehmen soll.

Zeile_4 Definiert eine Var. vom Typ int, diese soll als Index benutz werden.

Zeile_5 Hier erhält ptrInput als Inhalt die Start.Adr. des Array's iX[].

Zeile_6 y enthält den Wert der in iX[0] steht, da der Pointer diese Adresse als Ziel hat.

Zeile_7 i erhält einen neuen Wert. i = 5.

Zeile_8 y enthält den Wert der in iX[5] steht, da der Pointer diese Adresse als Ziel hat. Die Zieladresse des Pointers setzt sich jetzt zusammen, aus dessen Basus-Adresse und dem Indexwert.

Zeile_9 i erhält einen neuen Wert. i = 9.

Zeile_10 y enthält den Wert der in iX[9] steht, da der Pointer diese Adresse als Ziel hat. Die Zieladresse des Pointers setzt sich jetzt zusammen, aus dessen Basus-Adresse und dem Indexwert.

Das Thema "Pointer" hat noch viel mehr Möglichkeiten, die jedoch z.Z. hier in diesem Script noch nicht implementiert sind. Darum soll hier an dieser Stelle auf einschlägige Literatur verwiesen werden, wenn es darum geht, sich tiefer in die Arbeitweise und die Möglichkeinten des Pointer, einzusteigen.

3.2.4 Operatoren

Zunächst einmal hier die wichtigsten Operatoren, jeweils mit Beispiel, welche in der Sprache "C" benutzt werden:
(Beispiel: B = 24; C = 5;)

Operand	Symbol	Beispiel
Addition	+	Q = A + B;
Subtraktion	-	Q = C - D;
Multiplikation	*	Q = A * X;
Division	/	Q = Z / 100;
Modulo	%	Q = x % 10;

Vergleichs-Operanden

Operand	Symbol	Beispiel
größer	>	B > C
kleiner	<	D < Z
größer, gleich	> =	K > = J
kleiner, gleich	< =	F < = 10
identisch	= =	K = = H
nicht gleich	! =	G ! = 13
UND	&&	(A > B) && (A > K)
ODER	||	(C <= 0.0) || (C >= 10)

Bit - Operanden

Operand	Symbol	Beispiel
AND	&	A = B & C;
OR	|	A = B | C;
Invers	~	A = ~B;

Schift - Operand

Operand	Symbol	Beispiel
schiebe nach Rects	>>	K = B >> 1;
schiebe nach links	<<	F = B << 2;

Ergänzend dazu hier noch einige sehr wichtige schreibweisen von Anweisungszeilen. Als Startwert soll die Variable "Q" immer den Wert von 24 haben.

Lang-Form	Kurz-Form	Beispiel (Ergebnis)
Q = Q + 1;	Q++;	Q = 25
Q = Q - 1;	Q–;	Q =23
Q = Q + 5;	Q += 5;	Q = 29
Q = Q - 4;	Q -= 4;	Q = 20
Q = Q * 3;	Q *= 3;	Q = 72
Q = Q | 1;	Q |= 1;	Q= 25
Q = Q & 2;	Q &= 2;	Q = 2
Q = Q >> 1;	Q >>= 1;	Q = 12
Q = Q << 2;	Q <<= 2;	Q = 96

Diese sogennate "Kurzform" oder kurze schreibweise, sollte wenn möglich vom Programmierer immer benutzt werden. Dies Art der schreibweise signalisiert dem Compiler, daß dieser diese Anweisung versuchen soll mit einem speziellen Befehl oder einem anderem Konstrukt umzusetzen in der Assemblercode, als daß der Fall wäre, wenn die lange schrweibweise benutzt wird.

3.2.5 Strukturen

Hier einige der wichtigsten Kontrollstrukturen, welche die Sprache "C" bietet. Auch hier wieder das Ganze jeweils mit einem Beispiel.

3.2.5.1 for (...) Schleife

Diese sogenannte "for - Schlefe" wird solange durchlaufen, wie daß die im Kopf der "for-Schleife" definierte Variable den Grenzwert nicht erreicht. Der Start-Wert, wie auch der Grenz-Wert und die art der Modifikation der Schleifen-Variable, werden im Kopf (Header) der Scheife initiiert.

```
// Beisiel einer for - Schleife
// mit Start-Wert = 0 und einem MaxWert von 100.
// Die Schleife soll 100x durchlaufen werden
for (i = 0; i < 100; i++)
{
    Z[i] = Y[i] * K;
}
```

In dieser "for-Schleife" wird aus einem Array Y[] an der Stelle i der Wert ausgelesen, dieser dann mit dem Faktor K multipliziert und in das Feld (Array) Z an der Position i eingetragen. Sobald die "LaufVariable i" den Wert von 100 erreicht hat, wird diese Schleife verlassen und der Code zur Ausführung gebracht, der unterhalb dieses Abschnittes steht. i++ bedeutet hier soviel wie: "Führe alle Anweisungen innerhalb der "{" und "}" aus, dann erhöhe i um 1.

Wenn jetzt gewünscht wäre, daß i nicht mit der Schrittweite von 1 arbeiten soll, sondern mit der Schrittweite von 2, wie müßte dann die Kopf-Zeile der "for-Schleife" aussehen?

3.2.5.2 while(...) Schleife

Es gibt zwei unterschiedliche Arten von "While-Schleifen". Das wäre die "while ... do" und die "do ... while" Variante. Zunächst soll die "while ... do" Schleife betrachtet werden. Diese Schleife wird solange bearbeitet, wie die Bedingung in der Kopf-Zeile wahr ist.

```
// Version I
// Beispiel einer while ... do  - Schleife
```

```
k = 1;
while (n <= M )
{
     k = i << 1;  // K = i * 2
     y[k] = y[k - 1] + sin_tab[i] * InData[n];
     n++;
}
```

Was würde passieren, wenn die letzte Zeile (n++) nicht eingebaut wäre? welche Konsequenz hätte das zur Folge?

```
// Version II
// Beisiel einer do ... while - Schleife

k = 1;
do
{
     k = i << 1;  // K = i * 2
     y[k] = y[k - 1] + sin_tab[i] * InData[n];
     n++;
}
while (n < M)
```

Der entscheidende Unterschied ist hier, daß bei der Variante 1 (while...do) zunächst geprüft wird, ob die Bedingung erfüllt ist und dann entschieden wird, on der Code innerhalb der { ... } zur Ausführung gebracht wird, oder nicht. Bei der Version II, (do ... wihle) wird auf jeden Fall der Code innerhalb der { ... } zumindest 1x zur Anwendung gebracht und erst danach wird überprüft, ob dieser Code erneut zur Ausführung gebracht werden soll oder nicht.

3.2.5.3 if ... else ... Bedingung

Eine weitere sehr häufig eingesetzte Struktur ist die "if .... else" Bedingung. Sie arbeitet ähnlich wie die bereits oben aufgeführte "while ... do" Schleife. Nur, daß der Code welcher zur Ausführung gebracht wird, wenn die Bedingung in der Kopfzeile erfüllt ist, nur einmal durchlaufen wird und dann automatisch die nächten Anweisungen im Programm Code abgearbeitet werden. Sollte in Abhängigkeit der Bedingung, welche in der Kopfzeile der "if" Anweisung steht, unterschiedliche Code's ausgeführt werden, so kann dies mit der Option "else" erledigt werden. Der Code, welcher der "else" Bedingung zugewiesen ist, wied nur dann ausgeführt, wenn die Bedingung, welche direkt hinter if (....) steht nicht wahr sondern falsch ist. In einem Flußdiagramm kann dies sehr einfach dargestellt werden:

Der Quell-Code dazu sieht dann folgenermaßen aus:

```
//
// Beisiel einer if ... else ... Bedingung
```

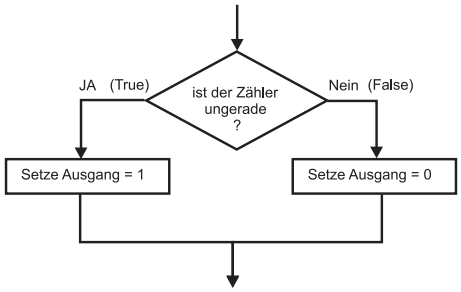

Abbildung 3.2: if ... else... - Bedingung

```
if (Zaehler &= 0x01) > 0)  // Testet ob das Bit0 gesetzt ist, dann ungerade
{
    P1OUT |= 0x01;     // Setzt das Bit0 im Port1 Ausgangs.Register
}
else
{
    P1OUT &= ~(0x01); // löscht das Bit0 im Port1 Ausgangs.Register
}
```

Wenn der Zweig mit "else" nicht benötigt wird, dann sieht das Ganze so aus:

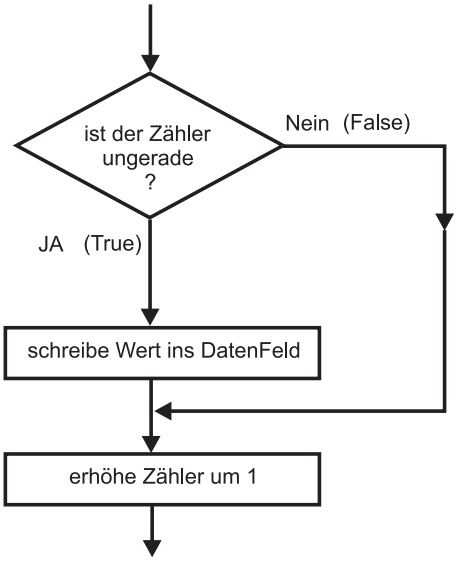

Abbildung 3.3: if .... Befingung

Der Quell-Code dazu sieht dann wiederum so aus:

```
//
// Beisiel einer if ... else ... Bedingung

if (Zaehler &= 0x01) > 0)  // Testet ob das Bit0 gesetzt ist, dann ungerade
{
    Daten[i] = P6IN;       // Schreibt die Daten ins Datenfeld
}
i++;                       // erhöhe Index
```

3.3 Zugriffe auf Register & Adressen

Mit einer der wichtigsten Operationen in der MircoController Welt ist das Manipulieren von einzelnen, oder Gruppen von Bit's in den sogenannten Registern. Auch sind Kenntnisse über die Bedeutung der Bits hier von größter Wichtigkeit. Zum Einen um Zugiffe zu optimieren und zu Anderen um effektiven Code zu erstellen. Anders wie in der Welt der PC's ist hier der RAM und auch der Programmspeicher begrenzt und somit kostbar. Alles was dazu beiträgt um Platz und Zeit zusparen ist in der Welt der MicroController wichtig.

3.3.1 Bits und Bytes, allgemein

Zunächst einiges (als Wiederholung) über Bit's und Byte's.

Ein Byte (= DatenByte [DB]) hat 8 Bit. 2 Byte sind ein Word (= DatenWord [DW]) und haben somit 16 Bit. Ein sogenanntes Doppel-Word [DD] besteht aus 2 DW oder aus 4 DB und hat somit 32Bit. Jedes Bit hat eine, für sich spezielle Wertigkeit, in Abhängigkeit wo es sich im DB, DW oder DD befindet. Den Zusammenhang zwischen Bit.Nr. und Wertigkeit sieht man am Besten in grafischer Form. Der Einfachheit halber, und um die Zahlen nicht allzu groß werden zu lassen wird hier da DB (DatenByte) dargestellt.

Bit.Nr.:	7	6	5	4	3	2	1	0
Wertigkeit :	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
in Dezimal :	128	64	32	16	8	4	2	1
in Hex :	80h	40h	20h	10h	8h	4h	2h	1h

Wenn man sich die Zeile mit den Hex-Werten genauer betrachtet, fällt auf, daß immer in 4er Gruppen die Ziffern 8, 4, 2 und 1 vorkommen. Dieser Umstand macht es auch sehr einfach, Hex-Zahlen in ihre binärer Schreibweise darzustellen, oder umgekehrt. Hier ein paar Beispiele:

Dezimal	Hex		7	6	5	4	3	2	1	0	Bit
			80h	40	20h	10h	8h	4h	2h	1h	Bit-Wertigkeit [Hex]
71	47h		0	1	0	0	0	1	1	1	
202	CAh		1	1	0	0	1	0	1	0	
			1	0	0	1	1	1	1	0	
			0	1	1	0	0	0	1	1	
	FFh										
127											

Ergänzen Sie die offenen Stellen in dieser Tabelle, um sich mit der Umrechnung Hex - Dez. - Binär vertraut zu machen.

Das Bit welches ganz links steht wird aus das MSB - Bit genannt. Dieses hat die größte Wertigkeit. Das Bit ganz rechts, also das bit mit der geringsten Wertigkeit wird als LSB - Bit bezeichnet. Diese beiden Bits weisen wichtige Merkmale auf, die bekannt sein sollten. Betrachten wir uns zunächst das LSB - Bit. Dieses Bit representiert die kleinste Einheit einer Zahl, die dargestellt werden kann. Das ist wichtig bei Verwendung von ADC's bzw. DAC's. Zudem kann durch dieser Bit herausgefunden werden, ob eine Zahl "gerade" oder "ungerade" ist. Wenn die Zahlen nicht vorzeichenlos anzusehen sind, sondern mit Vorzeichen, dann representiert das MSB - Bit das Vorzeichen. Sollte diese Bit der Wert "1" haben, so ist die Zahl negativ anzusehen, wenn diese Zahl als "signed " interpretiert wird. Das bedeutet auber dann auch, daß zur Darstellung von Werten im einen Datenbyte von den ursprüngliche 8 Bit nur 7 Bit für die eigendliche Zahl übrig

bleiben.

Zugriff auf Bits

Jetzt ist es an der Zeit endlich etwa mit diesen Byte's anzufangen. Um das Ganzen anschaulich zu gestalten soll dies an einem Beispiel gemacht werden. Dazu stellen wir uns vor, daß über ein Control-Byte ein Motor gesteuert werden soll. Diese Controlbyte soll eine feste Adresse haben, die 0FAh ist. Hier noch mal als Erinnerung: statt 0FAh könnte auch die schreibweise 0x0FA benutzt werden. Zunächt wird eine Festlegung getroffen welches Bit für welche Funktion im "Control-Byte" des Motors steht.

Bit.Nr	Funktion
0	Motor ON
1	Motor Rechts.Lauf
2	Motor Links.Lauf
3	Kühlung ON
4	RM: Motor ON
5	RM: Temp. > max
6	frei
7	Störung

Damit das Ganze später lesbarer wird wollen wir zunächt der Adresse 0x0FA dem Pointer mit dem Namen "MOTCTL" zuweisen. Dies geschieht mit Hilfe der Anweisung:

```
// Präprozessor Anweisung. Diese werden in der Regel in
// sogenannte Header-Files ausgelagert. (xxx.h)
//
#define MOTCTL_ (0x0FA)
DEFW(MOTCTL, MOTCTL_)
//
```

Wenn wir also jetzt etwas in die Adresse 0x0FA schreiben wollen, dann benutzen wir stattdessen "MOTCTL". Ausgangszustand soll sein, alle Bit's stehen auf ihrem Initial.Wert von NULL. Der Wert der Adresse 0x0FA ist somit 00h. Jetzt soll der Motor im Rechtslauf eingeschaltet werden. Dazu müssen die Bit's Bit_0 und Bit_1 auf "1" gesetzt werden.

```
//
//
   MOTCTL = 0x03;  // Motor im Rechtslauf, Motor = ON
//
//
```

Nach setzen dieser Anweisung arbeitet der Motor wie gewünscht. In der Adresse von "MOTCTL" befindet sich zunächst der Wert 0x03, den wir so dorthin geschrieben haben.

Da jedoch in diesem Control-Byte auch Rückmeldungen des Motors mit abgelegt werden, kann nachdem der Motor einmal gearbeitet hat nicht einfach eine Anweisung 0x0B eingetragen werden um den Motor in den Linkslauf zu überführen. Sollte dies dennoch gemacht werden, so würden Rückmeldungen des Motors alle auf den Wert "0" gestellt, was zu Problemen führen könnte. Es muß also ein Weg gefunden werden um einzelnen Bit's zu setzten bzw zu löschen. Betrachten wir hier zunächst den Fall, daß einzelne Bit's gesetzt werden sollen. In unserem Fall soll das Bit, welches die Kühlung des Motor's einschaltet gesetzt werden. Die Bit's, welche als Zustand ein "?" haben, werden vom Motor gesetzt und wir wissen somit nicht welchen Zustand sie haben. das Bit welches mit "X" gekennzeichnet ist, soll auf "1" gesetzt werden. Dazu machen wir von der Operation "Bitweises ODER" gebrauch.

	7	6	5	4	3	2	1	0	Bit
	?	0	?	?	X	0	1	1	MOTCTL_{alt}
OR	0	0	0	0	1	0	0	0	0x08
	?	0	?	?	1	0	1	1	MOTCTL_{neu}

In "C" sieht das ganze dann so aus. Zunächst in der Langen Form und dann darunter in der allgemein gebräuchlichen Kurzform. Diese Kurzform wird meist deshalb gewählt, weil der Compiler diese effektiver umsetzt (ist si implementiert).

```
//
// Lang-Form
//
   MOTCTL = MOTCTL | 0x04;

//
// Kurz-Form
//
   MOTCTL |= 0x04;
//
```

Als nächstes woolen wir den Motor aus dem Rechtslauf, in den Linklauf überführen. Dazu muß zunächst das Bit, welches für den Rechtslauf zuständig ist gelöscht werden und nach einer Zeit (damit der Motor auslaufen kann) das Bit für den Linklauf gesetzt werden. Auch hier wieder benutzen wir eine Bitmanipulations Operation. Jedoch diesesmal nit das "Bitweise OR" sonder das "Bitweise UND".

	7	6	5	4	3	2	1	0	Bit
	?	0	?	?	1	0	1	1	MOTCTL_{alt}
	0	0	0	0	0	0	1	0	0x02
INV	1	1	1	1	1	1	0	1	
	?	0	?	?	1	0	1	1	MOTCTL_{alt}
&	?	0	?	?	1	0	0	1	MOTCTL_{neu}

Es muß also das Bitmuster zuerst invertiert werden und dieser Wert dann bitweise mit

dem Inhalt von MOTCTL über UND verkrüpft werden. Anschließend kann die Wartezeit wirken und dannach wiederum kann der Motor in die Betriebsart "Linkslauf" gesetzt werden.

```
//
// Lang-Form
//
   MOTCTL = MOTCTL & ~(0x02);
//
// Kurz-Form
//
   MOTCTL &= ~(0x02);
//
   iWait = 1000;          // Warte Zeit setzen
   while(iWait > 0)       // warte bis 0
   {
     iWait--;             // iWait = iWait - 1
   }

   MOTCTL |= 0x04;
```