9. Beschreibung des Prozessors MSP 430

9.1 Die Register des MSP430

Der MSP430 hat 16 Register, bezeichnet mit R0 - R15. Davon haben die ersten vier eine spezielle Bedeutung.

Bemerkung:
Andere Assembler erlauben die Verwendung der Bezeichnungen PC, SP, SR für solche speziellen Register. Bei den GNU msp430-tools müssen jedoch die Registernamen R0 bis R3 verwendet werden.

Der Befehlszähler

R0: **Program Counter**, kurz PC. Schreibt man einen Wert in R0, wird dieser vom Prozessor als die Adresse interpretiert, an der der nächste auszuführende Befehl steht. Ein Laden von R0 entspricht einem nachfolgenden Sprung an die entsprechende Adresse.

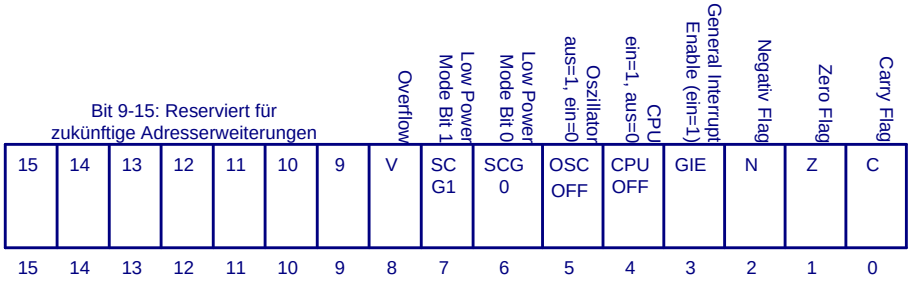
Grundregel: Der PC ist immer gerade! Manche Befehle belegen im Speicher aber auch n*2 byte. Daher ist nicht jede gerade Adresse zwingend ein gültiger Befehl.

Der Stackpointer

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 steht nicht ausschließlich unter der Kontrolle des Programmes, Inter-rupt-Routinen können ihn jederzeit verändern, müssen die Änderungen aber vor der Rückkehr revidieren.

Das Statusregister

R2: **Status Register**, kurz SR. Jedes Bit hat hier eine spezielle Bedeutung.



Das Konstantenregister

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

Das Konstantenregister wird vor allem verwendet, um die Konstanten #-1, #0, #1, #2, #4 und #8 effizient bereit zu stellen.

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 bei Verwendung des Pseudo-Registers R3 in Form von Flags noch im eigentlichen Befehls-wort Platz, erfordern also keinen extra Speicherzugriff..

Folge: Kürzere Programme und schnellere Ausführung des Befehls

9.2 Aufbau des Speichers

0xFFE0-0xFFFF	16 Adressen für Unterprogramme	Interrupt-Vektoren
0x1100-0xFFDF	Wird in der Regel einmal vor Inbetriebnahme des Rechners geschrieben, kann jedoch auch in 512-Byte-Bänken während des Betriebes verändert werden.	ca. 60 kByte Flash-ROM für Firmware, Programme, Daten, Tabellen. Bei ausreichend Spannung auch vom Programm aus schreibbar!
0x1000-0x10FF	Zwei kleine Bänke	2x128 Byte Flash-ROM
0x0A00-0x0FFF	für Programme via Scatter Flasher	Boot-Loader ROM (fix)
0x0200-0x09FF	Nur 2kB schnelles RAM	RAM (für Variable, Stack)
0x0100-0x01FF	kein echter Speicher hinter diesen Adressen, sondern Verbindung mit der „Aussenwelt“ (memory-mapped device buffers)	16-Bit-Peripherie (memory mapped). Nur wortweise (16 Bit) lesen!
0x0000-0x00FF		8-Bit-Peripherie (memory mapped) Nur byteweise lesen!

9.3 MSP430 Interrupts

Zum Vergleich: 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
```

Interrupts als Alternative zum “busy waiting”

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

Im Beispiel entschärft die Anweisung `sleep(100ms)`diesen Sachverhalt, wenn kein Ereignis vorliegt. Der Prozess wird vom Scheduler des Betriebs-systems dann für 100 ms nicht beachtet, und es können andere Prozesse die Rechenressourcen erhalten. Dadurch verschlechtert sich die Reaktions-zeit auf das Ereignis aber auch auf 50 ms im Mittel.

Reaktive Programmierung (1)

Das erwarten viele moderne Betriebssysteme:

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

void main(...) // wird nach Programmstart zuerst angesprungen
{
    InitAllYouNeed(...);
    OperatingSystem.RegisterHWEventHandler(HandleHardwarEvent)
;
} // end main
```

Reaktive Programmierung (2)

Nach dem Programmstart kann die Anwendung alles nötige initialisieren. Ansonsten definiert das Programm Funktionen für Ereignisse, die es ab-arbeiten möchte. Diese Funktionen werden beim Betriebssystem ange-meldet. Tritt später ein Ereignis tatsächlich ein, so ruft das Betriebssystem die zuvor registrierte Programmfunktion auf.

Grundsätzlich funktioniert die Programmierung des MSP430 reaktiv. Wenn nichts geschieht, schaltet sich der Prozessor so schnell wie möglich ab, um Strom zu sparen.

9.4 Der Watchdog

Das Programm kann sich von einer Reihe von Sensor-Zuständen über so genannte **Comparatoren** wecken lassen, wenn z. B. bestimmte Pegel einen Sollwert 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 vom Programm aus auch deaktiviert werden.

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

Der Watchdog arbeitet ähnlich wie der “Totmannknopf” in Zügen, der das Einschlafen des Fahrers verhindern soll.

Steuerung des Watchdog

Das 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 xFFFE)