Hard- und Software-Engineering
Übersicht

Stand: 9. 1. 2006

1. Mikrocontroller

Mikrocontroller sind programmierbare Universalrechner, die für den Einsatz in Embedded Systems optimiert sind. Kennzeichnende Merkmale:

- Implementierung der jeweiligen Informationswandlungen auf Grundlage herkömmlicher Programmiermodelle,
- Kostenoptimierung,
- Einbau in eine Anwendungsumgebung,
- Programmierung auf den jeweiligen Anwendungszweck hin (Controller wird durch Programmierung zur Einzweckmaschine),
- Programme typischerweise vom Nutzer nicht änderbar,
- wir dürfen programmieren (Narrenfreiheit, potentielle Funktionsvielfalt),
- wir müssen programmieren (Zwang zur Rückführung der anwendungsseitig geforderten Informationswandlungen auf elementare Programmschritte; Serialisierung der Informationsverarbeitung, Problemlösung entspricht nicht mehr dem anwendungsseitigen Datenflußschema).

Anwendungsfälle:

- Realzeitraster im ms-Bereich oder langsamer,
- es stehen keine anderweitigen Forderungen entgegen (extremes Stromsparen, EMV, Sicherheitsvorschriften).

Infrastruktur:

- Stromversorgung,
- Takt,
- Rücksetzen,
- E-A-Anschaltung,
- ggf. Speicheranschaltung.

Auswahlkriterien:

1. Kosten,
2. frei nutzbare E-A-Anschlüsse (Ports),
3. eingebaute Peripherie,
4. Programmspeicherkapazität,
5. Datenspeicherkapazität,
6. Registersatz,
7. Leistungsvermögen der Befehlsliste[1)],

1) wir merken uns: alle Befehlslisten sind äußerst leistungsfähig (powerful), manche mehr, manche weniger.

8. Geschwindigkeit,
9. technische Einsatzbedingungen (Gehäuse, Speisespannung, Treibvermögen, Takterzeugung, Programmierverfahren usw.),
10. Kompatibilität,
11. Bezugsmöglichkeiten,
12. Entwicklungsunterstützung.

Der typische moderne (kleine)Mikrocontroller hat eingebaute Programm- und Datenspeicher. Somit stehen alle E-A-Anschlüsse zum Lösen der Anwendungsaufgabe zur Verfügung. Seine Festwertspeicher sind Flash-ROMs oder EEPROMs. Sie können in der Einsatzumgebung programmiert werden (In-System Programming ISP – kein Programmiergerät, kein UV-Licht zum Löschen).

Lösen von Anwendungsaufgaben:
Hard- und Software sind im Verbund zu betrachten. Wir beginnen mit der Hardware.

Erstes Ziel:
Zusatzbeschaltung vermeiden! Im Idealfall besteht – von der Infrastruktur (Takt, Rücksetzen, Stromversorgung) abgesehen – das System nur aus dem Mikrocontroller und der gemäß Anwendungsaufgabe anzuschließenden Hardware (Tasten, Schalter, Leuchtanzeigen, Sensoren usw.).

Zweites Ziel:
Mit möglichst wenigen E-A-Anschlüssen auskommen (genauer: mit dem "kleinst-möglichen" Gehäuse (Kostenfrage)).

Oftmals unvermeidlich:

- Hardware zur Signalwandlung und -aufbereitung,
- Treiberstufen.

In diesem Rahmen kann (und sollte) getrickst und optimiert werden (Analogmultiplexer, Schieberegister-Interfaces usw.). Maßgebend ist stets Kostenrechnung "über alles"!

Externe Logikschaltungen sollten aber nicht über einzelne Gatter, Treiber o. dergl. hinausgehen. CPLDs sind beträchtlich teurer als Mikrocontroller!

Schritte der Problemanalyse:

1. Was ist – als Hardware – zu bauen?
Die vorgegebenen Schnittstellen der anzuschließenden Einrichtungen untersuchen:

1. um welche Signale geht es eigentlich? – Signalflußrichtung, Pegel, Flanken, Ströme usw.
2. ist Direktanschluß an den Mikrocontroller möglich (aus elektrischer Sicht)? – Das hängt ab zum einen von der Art des Signals und zum anderen von Praxisfragen ab (Leitungslänge, Störbeeinflussung, Störstrahlung).
3. welche Anschaltungsmaßnahmen sind ansonsten erforderlich (Signalaufbereitung und Wandlung, Treiberfunktionen usw.)?
4. welche Bauelemente bzw. Schnittstellen sind hierfür vorteilhaft einsetzbar? – Anregungen: Erfahrungwissen, Literatur, Internet. Motto: seht Euch um und laßt Euch was einfallen...
5. welche eingebauten Peripheriefunktionen der Mikrocontroller sind vorteilhaft einsetzbar? – Das läuft meist darauf hinaus, verschiedene Alternativen zu untersuchen (Kostenrechnung), z. B. externer A/D-Wandler oder Mikrocontroller mit eingebautem A/D-Wandler.

Ergebnis: Schaltungslösung. Dokumentation: wenigstens Blockschaltbild (hinreichend detailliert).

Beim professionellen Entwickeln: jetzt schon prüfen:

- Beschaffung,
- Fertigung,
- Prüfung,
- EMV-Probleme.

Es nützt gar nichts, Bauelemente einzusetzen, die zwar gut aussehen, deren Beschaffung aber im vorauszusehenden Fertigungszeitraum nicht gewährleistet ist.

Der Entwurf muß sich auch wirklich fertigen lassen! Prüfen, ob Bedingungen der Leiterplattenfertigung erfüllt werden. Es nützt nichts, Bauelemente einzusetzen, die im gegebenen Kostenrahmen nicht zu bestücken sind (Anschlußabstände, Lötverfahren usw.).

Jede Hardware braucht den CE-Kuckuck! Jetzt schon an die erforderlichen Vorkehrungen denken! (Z. B. an Platz für Filter und Suppressordioden.)

Es kann durchaus sein, daß wegen solcher Praxis- bzw. Trivialprobleme ein an sich guter Entwurf geändert werden muß – womöglich radikal. Aber besser jetzt als später!

2. Was ist – als Software – zu programmieren?
Auszuführende Funktionen verstehen. Vorgehensweise ergibt sich aus den Anforderungen. Akademische Entwurfsprinzipien nur bedingt hilfreich. Von Anfang an vorsortieren:

- was muß unbedingt sein?
- was dient nur der Schönheit? – Was kann also geopfert werden, wenn Terminvorgaben drängen oder wenn nicht mehr alles in den Speicher paßt?[1)]
- was hat Zeit? (Programmierung unkritisch.)
- wobei sind bestimmte Laufzeitvorgaben einzuhalten? – Ausführbarkeit zeitkritischer Schleifen ggf. anhand von Probeprogrammen überprüfen! (Noch ehe wir in die eigentliche Programmierung einsteigen.)

Gleich am Anfang: die zeitkritischen Programmstücke erkennen (Erfahrungs- und Intuitionssache) und zusehen (Probeprogrammierung), ob es reicht oder nicht. Hieraus ergeben sich gelegentlich Rückwirkungen auf die Hardware (Auswahl des Mikrocontrollers).

Anregungen: die Applikationsschriften der Hersteller und Nutzervereinigungen (Internet). Achtung - nicht blindlings vertrauen, sondern selbst erproben!

Entwicklungsplattform wählen:

1. Programmiersprache. Nicht in der Herde mitrennen. Selbständig denken! Es gibt nicht nur C, sondern auch Forth, Basic usw. Eine Sprache wie C hat nur dann wirkliche Vorteile, wenn die Bibliotheksfunktionen ausgiebig genutzt werden – wir müssen uns weder die Arithmetik selbst schreiben noch uns über die Parameterübergabe beim Unterprogrammruf Gedanken machen. C-Programme für einfache Steuerungsabläufe sehen aus wie Assemblerprogramme, nur ist die Syntax eine Zumutung.

---

1) wir merken uns: Termine sind immer zu knapp, Speicher immer zu klein ...

2. Entwicklungsumgebung (nur PC mit Programmierschnittstelle (Download-Kabel) oder PC + In-Circuit-Emulator). Viele Aufgaben sind mit einfachen Hilfsmitteln (in Hard- und Software) zu lösen - man muß sich nur zu helfen wissen ...
3. Erprobungsumgebung (fertiges Starterkit, Labormuster, V-Muster des eigentlichen Systems). Handverdrahtete Labormuster kritisch bei hohen Frequenzen, Analogsignalen usw.

Elementare Festlegungen (Programmierkonventionen):

- Belegung der E-A-Schnittstellen,
- Registerbelegung. Daran denken, daß es meist keine echten Universalregister sind (es ist nicht so, daß mit jedem Register alles geht). Arbeitsregister und Register für ggf. benötigte Sonderfunktionen freihalten.
- Arbeitsspeicherbelegung. Ggf. an den Stack denken.
- EEPROM- Belegung (Speicher mit Datenerhalt),
- Programmspeicherbelegung (gelegentlich ist an zu speichende Konstanten, an Sprungweiten usw. zu denken),
- Parameterübergabe beim Unterprogrammruf,
- Debugging-Vorkehrungen.

Alles symbolisch deklarieren (z. B. mit EQU- oder DEFINE-Anweisungen)!

2. E-A-Ports

2.1 Universelle E-A-Ports

Mikrocontroller müssen universell verwendbar sein. Deshalb sind die E-A-Anschlüsse einzeln programmseitig steuerbar (Eingang oder Ausgang).

Nutzung:

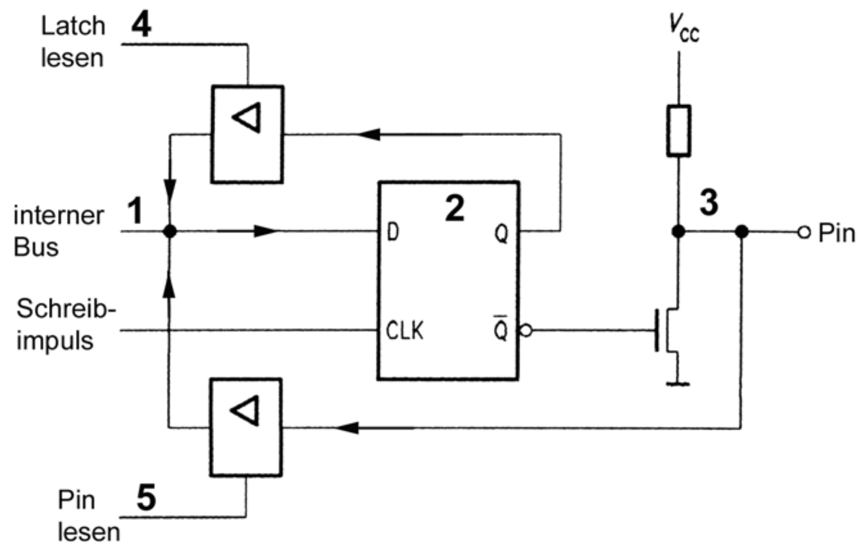
- zum Einstellen des jeweiligen Anschlusses auf die betreffende Funktion,
- zum Betreiben bidirektionaler Verbindungen (beide Übertragungsrichtungen),
- zum Freigeben von Signalwegen, so daß sie von anderen Einrichtungen benutzt werden können (Tri-State-Bus).

Einfache Lösungen:

- Open Collector- oder Open-Drain-Ausgänge. Der Port hat nur ein Register (Abb. 5.90).
- Tri-State-Ausgänge. Der Port hat zwei Register: Richtungssteuerregister und Ausgangsregister (Abb. 5.91).

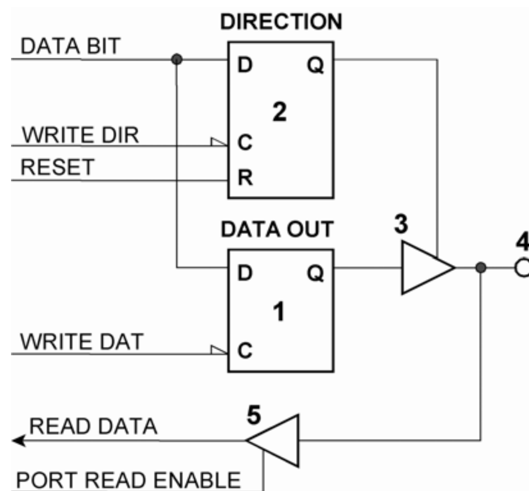
Abb. 5.90 veranschaulicht, wie die E-A-Ports der 8051-Mikrocontroller aufgebaut sind.

- Ausgabe: Betreffende Bitposition des Datenregisters mit dem gewünschten Wert laden.
- Eingabe: Betreffende Bitposition des Datenregisters mit einer Eins laden (muß dauernd so stehenbleiben, wenn der Anschluß als Eingang genutzt werden soll). Die Treiberstufe 3 wird dann nicht angesteuert, und es ist möglich, die Belegung des Anschlusses über die Aufschaltstufe 5 zurückzulesen. (Die Einrichtung, die die Daten liefert, wird allerdings, wenn sie einen Low-Pegel aufschaltet, über die Pull-up-Vorkehrungen mit einem zusätzlichen Stromfluß belastet.)



1 - interner Datenbus; 2 - Datenregister; 3 - Open-Drain-Treiberstufe mit E-A-Anschluß; 4 - Aufschaltung auf internen Datenbus beim Lesen des Datenregisterinhalts; 5 - Aufschaltung auf internen Datenbus beim Lesen der Anschlußbelegung.

Abb. 2.1 E-A-Port mit Open-Drain-Ausgang (eine Bitposition)



1 - Datenregister; 2 - Richtungssteuerregister; 3 - Ausgangstreiber (Tri State); 4 - E-A-Anschluß; 5 - Lesesignaltreiber.

Abb. 2.2 E-A-Port mit Tri-State-Ausgang (eine Bitposition)

Abb. 2.2 veranschaulicht den grundsätzlichen Aufbau eines Tri-State-Ports, wie er in vielen modernen Mikrocontrollern zu finden ist. Jede Bitposition kann einzeln als Eingang oder als Ausgang konfiguriert werden. Hierzu ist das Richtungssteuerregister entsprechend zu laden.

- Ausgabe: Richtungssteuerregister 2 mit Eins laden. Inhalt des Datenregisters 2 erscheint am Anschluß 4.

- Lesen: Gelesen wird durch Aktivieren des Lesesignaltreibers 5. Es wird stets die Signalbelegung am Anschluß 4 gelesen. Je nach Inhalt des Richtungssteuerregisters 2 handelt es sich um den Inhalt des Datenregisters 1 (Ausgabe) oder um ein außen anliegendes Signal (Eingabe).
- Eingabe: Richtungssteuerregister 2 mit Null laden. Ausgangstreiber 3 wird hochohmig. Somit darf der Anschluß 4 von außen belegt werden.
- Der Anfangszustand (nach dem Rücksetzen): alles auf Eingabe (Anschlüsse hochohmig).
- Einstellen: erst Datenbelegung, dann Richtungssteuerung (nur so erscheinen von Anfang an korrekte Ausgangsbelegungen und nicht solche, die sich womöglich aus einer zufälligen Belegung des Datenregisters ergeben).

Rücksetzen und Initialisierung

Nach dem Einschalten müssen die Anschlüsse zunächst hochohmig sein, um Buskonflikte zu vermeiden. Zu den ersten Aufgaben der Software gehört die Initialisierung der E-A-Ports (Belegen mit Anfangswerten, Einstellen der Übertragungsrichtung). Die Grundfrage: was wird wirklich hardwareseitig zurückgesetzt?

- die Optimallösung: alle Register, und zwar auf einen Wert, der nicht schadet (z. B. Richtung auf Eingang, Daten auf Null),
- eine gelegentlich zu findende Sparlösung: nur das Richtungsregister (Beispiel. PIC16C5x). Wichtig: erst das Datenregister mit der Anfangsbelegung laden, dann die Übertragungsrichtung einstellen (sonst stellen sich an den Anschlüssen kurzzeitig die anfänglichen Zufallsbelegungen des Datenregisters ein – was sehr häßliche Nebenwirkungen haben kann...).

Praxistip:

Stets eine Software-Routine zum Rücksetzen vorsehen (die alle Ports, Steuerregister usw. einstellt) – auch dann, wenn in der Hardware an sich alles richtig zurückgesetzt wird. Anwendung: zum programmseitig ausgelösten Rücksetzen zwecks Wiederanlauf (z. B. als Reaktion auf Fehlermeldungen und fehlerhafte Interrupts).

Ports zurücklesen

Manchmal ist es erforderlich, die Belegung von Ausgängen wieder einzulesen. Typische Anwendungen:

- zum Modifizieren von Bitpositionen und Feldern (Read - Modify - Write),
- zu Testzwecken (nachsehen, ob die Ausgänge tatsächlich richtig schalten).

Wie werden Einzelbits und Bitfelder beeinflußt?

Der übliche Weg führt vom Port in die Arithmetik-Logik-Einheit (ALU) des Prozessors und von dort zurück zum Port. Die Bitbeeinflussung in der ALU beruht auf elementaren bitweisen Verknüpfungen:

- Setzen von Bits: ODER mit Einsen an den zu setzenden Positionen (sonst Nullen),
- Löschen von Bits: UND mit Nullen an den zu löschenden Positionen (sonst Einsen),
- wechseln (Invertieren) von Bits: XOR mit Einsen an den zu invertierenden Positionen (sonst Nullen),
- Eintragen eines Wertes in ein Bitfeld: erst Löschen des Feldes (UND mit Nullen in den Feldpositionen), dann Setzen der Einsen (ODER mit dem neuen Feldinhalt).

Auch Controller und Prozessoren mit mehr oder weniger komfortablen Bitbefehlen erledigen das so. Probleme bei Portzugriffen können deshalb zu Datenverfälschungen und somit zu Fehlern führen, die nur zeitweilig auftreten (nämlich dann, wenn tatsächlich abweichende Bitbelegungen entstehen) und nur schwer zu finden sind (datenabhängige Fehler).

Wann ist ungestörtes bitweises Modifizieren besonders wichtig? – Vor allem beim echten Multitasking, wenn verschiedene Tasks verschiedene Interfaces steuern, die an gemeinsame Ports angeschlossen sind. Beispiel: Task 1 steuert die Bits 2 und 5; Task 2 steuert die Bits 1, 6, und 7. Jede Task muß sich darauf verlassen können, daß sie „ihre“ Bits immer so vorfindet, wie sie sie hinterlassen hat.

Rücklesen der Portbelegung (1). Die Zeitfrage
Ausgegebene Signalbelegungen sind typischerweise nicht sofort zurücklesbar (lesen wir zu zeitig zurück, so lesen wir die alte Belegung). Hierfür gibt es zwei Ursachen:

- die kapazitive Belastung des Anschlusses (Schaltungsabhängig),
- die zur Synchronisation nötige Eintaktierungszeit (fest).

Beispiel:

Programmierabsicht:

SET Bit 4 (= OR 10H)
SET Bit 5 (= OR 20H)

Es soll erst Bit 4 eingeschaltet werden und einen Befehl später Bit 5. Der Befehl SET Bit 5 trifft aber auf die Ausgangsbelegung *vor* SET Bit 4, so daß Bit 4 als Null gelesen und beim Modifizieren wieder gelöscht wird (Abb. 2.3).

Abb. 2.3 Theorie und Praxis. a) Programmierabsicht; b) der tatsächliche Signalverlauf an den Anschlüssen

Die Anzahl der Takte, die zwischen Ausgabe und Rücklesen mindestens durchlaufen werden müssen, steht im Datenblatt. Wie schnell sich eine Änderung der Belegung des Daten-Latch am Anschluß bemerkbar macht (und somit zurückgelesen werden kann), hängt von der kapazitiven Belastung des Anschlusses ab. Bei hoher kapazitiver Belastung kann die zwischen Ausgabe und Rücklesen abzuwartende Zeit länger sein als eine Taktperiode (vor allem bei hohen Taktfrequenzen).

Abhilfe 1
Zwischen Ausgabe und Rücklesen Wartebefehle (einfachste Lösung: NOPs einfügen).

Praxistips:

1. Das Schaltverhalten an den Anschlüssen beobachten (Oszilloskop) - und lieber einen NOP zuviel einfügen als einen zuwenig.
2. Programmieren des Intervalls zwischen Schreiben und Lesen: nicht einfach NOPs hineinhacken, sondern einen Makro definieren (der nach Bedarf angepaßt werden kann - bei einem langsamen Prozessor wird er z. B. als ein einziger NOP implementiert, bei einem schnellen Prozessor hingegen als Warteschleife).

Abhilfe 2
Solche Sequenzen stets durch Schreiben ganzer Bytes programmieren[1]:

statt

    SET Bit 4
    SET Bit 5

also z. B.

    OUT 10H
    OUT 11H

Rücklesen der Portbelegung (2). Inhaltsverfälschungen
Wir wollen den Datenregisterinhalt modifizieren, können aber nur die Portbelegung an den Anschlüssen zurücklesen (vgl. Abb. 2.1). Wir nehmen an, daß nicht vorzeitig zurückgelesen wird. Trotzdem ist achtzugeben, und zwar abhängig von der eingestellten Übertragungsrichtung:

- wenn Ausgang, so ist es o. k. (Portbelegung = Registerinhalt),
- wenn Eingang, wird die Portbelegung gelesen. Sie wird so zum neuen Registerinhalt. Wird jetzt der Port zum Ausgang, so wird nicht ein ggf. zuvor geladener Registerinhalt wirksam, sondern die zuletzt gelesene Portbelegung.

Ist das tatsächlich ein Problem? – Es kommt drauf an:

- nein, wenn Eingänge für immer Eingänge bleiben,
- ja, wenn die Übertragungrichtung immer wieder umgestellt wird und wenn wir uns auf die im Datenregister befindliche Belegung verlassen. Beispiel: die Nachbildung des Open-Collector-Verhaltens an einem Tri-State-Port. Das Datenregisterbit ist stets Null, und das Richtungssteuerregisterbit bestimmt die Anschlußbelegung. Bit = High: Eingangsrichtung (Port hochohmig; Anschluß wird über externen Pull-up-Widerstand auf High gezogen). Bit = Low: Ausgangsrichtung; Anschluß wird mit der Null aus dem Datenregister belegt. Der naive Programmierer setzt das Datenregisterbit bei der Initialisierung auf Null und kümmert sich dann nie mehr darum. Nun stehe ein solcher Anschluß auf High (Eingangsrichtung). Jetzt finde – in ganz anderem Zusammenhang – ein Lesevorgang statt. Dann wird die besagte Null im Datenregister von außen mit einer Eins überschrieben werden. Infolgedessen wird unsere Open-Collector-Nachbildung nie mehr einen Low-Pegel ausgeben können...

Was wird bei Lesezugriffen wirklich gelesen?
Es gibt verschiedene Auslegungen:

- nur die Portbelegung (Beispiel: PIC 16C5x). Richtungsregister kann nicht modifiziert werde. Deshalb besondere Ladebefehle (PIC: TRIS).
- Richtungsregister und Portbelegung (Beispiel: diverse PIC-Typen). Erlaubt Modifizieren des Richtungssteuerregisters.
- Steuerung über Richtungssteuerregister: wenn Eingang, dann Portbelegung, wenn Ausgang, dann Register (Beispiel: Renesas H8/300H; vgl. Abb. 2.5). Beseitigt das Zeitproblem (unmittelbar zuvor geänderte Ausgangsbelegungen werden aus dem Register geholt, also nicht der Zeitverzögerung am Port unterworfen). Die Inhaltsverfälschungen (Überladen von Datenregisterpositionen mit Eingangsbelegungen) bleiben aber.
- Richtungssteuerregister, Datenregister und Portbelegung (Beispiel: Atmel AVR).

---

1) dazu müssen wir allerdings wissen, wie die anderen Bits belegt sind. Beim Multitasking funktioniert das nicht.

Nicht rücklesbare Register können nur insgesamt überschrieben, nicht aber bitweise in ihrem Inhalt geändert werden.

Abhilfe bei fehlender Rücklesbarkeit:
Software-Kopien der Registerinhalte verwalten. Zugriffe über Makros oder Unterprogramme.

Eine Auslegung, die diese Probleme vermeidet
Jeder Port muß drei Lesezugriffe unterstützen (vgl. Atmel AVR): vom Richtungssteuerregister, vom Datenregister und von den Anschlüssen (Abb. 2.4; vgl. auch Abb. 2.6). Zugriffe:

- zum Modifizieren der Signalflußrichtung: auf das Richtungssteuerregister,
- zum Modifizieren der Ausgangsbelegung: auf das Datenregister,
- zur Eingabe: auf die Anschlüsse.

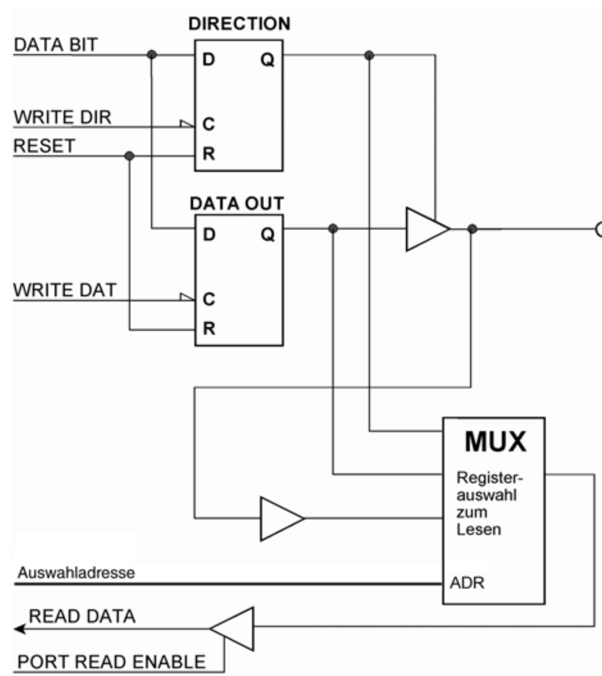


Abb. 2.4 Dieser E-A-Port unterstützt drei Lesezugriffe

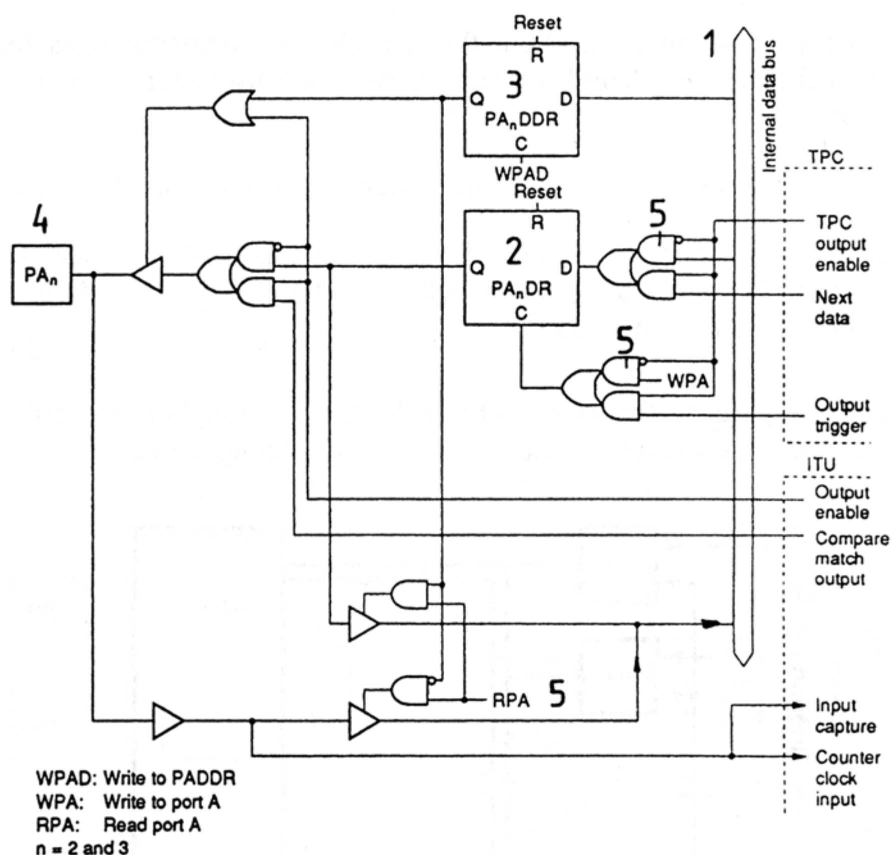
Alternative Auslegungen:

- befehlsgesteuertes Rücklesen (z. B. 8051). Was der Prozessor zurückliest, hängt vom jeweiligen Maschinenbefehl ab. Alle Befehle, die durch einen Read-Modify-Write-Ablauf gekennzeichnet sind (Tabelle 2.1), lesen das Datenregister, alle anderen die E-A-Anschlüsse.
- die Bitbeeinflussung erfolgt nicht über die ALU des Prozessors, sondern direkt im E-A-Port. Beispiel: die Mikrocontroller der Fa. Cyan unterstützen ein hardwareseitiges Modifizieren (Setzen/Löschen) der E-A-Bits.

Befehl	Wirkung
ANDL; ORL; XRL; CPL	logische Verknüpfungen UND, ODER, XOR, Negation
JBC	Verzweigen, wenn Bit = 1 und Bit löschen
INC, DEC	Erhöhen bzw. Vermindern um 1
DJNZ	Vermindern und verzweigen, wenn nicht Null
MOV Px.y,C	Übertrags-Flagbit in Bit y des Ports x schreiben
CLR Px.y; SET Px.y	Bit y des Ports x löschen bzw. setzen

Tabelle 2.1 8051-Befehle, die ein Lesen des Datenregisters bewirken

Weitere Peripherie
In modernen, mit reichhaltiger Peripherie ausgestatteten Mikrocontrollern hat nahezu jeder Port-Anschluß mehrere Nutzungsfälle (nutzt man beispielsweise einen Zeitgeber oder einen Impulsmustergenerator aus, so sind die diesen Anschlüssen zugeordneten Bitpositionen der universellen E-A-Ports nicht mehr verfügbar; Abb. 2.5). Ausgiebiges Handbuchstudium erforderlich!



1 - interner Datenbus; 2 - Datenregister; 3- Richtungsteuerregister; 4 - Tri-State-Treiberstufe mit E-A-Anschluß; 5 - Schaltmittel und Steuerwege für "gewöhnliche" E-A-Zugriffe (WPA - Write Port A, RPA - Read Port A; ITU - Integrated Timer Unit (Zeitgeberfunktionen); TPC - Timing Pattern Controller (Impulsmustergenerator)).

Abb. 2.5 Peripherieanschaltung an einen E-A-Anschluß. Ausführungsbeispiel: H8/3002, Port A, Anschlüsse PA_2, PA_3 (Hitachi)

Praxistip:
Im Problemfall Ports stets für die höher integrierte Funktion im µC ausnutzen, dafür die einfachen I/O-Funktionen auslagern. Ausnahme: wenn die internen Funktionseinheiten ohnehin unzureichend sind (z. B. Schnittstellencontroller oder A-D-Wandler).

Asynchrone Eingangssignale und Metastabilität
Kein Problem. Eingangsbelegungen werden in der Hardware synchronisiert. Synchronisations-Flipflops sind vorhanden, aber meist nicht dargestellt. Abb. 2.6 zeigt ein Beispiel *mit* solchen Flipflops..

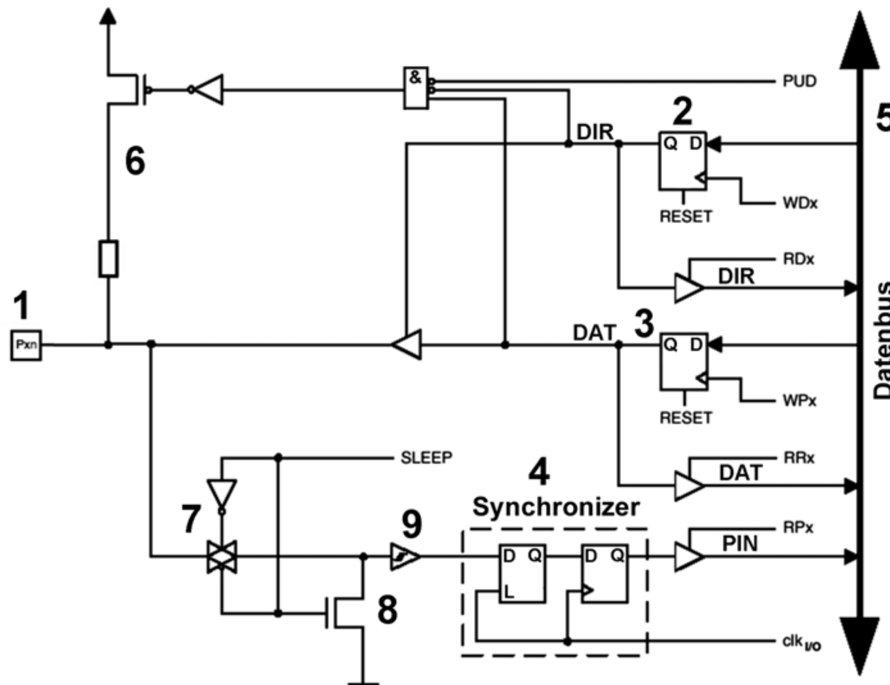


Abb. 2.6 E-A-Port mit Synchronisationsvorkehrungen (Atmel)

Die Abbildung veranschaulicht mehrere der zuvor erläuterten Schaltungsvorkehrungen. 1 - E-A-Anschluß (Pin); 2 - Richtungsregister; 3 - Datenregister; 4 - Synchronisationsstufe*) mit Latch und D-Flipflop; 5 - der interne Datenbus; 6 - schaltbarer Pull-up-Widerstand. Kann aktiviert werden, wenn die Bitposition als Eingang geschaltet, aber nicht belegt ist (offener Eingang). 7- Schalterelement; 8 - Pull-down-Transistor; 9 - Eingangspuffer. In Schlafzuständen (SLEEP) wird der Anschluß von der Innenschaltung abgetrennt (7) und der Eingangspuffer 9 mit dem Festwert Low belegt (8).

Partial Power Down (ausgeschalteter Controller in eingeschalteter Umgebung)
Achtung! - Es gibt Controllertypen, die im ausgeschalteten Zustand die Pins eigens hochohmig schalten, so daß das Problem nicht mehr besteht. Ist aber nicht immer der Fall.

Richtwerte zur Treibfähigkeit (bei VCC = + 5 V)
High: bis 3 mA, Low: 10 bis 20 mA (manche Ports). Auf das Kleingedruckte achten! (Stichwort: Zulässige Gesamtstrombelastung (über alle Ports.) Wichtig (Gegenkontrolle): Was darf über die Masse- und V_{CC}-Pins fließen?

Ungenutzte Ports:

- als Ausgänge und mit Festwert belegen (nur, wenn garantiert unbeschaltet),
- als Eingänge mit externem Pull-up,

- als Eingänge mit internen Pull-up's (diese ggf. scharfmachen),
- NIE als wirklich offene Eingänge (falls nicht ausdrücklich im Datenblatt als zulässig spezifiziert.)

2.2 Zweckgebundene E-A-Ports

Eine hinreichende Anzahl an Ein- und Ausgabeports bildet die Grundlage der anwendungsseitigen Schnittstellenentwicklung. Das gilt auch für das Anschließen von Leitungstreibern, Analog-Digital-Wandlern, Zeitstufen usw. Steht das Anwendungsvorhaben fest, so ist klar, wieviele Ein- und Ausgänge zu unterstützen sind. Es liegt nahe, Mikrocontroller einzusetzen, die eine ausreichende Anzahl an Ein- und Ausgängen haben. Dem stehen aber oftmals hohe Kosten entgegen, und für manche Anforderungen gibt es gar nichts Passendes. Viele Entwicklungsaufgaben sind deshalb mit vergleichsweise wenigen Interfacesignalen zu lösen (z. B. mit kostengünstigen Mikrocontrollern, die typischerweise 4...32 E-A-Anschlüsse haben).

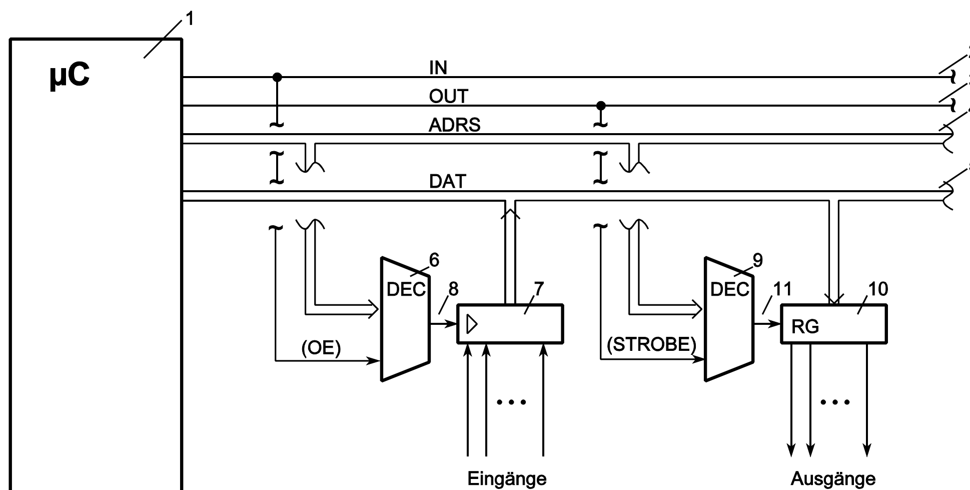
Ausgänge
Ausgänge sind typischerweise über Register bereitzustellen. Brauchen wir mehr Ausgänge, als das Interface Datensignale hat, bieten sich folgende Lösungen an:

- paralleler Bus, an den mehrere einzeln ladbare Register angeschlossen sind (Abb. 2.7),
- Schieberegister.

Eingänge
Eingangsbelegungen sind auf die Datenleitungen des Interfaces aufzuschalten. Brauchen wir mehr Eingänge, als das Interface Datensignale hat, bieten sich folgende Lösungen an:

- paralleler Bus. Aufschaltung nach dem Tri-State-Prinzip (Abb. 2.7).
- Abfrage über Schieberegister,
- Abfrage über Multiplexer (Abb. 2.8).

Erfordert das Interface eine synchrone Signalaufschaltung, so sind die Eingänge zunächst einzutaktieren (Synchronisation). Vor der eigentlichen Aufschaltung ist also ggf. ein taktflankengesteuertes Register anzuordnen.



1 - Mikrocontroller; 2 - Eingabesteuersignal (Strobe, Takt); 3 - Ausgabesteuersignal (Output Enable); 4 - Adreßbus; 5 - Datenbus; 6 - Adreßdecoder; 7 - Koppelstufe für Eingabe; 8 - Aufschaltsignal; 9 - Adreßdecoder; 10 - Ausgaberegister; 11 - Übernahmesignal.

Abb. 2.7 Anschluß mehrerer Einrichtungen an einen Port (Busprinzip)

Zunächst wird die jeweilige Adresse auf den Adreßbus 3 gelegt. Der weitere Ablauf hängt von der Art des Zugriffs ab:

Eingabe:
Der Mikrontroller belegt den Datenbus und gibt einen IN-Impuls ab. Erkennt der Adreßdecoder 6 die betreffende Adresse, so werden über die Koppelstufe 7 die betreffenden Eingänge auf den Datenbus 5 geschaltet.

Ausgabe:
Der Mikrontroller schalten den Datenbus auf Eingabe um (Bus wird hochohmig) und gibt einen OUT-Impuls ab. Erkennt derAdreßdecoder 9 die betreffende Adresse, so wird die Belegung des Datenbus 5 in das Ausgaberegister 11 übernommen. Die Belegung der angeschlossenen Ausgangsleitungen entspricht dann der besagten Datenbusbelegung.

Zu Position 10: Register oder Koppelstufe?
Das hängt davon ab, ob der Datenport des Mikrocontrollers metastabile Zustände verträgt oder nicht. Wenn nein, dann ein Register. Wenn ja, kann ggf. auch eine Koppelstufe gewählt werden.

Achtung: Beim Aufschalten von Bustreibern Buskonflikte vermeiden (Richtungssteuerung).

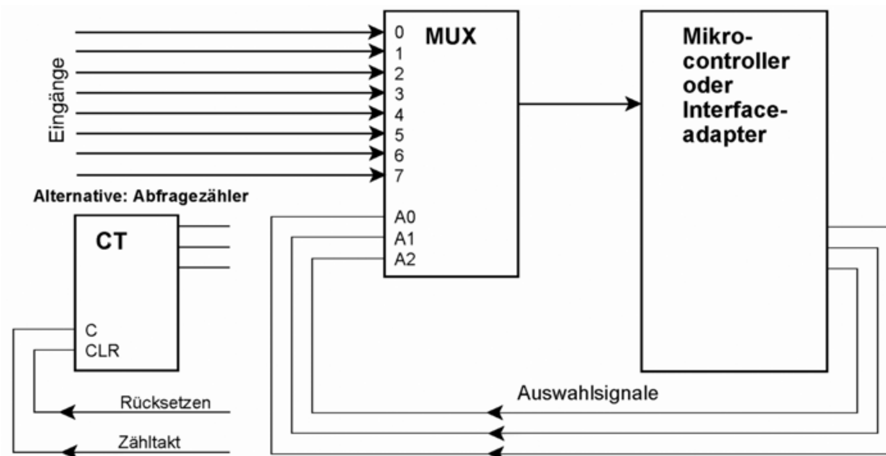


Abb. 2.8 Informationseingabe über Multiplexer

Eine Multiplexeranordnung zum Abfragen von n Bits braucht ld n Adreßeingänge, die von einem Ausgabeport zu treiben sind (Auswahlsignale). Alternative: ein externer Adreßzähler (Abfragezähler). Zu dessen Ansteuerung benötigen wir lediglich ein Takt- und ein Rücksetzsignal.

Wie aufwendig ist eine bestimmte Art der Schnittstellenerweiterung? – Wichtige Auswahlkriterien:

- die Anzahl der zur Ansteuerung erforderlichen Anschlüsse (Pin Count),
- die Zugriffszeiten,
- Anzahl und Kosten der erforderlichen Schaltkreise.

Die Wichtigkeit entspricht typischerweise der Reihenfolge unserer Aufzählung, denn meist denkt man über solche Erweiterungsmaßnahmen deshalb nach, um mit einem kleinen Mikrocontroller auszukommen. Die Anschlußzahl hat also oft Vorrang.

Tabelle 2.2 enthält eine Gegenüberstellung typischer Erweiterungsprinzipen. Dabei beziehen wir uns auf naheliegende, einfache Auslegungen.

paralleler Bus (Ein- und Ausgabe)	Schieberegister (Ein- und Ausgabe)	Multiplexer (nur Eingabe)
Datenleitungen gemäß Zugriffsbreite, Adreßleitungen gemäß ld Ports[1)], Lesesteuerleitung (RD), Schreibsteuerleitung (WR)	4 (Dateneingang, Datenausgang, Takt, Übernahme)	1 Dateneingang, Adreßleitungen gemäß ld Bits[2)] Mit externem Abfragezähler: 3 (Dateneingang, Takt Rücksetzen)
Beispiel: der Zugriff auf 32 Bitpositionen erfordert...		
8 • Daten + 2 • Adresse + 2 Steuerung = 12 Leitungen	4 Leitungen	1 • Daten + 5 • Adresse = 6 Leitungen. Mit externem Abfragezähler 3 Leitungen
Zeitbedarf (auf das Beispiel bezogen)		
1 Byte je Zeitintervall[3)] (4 Zeitintervalle)	1 Bit je Zeitintervall[3)] (32 Zeitintervalle)	1 Bit je Zeitintervall[3)] (32 Zeitintervalle)

1): Zweierlogarithmus der Portanzahl; 2): Zweierlogarithmus der Bitanzahl; 3): gemeint ist die Zeit, die ein einzelner programmseitiger Zugriff erfordert (das Ausgeben eines Bytes, das Schieben eines Bits usw.)

Tabelle 2.2 E-A-Schnittstellenerweiterungen im Vergleich

Sehr breite Anwendungsschnittstellen (Assembly/Disassembly)

Gelegentlich sind viele Signalleitungen anzusteuern oder abzufragen. Eine Grundfrage: ist es zulässig, das gleichsam stückweise zu erledigen (z. B. Byte für Byte) oder müssen alle Signalbelegungen auf einmal gestellt oder abgeholt werden? – Ist die stückweise Erledigung zulässig, genügen die bisher angesprochenen Erweiterungslösungen. Ansonsten müssen wir uns etwas einfallen lassen (Abb. 2.9 und 2.10).

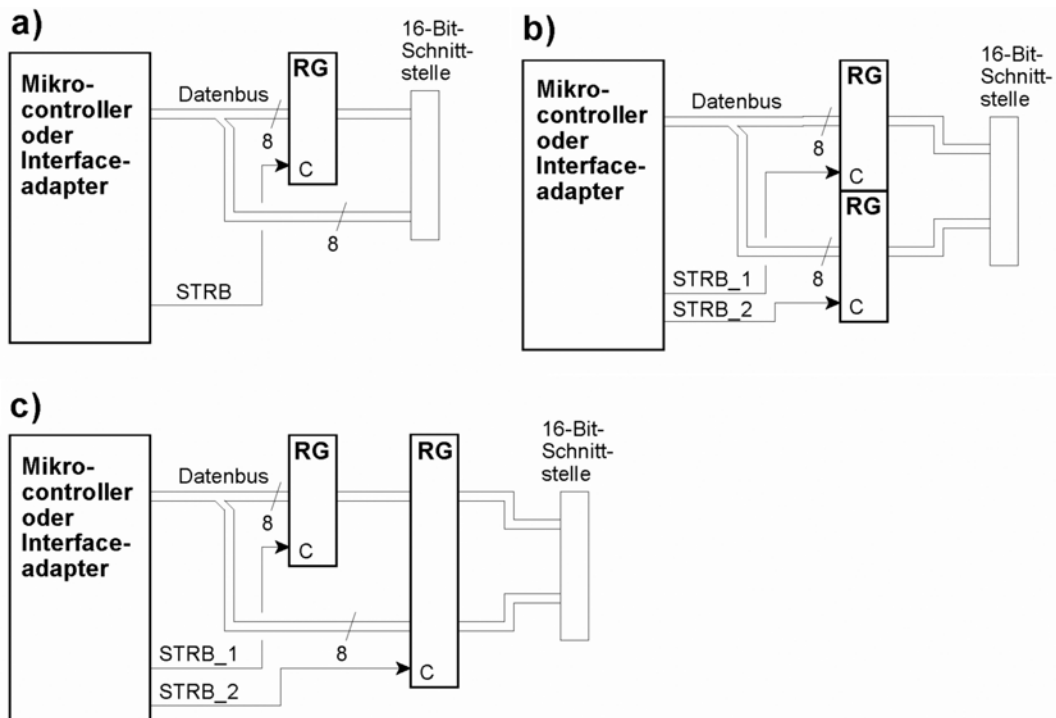


Abb. 2.9 Porterweiterungen (1). Ausgabe (Beispiele)

a) 16-Bit-Schnittstelle mit 8-Bit-Port. 8 Bits werden aus einem Pufferregister geliefert, 8 weitere Bits direkt vom Port. Laden des Registers über programmierbares Strobe-Signal. Problem: um das Register zu laden, muß der Port mit dem Registerinhalt belegt werden. Somit ändert sich die Belegung der verbleibenden 8 Bits.

b) 16-Bit-Schnittstelle mit 8-Bit-Port. Zwei Register. Belegung bleibt außen stabil, erscheint aber in 8-Bit-Abschnitten mit zeitlichem Versatz.

c) 16-Bit-Schnittstelle mit 8-Bit-Port und Assembly-Funktion. Außen erscheinen die 16-Bit-Belegungen stets auf einen Schlag (kein Zeitversatz). Nutzung: (1) das 8-Bit-Register laden, (2) die verbleibenden 8 Bits auf den Port geben und das 16-Bit-Register laden.

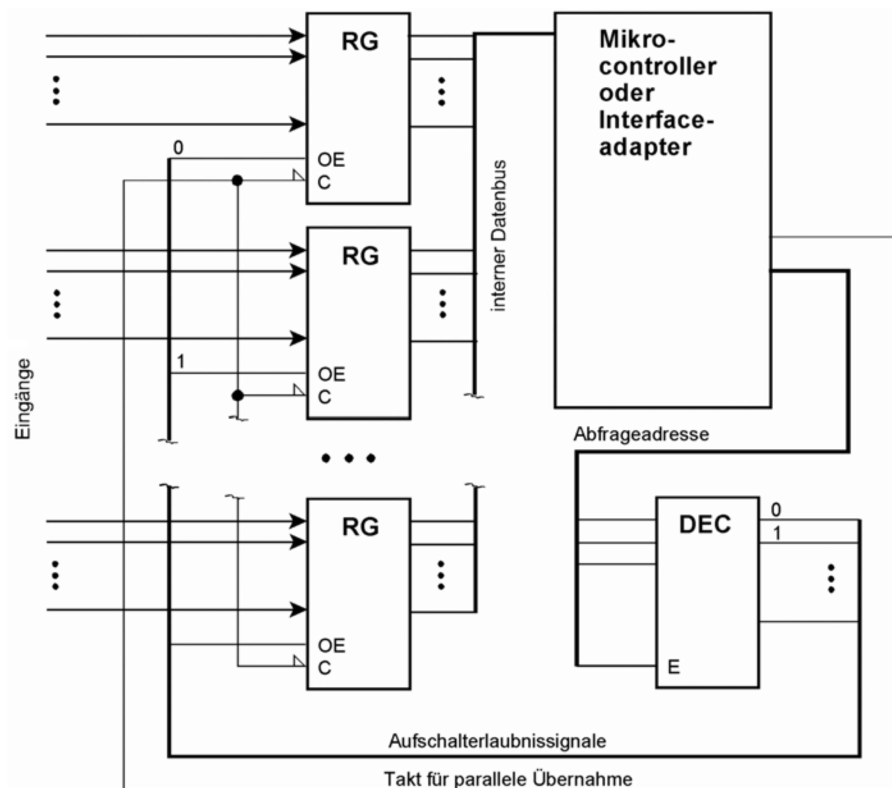
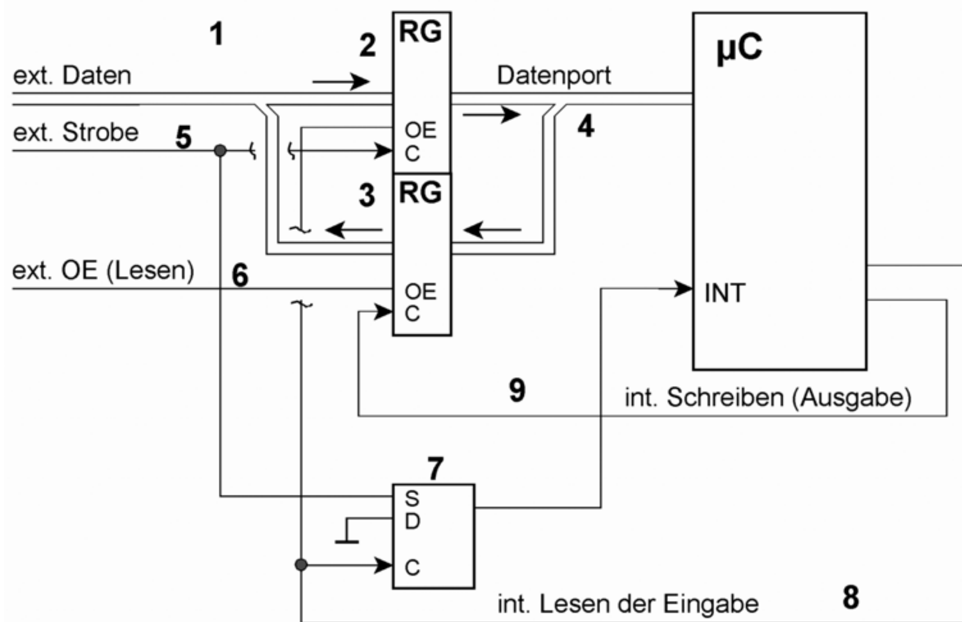


Abb. 2.10 Porterweiterungen (2). Eingabe (Beispiel)

Die eingangsseitige Datenbelegung wird in alle Register parallel übernommen. Anschließend können die Registerinhalte einzeln über den internen Datenbus gelesen werden (Disassembly-Funktion).

Zugriffe von außen steuern (SlavePorts)

Manchmal werden von anderen Einrichtungen Signalbelegungen angefordert oder geliefert, und zwar in einem Zeitraster, dem keine Software gewachsen ist (die entsprechenden Erlaubnis- oder Strobe-Signale sind vielleicht 20...200 ns lang aktiv). Beispiele: Anschluß an den ISA-Bus, an die Parallelschnittstelle oder an einen Prozessorbus. Eine entsprechende E-A-Schnittstelle muß sich gegenüber der Anwendungsseite so verhalten wie ein Register oder wie ein Speicherschaltkreis – mit anderen Worten: wie eine Slave-Einrichtung an einem üblichen Bussystem. Einige Mikrocontroller haben derartige Slave-Ports. Gibt es keine, müssen die Funktionen mit Buskoppelschaltkreisen (oder in einer CPLD) nachgebildet werden (Abb. 2.11).



1 - externer Datenbus (bidirektional); 2 - Eingaberegister; 3 - Ausgaberegister; 4 - interner Datenbus (bidirektional); 5 - externe Strobeleitung (Übernahmetakt); 6 - externe Leseerlaubnisleitung (Datenaufschaltung); 7 - Interruptanforderungsflipflop; 8 - Eingabelesesignal; 9 - Ausgabeschreibsignal.

Abb. 2.11 Ein Slave-Port

Eingabe:
Strobeleitung 5 bewirkt Datenübernahme nach Register 2. Dabei wird zugleich Flipflop 7 gesetzt. Das löst einen Interrupt im Mikrocontroller aus. Der Interruptbehandler liest den Inhalt des Registers 2, indem er den Bus 4 auf Eingabe schaltet und das Lesesignal 8 aktiviert. Hierdurch wird zugleich das Flipflop 7 gelöscht.

Ausgabe:
Der Mikrocontroller stellt die auszugebende Daten im Register 3 bereit (Bus 4 auf Ausgabe, Datenübernahme mittels Schreibsignal 9). Die angeschlossene Einrichtung liest die Daten durch Erregen der Leitung 6.

2.3 Elementare Buskoppelschaltkreise

Trotz des heutigen Standes der Schaltungsintegration werden Buskoppelschaltkreise nach wie vor in großen Stückzahlen eingesetzt. Typische Gründe dafür:

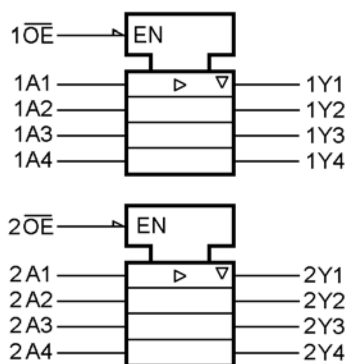
- besondere Forderungen an die Treibfähigkeit,
- besondere anwendungsseitige Forderungen. Beispiele: Ziehen/Stecken von Einrichtungen während des Betriebs (Hot Plugging), Schutz der funktionellen Schaltkreise, Pegelanpassung, vereinfachte Fehlerbeseitigung – ein über den Bus verursachter Ausfall betrifft nur den (kostengünstigen) Koppelschaltkreis, nicht aber die (teuren) funktionellen Schaltkreise.
- Zusammenfügen verschiedener Technologien, Aufbau der Einrichtung aus verschiedenartigen Bauelementen (wie bei kleinen bis mittleren Stückzahlen allgemein üblich).
- Kostensenkung.

Es liegt nahe, Typen zu bevorzugen, die bewährten Industriestandards entsprechen (Abb. 2.12 bis 2.14) und ggf. kleinere Nachteile (im Vergleich zu diesem oder jenem Exoten) in Kauf zu nehmen.

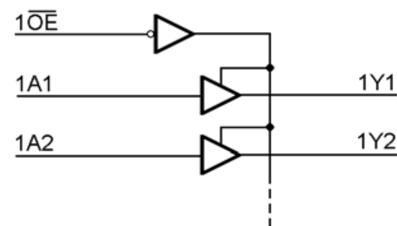
Woran erkennen wir, ob ein Schaltkreis ein „Industriestandard" ist oder nicht? – Industriestandard-Typen

- gibt es seit einiger Zeit,
- werden in nahezu allen Baureihen angeboten,
- werden auch für neue Baureihen angekündigt,
- haben in den Typenliste der Hersteller keine einschränkenden Vermerke,
- werden in Massenprodukten eingesetzt (z. B. auf Speichermoduln),
- sind bisweilen auffallend preisgünstig.

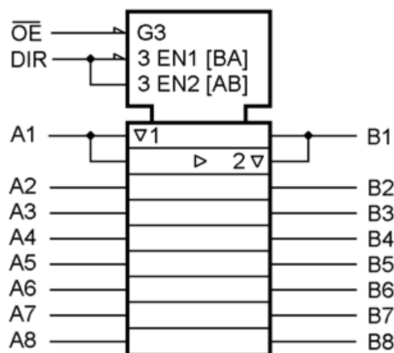
a) Bustreiber (Puffer) '244



Der Schaltkreis enthält zwei Blöcke zu vier Tri-State-Stufen mit gemeinsamem Erlaubnissignal (Enable).



b) Transceiver '245



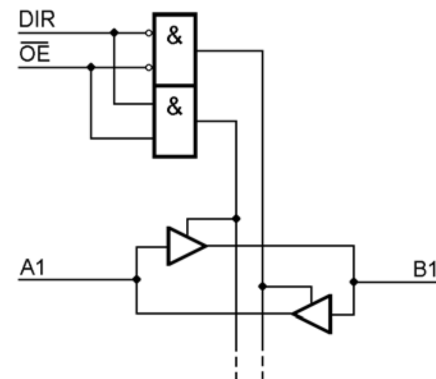


Abb. 2.12 Herkömmliche Buskoppelschaltkreise (Beispiele)

a) Bustreiber (Puffer). Eine Signalflußrichtung (von A nach Y). A - Dateneingänge; Y - Ausgänge (Busanschlüsse); OE bzw. EN - Erlaubniseingänge (betreffen je 4 Datensignale). Bei aktivem Erlaubnissignal werden die zugehörigen Datensignale zum Bus durchgeschaltet. Ist das Erlaubnisignal inaktiv, sind die zugehörigen Ausgänge (Y) hochohmig.

b) bidirektionale Koppelstufe (Transceiver). Eine von zwei Signalflußrichtungen wählbar (Tabelle 2.3). Beide Seiten A, B, haben Tri-State-Anschlüsse. OE - Erlaubniseingang; DIR - Richtungssteuereingang (Direction Control Input).

DIR	OE	Wirkung
0	0	Richtung von B nach A; A-Ausgänge aktiv
0	1	Richtung von B nach A; A-Ausgänge hochohmig
1	0	Richtung von A nach B; B-Ausgänge aktiv
1	1	Richtung von A nach B; B-Ausgänge hochohmig

Tabelle 2.3 Zur Wirkungsweise der Steuersignale des Transceivers ‘245

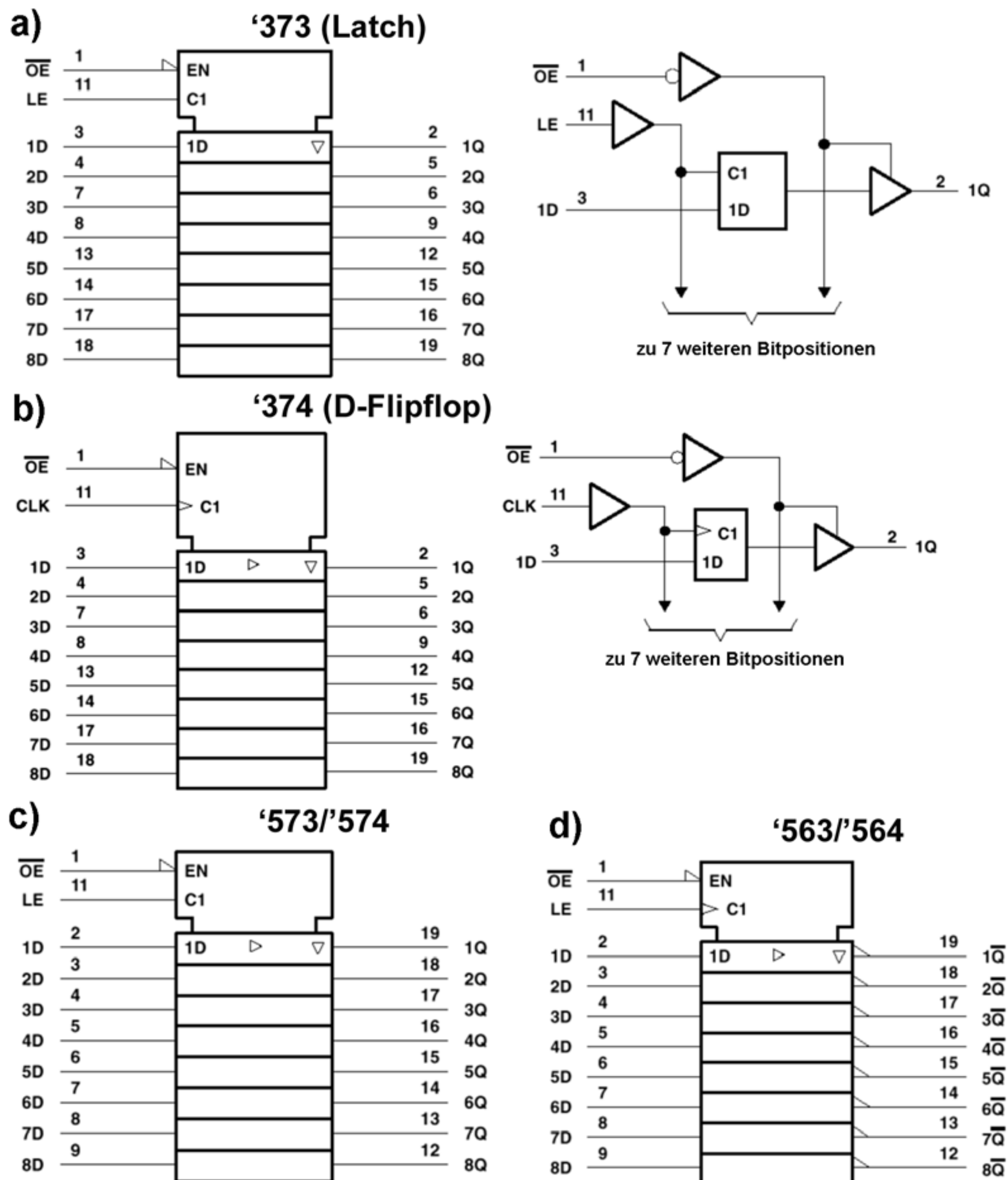


Abb. 2.13 Typische Register - eine kleine Auswahl. a) - Latch-Register; b) - D-Flipflop-Register; c) - Register mit anderer Anschlußbelegung; d) - Register mit negierten Ausgängen.

Die Schaltkreise haben Tri-State-Ausgänge. Braucht man zweiwertige Ausgänge, ist der Erlaubniseingang (OE) fest mit Masse zu verbinden. Gängige Breiten (Anzahl der Bitpositionen): 8, 16, 18, 20, 32, 36. Es gibt verschiedene Anschlußbelegungen (Pinouts). Bei a) und b) sind Ein- und Ausgänge gleichsam gemischt (Anschluß 2 = Ausgang, Anschluß 3 = Eingang usw.), bei c) und d) liegen sich hingegen Ein- und Ausgänge auf beiden Seiten gegenüber (1, 2 usw. sind Eingänge, 19, 18 usw. sind Ausgänge). Der Zweck: die Auslegung kostengünstiger Leiterplatten zu erleichtern[1)].

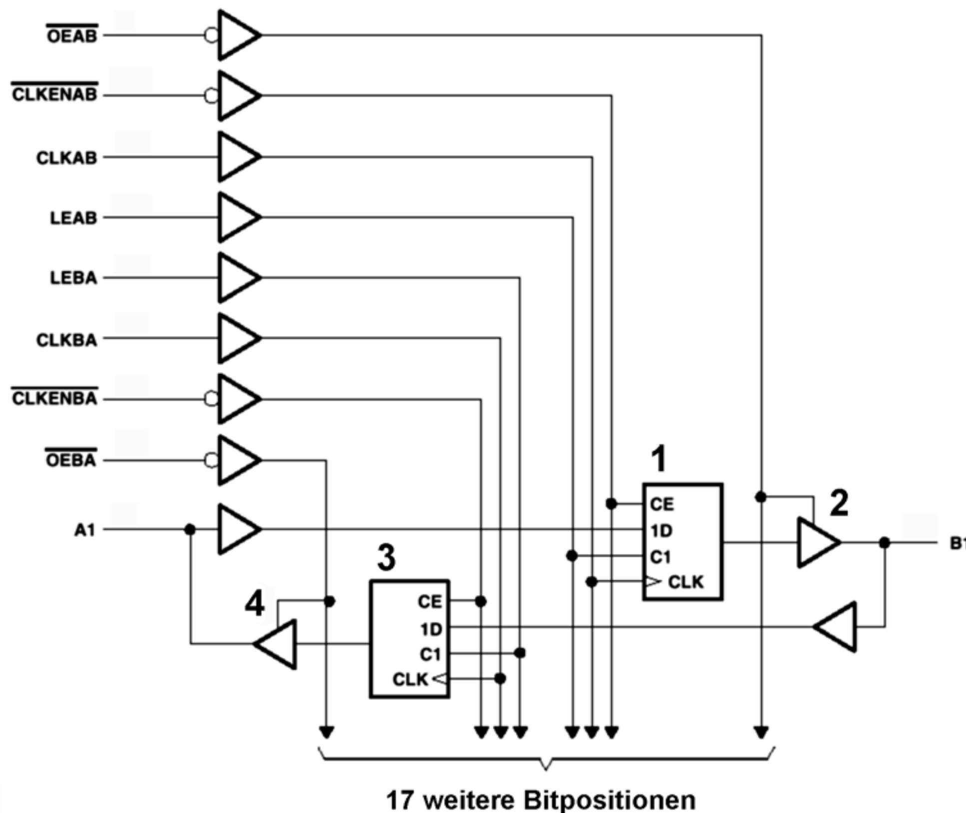


Abb. 2.14 Eine Stufe eines Universal Bus Transceivers (74LVT16601; Texas Instruments)

Ein Schaltkreis des angegebenen Typs enthält insgesamt 18 solcher Stufen. Die Steuereingänge sind allen Stufen gemeinsam. 1 - Datenspeicher für Richtung A - B; 2 - Tri-State-Treiber für B-Anschluß; 3 - Datenspeicher für Richtung B - A; 4 - Tri-State-Treiber für A-Anschluß.

Die Funktionsweise wollen wir zunächst anhand der Signalflußrichtung von links nach rechts (von A nach B) beschreiben (A ist Eingang, B Ausgang). OEAB - Output Enable Richtung A-B; CLKENAB - Clock Enable Richtung A-B; CLKAB - Takt Richtung A-B; LEAB - Latch Enable Richtung A-B. Sinngemäß erklären sich die Steuersignale der Gegenrichtung (B-A)

Aktivieren und Deaktivieren des Ausgangs (B): mittels OEAB (Low: Ausgang aktiv, High: Ausgang hochohmig).

Verhindern, daß der Eingang (A) über den Weg B - A beeinflußt wird: OEBA auf High (deaktiviert Treiber 4).

1) einfache Strukturen – z. B. vom Mikrocontroller zum Adreß-Latch, vom Adreß-Latch zum Speicher (vgl. 8051) – passen bei Anschlußbelegung c) auf eine einzige Leiterplattenebene.

Steuerung der Betriebsweise des Datenspeichers 1: über LEAB:

- LEAB = Low: Datenspeicher wirkt als D-Flipflop. Übernimmt mit der Low-High-Flanke des Taktes CLKAB die Belegung des A-Eingangs (Registerfunktion). Damit CLKAB wirksam werden kann, muß CLKENAB = Low sein (Übernahmesteuerfunktion).
- LEAB = High: Datenspeicher wirkt als Latch oder als einfache Durchreiche.

Durchreiche (keine Speicherfunktion): Solange LEAB = High ist, wirkt der Datenspeicher als Durchreiche.

Latch-Funktion: Wird LEAB von High auf Low geschaltet, so wirkt der Datenspeicher als Speicherglied und hält die Eingangsbelegung zum Zeitpunkt der High-Low-Flanke von LEAB (Verhalten eines transparenten Latches). In dieser Betriebsart darf CLKAB bei LEAB = Low nicht wirksam werden (sonst: Informationsübernahme mit Low-High-Flanke).

Durch entsprechendes Beschalten der Steuereingänge können viele der gängigen Buskoppelstufen nachgebildet werden. Beispiele:

1. Verhalten ähnlich '244 (einfache Durchreiche mit Tri-State-Ausgängen)
OEAB wirkt als OE. LEAB und OEBA fest auf High. Restliche Steuersignale auf beliebige Festwerte.

2. Verhalten ähnlich '245 (bidirektionaler Treiber)
OEAB und OEBA wirken als Erlaubnissignale auf der B- bzw. auf der A-Seite. LEAB und LEBA fest auf High. Restliche Steuersignale auf beliebige Festwerte. Wird eine zum '245 kompatible Steuerung mit DIR und OE gewünscht, sind OEAB und OBEA über UND-Gatter gemäß Abb. 5.87b anzusteuern.

3. Verhalten ähnlich 373/573 (Latch-Register)
OEAB wirkt als OE. OEBA fest auf High. Datenübernahme mit LEAB (wirkt als Takt). Restliche Steuersignale auf beliebige Festwerte.

4. Verhalten ähnlich 374/574 (D-Flipflop-Register)
OEAB wirkt als OE. OEBA fest auf High. Datenübernahme mittels Takt an Takteingang CLKAB. Übernahmesteuerung muß aktiv sein. Dazu CLKENAB auf Low. CLKENAB kann als Erlaubnissignal verwendet werden (vgl. Eingang CE in Abb. 5.23b). LEAB auf Low. Restliche Steuersignale auf beliebige Festwerte.

2.4 Das Schieberegister als Universalinterface

Das Schieberegister hat einen bedeutsamen Vorteil: man kann es so lang auslegen wie man will und braucht, um Bits zu transportieren, nur einen Dateneingang, einen Datenausgang und einen Takt. Da man mit den übertragenen Bits natürlich noch etwas anfangen will, braucht man zusätzlich irgendwelche Steuersignale. Dafür genügt eine einzige Leitung: wird sie aktiviert, so wird die eingeschobene Information zwecks Ausgabe übernommen und durch einzugebende Information ersetzt, die dann nur noch herausgeschoben werden muß (Abb. 2.15).

Die einzelnen Einrichtungen enthalten Schieberegisteranordnungen mit parallelen Ein- und Ausgängen. Die auszugebende Information wird Bit für Bit durchgeschoben. Anschließend wird STROBE erregt. Dies bewirkt zum einen das Laden der Parallelregister mit den eingeschobenen Daten (Ausgabe) und zum anderen das Laden der Schieberegister mit den Daten der jeweiligen Umgebung (Eingabe). Mit weiteren Schiebetakten wird dann die übernommene Information Bit für Bit herausgeschoben.

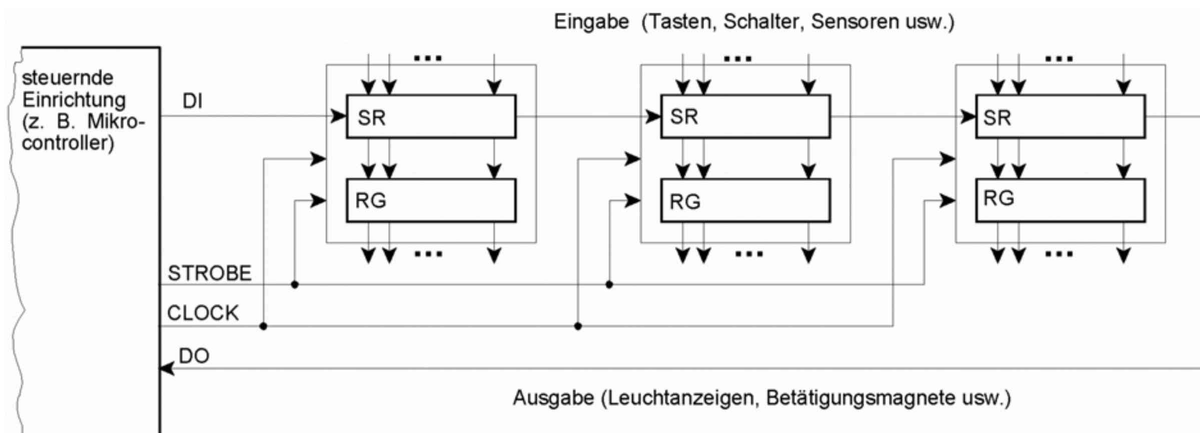


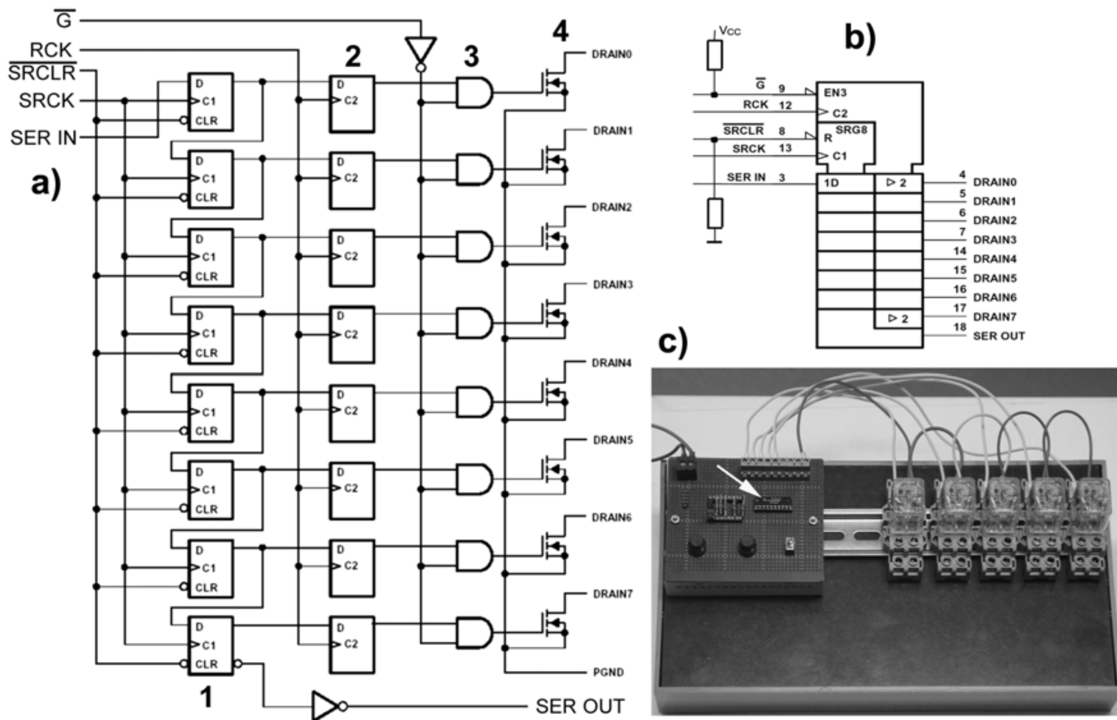
Abb. 2.15 Ein einfaches Schieberegister-Interface. SR - Schieberegister; RG - Parallelregister (Halteregister); DI - Dateneingang; DO - Datenausgang; CLOCK - Schiebetakt; STROBE - Parallelübernahmeimpuls

Es gibt viele Schaltkreise mit solchen Schnittstellen. Einige sind als Privatlösungen anzusprechen (Abb. 2.16, 2.17), einige haben den Charakter von Industriestandards. Die wichtigsten: I^2C-Bus, Microwire und SPI (Tabelle 2.4, Abb. 2.18 bis 2.24).

Die Anordung von Abb. 2.16 entspricht dem in Abb. 2.15 dargestellten Prinzip. Einschieben der Bits über SER IN mit Schiebetakt SRCK. Parallelübernahme mit RCK. Es können mehrere Schaltkreise hintereinandergeschaltet werden (Verbindung SER OUT - SER IN). Anwendungpraktisch wichtig ist, daß kein Leistungs-FET unberechtigt aktiv werden darf (z. B. nach dem Rücksetzen). Deshalb gibt es eine direktwirkende Sperrmöglichkeit über die Gatter 4. Sperrsignal G wird über Pull-up-Widerstand auf High gezogen und erst nach vollständiger Initialisierung programmseitig aktiviert. Zudem kann das Schieberegister gelöscht werden (Eingang SCLR). Achtung: Das Halteregister 2 läßt sich nicht direkt löschen. Vor der Freigabe der Sperrgatter 4 ist zunächst der Inhalt des gelöschten Schieberegisters 1 ins Halteregister 2 zu übernehmen (Impuls auf RCK).

Es gibt verschiedene Abwandlungen des einfachen Schieberegisterprinzips von Abb. 2.15. Abb. 2.17 zeigt ein Beispiel. Das Ziel der Entwickler: mit nur zwei Signalleitungen (Takt- und Datenleitung) auszukommen. Hier hat man das Start-Stop-Prinzip der seriellen Schnittstelle aufgegriffen. Das erste gesendete Bit wird als Startbit interpretiert. Dann folgen die eigentlichen Informationsbits (hier: 35). Die Takte werden intern mitgezählt. Nach dem 36. Takt gelangt der Schaltkreis wieder in den Ausgangszustand (internes Rücksetzen) und wartet auf das nächste Startbit.

Hinweis:
Bei derartigen Interfaces auf die Impulsbreiten bzw. Taktfrequenzen achten (Datenblatt). Der Schaltkreis von Abb. 2.16 kommt noch mit 20 ns breiten Impulsen zurecht (entspricht einem Schiebetakt von 25 MHz), der Schaltkreis von Abb. 2.16 hingegen hat eine maximale Taktfrequenz von nur 500 kHz. Naiv programmierte Ausgabeschleifen sind typischerweise zu schnell (auch in „langsamen“ Mikrocontrollern)...



a) - Schaltbild (nach Texas Instruments); b) - Schaltsymbol mit Widerstandsbeschaltung; c) - Anwendung als Relaistreiber (Labormuster). 1 - Schieberegister; 2 - Halteregister; 3 - Sperrgatter; 4 - Leistungs-FETs.

Abb. 2.16 Leistungschaltkreis mit Schieberegisterinterface

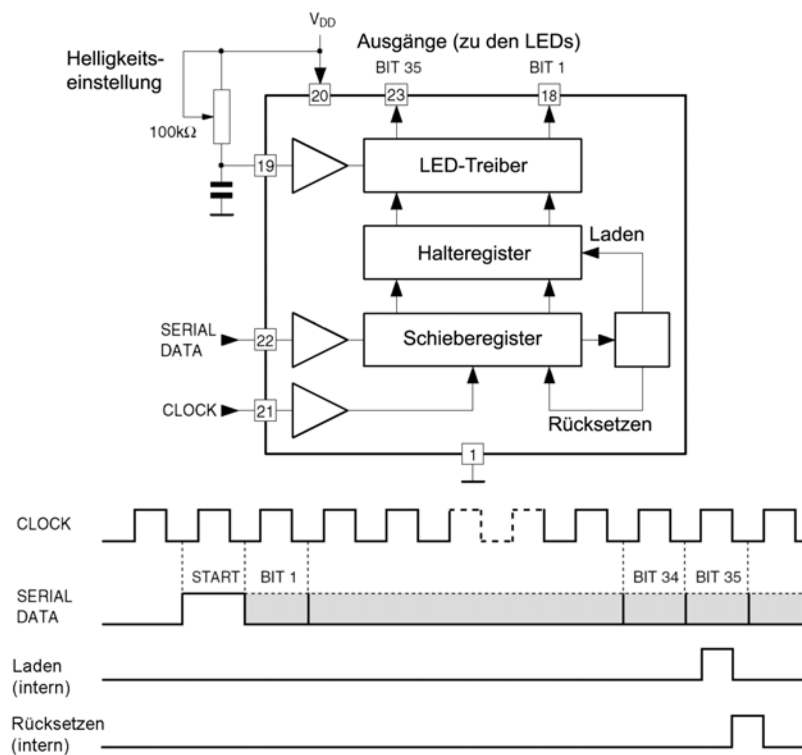
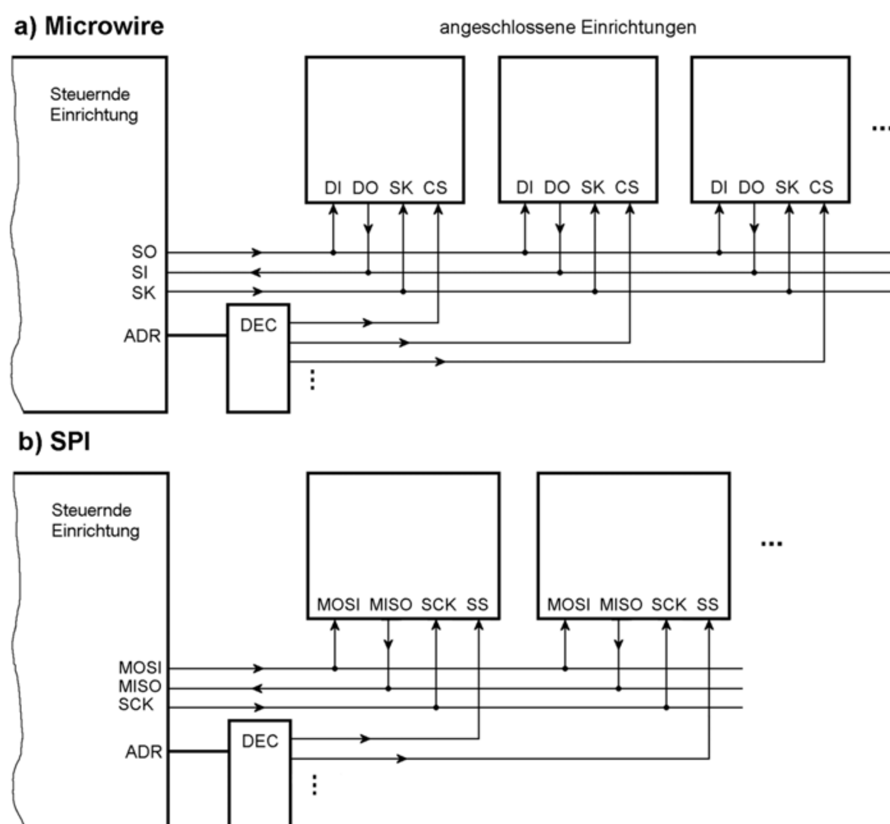


Abb. 2.17 LED-Treiber MC5451 (nach ST Microelectronics)

	I^2C	Microwire	SPI
Entwicklungsfirma	Philips	National Semiconductor	Motorola
Anzahl der Signalleitungen	2	3 bzw. 4*)	3 bzw. 4*)
Architektur	Multimaster-Bus	Bus mit zentralem Master	Bus mit zentralem Master
Schaltkreisauswahl	über Adreßdecodierung	zentrale Auswahl über CS-Eingang	zentrale Auswahl über CS-Eingang
Taktfrequenz**) (Richtwerte)	100 bzw. 400 kBits/s, 3,4 MBits/s	1...3 MHz	1...5 MHz
Signalprotokolle	komplizierter	einfacher	einfacher

*): 3 Interfaceleitungen + 1 Schaltkreisauswahlsignal.**): entspricht der maximalen Datenrate in Bits/s. Betriebsspannungsabhängig (je geringer die Betriebsspannung, desto geringer die spezifizierte maximale Taktfrequenz)

Tabelle 2.4 Industriestandards im Überblick



a) Microwire: Dateneingang DI, Datenausgang DO, Schiebetakt SK und Schaltkreisauswahl CS. Es ist möglich, DI und DO zu einer bidirektionalen Datenleitung zusammenzuschalten.

b) SPI: Dateneingang MOSI, Datenausgang MISO; Schiebetakt SCK, Schaltkreisauswahl SS. (MOSI = Master Out/Slave In; MISO = Master In/Slave Out; SCK = Shift Clock; SS = Slave Select.)

Abb. 2.18 Microwire und SPI

Microwire

Die zentrale Steuerung wählt jeweils eine angeschlossene Einrichtung aus, indem sie die betreffende CS-Leitung aktiviert. Auszugebende Information wird über DO geliefert und in der ausgewählten Einrichtung mit der Vorderflanke der SK-Impulse übernommen; einzugebende Information wird über DI mit SK-Taktimpulsen Bit für Bit abgeholt. Das Schieben ist byteweise organisiert. Was die einzelnen Bytes bedeuten, ist schaltkreisspezifisch.

SPI

Der jeweilige Slave wird durch Aktivierung des Eingangs SS ausgewählt. Der Datentweg zwischen Master und ausgewähltem Slave ist eine Ringstruktur aus zwei Schieberegistern (Abb. 2.19, 2.20).

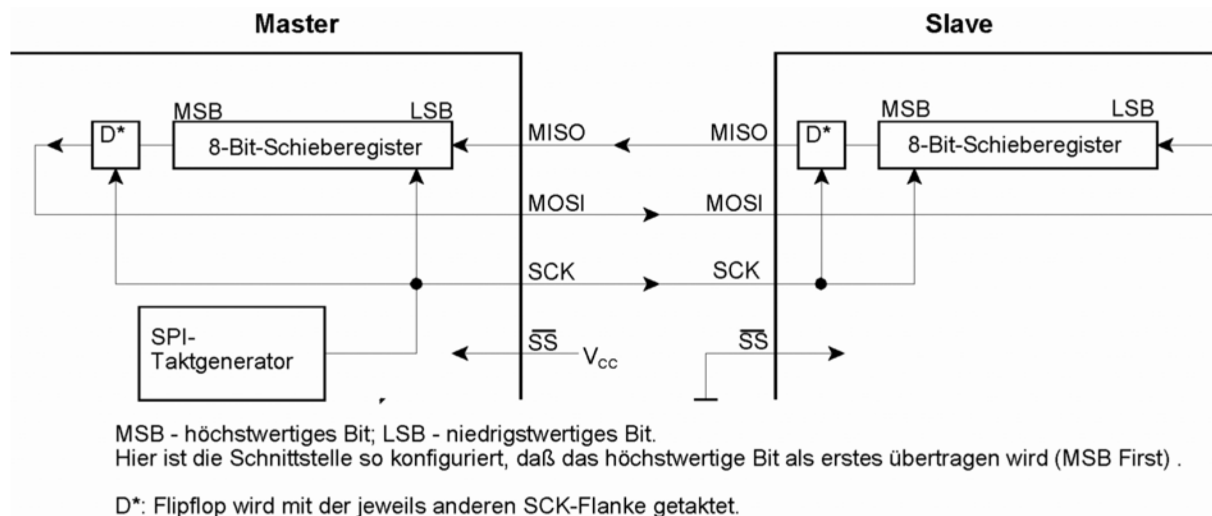


Abb. 2.19 SPI: der Datenweg zwischen Master und Slave

Abb. 2.19 zeigt eine Art Ersatzschaltung, um die typischen Besonderheiten zu veranschaulichen. Es wird byteweise geschoben. Das Byte im Schieberegister des Masters gelangt in den Slave, während gleichzeitig das im Schieberegister des Slaves stehende Byte in den Master geschafft wird. Dabei wird mit der einen Taktflanke der auf der jeweiligen Datenleitung anliegende Wert übernommen und in das jeweils empfangende Schieberegister eingeschoben, mit der anderen wird das jeweils nachfolgende Bit ausgeschoben. Diese Betriebsweise vermeidet Takttoleranzprobleme – deshalb kann SPI mit höheren Taktfrequenzen betrieben werden als Microwire.

Die SPI-Hardware in einem Mikrocontroller ist typischerweise als autonom arbeitende State Machine ausgelegt, die – von der Software gesteuert – wahlweise als Master oder als Slave betrieben werden kann. Die Konfigurationsmöglichkeiten sind schaltkreisspezifisch. Programmseitig einstellbar sind u. a.:

- das Übertragungsformat: Einzelbyteübertragung (nach Übertragung eines jeden Bytes wird SS wieder inaktiv) oder Mehrbyteübertragung (aufeinanderfolgende Bytes werden übertragen, während SS aktiv bleibt),
- die Polarität des SCK-Signals (aktiv High oder aktiv Low),
- die Schiebeordung (höchstwertiges oder niedrigstwertiges Bit zuerst).

Multi-Master-Betrieb

SPI kann als echter Bus betrieben werden, auf den sich mehrere Einrichtungen wahlweise aufschalten können. Die Mastervermittlung muß aber gesondert verwirklicht werden. (Typischerweise kann man den SS-Anschluß im Master-Betrieb so konfigurieren, daß dessen Aktivierung das Freigeben des Busses veranlaßt.)

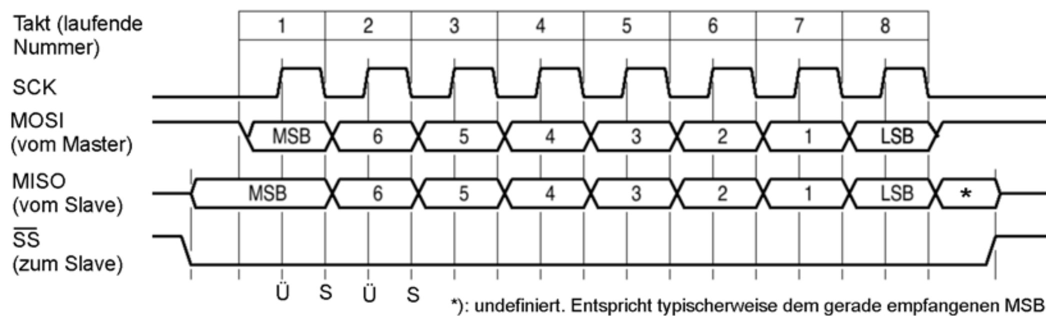


Abb. 2.20 Datentransport über SPI (nach Motorola). Ü - Datenübernahme; S - Ausschieben

Wir zeigen hier nur einen der möglichen Betriebsfälle. Das Schieben eines Bytes erfordert 8 Takte. Zuvor ist SS zu aktivieren und nach dem Schieben wieder zu deaktivieren. Datenübernahme mit der Low-High-Flanke von SCK, Weiterschieben mit der High-Low-Flanke. Die letzte High-Low-Flanke bewirkt im Slave ein weiteres Schieben, in dessen Ergebnis MISO typischerweise mit dem ersten der zuvor eingeschobenen Bits belegt wird.

Der I^2C-Bus

I^2C = IIC = Inter Integrated Circuit. Das Bussystem wurde entwickelt, um Schaltkreise auf einer größeren Leiterplatte auf kostengünstige Weise miteinander zu verbinden. Der I^2C-Bus verbindet mehrere Slave-Einrichtungen mit einer steuernden Einrichtung (dem Master). Es gibt nur zwei Leitungen:

- Serieller Takt SCL (Serial Clock). Diese Leitung wird ausschließlich vom Master angesteuert.
- Serielle Daten SDA (Serial Data). Diese Leitung ist bidirektional; je nach Übertragungsrichtung wird sie vom Master oder vom jeweils ausgewählten Slave angesteuert.

Treiberstufen

Die Treiberstufen des I^2C-Bus sind Open-Drain-Ausgänge. Deshalb sind die Busleitungen mit Pull-up-Widerstände beschaltet (z. B. mit 4,7 kΩ). Typischer Treiberstrom: 3 mA.

Geschwindigkeit

Typischerweise liegt die maximale Taktfrequenz bei 100 kHz (praxisüblich: bis zu 80 kHz). Zudem gibt es zwei weitere Betriebsarten: (1) bis 400 kHz (Fast Mode), (2) bis 3,4 MHz (High Speed Mode). Der I^2C-Bus ist voll statisch (also mit beliebig geringer Taktfrequenz) betreibbar.

Slave-Auswahl (1). 7-Bit-Adressierung

Das erste vom Master gesendete Byte (Steuerbyte) enthält die Slave-Adresse und die Zugriffskennung (Abb. 2.21). Mit den 7 Bits der Slave-Adresse könnte man theroetisch 128 Einrichtungen adressieren. Einige Belegungen sind aber reserviert, und für bestimmte Arten von Einrichtungen (z. B. für serielle EEPROMs) sind bestimmte Adreßformate festgelegt (Abb. 2.21b).

Slave-Auswahl (2). 10-Bit-Adressierung

Dem Steuerbyte folgt ein weiteres Adreßbyte nach. Die Bits 7...3 des Steuerbytes kennzeichnen die Adressierungsweise (Belegung mit Festwert 11110B), die Bits 2 und 1 enthalten zwei der 10 Adreßbits.

Signalfolgen und Zugriffsabläufe

Tabelle 2.5 und Abb. 2.22 veranschaulichen die elementaren Signalfolgen auf den beiden Busleitungen. Anhand von Abb. 2.23 wollen wir das Grundsätzliche der Zugriffsabläufe erläutern. Unser Beispiel: ein serieller EEPROM.

a) allgemeine Struktur

7			**Slave-Adresse**			1	0
							R/W

b) Beispiel (serielle EEPROMs)

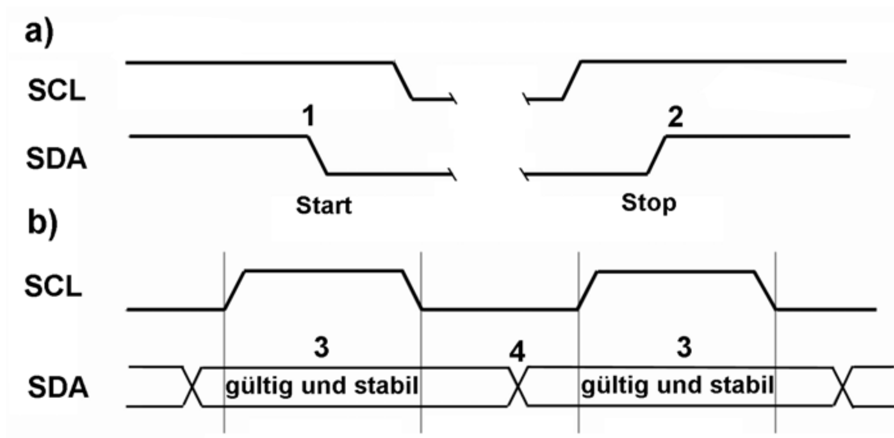
Typkennung				**Slave-Adresse**			**R/W**

Abb. 2.21 Das Steuerbyte

Die Slave-Adresse wählt die Einrichtung (= einen Schaltkreis) aus, die Typkennung kennzeichnet die Art der Einrichtung, die Zugriffskennung (R/W) bestimmt die Richtung der Datenübertragung (0 = Schreiben, 1 = Lesen).

Signalfolge	Leitung SCL	Leitung SDA
Start	High	Flanke High => Low
Stop	High	Flanke Low => High
Datenübertragung	Flanke Low => High	Datenbit (muß während der Low-High-Flanke des Taktes stabil anliegen)
Bestätigung (Acknowledge)	Low-High-Flanke (9. Takt der Datenübertragung)	wird von der jeweils empfangenden Einrichtung (Master oder Slave) auf Low gezogen

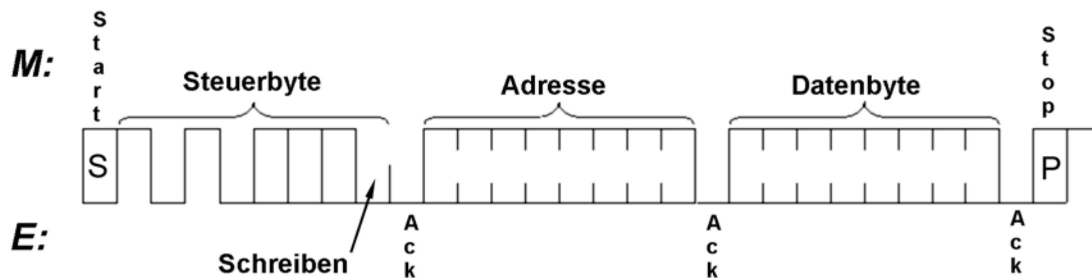
Tabelle 2.5 Elementare Signalfolgen des I^2C-Bus



a) - Start- und Stopsignalisierung; b) - Gültigkeit der Datenbelegung in Bezug auf den Takt. 1 - Start = SDA-Flanke von High nach Low bei SCL = low ; 2 - Stop: SDA-Flanke von Low nach High bei SCL = Low. 3 - Datenbelegung liegt stabil an, wenn SCL = High; 4 - Datenbelegung darf sich nur dann ändern, wenn SCL = Low.

Abb. 2.22 Elementare Signalfolgen des I^2C-Bus

a) Schreiben



b) Lesen

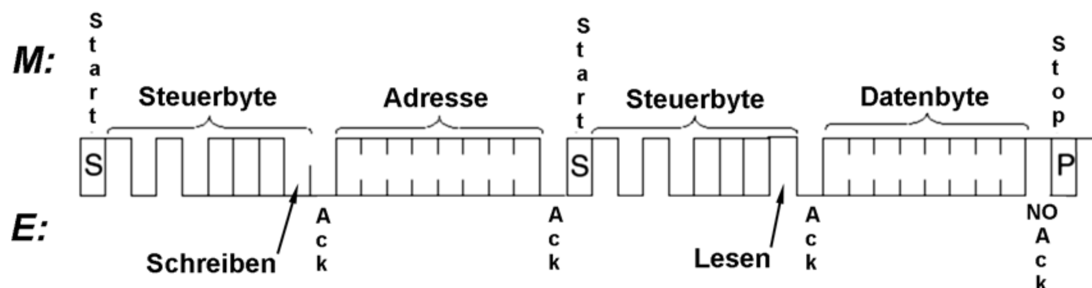


Abb. 2.23 Elementare Zugriffsabläufe am Beispiel eines seriellen EEPROMs (nach Microchip)

Es sind die einzelnen Bitpositionen des seriellen Datenstroms auf der Leitung SDA dargestellt. Jeder Bitposition entspricht ein Taktimpuls auf der Leitung SCL. M - Aktivitäten des Masters; E - Aktivitäten des EEPROMs; Start (S) - Startbedingung; Ack - Bestätigung (Acknowledge); No Ack - keine Bestätigung; Stop (P) - Stopbedingung.

a) Schreiben
Nach der anfänglichen Startbedingung kommt das Steuerbyte. R/W = 0 zeigt einen Schreibzugriff an. Das Steuerbyte (Schreibkommando) wird von EEPROM bestätigt (Acknowledge). Darauf folgt die Adresse. Bei entsprechend geringer Speicherkapazität genügt ein Adreßbyte (EEPROMs mit größerer Speicherkapazität erfordern 2 Adreßbytes). Der Empfang des Adreßbytes wird vom EEPROM bestätigt (Acknowledge). Dann folgt das Datenbyte. Nach dessen Bestätigung (Acknowledge) sendet der Master eine Stopbedingung. Daraufhin wird im EEPROM der Programmiervorgang ausgelöst.

b) Lesen
Das Lesen beginnt mit einem Schreibkommando, um die Adresse in den EEPROM zu schaffen. Nach der Bestätigung (Acknowledge) der Adreßübertragung sendet der Master eine weitere Startbedingung und ein Steuerbyte mit R/W = 1 (Lesekommando). Nach dessen Bestätigung beginnt der EEPROM, das Datenbyte zu senden. Es gibt zwei Ablaufvarianten:

- Lesen eines einzelnen Bytes (Random Read). Der Master bestätigt den Empfang des Datenbtes *nicht* (No Acknowledge), hält also während der betreffenden Taktperiode SDA auf High, und sendet dann eine Stopbedingung (vgl. Abb. 2.23b).
- Fortlaufendes Lesen (Sequential Read). Der Master bestätigt den Empfang des ersten Datenbytes (Acknowledge). Darauf folgt sofort das zweite usw. Jede weitere Bestätigung führt zum Liefern eines weiteren Datenbytes. Um das Lesen zu beenden, muß der Master auf das jeweils letzte Datenbyte so antworten, wie in Abb. 2.23b gezeigt (keine Bestätigung, dann Stopbedingung). Die EPROMs haben Adreßzähler, die den gesamten Speicher fortlaufend adressieren (man könnte also den vollständigen Speicherinhalt im Rahmen eines einzigen Lesekommandos abholen).

Schieberegisterinterfaces selbstgebaut
Das Nachbauen von SPI, I^2C, IEEE 1149 usw. lohnt sich meist nicht (zu kompliziert). Die typische Eigenentwicklung beruht meist auf dem einfachen Prinzip von Abb. 2.15. Manchmal sind weitere Vereinfachungen möglich.

Schieberegisterschaltkreise der Logikbaureihen
Das herkömmmliche Sortiment ist reichhaltig (Tabelle 2.6). Schieberegister sind heutzutage allerdings eher Exoten (oft teurer, nur in wenigen Baureihen erhältlich). Manche Typen (vor allem der LS-Baureihe) sind für Neuentwicklungen nicht zu empfehlen (veraltet, keine garantierte Verfügbarkeit).

Typ	Ausführung	Typ	Ausführung
74x164	8 Bits, parallele Ausgänge	74x195	4 Bits, parallele Eingänge, parallele Ausgänge
74x165	8 Bits, parallele Eingänge	74x595	8 Bits, parallele Ausgangs-Latches
74x166	8 Bits, parallele Eingänge	74x597	8 Bits, parallele Eingangs-Latches
74x194	4 Bits, parallele Eingänge, parallele Ausgänge	74LS671/ 672	4 Bit parallele Eingänge, paralleles Ausgangsregister

Tabelle 2.6 Schieberegisterschaltkreise (Auswahl)

Schieberegister mit Buskoppelschaltkreisen
Eine Alternative: wir verwenden „Industriestandard“-Buskoppelschaltkreise mit D-Flipflop-Registern (Abschnitt 2.3) und bilden den Schiebeweg auf der Leiterplatte nach (Abb. 2.24 bis 2.26).

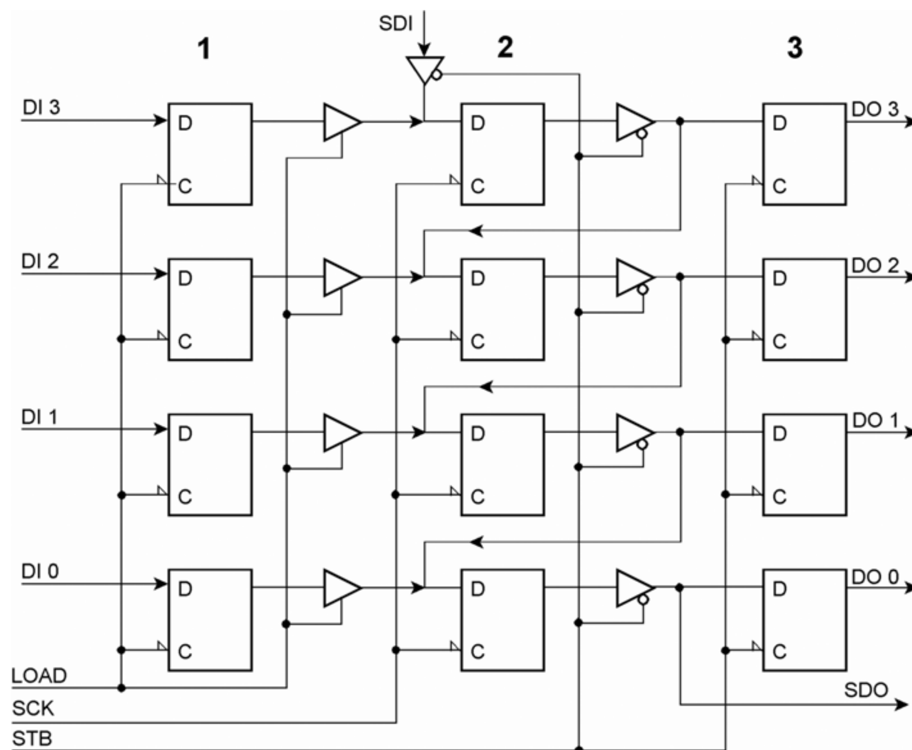
Parallele Ausgabe
Kein Problem, da alle Ausgänge zugänglich sind. Ggf. wird ein weiteres Register als paralleles Ausgaberegister angeschlossen.

Parallele Eingabe
Es liegt nahe, 2-zu-1-Multiplexer einzusetzen (Umschaltung zwischen Schieben und Parallelübernahme). Das erfordert aber vergleichsweise viele Schaltkreise. Womöglich günstiger: Tri-State-Koppelstufen oder entsprechende Register.

Schieberegister in CPLDs
Ein Flipflop kostet eine ganze Makrozelle. Die Parallelübernahme von Eingangssignalen bedeutet praktisch keinen Zusatzaufwand, ein paralleles Ausgangsregister erfordert hingegen eine weitere Makrozelle je Bitposition (Abb. 2.27). Wichtig: soviel wie möglich mit einem gemeinsamem Takt erledigen (ähnlich JTAG 1149, nur einfacher)[1)].

Auf das parallele Ausgangsregister verzichten
Das ist dann möglich, wenn es nicht stört, daß die nachgeordneten Schaltungen den Schiebevorgang mitbekommen. Anwendungsbeispiel: LED-Anzeigen mit Einzelansteuerung (kein Multiplexbetrieb). Voraussetzungen: (1) es wird immer nur dann eine neue Belegung eingeschoben, wenn sich die Anzeige ändert (Sache der Anwendungsprogrammierung), (2) die Änderungen kommen nicht allzu oft vor (das ist z. B. dann der Fall, wenn sich Änderungen der Anzeige auf Grund von Bedienhandlungen ergeben).

1) individuelle Takte belegen zumeist weitere Ressourcen in den Makrozellen.



1 - Eingaberegister; 2 - Schieberegister; 3 - Ausgaberegister. SCK = Schiebetakt; LOAD = Parallelübernahme der Eingangsbelegung; STB = Parallelübernahme der Ausgangsbelegung.

Abb. 2.24 Schieberegisteranordnung mit Ein- und Ausgaberegistern

Um die Schaltung einfach zu halten, werden zwei Steuerleitungen verwendet. Der Schiebeweg ist über Tri-State-Stufen geführt. Beim Schieben sind LOAD und STB = Low. Somit sind die Tri-State-Treiber des Schiebergisters 2 aktiv. Nach dem Schieben schalten wir STB auf High. Damit wird die Schieberegisterbelegung ins Ausgaberegister 3 übernommen, und die Tri-State-Stufen des Schieberegisters 2 werden freigegeben. LOAD bewirkt, daß die Eingangsbelegung ins Eingaberegister 1 übernommen wird und daß dessen Tri-State-Koppelstufen aktiviert werden. Ein Impuls auf SCK veranlaßt die Übernahme ins Schieberegister 2. Abschließend wird zunächst LOAD und dann STB auf Low gebracht. Nun kann ein neuer Schiebevorgang beginnen.

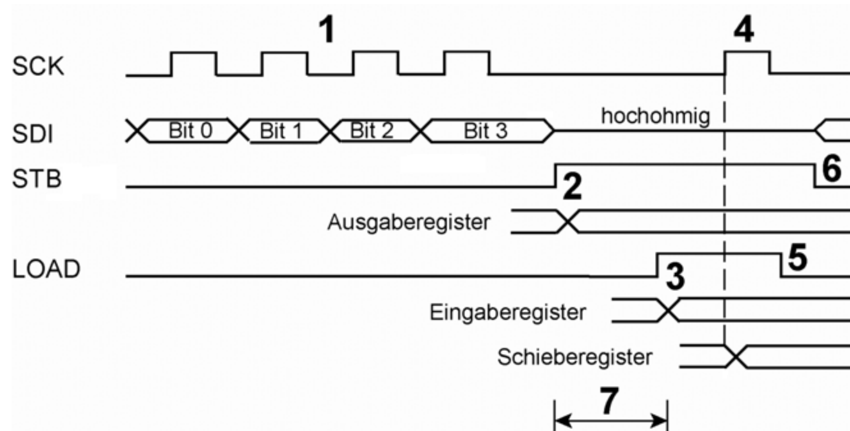


Abb. 2.25 Ein Schiebeablauf

Erklärung zu Abb. 2.25:
1 - Einschieben von vier Bits; 2 - Übernahme ins Ausgaberegister. Tri-State-Koppelstufen des Schieberegisters werden hochohmig. 3 - Übernahme der Eingangsbelegung ins Eingaberegister. Dessen Tri-State-Koppelstufen werden aktiv. 4 - Übernahme der Eingangsbelegung ins Schieberegister. 5 - Deaktivieren der Tri-State-Koppelstufen des Eingangsregisters. 6 - der Schiebeweg wird wieder aktiviert. 7 - wir bringen erst die Ausgänge zur Wirkung (2) und holen die Eingänge später ab (3). Somit entspricht die Eingangsbelegung bereits der Reaktion auf die soeben eingestellte Ausgangsbelegung. Wir können sie also schon im folgenden Schiebevorgang abholen. Um diesen Effekt auszunutzen, muß ein hinreichend langer Abstand zwischen STROBE und LOAD vorgesehen werden.

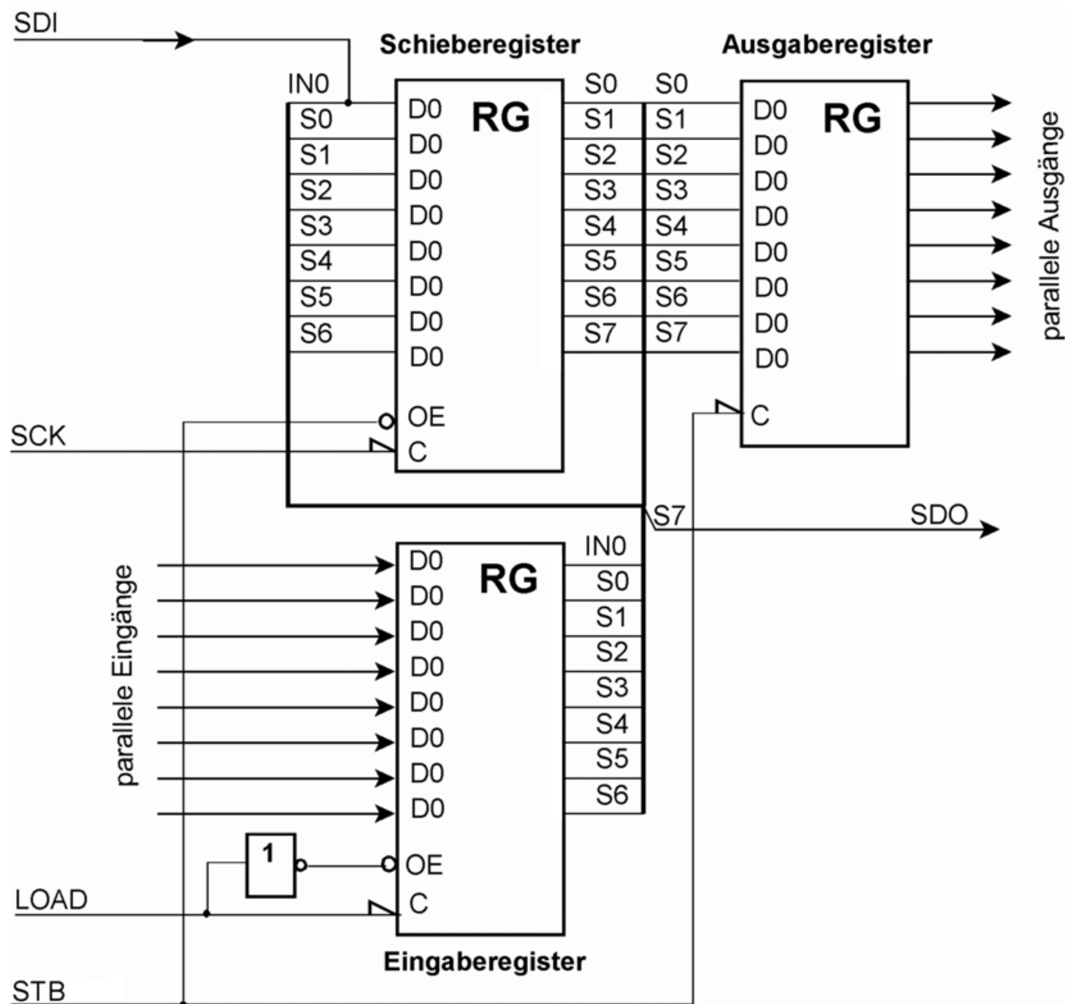


Abb. 2.26 Schieberegisteranordnung mit drei universellen Registerschaltkreisen

Das Prinzip von Abb. 2.24 wurde hier mit drei D-Flipflop-Registern (z. B. 74x374) verwirklicht. Mit modernen universellen Bustreibern lassen sich bei gleicher Schaltkreiszahl mehr E-A-Anschlüsse (z. B. 18) vorsehen (vgl. Abb. 2.13). Im Vergleich zu „echten" Schieberegistern brauchen wir zwar mehr Schaltkreise, es sind aber Typen aus der Massenfertigung, die es in nahezu allen Logikbaureihen gibt – auch in den ganz modernen. Somit lassen sich ggf. Probleme der Pegelwandlung, des Partial Power Down usw. durch Auswählen entsprechender Typen lösen. Hinweis: In der Schaltung von Abb. 2.26 muß der steuernde Mikrocontroller dafür sorgen, daß die Leitung SDI hochohmig geschaltet wird (vgl. die Positionen 2 und 6 in Abb. 2.25) – es sei denn, man nutzt das erste (an SDI angeschlossene) Flipflop des Schieberegisters nicht zu Eingabezwecken.

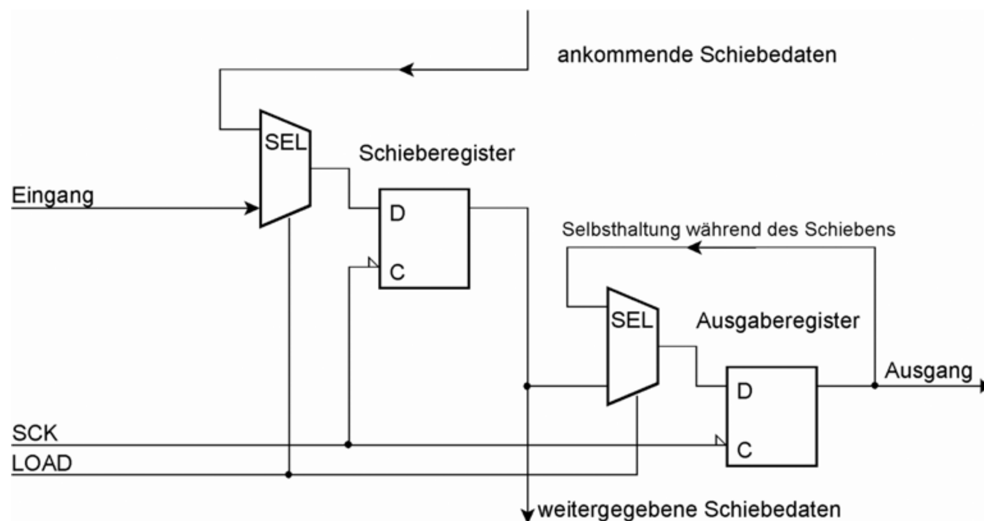


Abb. 2.27 Eine Registerposition in einem CPLD

SCK ist das einzige Taktsignal. Bei LOAD = Low ist der Schiebeweg aktiv, und die Belegung des Ausgaberegisters wird über die Rückführung gehalten. Ist LOAD = high, so bewirkt ein Impuls auf SCK die Übernahme der Schieberegisterbelegung ins Ausgaberegister und die Übernahme der Eingangsbelegung ins Schieberegister. Hier wird – wie bei Boundary Scan – die jeweils vorhergehende Eingangsbelegung übernommen, so daß, um die Reaktion auf die aktuelle Ausgabe abzuholen, ein weiterer Schiebeablauf erforderlich ist.

2.5 Interfaceanschlüsse trickreich ausnutzen

Oft geht es darum, mehrere Einrichtungen anzuschließen, dabei aber mit einer möglichst geringen Zahl an Interfacesignalen auszukommen. In den Applikationsschriften der Mikrocontroller-Hersteller, in einschlägigen Zeitschriften usw. finden wir hierzu eine Vielzahl von Anregungen. Eine naheliegende systematische Vorgehensweise:

- wir schließen zunächst die „dickste" Einrichtung – jene mit den meisten Interfacesignalen – an,
- wir überlegen, wie wir diese Signale anderweitig ausnutzen können. Die Mehrfachnutzung beruht auf der Tatsache, daß es verschiedene Arten von Interfacesignalen gibt (Abb. 2.28).

In vielen Fällen können wir folgende Arten von Interfacesignalen erkennen:

- Datensignale,
- Adreß- und Auswahlsignale,
- echte Steuersignale. Das sind solche, die tatsächlich irgendwelche Vorgänge (Informationsübernahme, Aufschaltung usw.) auslösen (typische Bezeichnungen: Clock, Enable, Strobe, Write, Chip Select usw.).
- Rückmeldungen (Zustands- und Fehleranzeigen, Interruptsignale usw.).

Das Prinzip der Mehrfachnutzung: sind die echten Steuersignale inaktiv, so dürfen die anderen Signale beliebig schalten. Abb. 2.29 zeigt ein entsprechendes Beispiel.

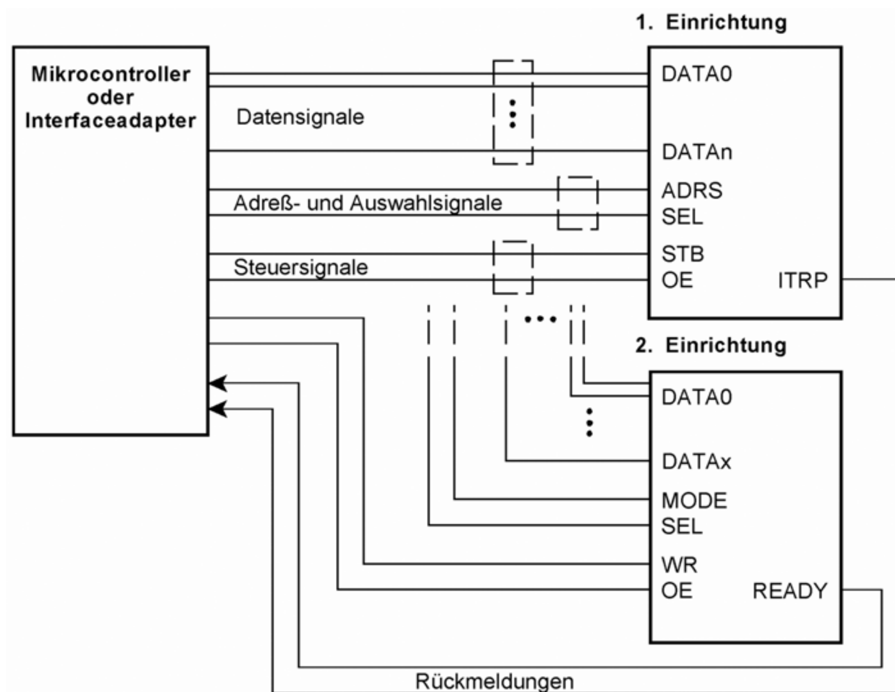


Abb. 2.28 Verschiedene Arten von Interfacesignalen

Praxistips:

1. Datenwege gemeinsam nutzen. Eine Art Privatbus aufbauen.
2. Adreß- und Auswahlsignale mehrfach nutzen.
3. Nur die echten Steuersignale müssen für jede Einrichtung gesondert vorgesehen werden.

Rückmeldungen
Im Prinzip ist ein Zusammenfassen über Auswahlschaltungen (Multiplexer) oder eine Art Bussystem möglich. Es kommt aber darauf an, wie sich die Signale verhalten und wie wir sie auswerten (direkte Wirkung auf die Hardware[1)], Auslösung von Interrupts, programmseitige Abfrage).

Praxistips:

1. Soll der Mikrocontroller oder Interfaceadapter mit mehreren Einrichtungen gleichzeitig arbeiten, müssen jene Signale, die direkt auf die steuernde Hardware einwirken (Wartezustände, Handshaking, Interruptauslösung usw.), einzeln angeschlossen werden.
2. Arbeitet der Mikrocontroller oder Interfaceadapter zu einer Zeit nur mit einer einzigen Einrichtung, wäre eine Zusammenfassung möglich.
3. Bei rein programmseitiger Abfrage (Polling) ist die Zusammenfassung stets möglich.
4. Das Zusammenfassen lohnt sich nur dann, wenn hierdurch tatsächlich weniger Anschlüsse benötigt werden (an die erforderlichen Auswahlsignale denken...)

1) z. B. zum Hervorrufen von Wartezuständen oder in Handshaking-Signalspielen.

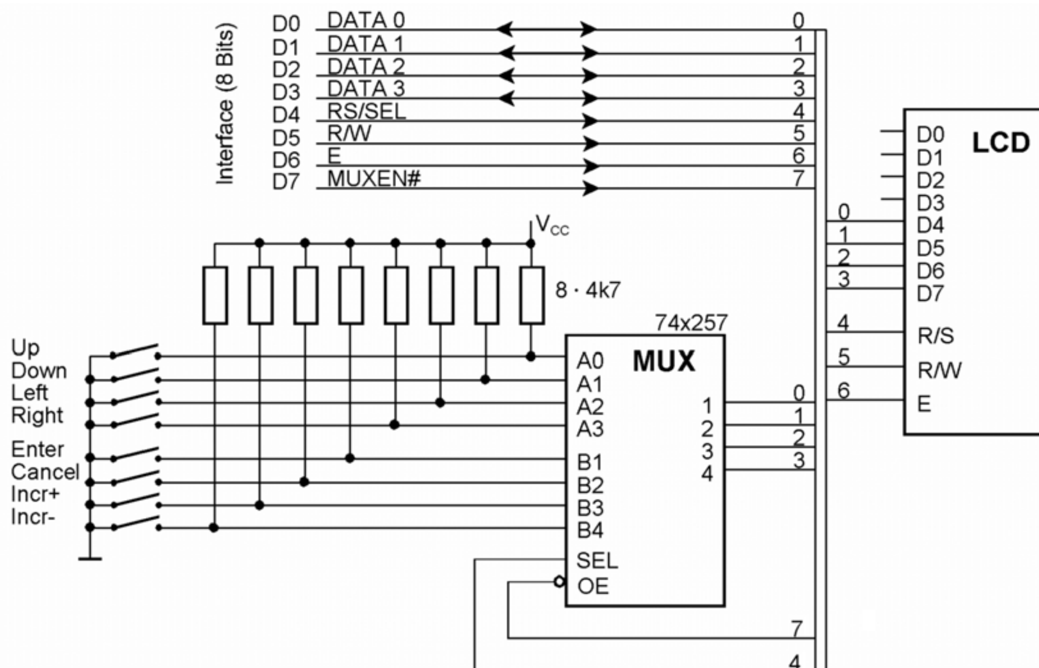


Abb. 2.29 Praxisbeispiel: eine Bedien- und Anzeigetafel

Die Bedien- und Anzeigetafel enthält sechs Tasten, einen Incrementalgeber und eine LCD-Punktmatrixanzeige (vgl. auch Abb. 2.51). Diese Einrichtungen werden über einen einzigen universellen 8-Bit-Port angeschlossen. Vier Bits (D3...0) bilden einen bidirektionalen Datenbus. Bit D6 treibt das Steuersignal (E) der LCD-Anzeige, Bit D7(MUXEN#) aktiviert den Multiplexer. Ausgabe auf die LCD-Anzeige: Bit D7 auf High. Bit D6 wird bei jedem Zugriff aktiviert. Abfrage der Bedienelemente: Bit D6 bleibt inaktiv (Low). Bit 7 auf Low. Der Multiplexer schaltet jeweils vier Kontaktbelegungen auf den Datenbus auf. Auswahl über Bit D4. Ist die LCD-Anzeige inaktiv, so dürfen die Bits D4 und D5 beliebig schalten. Nutzt man beide Bits aus, können insgesamt 4 • 4 = 16 Kontakte oder andere Eingangssignale abgefragt werden.

3. Periphere Funktionseinheiten in Mikrocontrollern (Überblick)

(Bildquellen: Atmel, Renesas)

Zähler und Zeitgeber (Timer/Counter)

Zähler (Counter):

- Zählimpulse kommen von außen,
- Vorwärtszählen von geladenem Wert an,
- Nulldurchgang (Wrap Around) wird registriert. Kann abgefragt werden oder Interrupt auslösen.

Zeitgeber (Timer):

- zählt mit internem Takt,
- Takt kann über Vorteiler (Prescaler) geführt werden (Abb. 3.1). Typische Teilerverhältnisse sind Zweierpotenzen,
- Vorwärtszählen von geladenem Wert an,
- Nulldurchgang (Wrap Around) wird registriert. Kann abgefragt werden oder Interrupt auslösen.

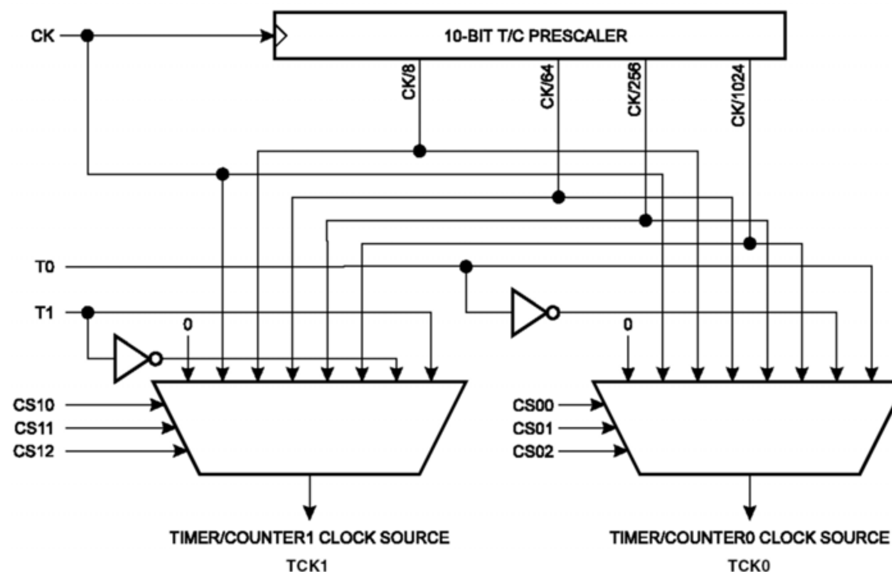
Zeiterfassungsfunktion (Capture)
Externes Signal bewirkt Übernahme des aktuellen Zählerstandes in ein programmseitig abfragbares Halteregister. Das Auftreten des externen Signals wird registriert. Kann abgefragt werden oder Interrupt auslösen.

Vergleichsfunktion (Compare)
Zählerstand wird mit dem Inhalt eines programmseitig ladbaren Vergleichsregisters verglichen. Bei Gleicheit wird eine entsprechende Bedingung gesetzt. Kann abgefragt werden oder Interrupt auslösen. Ggf. weitere Wirkungen (programmseitg einstellbar):

- bei Gleichheit Zähler löschen (so daß er von Null an weiterzählt) oder durchlaufen lassen
- Ausgangssignal setzen / löschen /umschalten

Interrupts einer Zähler-Zeitgeber-Einheit:

- Nulldurchgang (Überlauf),
- Capture,
- Compare.



Clock 0 Prescale Select

CS02	CS01	CS00	Description
0	0	0	Stop, the Timer/Counter0 is stopped.
0	0	1	CK
0	1	0	CK/8
0	1	1	CK/64
1	0	0	CK/256
1	0	1	CK/1024
1	1	0	External Pin T0, falling edge
1	1	1	External Pin T0, rising edge

Abb. 3.1 Vorteiler (Prescaler)

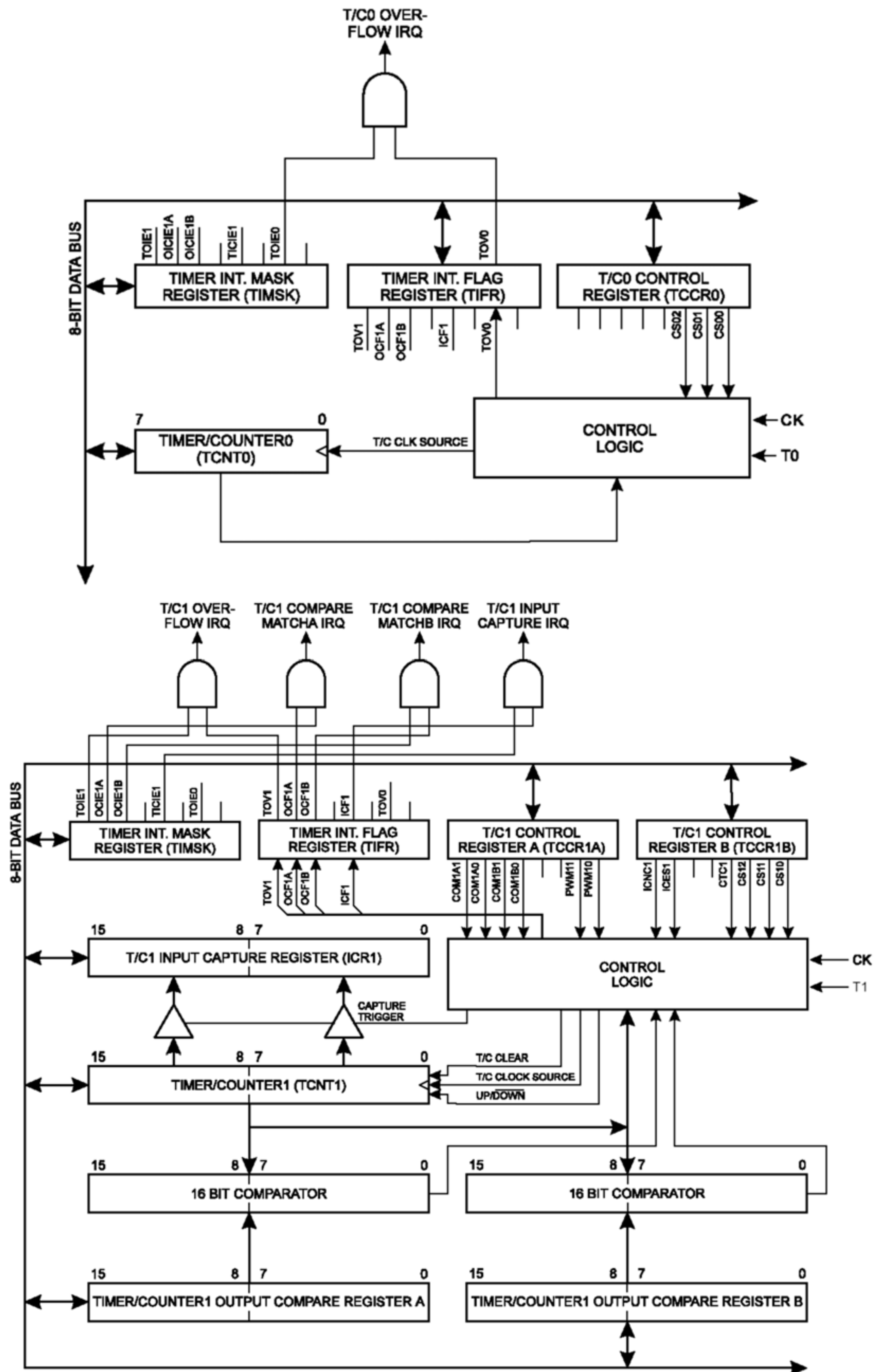


Abb. 3.2 Zwei Counter/Timer-Einheiten. Oben 8 Bits, darunter 16 Bits

Compare 1 Mode Select

COM1X1	COM1X0	Description
0	0	Timer/Counter1 disconnected from output pin OC1X
0	1	Toggle the OC1X output line.
1	0	Clear the OC1X output line (to zero).
1	1	Set the OC1X output line (to one).

Note: X = A or B

PWM Mode Select

PWM11	PWM10	Description
0	0	PWM operation of Timer/Counter1 is disabled
0	1	Timer/Counter1 is an 8-bit PWM
1	0	Timer/Counter1 is a 9-bit PWM
1	1	Timer/Counter1 is a 10-bit PWM

Timer TOP Values and PWM Frequency

PWM Resolution	Timer TOP Value	Frequency
8-bit	$00FF (255)	$f_{TCK1}/510$
9-bit	$01FF (511)	$f_{TCK1}/1022$
10-bit	$03FF(1023)	$f_{TCK1}/2046$

Compare1 Mode Select in PWM Mode

COM1X1	COM1X0	Effect on OCX1
0	0	Not connected
0	1	Not connected
1	0	Cleared on compare match, up-counting. Set on compare match, down-counting (non-inverted PWM).
1	1	Cleared on compare match, down-counting. Set on compare match, up-counting (inverted PWM).

Note: X = A or B

PWM Outputs OCR1X = $0000 or TOP

COM1X1	COM1X0	OCR1X	Output OC1X
1	0	$0000	L
1	0	TOP	H
1	1	$0000	H
1	1	TOP	L

Note: X = A or B

Abb. 3.3 Überblick über programmseitige Einstellmöglichkeiten von Counter/Timer-Einheiten

PWM (Pulsweitenmodulation):

- fortlaufendes Zählen
- Zählperiode fest, Duty Cycle veränderlich (Vergleichsregister)

PWM-Betriebsarten
Sind von Bedeutung, wenn mehrere synchrone PWM-Signale zu erzeugen sind.

1. Synchronisation durch Rücksetzen (Abb. 3.4). Zähler zählt aufwärts und wird am Ende der Periode wieder gelöscht. Alle PWM-Impulse starten bzw. enden (je nach Polarität) zur gleichen Zeit
2. Komplementäre Zählweise (Abb. 3.5). Zähler zählt aufwärts bis zur Häfte der Zählweite und dann wieder abwärts bis Null. Der Zähler durchläuft den jeweiligen Vergleichswert zweimal. Der eine Durchlauf bewirkt das Einschalten des PWM-Signals, der andere das Ausschalten.

Schrittmotorsteuerung
Wir brauchen Impulse, die gegeneinander um 90° phasenverschoben sind. PWM-Vorkehrungen ungeeignet. Abhilfe: das Zeitraster der Schritte mit Timer darstellen lassen, Interrupt auslösen und Impulse mit billiger Interruptroutine erzeugen. Richtwert: 500...1000 Schritte/s; demgemäß wird z. B. alle 1...2 ms ein Interrupt ausgelöst.

Erzeugung komplizierter Impulsverläufe
Impulsverläufe als Bitmuster im Speicher darstellen und gemäß dem jeweiligen Zeitraster zyklisch ausgeben. Manche Mikrocontroller haben hierfür Hardware-Unterstützung (Timing Pattern Controller).

Nützliche Besonderheiten:

- Ausgabe-Pufferregister. Software lädt dieses Pufferregister. Eigentliche Ausgabe wird hardwareseitig vom Vergleicher (Comparator) des Zeitgebers ausgelöst. Behelf: dem E-A-Port des Controllers ein Register nachschalten. Dessen Takt ist das vom Comparator gesteuerte Ausgangssignal. Erweiterung: FIFO-Puffer. Nicht vergessen: den Puffer initialisieren und ggf. in das Rücksetzen einbeziehen.
- Abruf der Impulsmuster mit DMA-Vorkehrungen (falls vorhanden). Behelf: das komplette Impulsmuster in einen FIFO-Schaltkreis laden und daraus zyklisch abrufen. Retransmit-Funktion ausnutzen.

Achtung: Das programmseitige Aufbereiten eines neuen Impulsmusters darf das Ausgeben des alten nicht beeinträchtigen. Abhilfe: zwei Bereiche (alt – neu) und Synchronisation bei Beginn des zyklischen Auslesens vorsehen.

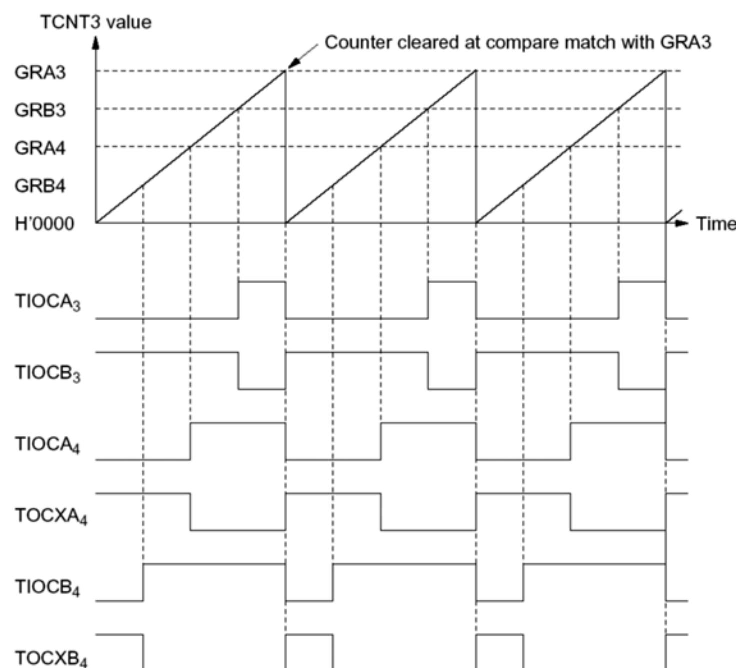


Abb. 3.4 PWM-Signalverläufe (1). Synchronisation durch Rücksetzen

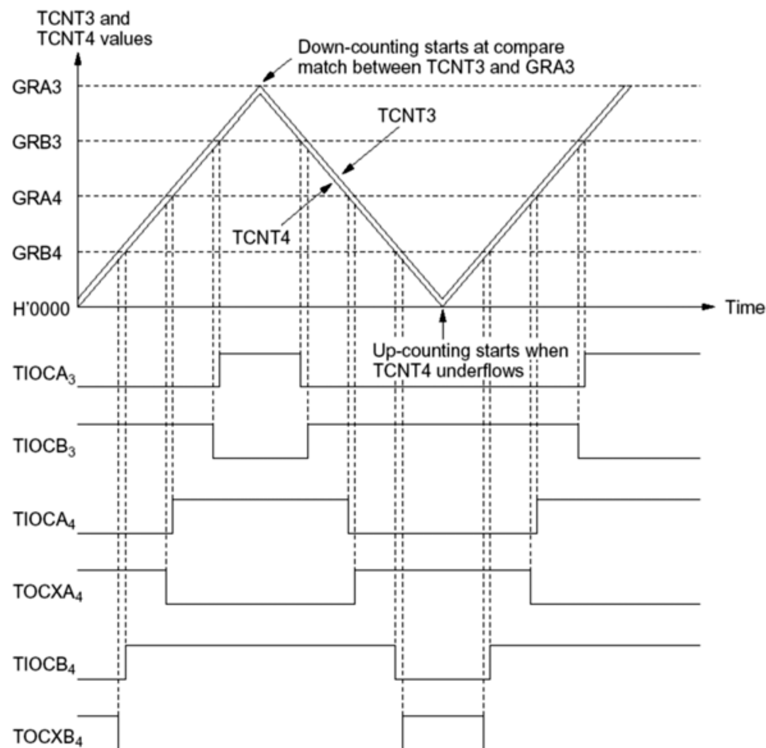


Abb. 3.5 PWM-Signalverläufe (2). Komplementäre Zählweise

Weitere periphere Funktionen (Auswahl):

- Einzelsignalen zur unmittelbaren Interruptauslösung nutzbar. Wann der Interrupt im einzelnen ausgelöst wird, ist programmseitig steuerbar (Abb. 3.6).
- serielle Schnittstellen (UARTs),
- Schnittstellen für serielle einafchbussysteme (I^2C, SPI),
- analoge Comparatoren (können auch Interrupts auslösen),
- Analog-Digital-Wandler (Abb. 3.7).

ISC11	ISC10	Description
0	0	The low level of INT1 generates an interrupt request.
0	1	Reserved
1	0	The falling edge of INT1 generates an interrupt request.
1	1	The rising edge of INT1 generates an interrupt request.

Abb. 3.6 Programmseitige Steuerung der Interruptauslösung über eine Einzelsignal

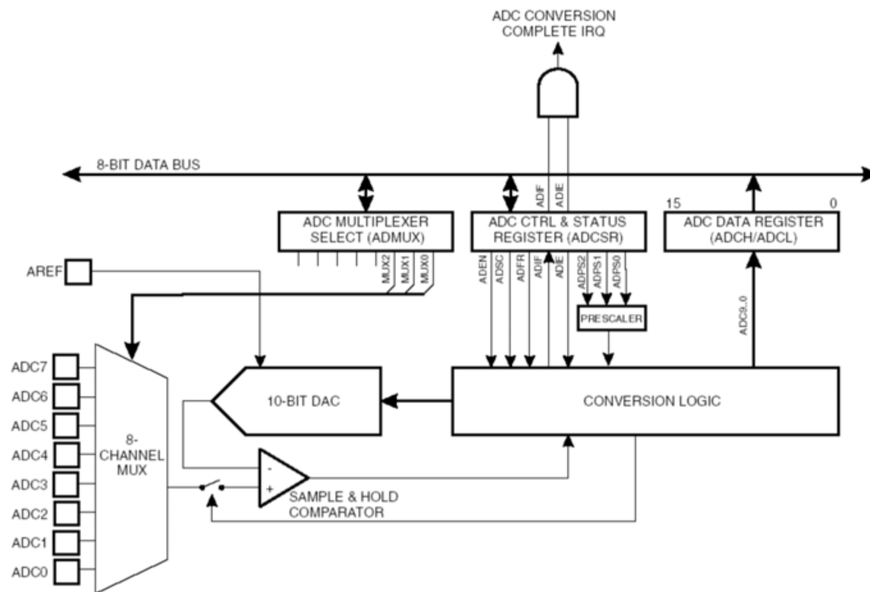


Abb. 3.7 A-D-Wandler

4. Programmiermodelle

4.1 Allgemeiner Überblick

1. intuitives prozedurales Programmieren (frisch, fromm, fröhlich, frei drauflos; IF...THEN...GOTO)
2. Abfragesteuerung. Prinzip der Steuerschleife.
3. automatentheoretisches Modell (State Machine)
4. Mehrprozeßmodell (Multitasking)
5. Ereignissteuerung und Nachrichtenweitergabe (Message Passing)
6. problembezogene Datenflußmaschine - Programmieren im Blockschaltbild (fiktive Hardware)

Es ist zu unterscheiden zwischen

1) Sprach- und Entwicklungsumgebung (was nehme ich als Werkzeug?)

   und

2) Programmierphilosophie (wie gehe ich an die Problemlösung ran?)

Die Entscheidungen in betreff 1) und 2) sind weitgehend unabhängig voneinander. Ausnahme: integrierte Entwicklungsumgebungen.

Von Hand programmieren – Vorteile:

- bessere Ausnutzung der Hardware im Hinblick auf zeitkritische innerste Schleifen,
- weniger Overhead beim Unterprogrammruf,
- Nutzung ungewöhnlicher Datentypen (“bis aufs Bit”),
- Implementierung ungewöhnlicher Programmiermodelle(Multitasking, Memory Swapping, State Machines usw.) – und zwar von Grund auf, also ohne auf ein (ggf. kostspieliges) fertiges Betriebssystem angewiesen zu sein

In C usw. programmieren – Vorteile:

- fertige Arithmetik
- fertige Mechanismen für Unterprogrammaufruf und Parameterübergabe
- fertige Bedingungsabfrage- und Schleifenkonstrukte
- Optimierung der Belegung größerer Registersätze

- Bei nur wenigen Registern (z. B. Intel) kann auch der beste Compiler kaum etwas tun. -

Was kann man von Hand besser optimieren?

- komplizierte Algorithmen,
- Umgang mit Datenstrukturen, die von der Hochsprache nicht direkt unterstützt werden (betrifft u. a. schon das richtige Multiplizieren und Dividieren mit Integers),
- Ausnutzung besonderer Eigenschaften des Prozessors.

Wann lohnt sich die Handoptimierung kaum?
Bei einfachen bzw. allgemein üblichen Abläufen – wenn ersichtlich weder Anlaß noch Gelegenheit zum Tricksen vorliegt (mit anderen Worten: wenn es so simpel ist, daß einem eh nichts weiter einfällt ...).

Praxistip:
Handoptimierte Abläufe zusammenfassen (z. B. in bestimmten Funktionen); nicht wahllos einstreuen.

Gesamt-Vorgehen:
Top Down und Bottom Up gleichzeitig. Bottom Up ermöglicht mehr Parallelisierung. wäöhren das gesamt-Projekt ausspezifiziert wird (Top Down) kann man bereits die unbedingt erforderlichenWerkzeuge und Grundfunktionen schaffen und für sich austesten.

Ggf. Testumgebungen vorsehen – bis hin zu Hardware-Simulatoren (mit weiteren PCs, Mikrocontrollern usw. improvisieren).

4.2 Konventionen der elementaren Mikrocontroller-Programmierung

Speicherung der Variablen:

a) in Registern,
b) im Arbeitsspeicher.

Bei Assemblerprogrammierung Variable genauso definieren wie in einer höheren Programmiersprache.

– Ordnung muß sein –

Sprachmittel (Assembler):
DSEG für Speichernutzung, DEF für Registernutzung.

Programmiermodelle:

1. alle Variable passen in die Register. Achtung: *Genügend Arbeitsregister frei lassen!*
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 (für Atmel):

- 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...
```

4.3 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: 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. 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 (Abb. 4.1).

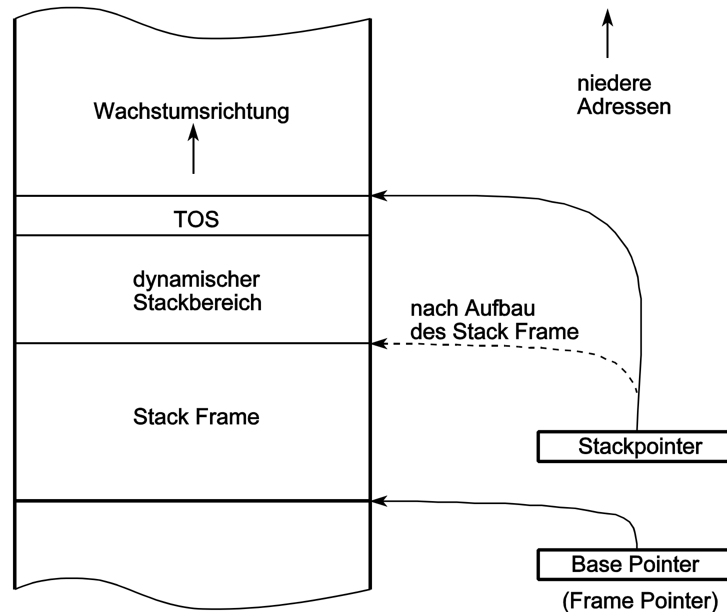


Abb. 4.1 Stack-Organisation mit Stack Frame

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). Abb. 4.2 veranschaulicht ein Prinzip, das häufig implementiert wird.

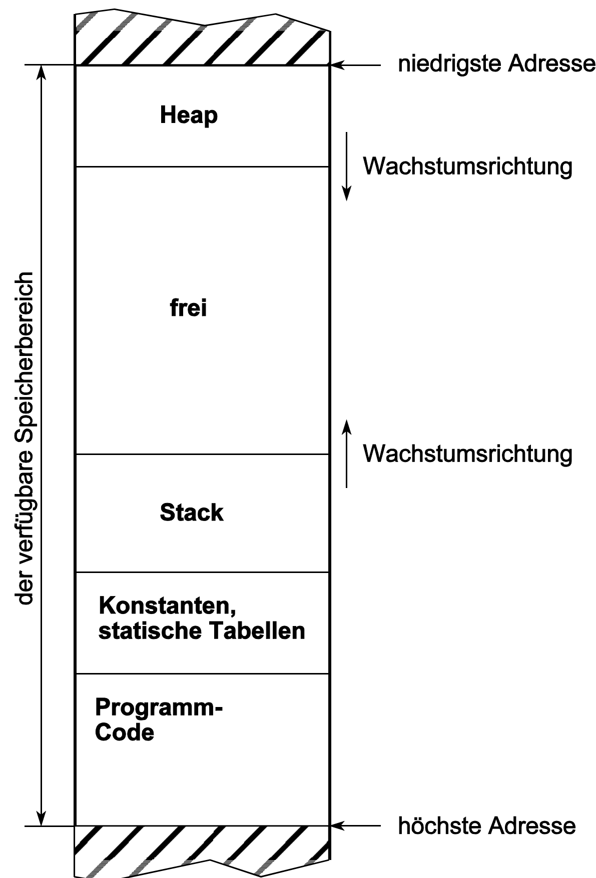


Abb. 4.2 Zum Prinzip der Speicheraufteilung (Beispiel)

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 4.1.

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 4.1 Zur Verwendung von Stack und Heap

Die UNIX-Stackorganisation
Für jeden Prozeß werden zwei Stacks verwaltet (Abb. 4.3):

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

Erklärung zu Abb. 1.22:
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[1)].
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[2)].
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.

1) 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.).

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

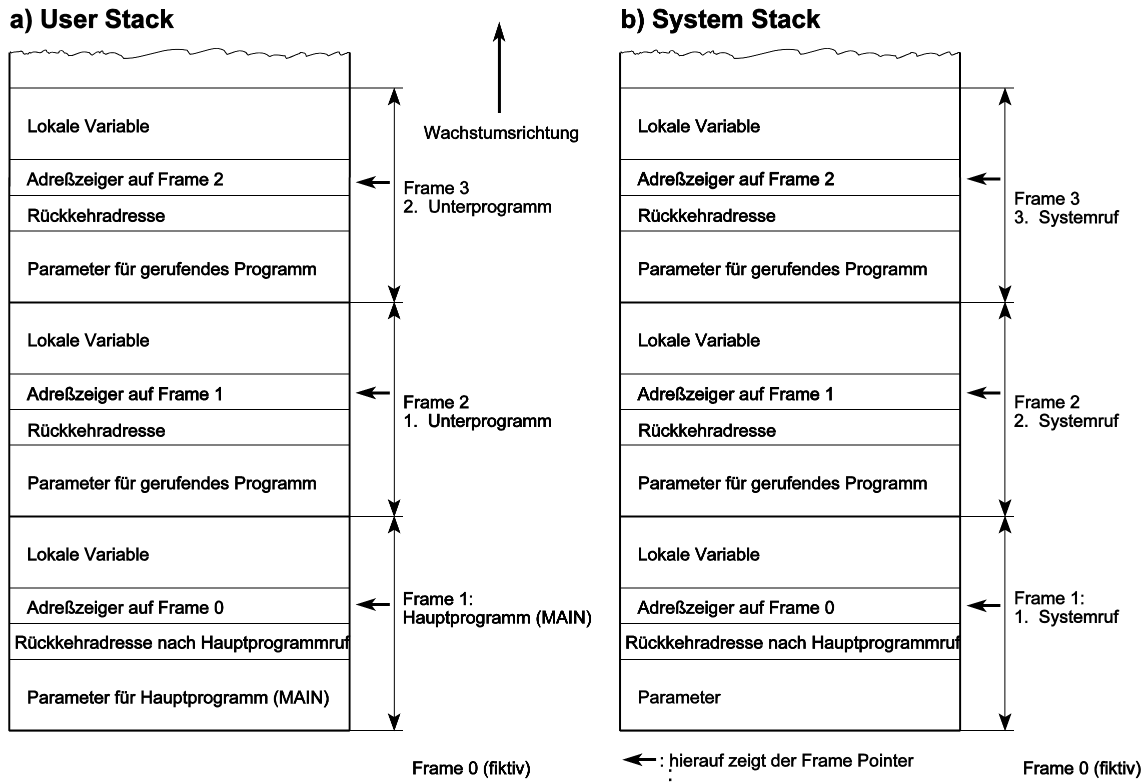


Abb. 4.3 Die UNIX-Stackorganisation

a) rufendes Programm:

PUSH Parameter

CALL Prozedur (PUSH Rückkehradresse)

b) gerufenes Programm (Funktion, Prozedur)

ENTER-Ablauf (Eintritt):
PUSH alten Frame Pointer
Stackpointer wird neuer Frame Pointer (SP => FP)
DECREMENT SP -- Platz schaffen für lokale Variable

-- der eigentliche Programmablauf --

Rückgabe von Ergebnissen bzw. Funtionswerten: der Parameterbereich ist über den Frame Pointer mit positiven Displacements erreichbar

LEAVE-Ablauf (Rückkehr):
Stackpointer mit Frame Pointer überladen (FP => SP)
POP alten Frame Pointer (wird wiederhergestellt)
RETURN

POP Parameter (Stack säubern)

Abb. 4.4 Unterprogrammaufruf in einer Laufzeitumgebung, die auf Stack Frames beruht

Typische Konventionen des Unterprogrammrufs
Siehe Tabelle 4.2.

	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 4.2 Typische Konventionen des Unterprogrammrufs

Die Stackbelegung anhand eines Beispiels

Deklaration einer Funktion:

```
int MAUSI (int A, int B, double C);

{
int X, Y;
double Z;
float H, I;
...

....

return (Y);

}
```

Jetzt wird die Funktion aufgerufen:

```
...

OMEGA = MAUSI (ALPHA, BETA, GAMMA);

...
```

Zunächst werden die Parameter auf den Stack gelegt, dann wird die Funktion aufgerufen:

```
PUSH_DOUBLE GAMMA -- Übergabe beginnt von hinten (C-Konvention)
PUSH BETA
PUSH ALPHA
CALL MAUSI
```

Eintritt in die Funktion. Es wird zunächst der Eintrittscode ausgeführt (ENTER-Ablauf):

```
MAUSI:  PUSH   FP       -- Frame Pointer auf Stack
        MOV    SP, FP   --  neuen FP einrichten
        DEC    SP, 24   -- die lokalen Variablen brauchen 24 Bytes

        .... jetzt kommt der Funktionskörper von MAUSI ....

        Rückkehr (LEAVE-Ablauf):

        MOV Y, (FP + 20)    -- Rückgabewert in Stack schreiben
        MOV FP, SP          -- Zurückstellen des Stackpointers
        POP FP              -- alten FP aus Stack zurückholen
        RETURN
```

Weiter mit dem rufenden Programm:

```
POP_DOUBLE          -- Stack freimachen
POP
POP OMEGA        -- mit dem letzten POP wird der Rückgaberwert zugewiesen
```

Programmoptimierung:

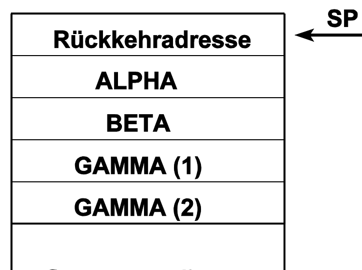
- kurze Funktionen nicht rufen, sondern als Inline-Code einfügen,
- Funktionen ohne Parameter und lokale Variable werden nach einem verkürzten Verfahren aufgerufen. Also: nicht nach der reinen Lehre, sondern auf Leistung programmieren...

Zum Fehlersuchen (Debugging) müssen diese Optimierungen ggf. ausgeschaltet werden (damit man im Speicherausdruck (Memory Dump) erkennen kann, was eigentlich losgewesen ist):

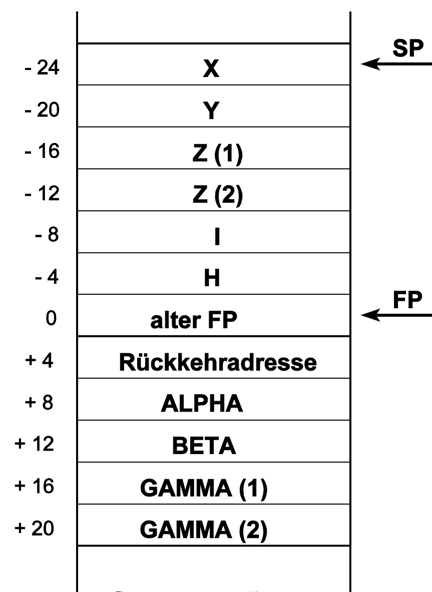
- kein Inline-Code,
- es wird stets ein richtiger Stack Frame erzeugt.

a) Ausgangszustand

b) Stack bei Verzweigung zu MAUSI



c) mit dieser Stackbelegung beginnt die Ausführung des Porgrammkörpers von MAUSI



Beispiel der Steuerung eines C-Compilers: