Aufbau des Speichers

16 Adressen f. Unterprog.	0xFFE0-0xFFFF	Interrupt Vektoren
Wird i.d.R. einmal vor Inbetriebnahme beschrieben, kann jedoch in 512 Byte Bänken während Betr. verändert werden.	0x1100-0xFFDF	ca. 60 kByte Flash-ROM für Firmware, Programme, Daten, Tabellen Bei ausreichend Spannung auch innerhalb des Programmes schreibbar!
Zwei kl. Bänke	0x1000-0x10FF	2x128 Byte Flash-ROM
f. Programm. via Scatt.Fl.	0x0A00-0x0FFF	Boot-Loader ROM (fix)
Nur 2kB schnelles RAM	0x0200-0x09FF	RAM (f. Variablen, Stack)
Kein echter Speicher hinter diesen Adressen, sondern Verbindung mit der „Aussenwelt“ (memory-mapped)	0x0100-0x01FF	16-Bit Peripherie (Memory mapped) — Nur wort-weise (16 Bit) lesen
	0x0000-0x00FF	8-Bit Peripherie (Memory mapped) — Nur byte-weise (8 Bit) lesen

Die Register des MSP430

Der MSP430 hat 16 Register R0-R15. Davon haben die ersten vier eine festgelegte Bedeutung.

Bemerkung: Andere Assembler erlauben die Verwendung der Bezeichnungen PC, SP, SR für die speziellen Register. Bei den GNU msp430 tools müssen jedoch die (richtigen) Registernamen verwendet werden. Natürlich kann sich jeder selbst eine Ersetzung definieren (dann aber aufpassen, dass durch die Ersetzung keine Seiteneffekte entstehen!).

R0: Program Counter, kurz PC. Schreibt man einen Wert in R0, wird dieser vom Prozessor als die Adresse interpretiert, ab der der nächten auszuführende Befehl steht. Ein Laden von R0 entspricht einem Sprung an die entsp. Adresse. Grundregel: Der PC ist immer gerade! Manche Befehle belegen im Speicher aber auch n*2 Byte. Daher ist nicht jede gerade Adresse zwingend ein gültiger Befehl.

R1: Der Stackpointer, kurz SP. Er ist ebenfalls immer gerade. Das LSB (least significant Bit) ist gar nicht implementiert. Der Stack wird mit seiner höchsten Adresse initialisiert und wächst nach unten. Der Stack ist nie vollkommen unter der Kontrolle des Hauptprogrammes. Interrupt Routinen können ihn jederzeit verändern, müssen die Änderungen aber vor der Rückkehr revidieren.

Die Register des MSP430

R2: Status Register, kurz SR. Jedes Bit hat hier eine eigene, unabhängige Bedeutung.

Bit 9-15: Reserviert für zukünftige Adress-erweiterungen

- Overflow
- Low Power Mode Bit 1
- Low Power Mode Bit 0
- Oszillator schalten (aus=1, ein=0)
- CPU schalten (ein=1)
- General Interrupt Enable (ein=1)
- Negativ Flag
- Zero Flag
- Carry Flag

15	14	13	12	11	10	9	V	SCG 1	SCG 0	OSC OFF	CPU OFF	GIE	N	Z	C
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Die Register des MSP430

R3: Konstantes Register. Auslesen des Registers resultiert immer in einer Null. Schreiben ist möglich, hat aber keinen Effekt (wie /dev/null unter Linux).

Das konstante Register wird vor allem intern verwendet, um die Konstanten #-1, #0, #1, #2, #4 und #8 zu erzeugen. Warum?

Eine unmittelbare Kontante wie #61 muss nach dem Befehlswort im Speicher abgelegt werden. Die sechs Konstanten oben werden jedoch sehr häufig in Programmen verwendet. Sie finden in Form von Flags noch im eigentlichen Befehlswort Platz. Folge: Kürzere Programme und schnellere Ausführung, da ein Zugriff weniger auf den Speicher erfolgt.

Adressierungsarten

Unterschiedliche Adressierungsarten erlauben den Speicher in unterschiedlicher Art und Weise zu lesen und zu schreiben. Klassische RISC Prozessoren haben häufig nur zwei explizite Adressierungsbefehle LOAD und STORE. Alle anderen Operationen können dabei nur auf Register zugreifen.

Beim MSP430 können alle Adressierungsarten mit allen Befehlen verknüpft werden.

Konstanten

#0x1F00	Hexadezimal (genauer eigentlich Sedezimal)
#49152	Dezimal

Konstanten können natürlich nur Quelloperanden, niemals aber Zieloperanden sein

Beispiel: mov #1234, R7 // Lade die Zahl 1234 in Register R7

Adressierung rel. zum Program Counter (PC)

nnnn	Adressiert die „Speicherzelle“ PC+nnnn Dies ist die std. Adressierung des MSP430. Ein Abzählen der Adresse nnnn rel. zum PC ist jed. nicht nötig. Dies erledigt der Assembler.

Adressierungsarten

Absolute Adressierung

`&nnnn`	Adressiert die Speicherzelle mit der Hausnummer nnnn
Beispiel:	`mov.b #7, &0029` // Adressiert Speicherzelle

Register (direkt)

Rn	Als Operand dient Register n.
Beispiel:	`mov R5, R6` // Läd Register 6 mit Inhalt von Register 5

Register (indirekt)

@Rn	Register n enthält die „Hausnummer“ der Speicherzelle, deren Inhalt adressiert wird
Beispiel:	`mov #FFFE, R15` // Läd Register 15 mit Adresse FFFE `mov #2100, @R15` // Schreib den Wert 2100 in (ab) FFFE

Adressierungsarten

Register (indexiert)

```
offset(Rn)      Register n enthält die „Hausnummer“ + offset der
                Speicherzelle, deren Inhalt  adressiert wird

Beispiel:  mov #FFFF, R15      // Läd Reg. 15 mit Adresse (bzw. Zahl)
                               // FFFF
Beispiel:  mov #0123, -1(R15)  // Schreibt 123 ab Adresse FFFE
```

Register indirekt mit Postinkrement

```
@Rn+            // Adressiert Register Rn und inkrementiert
                // es sofort nach der Adressierung um 2

Beispiel:  mov #1234, &0200    // Schreibt 1234 ab Speicherzelle 200
Beispiel:  mov #0200, R7       // Läd Register 7 mit Wert (Adresse) 200
           mov @R7+, R1        // Läd Register R1 mit 1234
                               // danach steht 202 in R7
```

Was tut folgender Ausdruck: add @R15+, -2(R15)

(Bedingte) Sprünge

Insbesondere bedingte Sprünge finden nach Vergleichs- oder Rechenoperationen statt, die ihre Spuren in den Flags des Statusregisters hinterlassen. Abhängig von diesen Flags können folgende bedingte Sprünge durchgeführt werden.

Hinweis zum besseren Verständnis: Ein Vergleich **cmp src, dest** wird als Subtraktion von **dest-src** implementiert.

JNE/JNZ	„Jump Not Zero“	Jump if Z==0 (if src != dest)
JEQ/JZ	„Jump Zero“	Jump if Z==1 (if src == dest)
JN	„Jump Negative“	Jump if N==1 (if dest < src)

stimmt nur, falls kein Überlauf

Vorsicht vor Überlauf: cmp.b 1, -127

geprüft wird -127-1=-128
-128 ist bereits außerhalb des Wertebereichs. Gesetzt wird daher nur das Carry Bit! Das Negativ-Bit, das man eigentlich gern erhalten hätte, wird gewissermaßen durch das unerlaubte Bit der -128 überschrieben

JL	„Jump Less than“	Jump if N==1 xor C==1 (if dest < src)

Auch hier kann wegen des Überlaufs das Ergebnis der Subtraktion falsch sein, jedoch wird in Abhängigkeit von den Bits wenigstens richtig gesprungen

(Bedingte) Sprünge

JNC	„Jump Not Carry“	Jump if C==0 (if (op1+op2)<=0xFFFF)
JC	„Jump Carry“	Jump if C==1
JGE	„Jump if Greater or Equal“	Jump if (N xor V) == 0 (if dest > src)

Hinweis: Analog zur Überlegung von JL vs. JN könnte man einen „Jump Greater or Equal“ einfach als „Jump Not Negativ“ implementieren. Jedoch entsteht hier das gleiche Problem mit dem Übertrag bei JN mit großen Zahlen. Dieses wird durch die Hinzunahme des Carry-Bits gelöst. Nur wenn beide gesetzt sind gilt „greater or equal“.

JMP	„Jump“	Jump in any case

Zwei Operanden Befehle

Hinweis: ~ ist der bitweise NOT operator (so wie z. B. in C definiert)
alle Befehle können Byte „.b“ oder Wörter „.w“ als Argumente besitzen. „.w“ ist optional und kann weggelassen werden.

```
mov.b &0100, &0200 // Inhalt Speicherzelle &200 = &100
mov.w &0100, &0200// &200 = &100 und &201 = &101
```

mov src, dest	dest = src
add src, dest	dest = dest + src
addc src, dest	dest = dest + src + C (für > 16 Bit Berechnungen)
sub src, dest	dest = dest – src
subc src, dest	dest = dest - src + C
cmp src, dest	dest – src (wie Subtraktion, jed. ohne Speicherung)
dadd src, dest	dest = dest + src + C (Binary Coded Decimal)
bit src, dest	dest & src (jed. ohne Speicherung)
bic src, dest	dest = dest & ~src
bis src, dest	dest = dest | src
xor src, dest	dest = dest ^ src
and src, dest	dest = dest & src

Ein Operanden Befehle

Zur Erinnerung: Das 2er Komplement

+40 in binärer Darstellung (S = sign)

0	0	1	0	1	0	0	0
S	64	32	16	8	4	2	1

(a) Bitweise Invertierung

1	1	0	1	0	1	1	1
S	64	32	16	8	4	2	1

(b) +1 (= -40 als 2er Komplement)

1	1	0	1	1	0	0	0
S	64	32	16	8	4	2	1

Ein Operanden Befehle

rrc.b oder rrc.w	C->BIT(n)->BIT(n-1)->BIT(1)->C d. h. Rotation eine 9 bzw. 17 Bit Operanden nach rechts
swapb	Vertausche höherw. und niederw. Byte. Nur sinnvoll für Wort-Operanden
rra.b oder rra.w	Arithmetischer Rechtsschift, d. h. Division durch 2 BIT(n)->BIT(n) und BIT(n)->BIT(n-1)->BIT(1)->C

Warum wird BIT(n) sowohl gerettet, als auch kopiert? Weil sonst das 2er Komplement nicht erhalten wird. Beispiel:

Ein Operanden Befehle

rra.b oder rra.w — Arithmetischer Rechtsschift, d. h. Division durch 2
BIT(n)->BIT(n) und BIT(n)->BIT(n-1)->BIT(1)->C

Warum wird BIT(n) sowohl erhalten, als auch kopiert?
Damit das 2er Komplement richtig behandelt wird. Beispiel:

+40 in binärer Darstellung (S = sign)

0	0	1	0	1	0	0	0
S	64	32	16	8	4	2	1

rra.w (ergibt 20)

0	0	0	1	0	1	0	0
S	64	32	16	8	4	2	1

-40 (als 2er Komplement)

1	1	0	1	1	0	0	0
S	64	32	16	8	4	2	1

rra.w (ergibt -20)

1	1	1	0	1	1	0	0
S	64	32	16	8	4	2	1

Ein Operanden Befehle

push operand

Schiebt Operand auf den Stack und dekrementiert SP mit 2 (auch bei push.b! SP ist immer gerade.)

Beispiel:

```
push R5          // kann ersetzt werden durch
sub  2, SP       // SP zeigt immer auf das zuletzt auf den
                 // Stack geschobene Datum. Hier soll jedoch
                 // gleich ein neues gespeichert werden
mov  R5, @SP
```

call dst

Springt ein Unterprogramm an, dessen Adresse in dst gespeichert ist. VORSICHT: Es wird nicht zu dst selbst gesprungen. Die absolute Adressierung wäre &dst. Unterschied zu jmp: Ein komfortabler Rücksprung erfolgt mit ret.

Beispiel:

```
call dst         // kann ersetzt werden durch
sub 2, SP        // gleich soll die Rücksp.Adr. gerettet werden
mov PC, @SP      //
mov dst, PC      // der Sprung erfolgt durch direkte
                 // Manipulation des Program Counters PC
```

Ein Operanden Befehle

ret — So werden Unterprogramme beendet, die mit call aufgerufen wurden. Es erfolgt ein Rücksprung zur nächsten Adresse nach dem call.

Beispiel:
```
ret                 // kann emuliert werden durch

mov PC, @SP         // getrette Adresse vom Stack holen
add 2, SP           // Adresse wieder „vergessen“

mov @SP+, PC        // das gleiche in nur einem Ausdruck
```

reti — Return from Interrupt.

Beispiel:
```
reti                // wie ret, zusätzlich wird jed. das SR gerettet

mov SR, @SP         // gerettete Flags vom Stack holen
add 2, SP           // Adresse wieder „vergessen“

mov PC, @SP         // gerettete Adresse vom Stack holen
add 2, SP           // Adresse wieder „vergessen“
```

sxt operator — Erweitert 8 Bit Wort auf 16 Bit. Vorzeichen wird dabei erhlaten! Macht nur in der .w Version Sinn.

16 Bit Multiplikationen

Multiplikationen sind im Kern des MSP430 nicht vorgesehen. Jedoch existiert ein Hardware Multiplizierer, der über den Memory Mapped Speicherbereich für 16 Bit Peripherie (0x100-0x1FF) wie jedes andere externe Gerät (z. B. die Leuchtdioden) adressiert wird. TI bezeichnet die 4 Arten von Multiplikationen mit MPY, MPYS, MAC und MACS.

MPY: 1. Operand unsigned Multiplikation	0x130
MPYS: 1. Operand signed Multipliation	0x132
MAC: 1. Operand unsigned Mult. u. Add.	0x134
MACS: 1. Operand signed Mult. u. Add.	0x136
2. Operand signed/unsigned	0x138

RESLO: 16 LSW des Ergebnisses	0x13A
RESHI: 16 MSW des Ergebnisses	0x13C
SUMMEXT: Sum Extension	0x13E

16 Bit Multiplikationen

Aufrufen der Multiplikation aus dem Programm heraus:

(1) Auswahl des Speicherbereiches für den ersten 16 Bit Operanden und laden des Speicherbereiches mit dem 1. Operanden. Der Speicherbereich wird auch als Register des Multiplizierers bezeichnet.

```
mov #0xFFFF, &0x130      // damit ist die vorzeichenlose Multiplikation
                            gewählt
```

(2) Für alle Typen von Multiplikationen gilt: 0x138 ist das Register für den 2. Operanden. Im Moment des Schreibens in 0x138 wird der Multiplizierer (automatisch) gestartet. Das Schreiben triggert die Ausführung. Dabei bleibt der Prozessor unbelastet.

```
mov #0xFFFF, &0x138      // 0xFFFF ist der zweite Operand
```

(3) Die ersten 16 Bit des Ergebnisses stehen in den Adressen ab 0x13A (also 0x13A und 0x13B), die zweiten 16 Bit ab 0x13C (also 0x13C und 0x013D). SUMEXT enthält ein Carry bei unsigned Operationen. Bei unsigned Operationen enthält es das Vorzeichen.

Anm.: Das Ergebnis steht nach 2 Taktzyklen zur Verfügung. Bei kurzen Befehlen nach der Rechnung muss daher evtl. ein „nop“ eingefügt werden (oder man macht etwas anderes sinnvolles).
Hinweis: Vorsicht vor Interrupts!

MSP430 Interrupts

Klassische proaktive Programmierung:

Die gesamte Ablaufsteuerung liegt vollständig in der Verantwortung des Hauptprogrammes.

```
void main(...) // wird nach Programmstart zuerst angesprungen
{
    while(endless) // Läuft kontinuierlich bis Programmende
    {
        if hardware_event    then HandleHardwareEvent(...);
        if idle              then Sleep(100ms);
    } // end while
} // end main
```

Ausstiegspunkt

Wiedereinstiegspunkt

Es ist guter Stil den Prozessor nicht die gesamte Zeit mit der while(endless)-Schleife zu blockieren. Dies verbraucht in jedem Fall 100% der Prozessorleistung, egal wie schnell dieser ist.

Im Beispiel entschärft die Sleep()-Anweisung diesen Sachverhalt, wenn kein Event vorliegt. Der Prozess wird vom Scheduler des OS dann für 100ms nicht beachtet und es können andere Prozesse die Rechenressourcen erhalten. Dadurch verschlechtert sich die Reaktionszeit aber auch auf 50ms im Mittel.

MSP430 Interrupts

Reaktive Programmierung:

Das erwarten die meisten modernen Betriebssysteme:

```
void HandleHardwareEvent(...)
{
    ...
} // end HandleHardwareEvent

void main(...) // wird nach Programmstart zuerst angesprungen
{
    InitAllYouNeed(...);

    OperatingSystem.RegisterHWEventHandler(HandleHardwarEvent);
} // end main
```

Nach dem Programmstart kann die Anwendung alles nötige initialisieren. Ansonsten definiert das Programm Funktionen für Ereignisse, die es abarbeiten möchte. Diese Funktionen werden beim Betriebssystem angemeldet. Tritt später ein Ereignis tatsächlich ein, so ruft das BS die zuvor registrierte Programmfunktion auf. Das Programm ist damit gewissermaßen der Dienstleister des Betriebssystems. Grundsätzlich funktioniert die Programmierung des MSP430 reaktiv. Wenn nichts geschieht, schaltet sich der Prozessor so schnell wie möglich ab, um Strom zu sparen.

MSP430 Interrupts (Watchdog)

Das Programm kann sich von einer Reihe von Sensor-Zuständen über die Comparatoren wecken lassen, wenn z. B. bestimmte Pegel (z. B. Helligkeit) ein Soll überschritten haben.

Der Watchdog hat eine Sonderstellung, innerhalb der Interrupt-Logik. Er soll überwachen, ob sich das Programm aufgehängt hat. Nach dem Reset des Prozessors ist er aktiv, kann jedoch im Programm deaktiviert werden.

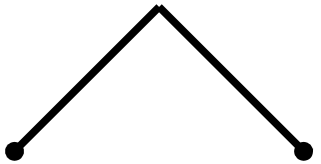
Der WG wartet eine vorgegebene maximale Anzahl von Taktzyklen. Wenn er bis dahin nicht zurückgesetzt wurde, können abhängig vom Modus des WD zwei Dinge passieren. Er kann einen Interrupt oder einen Reset auslösen. Damit soll verhindert werden, dass selten auftretende Probleme zum Totalausfall des Systems führen.

Der WD arbeitet ähnlich wie die automatische Notbremse in Zügen, die das Einschlafen des Fahrers verhindern sollen.

MSP430 Interrupts (Watchdog)

16-Bit Word-Register ab Adresse 0x0120 steuert den Watchdog.

Wird der Watchdog aktiv, so geschieht nichts anderes, als das Auslösen eines Interrupts.



WD Timer
(Vec 0xFFF4)

WG HW Interrupt
(Vec 0xFFFE)

Password
(wird vergl. mit 0x5A)

WDTHOLD (0=aktiv / 1=inaktiv)
WDTNMIES
WDTNMI
WDTTMSEL (0=HW / 1=Timer)
WDTCNTCL (1 = Counter reset)
WDTSSEL
Timer Einstellungen
WDTTIS1
WDTTIS0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0