Der STACK Teil 1 - Funktion und Nutzen
==========

Seine Existenz kennen die meisten. Wozu er im System dient ist wohl auch den meisten bekannt. Er speichert bei Unterprogrammaufrufen oder Interrupts die Rückkehradresse ins Hauptprogramm. Das war's dann aber oft schon. Die Befehle PUSH und POP sind vielleicht noch bekannt. Aber das Ding wird vom System gebraucht und man packt es erst mal nicht an.

Das der STACK aber auch in Anwenderprogrammen recht universell verwendbar ist, wissen ein Grossteil der Anwender meist nicht. Man setzt den Stackpointer im Programmheader nach Vorschrift und das war es. Hier möchte ich helfen, das Ding und seine Verwendung zu verstehen.

Was ist der STACK. Zuerst mal ein Speicherbereich im SRAM. Für seine Verwaltung braucht er einen STACK-POINTER, der ein Statusregister ist. Nach Vorschrift wird dieser Stackpointer auf das Ende des SRAM gesetzt. SPL = Low(RAMEND). Wird jetzt von irgendwas ein Wert auf dem STACK abgelegt, wird dieser STACK-POINTER pro Byte um eins vermindert. Holt irgendwas einen Wert vom STACK herunter, wird der STACK-POINTER um eins erhöht. Das passiert alles automatisch. Somit zeigt der STACK-POINTER immer auf ein noch nicht verwendetes Byte im SRAM. OK, das dürfte bekannt und verstanden sein. Wird in den Dokumentationen von Atmel auch immer recht verständlich so beschrieben. Nur hat mancher Seiteneinsteiger halt seine Probleme mit dem englichen Text.

Der STACK ist ein LIFO-Speicher. LIFO steht für "Last In, First Out" deutsch: "Zuletzt rein, Zuerst raus". Also kann man von ihm immer nur den Wert runter holen, der gerade zuvor geschrieben wurde. Er ist somit, wie der Name sagt, ein Stapelspeicher. Von einem Stapel kann man immer nur das oberste Teil sicher herunter nehmen. Zieht man was aus der Mitte, besteht die Gefahr, dass der ganze Stapel zusammen kracht.

Krachen kann er auch wenn man zuviel stapelt. Die Anzahl Bytes im SRAM ist ja begrenzt. Schreibt man z.B. ein Unterprogramm das sich immer wieder selbst per Call aufruft, läuft der Stapel irgendwann über. Rekursive Programmierung nennt man sowas.

```
Main:
        ...
        CALL CRASH
        ...

END

CRASH:
        ...
        CALL CRASH
        ...
RET
```

Also muss man etwas vorsichtig bei seiner Verwendung sein. Hinzu kommt folgendes: Man nutzt das SRAM ja ggf. auch selbst. Und der STACK könnte irgendwann in diesen Variablenbereich herein laufen und Werte überschreiben.

Verdeutlicht man sich das Prinzip des STACK erkennt man folgendes. Der STACK "wächt" vom oberen Ende des SRAM nach unten. Daher sollte man seine SRAM-Variablen immer vom unteren Ende nach oben

platzieren. Der Bereich dazwischen kann dann vom STACK genutzt werden. Man "reserviert Platz für den STACK". Wieviel Platz aber sollte berücksichtigt werden?

Gehen wir mal vom schlimmsten Fall aus. Mit ein paar Tricks kann man es schaffen alle Interruptroutinen eines 90S gleichzeitig aktiv zu setzen. Das wären beim 90S8434 17 Stück.
Jede Routine belegt für die Rückkehradresse 2 Byte. Macht 34 Byte. Hinzu rechnen muss man jetzt noch den Bedarf für eigene Unterprogramme. Setzen wir hier mal eine Schachtelungstiefe von 10 an. (Ein Unterprogramm ruft ein anderes auf. Bis zu 10 UP können hintereinander aufgerufen sein.) Das macht nochmal 20 Bytes. Und letztlich möchte man selbst 20 Bytes auf dem STACK zwischenspeichern. Diese WWV (Wilde wissenschaftliche Vermutung) bringt einen STACK-Bedarf von 34 + 20 + 20 = 75 Bytes. Mit 100 reservierten Bytes liegt man also schon weit über dem zu erwartenden Maximalbedarf. Der Rest kann frei verbraten werden.

Gut. Wir wissen jetzt wie wir den STACK einschätzen müssen. Aber wozu kann man ihn im Programm selbst brauchen ? Schauen wir mal auf folgenden Codeabriss:

```
MAIN:
        ...
        TST R0          ; Während dieser Bearbeitung tritt Int_Timer_0 auf !!!
        BREQ Dorthin
Hierhin:
        ...
Dorthin:
        ...
END

Int_Timer_0:
        ...
        CMP R16, $23
        BREQ Woandershin
Hierbleiben:
        ...
Woandershin:
        ...
RETI
```

Nun, was passiert. TST R0 wird sauber zu Ende ausgeführt und das Statusregister SREG entsprechend gesetzt. Jetzt kommt Int_Timer_0 daher und testet selbst irgendwas. SREG wird durch diesen Test überschrieben. Nach Rückkehr ins Hauptprogramm ist das Ergebis des TST R0 verloren. Das Programm verzweigt möglicherweise falsch. An solch einem Fehler kann man verzweifeln. 999 mal geht alles gut, dann plötzlich läuft das Programm falsch.

Man muss also bei Interruptroutinen IMMER dafür sorgen, dass das Statusregister gesichert wird. OK, ganz einfach, so geht's:

```
Int_Timer_0:
        MOV R17, SREG
        ...
        CMP R16, $23
        BREQ Woandershin
```

```
Hierbleiben:
        ...
Woandershin:
        ...
        MOV SREG, R17
RETI
```

Aber was ist wenn innerhalb eines INT ein weiterer auftritt. Nimmt man dort wieder R17 als Zwischenspeicher können sich wieder Fehler einschleichen. Also für jede INT-Routine ein eigenes Register. Gut, geht auch. Aber ist sicher nicht der schönste Weg. Hier bietet sich der STACK geradezu an.

```
Int_Timer_0:
        PUSH Temp
        MOV Temp, SREG
        PUSH Temp
        ...
        CMP R16, $23
        BREQ Woandershin
Hierbleiben:
        ...
Woandershin:
        ...
        POP Temp
        MOV SREG, Temp
        POP Temp
RETI
```

Was mache ich da ? Zuerst mal schiebe ich Temp auf den STACK, da ich nicht sicher sein kann das Temp gerade von keinem anderen Codeabschnitt verwendet wird. Dann hole ich SREG nach Temp und PUSHe den ebenfalls auf den STACK. Direkt PUSH SREG wäre schöner, geht aber nicht. PUSHen kann man nur R0 bis R31. Am Ende des INT stelle ich nun SREG und TEMP wieder im ursprünglichen Zustand her.

Wenn jede INT-Routine diese 6 Programmworte beinhaltet kann nichts mehr passieren. Egal wann und wieviel INT's auftreten. Zudem kann im INT mit Hilfsregistern ähnlich verfahren werden. Jedes temporär verwendete Register vorher PUSHen, nachher POPen. Macht die Routine für andere Programme recht flexibel. Sch..egal ob das Register im Hauptprogramm anderweitig genutzt wird.

Nicht vergessen darf man hierbei aber, diesen Zusatzbedarf bei der Berechnung des STACK-Bedarfs zu berücksichtigen. Ansonsten läuft man wieder gefahr, den STACK oder SRAM-Variablen zu zerstören.