Programmierkonventionen der elementaren Mikrocontroller-Programmierung

Stand: 7. 1. 03

Speicherung der Variablen:

a) in Registern,
b) im Arbeitsspeicher.

Bei Assemblerprogrammierung Variable genauso definieren wie in einer höheren Programmiersprache.

- Ordnung muß sein -

Sprachmittel:
DSEG für Speichernutzung, DEF für Registernutzung.

Genügend Arbeitsregister frei lassen!

Programmiermodelle:

1. alle Variable passen in die Register,
2. Variable in Register und Arbeitsspeicher. Naheliegender Ansatz, aber naiv und unschön, da Variable auf zweierlei Art zugänglich.
3. alle Variable im Arbeitsspeicher. Register sind nur Arbeitsregister. Läuft letzten Endes auf eine speicherorientierte Privatbefehlsliste hinaus. Einfachste Implementierung: über Makrodefinitionen. Ehrgeiziger: Emulation. Verschwendet Speicherplatz und Verarbeitungsleistung.
4. Aufteilung in globale und lokale Variable (enntsprechend der Anwendung, z. B. je Task oder auch je Unterprogramm). Register können alle globalen und einen Satz lokaler Variablen aufnehmen.
   a) nicht benötigte Variable auf den Stack legen und ggf. zurückhohlen (PUSH-POP),
   b) Registerbelegungen in fest zugeordnete Arbeitsspeicherbereiche auslagern (Swapping). Z. B. je Task einen solchen Bereich (Task Control Area). wird Task aktiv, den Registersatz der verdrängten (= zuvor aktiv gewesenen) Task auslagern (Swap Out) und den der aktuellen Task einlagern (Swap In).
5. globale und lokale Variable in Stack Frames im Arbeitsspeicher (vgl. Unix/C). Dann besser gleich eine höhere Programmiersprache nehmen und das Denken den Pferden (= den Compiler-Autoren) überlassen. Oder einen dickeren Controller (16/32 Bits) nehmen.

Vorteil der Emulation: erlaubt Befehlsausführung aus Arbeitsspeicher und hinreichende "Instrumentierung" zum Debuggen. Probleme:

- Geschwindigkeitsminderung (Emulation braucht zwischen 5 und 50 Befehle je zu emulierendem Befehl. Durchschnittliche Verminderung der rohen Verarbeitungsleistung auf 1/10 (falls man es geschickt anstellt...).

- zwei Entwicklungsaufgaben: (1) Befehlsliste, (2) Anwendungsproblem,
- entwicklungsseitige Unterstützung der neuen Befehlsliste.

Parameterübergabe an Unterprogramme:

1. in Registern (vgl. PC-BIOS),
2. in festen Speicherbereichen,
3. im Anschluß an den Befehl (nur Direktwerte),
4. im Stack (vgl. C/Unix).

3. und 4. setzen voraus, daß man an den Stackinhalt programmseitig herankommt (SP-relative Adressierung). Beim PIC 16/17 unmöglich, beim Atmel schwierig. Lösungen:

- SP über E-A-Zugriffe in Register X, Y oder Z holen,
- zwei Stacks: einen Rettungs-Stack mit SP und einen Parameterübergabe-Stack z. B. mit Y (die von den Compiler-Schreibern bevorzugte Lösung).

Rückgabe von Ergebnissen:

1. in Registern (vgl. PC-BIOS),
2. in festen Speicherbereichen,
3. im Stack (vgl. C/Unix).

Rückgabe von Ablaufinformationen (z. B. Unterscheidung o.k./Fehler):

1. in Register (vgl. PC-BIOS,
2. in den Flagbits (komfortabel, weii so leicht verzweigt werden kann),
3. im Stack (Fehlercode = Funktionswert (eine typische C-Programmiergepflogenheit)),
4. durch Änderung der Rückkehradresse (sehr komfortabel, weil es Abfragen grundsätzlich erspart). Z. B. wenn Fehler, Rückkehr auf Folgeadresse, wenn o.k., Rückkehr auf übernächste Adresse. Ermöglicht es, so zu programmieren:

```
CALL
JMP Error_handler
... Fortsetzung...
```

Grundlagen der Stackorganisation und -adressierung

Das Stack- (Kellerspeicher-) Prinzip ist in der Informatik von grundsätzlicher Bedeutung, namentlich was die Programmiersprachen, die Compiler und die Systemsoftware angeht. Manche Architekturen haben Vorkehrungen, um Stacks zu unterstützen, manche nicht (dann müssen Stacks als normale Datenbereiche vorgesehen werden, deren Verwaltung mit elementaren Befehlen auszuprogrammieren ist). Die Grundprinzipien bleiben stets die gleichen.

Grundlagen
Ein Stack ist eine Speicheranordnung, die eine gewisse Anzahl gleich langer Informationsstrukturen (*Stack-Elemente*) aufnehmen kann. Es gibt keinen wahlfreien Zugriff, sondern die Speicheranordnung wird implizit von einem Adreßzähler (*Stackpointer SP*) adressiert.

Stackzugriffe
Es gibt nur zwei grundlegende Zugriffsabläufe:

- ein *Push*-Ablauf legt ein Element auf den Stack,
- ein *Pop*-Ablauf entnimmt das zuletzt (vom letzten Push) auf den Stack gelegte Element (beim nächsten Pop wird dann das vom vorletzten Push abgelegte Element entnommen usw.).

Die Stack-Organisation wird deshalb gelegentlich auch als LIFO (Last In, First Out) bezeichnet.

Wachstumsrichtung
Es ist eine reine Konventionsfrage, ob bei Push-Abläufen der Inhalt des Stackpointers erhöht und bei Pop-Abläufen vermindert wird oder umgekehrt.

In vielen Architekturen *wachsen Stacks immer in Richtung niederer Adressen*, d. h. der Stackpointer zeigt anfänglich immer auf die höchstwertige Adresse. Sein Inhalt wird bei Push-Abläufen vermindert und bei Pop-Abläufen erhöht.

Adreßzähl- und Zugriffsreihenfolge
Ebenso ist es eine reine Konventionsfrage, ob bei einem Push zunächst der Stackpointer verändert und dann das neue Element gespeichert wird oder umgekehrt.

Die typische Auslegung (glit auch für Atmel): Der Stackpointer zeigt *immer auf das oberste Element im Stack* (Top of Stack, TOS), nicht auf die erste freie Stackposition. Bei einem Push wird deshalb der Stackpointer-Inhalt zunächst vermindert (Ausdehnungsrichtung!); dann wird das Element gespeichert. Umgekehrt wird bei einem Pop das Element entnommen und dann der Stackpointer-Inhalt erhöht (Zugriffsprinzip: Predecrement/Postincrement).

Stack-relative Adressierung
Es ist oft von Vorteil, wenn man zu Elementen des Stack auch wahlfrei zugreifen kann. So kann man auch untere Elemente im Stack erreichen, ohne die oberen zuvor entfernen zu müssen. Solche Zugriffe beziehen sich zweckmäßigerweise auf den Stackpointer, so daß das erste, zweite usw. Element im Stack für Lese- und Schreibzugriffe zugänglich ist, wobei der Stackpointer nicht verändert wird (explizite Stackzugriffe nach dem Prinzip Basis + Displacement mit dem Stackpointer als Basisadreßregister). Muß beim Atmel ausprogrammiert werden (SP über E-A-Zugriffe holen oder softwareseitiger Stack, z. B. auf Grundlage des Y-Registers).

Variabel lange Stackelemente
In den meisten Architekturen, so sie überhaupt Stacks vorsehen, sind alle Elemente in einem Stack *gleich lang*. Kürzere Angaben werden zwecks Ablage auf dem Stack entsprechend erweitert.

Stack Frames

Ein Stack Frame ist ein fester Bereich im Stack. Er dient vor allem dazu, die statischen Variablen des laufenden Programms aufzunehmen.

Statische und dynamische Variable

Statische Variable werden im Programmtext deklariert (jeder Variablenname wird angegeben, und es wird ihm ein Datentyp zugewiesen). Beispiel (wir verwenden der Anschaulichkeit halber eine an Pascal und Ada orientierte Syntax):

```
Artikel_Nr: Integer;                    -- 4 Bytes (ganze 32-Bit-Binärzahl)
Bezeichnung: String(64);                -- 64 Bytes (Zeichenkette)
Preis: Unpacked_BCD(16);                -- 16 Bytes (BCD-Zahl)
Länge, Breite, Höhe: Small_Integer;     -- je 2 Bytes (ganze 16-Bit-Binärzahlen)
Gewicht, Spezifisches_Gewicht: Float;   -- je 4 Bytes (32-Bit-Gleitkommazahlen)
usw.
```

Jeder diese Variablen muß der Compiler entsprechenden Speicherplatz zuweisen.

Dynamische Variable entstehen hingegen im Laufe der Verarbeitung (also ohne daß sie der Programmierer ausdrücklich deklarieren muß). Beispiel: der Programmierer schreibt hin:

```
Gewicht := Länge * Breite * Höhe * Spezifisches_Gewicht;
```

Der Compiler muß diese Formel in eine Folge von Maschinenbefehlen umsetzen (hierbei sind u. a. verschiedene Datentypen ineinander zu wandeln). Da die einzelnen Befehle nur ganz elementare Operationen ausführen können, fallen im Verlauf der Rechnung Zwischenergebnisse an. Dies sind die dynamischen Variablen, die typischerweise auf dem Stack abgelegt werden.

Sowohl statische als auch dynamische Variable werden im Stack untergebracht

Das muß nicht unbedingt so sein, hat sich aber bewährt. Und zwar vor allem deshalb, weil man gern Programme in Programme schachtelt (Unterprogrammtechnik). Dann liegt es nahe, die verfügbare Speicherkapazität im Sinne eines Stack zu verwalten und den Speicher vom oberen Ende her aufzufüllen (vgl. weiter unten Abbildung 1.11). Zuerst kommt der Stack Frame des ersten Programms. Darüber (in Richtung zu den niederen Adressen hin) werden die gerade aktuellen dynamischen Variablen auf den Stack gelegt. Wenn nun das Programm ein Unterprogramm aufruft, kommt dessen Stack Frame auf den Stack, darüber werden dessen dynamische Variable abgelegt usw.

Das Zugriffsproblem

Gemäß dem Rechenablauf wächst oder schrumpft der Stack. Andererseits sind aber immer die gleichen statischen Variablen zu adressieren. Würde man sich aber stets auf den Stackpointer (als Basisadresse) beziehen, so würden sich bei jedem Zugriff andere Displacements zu den statischen Variablen ergeben. Deshalb sieht man typischerweise ein weiteres Adreßregister vor, den sog. Frame Pointer oder Base Pointer (Abbildung 1.1).

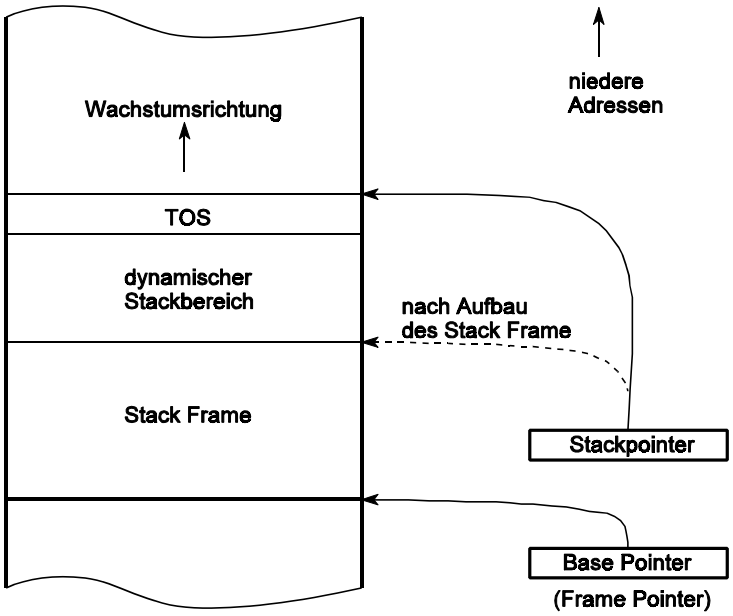

Abbildung 1.1 Stack-Organisation mit Stack Frame

Erklärung:
Der Base Pointer (Frame Pointer) zeigt stets auf den Anfang des aktuellen Stack Frame (d. h. auf das Wort an der jeweils höchsten Adresse). Alle Inhalte des aktuellen Stack Frame sind somit über negative Displacements (bezogen auf den Base Pointer) erreichbar.

Ruft das aktive Programm seinerseits ein Unterprogramm, so wird der aktuelle Inhalt des Stackpointers in den Base Pointer übernommen, und oberhalb des dynamischen Bereichs des rufenden Programms wird der Stack Frame des gerufenen aufgebaut. Spitzfindigkeiten erläuterm wir im folgenden anhand von UNIX .

Grundlagen der systemseitigen Speicherverwaltung
Die Speicherverwaltung hat die Aufgabe, den einzelnen Programmen im Arbeitsspeicher eine angemessene Speicherkapazität zur Verfügung zu stellen. Wieviel Speicher (z. B. in Bytes ausgedrückt) braucht aber ein Programm? - Es sind unterzubringen:

- das Programm selbst,
- die zugehörigen konstanten Daten,
- Arbeits- und Übergabebereiche,
- bedarfsweise Symbol- und Verweistabellen.

In einfachen Systemen kann man die verfügbare Speicherkapazität fest aufteilen (statische Speicheraufteilung). Moderne Hochleistungssysteme sind hingegen dadurch gekennzeichnet, daß sich die Speicherbelegung ständig ändert (dynamische Speicheraufteilung). Abbildung 1.2 veranschaulicht ein Prinzip, das häufig implementiert wird.

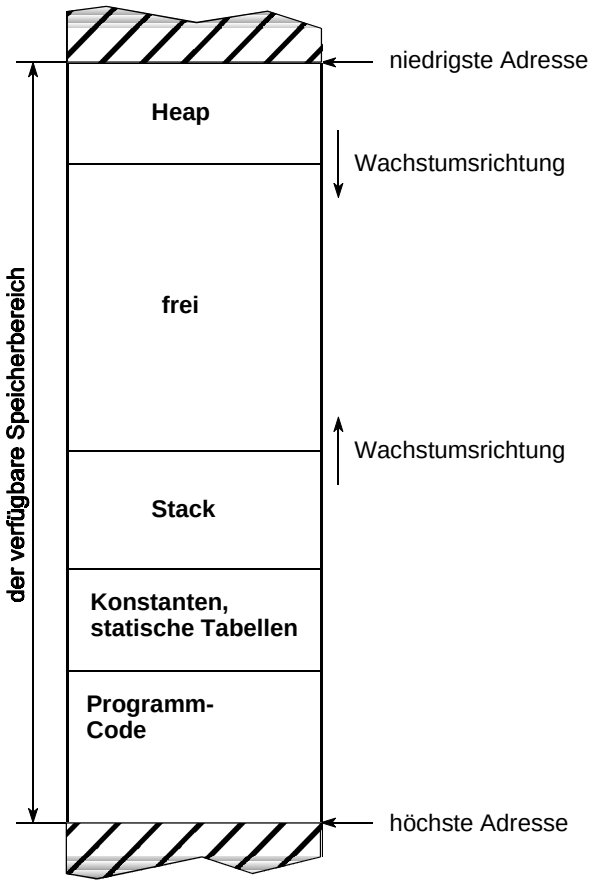

Abbildung 1.2 Zum Prinzip der Speicheraufteilung (Beispiel)

Erklärung:
Wir beginnen damit, daß ein "hinreichend" großer Speicherbereich zunächst bereitsteht. Dieser wird folgendermaßen belegt.

- der Programmcode an sich wird ganz hinten untergebracht,
- davor kommen die "statischen" - in ihrer Größe unveränderlichen Datenbereiche (Konstanten, Symboltabellen usw.),
- im Anschluß daran - zu den niederen Adressen hin - wird der Stack eingerichtet. Er nimmt dynamische Daten, Parameter, statische Variable, Zwischenergebnisse und Rückkehradressen auf. Er wächst in Richtung niederer Adressen.
- ergänzend zum Stack sieht man oft eine weitere veränderliche Struktur vor, den Heap (sprich: Hiep; wörtlich = Haufen). Der Heap wird am Anfang des Speicherbereichs angeordnet. Er wächst in Richtung höherer Adressen. Zur Verwendung von Stack und Heap siehe Tabelle 1.1.

Sowohl Stack als auch Heap wachsen oder schrumpfen während der Ausführung des Programms. Durch die Anordnung an entgegengesetzten Enden ist stets gewährleistet, daß sich ein möglichst großer freier Bereich zwischen Stack und Heap befindet. Nur in dem - vergleichsweise unwahrscheinlichen - Fall, daß beide Strukturen wachsen und wachsen, kann es vorkommen, daß irgendwann einmal nichts mehr frei ist, daß also der Stack versucht, ein Stück des Heap zu belegen oder umgekehrt. Die Schutzvorkehrungen der Hardware bzw. das

Laufzeitsystem der Software sollten dies erkennen und entsprechend reagieren (z. B. mit dem Abbruch der Programmausführung und einer entsprechenden Fehlermeldung). ***Das ist aber nicht immer der Fall!!!***

	Stack	**Heap**
Nutzung (gespeichert werden...)	Rückkehradressen, lokale Daten (verschwinden bei Rückkehr aus der jeweiligen Funktion)	dynamische Daten (bleiben solange erhalten, bis sie explizit (vom Programm) wieder freigegeben werden
Belegung und Freigabe (Auf- und Abbau)	automatisch gemäß dem LIFO-Prinzip	typischerweise (vgl. Programmier-sprache C) vom Programmierer anzufordern und freizugeben
besondere Eignung	für kleinere und einfachere Datenstrukturen (zu beispielsweise 32 oder 64 Bits)	für größere und kompliziertere Datenstrukturen (z. B. von 256 Bytes an aufwärts)

Tabelle 1.1 Zur Verwendung von Stack und Heap

Die UNIX-Stackorganisation
Für jeden Prozeß werden zwei Stacks verwaltet (Abbildung 1.3):

- der User Stack zum Aufrufen von Anwendungsprogrammen,
- der Kernel Stack zum Aufrufen der Systemfunktionen.

Erklärung zu Abb. 1.3:
Ein UNIX-Programaufruf läuft folgendermaßen ab:

1. das rufenden Programm legt die zu übergebenden Parameter auf den Stack,
2. der Aufruf wird ausgeführt. Dabei gelangt die Rückkehradresse auf den Stack*).
3. das gerufenene Programm kopiert den bisherigen Frame Pointer auf den Stack (Adreßzeiger als Rückverweis). Typischerweise wird der aktuelle Inhalt des Stackpointers zum neuen Frame Pointer**).
4. das gerufene Programm kopiert seine lokalen Variablen in den Stack (bzw. schafft auf dem Stack soviel Platz, daß die lokalen Variablen hineinpassen),
5. der aktuelle Frame bzw. Base Pointer wird eingerichtet.

*): erste Variante: automatisch mittels CALL-Befehl (z. B. IA.32). Zweite Variante: porgrammseitig, indem der Inhalt des Adreßrettungsregisters auf den Stack gebracht wird (die typischen RISC-Maschinen (Mips, PowerPC, Alpha usw.).

**): dieser Ablauf wird gelegentlich von der Hardware unterstützt (z. B. IA-32-ENTER-Befehl).

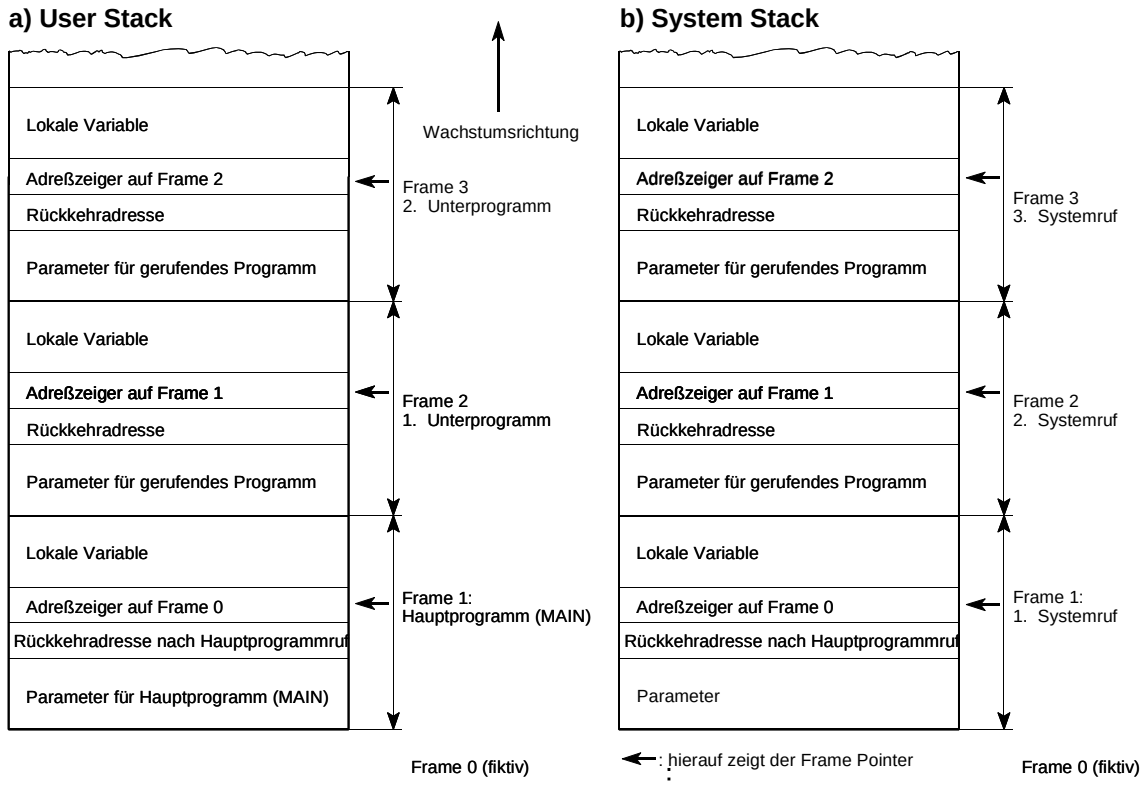

Abbildung 1.3 Die UNIX-Stackorganisation

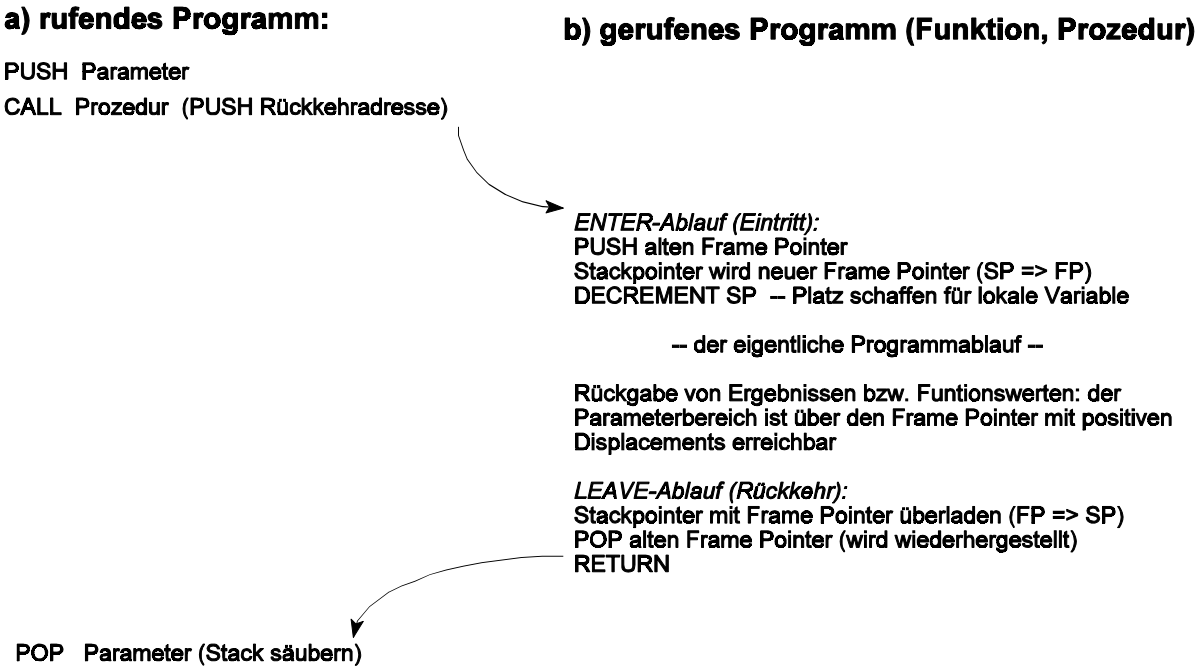

Abbildung 1.4 Unterprogrammaufruf in einer Laufzeitumgebung, die auf Stack Frames beruht

Typische Konventionen des Unterprogrammrufs
Siehe Tabelle 1.2

	Pascal	**C**
Reihenfolge der Parameterübergabe	von links nach rechts	von rechts nach links
wer stellt bei der Rückkehr die ursprüngliche Stackbelegung wieder her (Stack Cleanup)?	das gerufene Programm	das rufende Programm
Vor- und Nachteile der Stack-Cleanup-Konvention	▪ Cleanup-Ablauf nur einmal vorhanden (im gerufenden Programm), ▪ Funktionsaufrufe typischerweise nur mit fester Parameteranzahl	▪ Cleanup-Ablauf in jedem rufenden Programm erforderlich, ▪ erster Parameter (ganz links im Funktionsaufruf) kommt stets auf TOS zu liegen (erleichtert Implementierung von Funktionsaufrufen mit variabler Parameteranzahl)

Tabelle 1.2 Typische Konventionen des Unterprogrammrufs

Ein elementares Laufzeitsystem für Atmel AVR

Prinzip:

Parameter werden vorzugsweise in Registern übergeben.

Wir unterscheiden zwischen Variablenregistern und Arbeits- bzw. Scratch-Registern.

Variablenregister stehen dem jeweiligen Programm zur Verfügung. Sie sind ggf. vom betreffenden Programm zu initialisieren (lokale Variable).

Ein gerufenes Programm (Unterprogramm) muß Variablenregister, die es belegen möchte, auf den Stack retten und bei der Rückkehr wiederherstellen.

Parameter und Resultate werden in Arbeitsregistern übergeben.

Zu jedem Unterprogramm ist die Aufruf- und Rückgabekonvention anzugeben (z. B. R24 enthält die Zeichenzahl, R25 die erste Zeichenposition). Vgl. die Gepflogenheiten beim PC-BIOS.

Arbeitsregister, die nicht in der Aufrufkonvention genannt sind, müssen uverändert zurückgegeben werden (ggf. auf Stack retten und wiederherstellen).

Arbeitsregister, die in der Aufrufkonvention genannt sind, dürfen - sofern sie kein Resultat enthalten - bei der Rückkehr beliebige Inhalte haben (Rettung nicht erforderlich).

Variablenregister: R4...R23 (20 Stück). Reichen für 10 16-Bit-Worte bzw. für 5 32-Bit-Worte.

Arbeitsregister: R0...R3, R24...R27, R30, R31.

R0...R3 und R24...R27 können je ein 32-Bit-Wort aufnehmen (1. und 2. Operand für 16-Bit- und 32-Bit-Arithmetik).

Zur Vorzugsnutzung:

R0...R3: 1. Operand für Arithmetikfunktionen. Nutzung als Akkumulator.

R24...R27: 2. Operand für Arithmetikfunktionen. Direktwert-Parameter in diesen Registern anordnen. Ggf. R30, R31 hinzunehmen.

R30, R31 (= Z-Register): Übergabe einer Programmspeicheradresse.

R26, R27 (= X-Register): Übergabe einer Datenspeicheradresse.

R28, R29 (= Y-Register): reserviert für künftige Nutzung als Frame Pointer.

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15
R16
R17
R18
R19
R20
R21
R22
R23
R24
R25
R26 = X Low
R27 = X High
R28 = Y Low: reserviert (Frame Pointer)
R29 = Y High: reserviert (Frame Pointer)
R30 = Z Low
R31 = Z High