Interruptserviceroutinen (ISRs)

– Ein Überblick –

Interruptserviceroutinen (ISRs) sind Programme, die Unterbrechungsanforderungen bedienen. Eine Unterbrechung (Interrupt) kann als eine hardwareseitig erzwungene Programmumschaltung oder als ein automatisch ausgelöster Unterprogrammaufruf angesehen werden. Wichtig ist, daß das unterbrochene Programm nach der Rückkehr aus der Unterbrechungbehandlung fortgesetzt werden kann; es muß sich dann im selben Zustand befinden, als wenn die Unterbrechung nicht stattgefunden hätte (nur Zeitverzögerung, aber keine Beeinflussung des Ablaufs oder der Daten).

Grundzustand und Unterbrechungszustand
Grundzustand ist der Zustand, in dem "normale" Programme ablaufen. Es können mehrere Programme sein. Gibt es weniger Prozessoren als laufende Programme, ist irgend eine Form der Vermittlung erforderlich. Das übliche Vermittlungsprinzip beruht auf der Vergabe von Zeitscheiben (Programm A belegt den Prozessor für 5 ms, Programm B für 20 ms usw.).

Der Unterbrechungszustand ist der Zustand, in dem eine Interruptserviceroutine (ISR) abläuft. Er wird durch das Auslösen der ISR eingeleitet und durch das Verlassen der ISR beendet.

Die Interruptvorkehrungen sind im Grunde Sparlösungen
Sie dienen dazu, das zeitmultiplexe Ausführen mehrerer Anwendungsaufgaben (Tasks) auf vergleichsweise wenigen Prozessoren (in den meisten Fällen: auf einem einzigen) zu unterstützen.

Hätte man für n Anwendungsaufgaben n Maschinen, so würde das Problem gar nicht existieren. Das Reagieren auf externe Ereignisse der Hardware würde auf ein Abfragen (Polling) oder auf ein Aufwecken aus einer Art Schlafzustand hinauslaufen. (Aber auch bei massivem Einsatz von Prozessoren wäre man keineswegs alle Programmierschwierigkeiten los, denn das Zusammenwirken ist mit Zeitverlusten und Verwaltungsaufwand verbunden, wobei die kritischen Phasen des Zusammenwirkens (Abstimmung, Auftragsübergabe, Vermittlung) besondere Aufmerksamkeit erfordern.)

Einfache Interrupts und Systemereignisse
Was soll eine ISR leisten? – Im Hinblick darauf können wir zwei Arten unterscheiden:

1. Einfache (Lightweigth) Interrupts
Die ISR reagiert auf externe Ereignisse der Hardware und führt alle zugehörigen Funktionen selbst aus. Die im Grundzustand laufenden Programme bleiben unbeeinflußt (Interrupt = Problembehandlung).

2. Systemereignisse
Ein Systemereignis bewirkt, daß in der Maschinenbelegung / Laufzeitvergabe des Grundzustandes irgend etwas geändert wird (eine Task wird gestartet, eine andere erhält vorrangig Laufzeit usw.). Die ISR führt also keine Anwendungsaufgabe selbst aus, sondern schickt lediglich eine Nachricht an das System, das daraufhin die Arbeit im Grundzustand umorganisiert (Interrupt = Problemannahme).

Ganz einfache Interrupts
Die ISR ist in sich nicht unterbrechbar. Sie verwendet fest zugeordnete Bereiche im RAM und ggf. fest zugeordnete Register.

Jeder ISR sind – nach Bedarf – zuzuordnen:

- eigene Arbeitsbereiche im RAM (feste Adressen),
- eigene (fest zugeordnete) Register,
- transiente Register,
- temporäre Register (Arbeitsregister).

Die Inhalte der transienten Register bleiben beim Verlassen der ISR erhalten. Sie werden bei erneuter Auslösung wieder bereitgestellt. Um die Registerinhalte aufzubewahren, ist entsprechender Platz im RAM erforderlich.

Temporäre Register (Arbeitsregister, Scratchpad-Register) werden nur während der Ausführung der ISR benötigt. Ihr Inhalt kann bei der Rückkehr aufgegeben werden.

Der Ablauf einer ganz einfachen ISR:

1. Prozessorzustand auf Stack retten,
2. transiente und temporäre Register auf Stack retten,
3. die Variablen stehen in fest zugeordneten Registern und / oder im RAM,
4. ggf. Variable aus dem RAM in die zugeordneten (transienten) Register schaffen,
5. Funktion ausführen,
6. ggf. Variable (die sich geändert haben) aus den zugeordneten Registern in den RAM schaffen,
7. ggf. die auslösende Ursache (in der Hardware) zurückstellen,
8. transiente und temporäre Register aus Stack wieder einstellen,
9. Prozessorzustand aus Stack wieder einstellen,
10. Rückkehr aus Interrupt (RETI).

Das Registerproblem
Es hängt sehr von der Prozessorarchitektur ab, in Hinsicht auf Extravaganzen auch von der Programmentwicklungsumgebung. Typische Auslegungen:

- besondere Registersätze für die Interruptbehandlung (automatische Umschaltung oder Umschaltbefehl). Historische Beispiele: RCA Spectra, Zilog Z 80.
- ein nur kleiner Registersatz (z. B. Akkumulator(en), Indexregister und Stackpointer). Die Register sind grundsätzlich als Arbeitsregister anzusprechen; Variablen kann man darin beim besten Willen nicht unterbringen. Alles retten und wieder einstellen.Beispiele: 8051, 8086, 6800, JVM (hat als Stackmaschine gar keine architekturseitigen Register).
- größere Registersätze (Atmel AVR, ARM, MIPS usw.). Hier tritt das Problem überhaupt erst auf (z. B. einer ISR drei Register fest zuzuweisen). Wird dies von der Programmentwicklungsumgebung nicht unterstützt, so bleibt nichts anderes übrig, als jedesmal alle Register zu retten.

– Maschinen mit großen Universalregistersätzen sind im Grunde nichts für Multitasking-Realzeitanwendungen – mag doch die Werbung behaupten, was sie will ... –

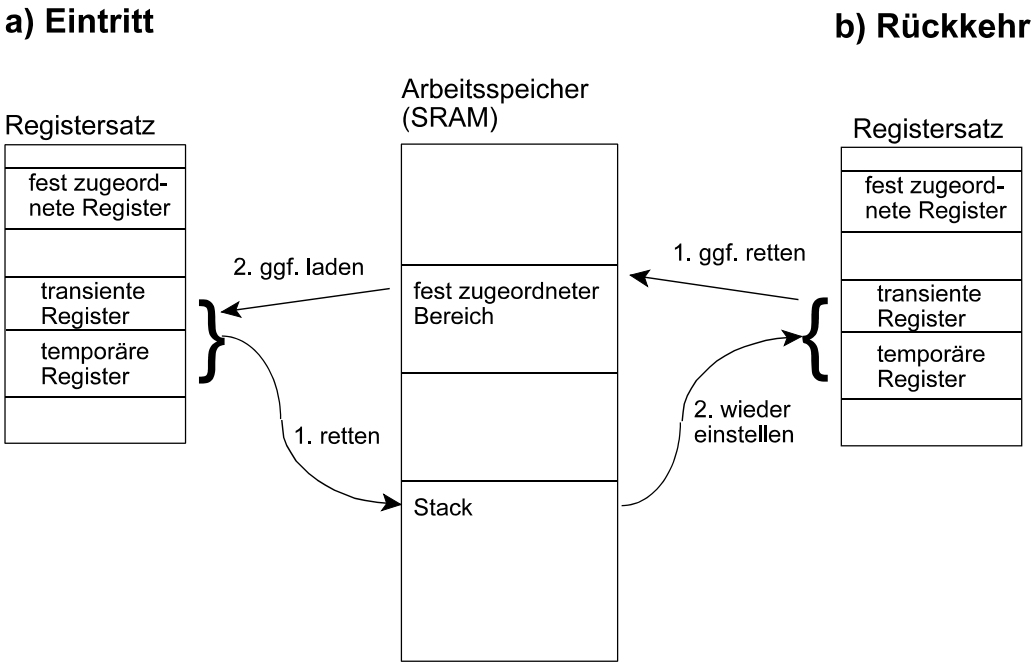


Abb. 1. Ganz einfache Interrupts: Eintritt und Rückkehr.

Was kostet es, alle Register des Atmel AVR auf den Stack zu retten und aus dem Arbeitsbereich der ISR neu einzustellen? – 1 SRAM-Zugriff = 2 Zyklen. 32 Register speichern und holen = 64 • 2 = 128 Takte. Bei 4 MHz (0,25µs) sind dies 32 µs. Bei der Rückkehr wird diese Zeit noch einmal fällig...

Unterbrechbare ISRs
Die laufende ISR macht sich ihrerseits unterbrechbar, nachdem sie alles auf den Stack gerettet hat, was zu retten ist. Hierdurch können weitere (höher priorisierte) Unterbrechungsanforderungen eher zur Wirkung kommen (Verringerung der Latenzzeit).

Das einfachste Ablaufschema: geschachtelte (Nested) Interrupts
Die zuletzt gerufene ISR hört auch zuerst auf. Es gibt keine Änderung der Ausführungsreihenfolge. Es genügt ein gemeinsamer Stack. Bereits laufende ISRs dürfen nicht noch einmal ausgelöst werden (das ist ggf. Sache der Hardware).

Wie wird die Unterbrechbarkeit hergestellt?
Das hängt von der Prozessorarchitektur ab. Im Umfeld der Unterbrechungsbehandlung sind drei Befehle von Bedeutung (sie heißen bei jedem Prozessor anders; wir verwenden hier typische Allgemeinbezeichnungen).

- Unterbrechung erlauben (Enable Interrupt, EI),
- Unterbrechung verhindern (Disable Interrupt, DI),
- Rückkehr aus der Unterbrechungbehandlung (Return from Interrupt, RETI).

Es kommt vor allem darauf an, was der RETI-Befehl leistet, was ihn vom RET unterscheidet. Es gibt zwei Auslegungen:

a) RETI ist praktisch ein RET, der zuätzlich nur die Unterbrechungserlaubnis setzt (Beispiel: Atmel AVR). Eine ISR kann man also einfach durch EI unterbrechbar machen. RETI am Ende schadet nicht.

b) RETI dient nicht nur zur Rückkehr, sondern auch zum erneuten Scharfmachen des Interruptmechanismus (Z 80, teils auch Intel IA-32). Beim Intel bewirkt der Rückkehrbefehl (IRET), daß die NMI-Auswertung wieder scharf wird. Zudem kann er die Rückschaltung zu einer Task veranlassen, der zuvor Laufzeit entzogen wurde. Beim Z80 muß RETI gegeben werden, damit die Hardware überhaupt wieder Interruptanforderungen bearbeiten kann. Um eine ISR unterbrechbar zu machen, ist ein kleines Trickprogramm erforderlich, z. B.:

```
        PUSH next_adrs
        RETI
next_adrs:
```

Kompliziertere Formen der Ablauforganisation
Die einfache Verschachtelung tut es nicht; die Verarbeitungsreihenfolge muß umorganisiert werden. Das läuft typischerweise darauf hinaus, die eigentlichen Aufgaben im Grundzustand zu erledigen (mit entsprechend eingerichteten Prioritäten usw.). Die ISR ist hierbei nur Auftragsannahme. Sie übergibt den jeweiligen Auftrag an den Laufzeitverwalter (Scheduler) des Betriebssystems (Ereignissteuerung).

Die Taskumschaltung mittels ISR
Das ist die am meisten verwendete Form der Taskumschaltung (z. B. Übergang zur nächsten Task nach Ablauf der aktuellen Zeitscheibe (Timer-Interrupt)). Es ist im Grunde auch eine Notlösung – weil die hardwareseitige Taskumschaltung einfach nicht in Mode ist (Beispiele (teils historisch): die E-A-Prozessoren der CDC 6600, Singer System 10, Denelcor HEP, Intel Pentium etc. mit HyperThreading).

Der Taskzustandsbereich (Task State Area TSA (Intel: Task State Segment TSS))
Jede Task hat einen solchen Bereich im Arbeitsspeicher. Er nimmt den aktuellen Taskzustand auf, wenn der Task Laufzeit entzogen wird.

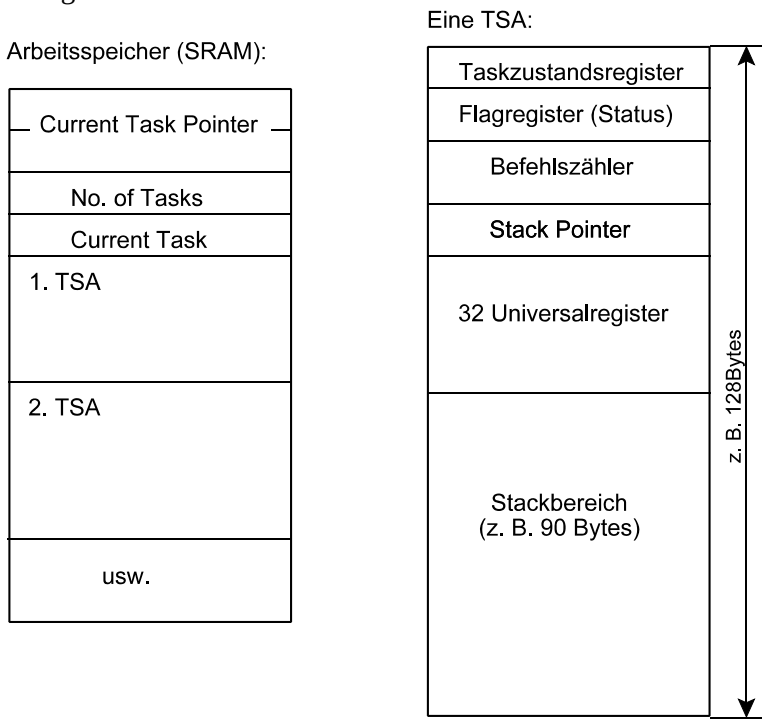


Abb. 2. Tasksteuer- und Zustandsangaben im Arbeitsspeicher. Links: allgemein. Der Current Task Pointer zeigt auf den Taskzustandsbereich der aktuellen Task. Rechts: Struktur eines Taskzustandsbereichs (beispiel für Atmel AVR).

Die Taskumschaltung ist im Grunde ganz einfach:

1. den aktuellen Prozessorzustand (Befehlszähler, Register) in den Taskzustandsbereich der aktuellen Task schaffen.
2. den Taskzustandsbereich der nächsten Task auswählen.
3. den Prozessorzustand aus diesem Taskzustandsbereich in den Prozessor schaffen.
4. mittels Rückkehrbefehl (RET, RETI) die betreffende Task zum Laufen bringen.

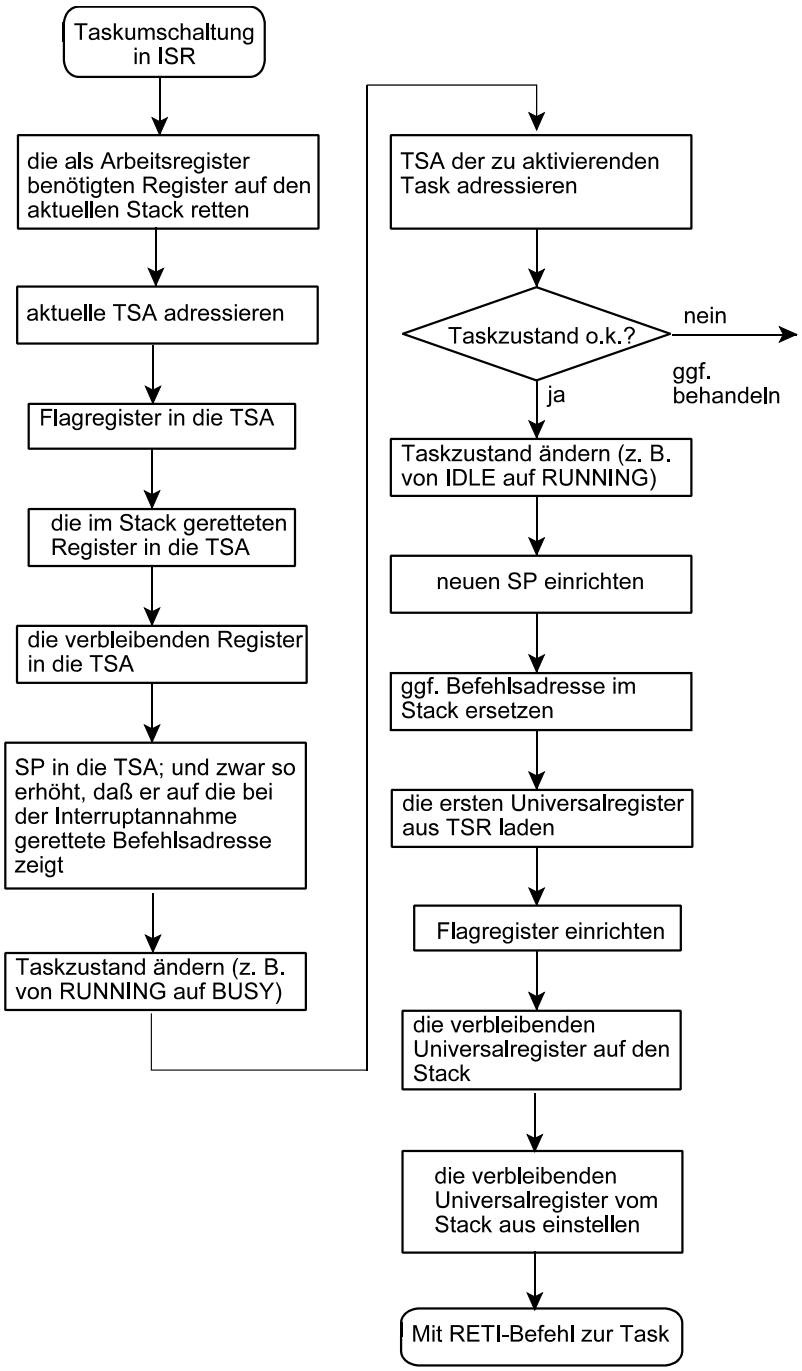


Abb. 3. Der Ablauf einer Taskumschaltung.

Die Taskumschaltung im Grundzustand – die Zeitscheibe
Das Problem: wie lange soll eine Zeitscheibe sein? Je kürzer, desto geringer die Latenzzeiten, desto länger der Overhead (für das Retten und Einstellen). Windows hat etliche ms, weil so viel zu retten und einzustellen ist...

Nehmen wir an, wir wollten uns nur 1% Verwaltungs-Overhead leisten. Beim Atmel AVR mit 4 MHz dauert das Retten und Einstellen der 32 Register 32 µs; also der Ablauf gemäß Abb. 3 rund 40...50µs. Daraus ergeben sich Zeitscheiben von 4...5 ms. Das ist für viele Anwendungen ausreichend (1 Netzhalbwelle schlimmstenfalls 8 ms, eine Bedienfeldabfrage z. B. alle 20 ms usw.; es können also ohne weiteres 3...5 Tasks mit Realzeitanforderungen in der Größenordnung von 10 ms laufen).

Aufträge an laufende Tasks übermitteln
Auch ganz einfache ISRs haben gelegentlich mit den Tasks des Grundzustandes zu kommunizieren, z. B. um Daten zu übergeben oder Parameter auszutauschen. Ein typisches Anwendungsgebiet ist die Interfacesteuerung (Kommunikation zwischen Interface und Datenpuffer). Hierzu sind entsprechende Kommunikations- und Pufferbereiche im RAM einzurichten. Die Kommunikation beruht auf dem Setzen und Abfragen von Steuerbits.

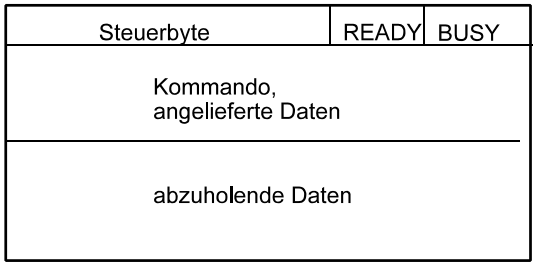


Abb. 4. Ein Kommunikations- und Pufferbereich.

Steuerbit	Funktion	Programm im Grundzustand	ISR
READY	Ausgabe	schaltet ein, um anzuzeigen, daß Daten bereitstehen	fragt ab und schaltet aus, um anzuzeigen, daß Daten übertragen wurden und Puffer wieder frei ist
BUSY	Eingabe	fragt ab und schaltet aus, um anzuzeigen, daß Daten intern abgeholt wurden und Puffer wieder frei ist	schaltet ein, um anzuzeigen, daß Daten geholt wurden und im Puffer bereitstehen

Was geschieht, wenn eine neue Unterbrechung kommt, und seitens der Task die vorherige Anforderung noch nicht bearbeitet wurde? – Das hängt vom Anwendungsproblem ab. Grundsätzlich gibt es verschiedene Möglichkeiten, z. B.:

- die Anforderung einfach ignorieren (z. B. Tastenbetätigungen),
- einfach warten oder verspätet senden (Wartezustände),
- den mit der Außenwelt ablaufenden Informationsaustausch irgendwie bremsen (Handshaking),
- Warteschlangen bzw. Datenpuffer (FIFOs) einrichten,
- eine Notmaßnahme einleiten,
- eine Fehlerbehandlungsmaßnahme einleiten.

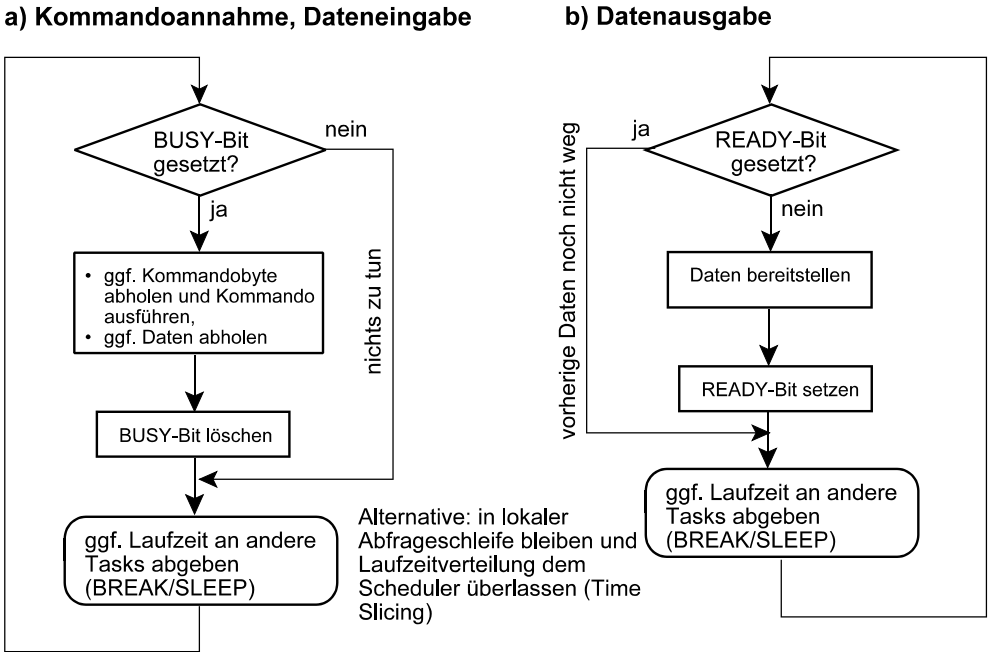


Abb. 5. Kommunikationssteuerung über Steuerbits (1). Die Task im Grundzustand.

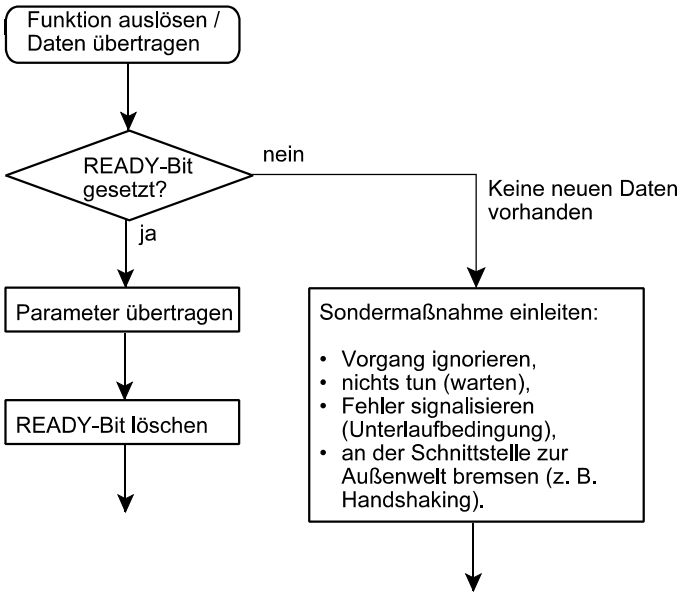


Abb. 6. Kommunikationssteuerung über Steuerbits (2). ISR für Ausgabe.

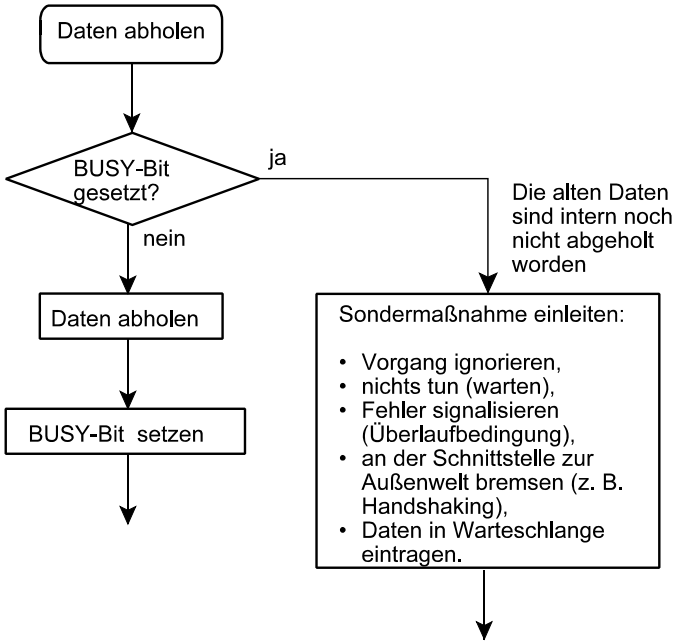


Abb. 7. Kommunikationssteuerung über Steuerbits (2). ISR für Eingabe.

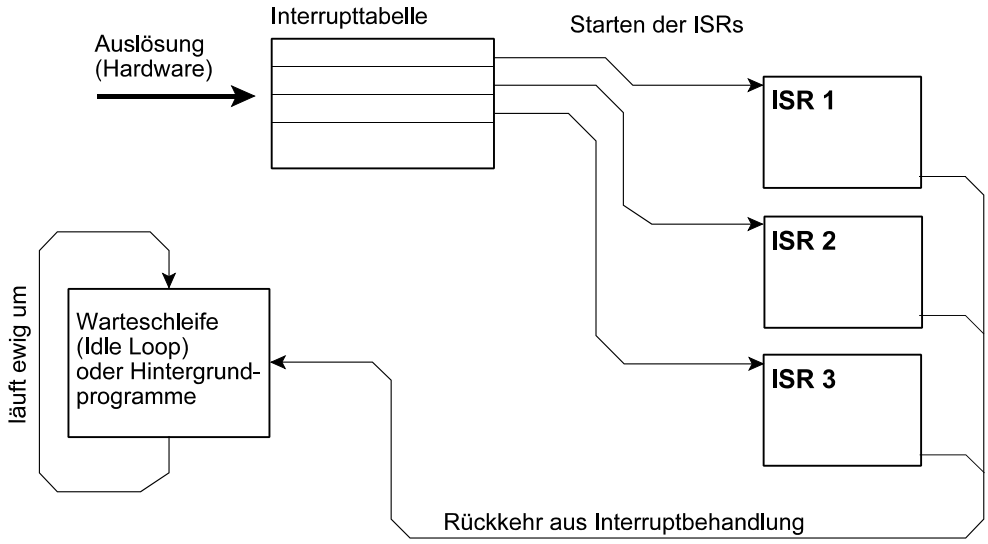


Abb. 8. Ablauforganisation einfachster ISRs.

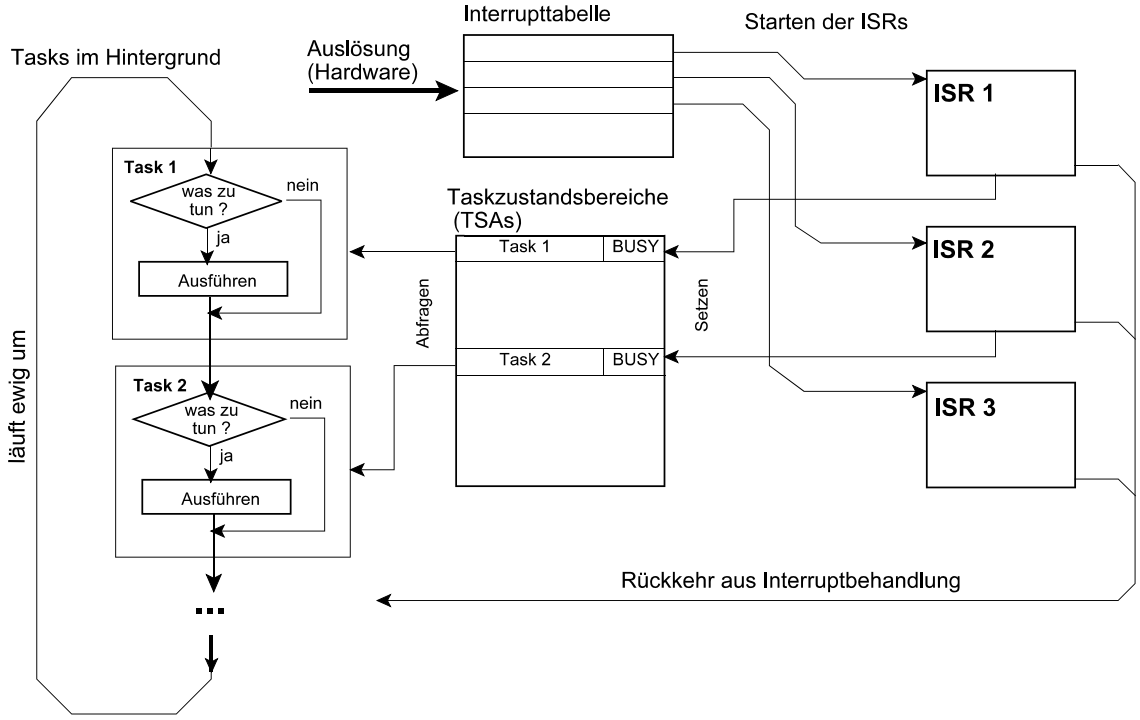


Abb. 9. Kommunikation zwischen einfachen ISRs und den Tasks im Grundzustand.

Wenn Interrupts in der ISR erlaubt sind:
Höher priorisierte Interrupts annehmen und niedriger priorisierte hinten anhängen (Warteschlange).

Sehr kurze Interrupts: Break Ins
Die ISR dient nur zur Auftragsannahme und zum unmittelbaren Reagieren an der jeweiligen Schnittstelle (z. B. Steuerung des Handshaking). Grundgedanke:die ISR startet nicht an einer festen Adresse, sondern kann die Fortsetzungsadresse (für den Programmstart bei der nächsten Unterbrechung) einstellen (z. B. bei der Rückkehr die Folgeadresse hinterlassen (Nachbildung des Break-In-Mechanismus des S/360)). Hiermit läßt sich der Befehlszähler der Interruptroutine verwenden, um den jeweiligen Programmzustand zu codieren.

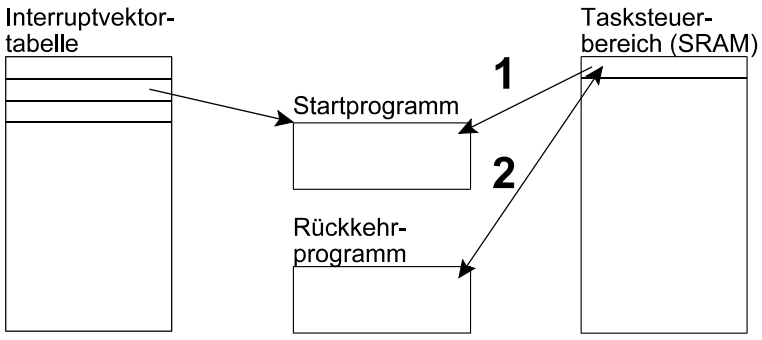


Abb. 10. Starten einer Break-in-Routine.

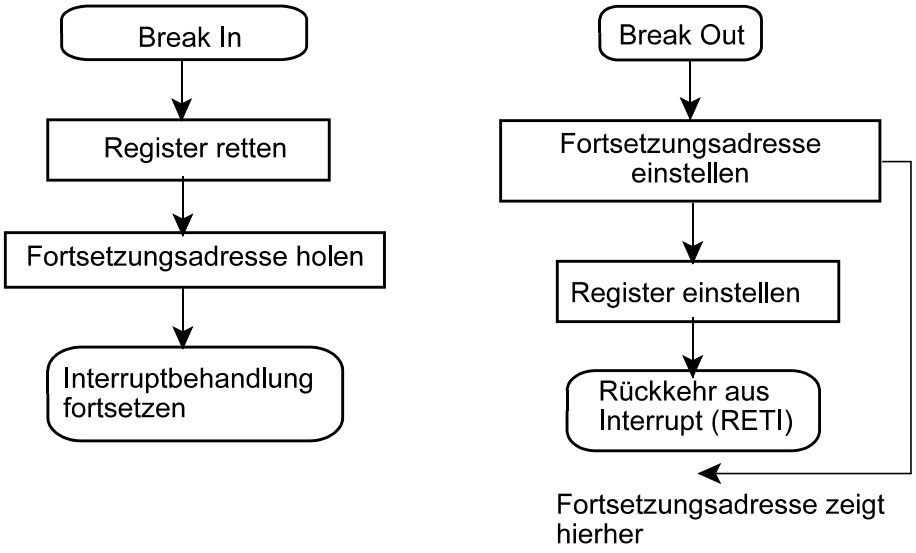


Abb. 11. Einleiten und Verlassen von Break-in-Routinen.

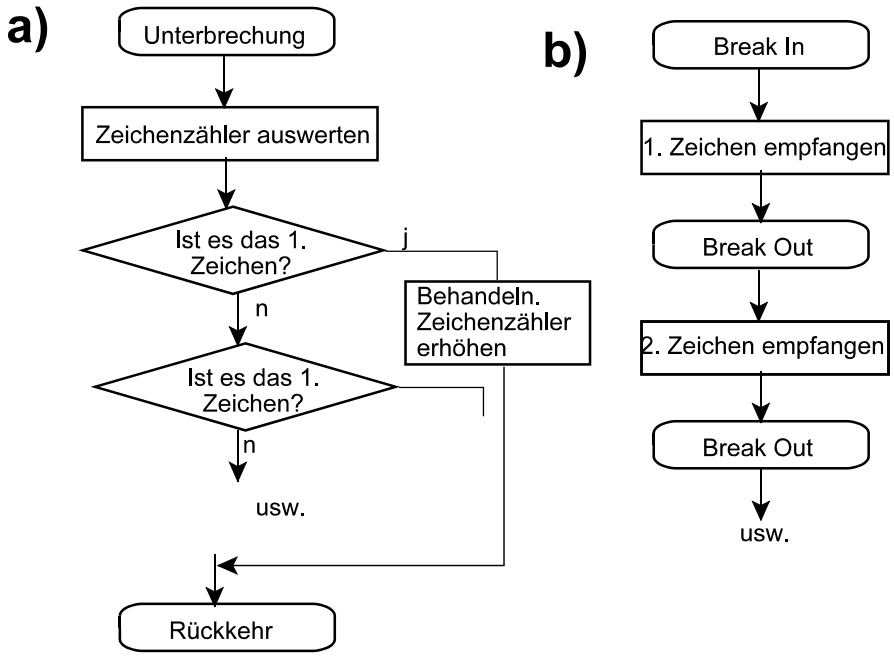


Abb. 12. Herkömmliche Unterbrechung vs. Break in am Beispiel des Akzeptierens einer Zeichenkette. a) Unterrbrechung. Start an fester Adresse. Maschinenzustand muß gesondert codiert werden, z. B. in einem Zeichenzähler. Bei jeder Unterbrechung muß gemäß aktuellem Zeichenzähler verzweigt werden (z. B. Switch-Anweisung). b) Break in. Man kann den Empfang der Zeichen fortlaufend ausprogrammieren. Hier ist der Maschinenzustand im Befehlszähler der Break-in-Routine codiert.

Versuch:
Multitasking mit Zwangsumschaltung Round Robin über Counter/Timer. Jede Task ist eine Art virtueller Maschine. Zunächst ganz schmucklos.

Bereiche im SAM:

a) Taskzustandsbereich. Besteht aus einem Zeiger auf die aktuelle Task und einem Speicherbereich je Task, der Zustandsbits sowie den aktuellen Stackpointer enthält. Der Stackpointer ist dann gültig, wenn die Task keine Laufzeit hat. Er zeigt auf das erste freie Byte im Stack (über dem geretteten Taskzustand).

b) Stackbereiche. Je Task ist ein Bereich vorgesehen. Größe: beispielsweise 90 Bytes. Der gerettete Taskzustand umfaßt 36 Bytes:

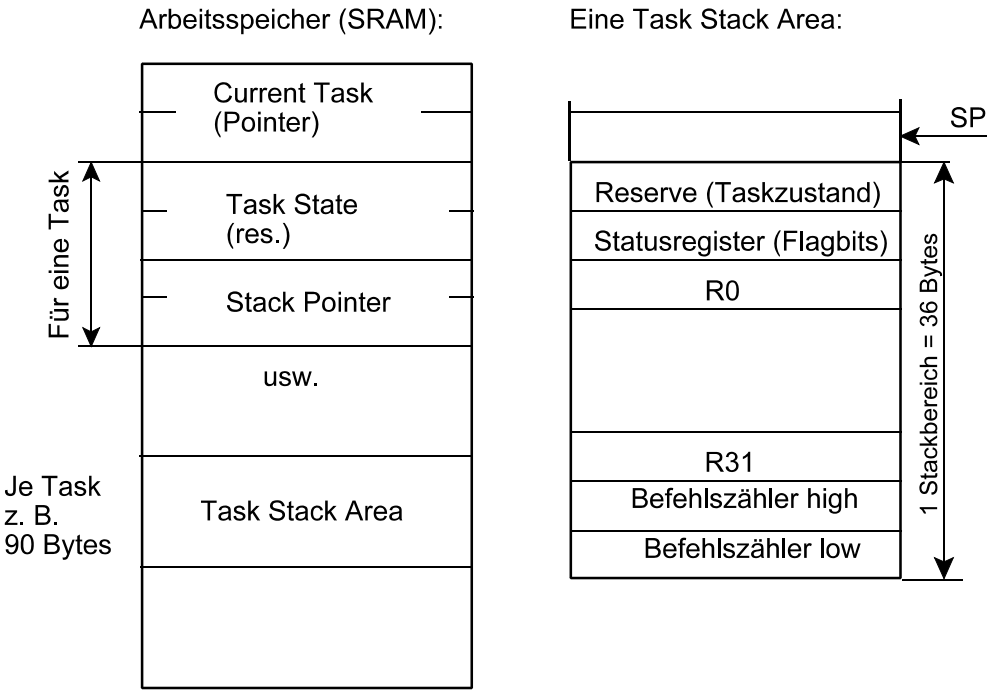
   - den Befehlszähler,
   - die 32 Register,
   - das Statusregister,
   - ein Zustandsbyte.

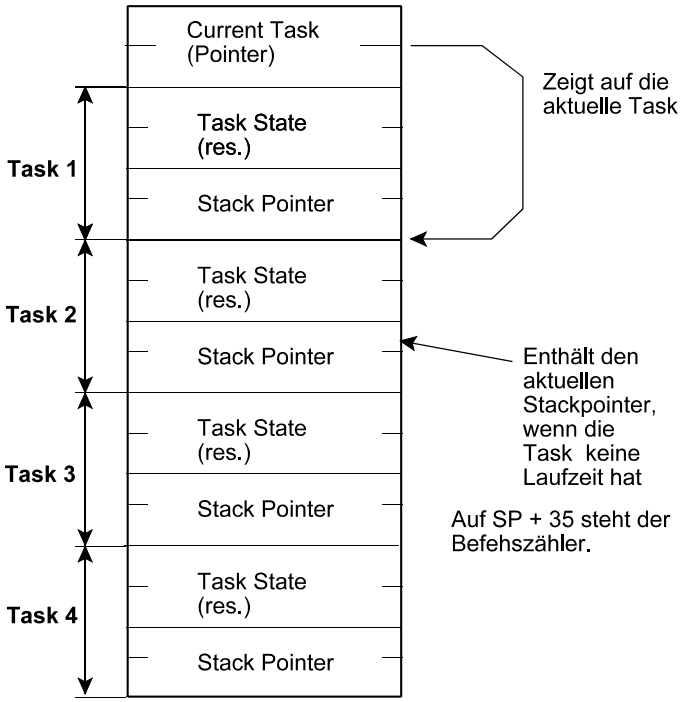
Taskumschaltung:

1. Eintritt mit Interrupt oder Call rettet den Befehlszähler auf den Stack.
2. Dann kommen alle Register einschließlich Flags.
3. Stackpointer wird in den Taskzustandsbereich gerettet.
4. Der nächste Stackpointer wird geholt und eingerichtet.
5. Alle Register (einschließlich Flags) werden aus dem Stack geholt.
6. Umschaltung erfolgt durch Holen des Befehlszählers mittels RET oder RETI.

Initialisierung:

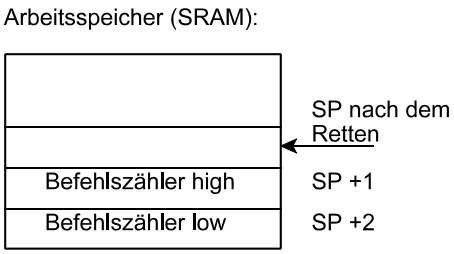
- Für jede Task wird der erste Stackbereich eingerichtet.
- Alle Befehlszähler in den Stacks zeigen auf ein lauffähiges Programm (und wenn es ein Sprung auf sich selbst ist.
- Dann wird der Timer-Interrupt scharfgemacht und darauf gewartet, daß etwas losgeht.


Arbeitsspeicher (SRAM):
Current Task (Pointer)
Task State (res.)
Stack Pointer
usw.
Task Stack Area
Für eine Task
Je Task z. B. 90 Bytes
Eine Task Stack Area:
SP
Reserve (Taskzustand)
Statusregister (Flagbits)
R0
R31
Befehlszähler high
Befehlszähler low
1 Stackbereich = 36 Bytes


Current Task (Pointer)
Task State (res.)
Stack Pointer
Task 1
Task 2
Task 3
Task 4
Zeigt auf die aktuelle Task
Enthält den aktuellen Stackpointer, wenn die Task keine Laufzeit hat
Auf SP + 35 steht der Befehszähler.

Offset v. SP	Inhalt	Offset v. SP	Inhalt	Offset v. SP	Inhalt	Offset v. SP	Inhalt
1	Zustand	10	R7	19	R16	28	R25
2	SReg	11	R8	20	R17	29	R26
3	R0	12	R9	21	R18	30	R27
4	R1	13	R10	22	R19	31	R28
5	R2	14	R11	23	R20	32	R29
6	R3	15	R12	24	R21	33	R30
7	R4	16	R13	25	R22	34	R31
8	R5	17	R14	26	R23	35	PC low
9	R6	18	R15	27	R24	36	PC high

Der gerettete Befehlszähler im Stack:



Achtung – oben das High-Byte, unten das Low-Byte, obwohl es sich eigentlich um eine Little-Endian-Maschine handeln sollte. Gurkensache ...