Druckerport-Programmierung: Parallelschnittstelle für Meß- und Steuerungsanwendungen

DIETER STOTZ

Komplexe Steuerungen erfordern oftmals spezielle Schnittstellenkarten. Für einfachere Anwendungen kommt man jedoch häufig mit der Standard-Parallelschnittstelle aus, wie sie zur Ansteuerung von (nicht vernetzten) Druckern gedacht ist. Wie aus dieser Schnittstelle mit der zugegeben begrenzten Leistungsfähigkeit dennoch erstaunliche Dinge herauszuholen sind, wollen wir hier demonstrieren.

Die Kommunikation eines Rechners mit der Außenwelt geschieht hauptsächlich über Schnittstellen. Man versteht darunter Datenanschlüsse, die einer Festlegung in bezug auf Signalpegel und Datenprotokoll unterliegen.
Dies erfordert natürlich einen bestimmten Hardware- und Software-Aufwand. Sollen viele Eingänge und Ausgänge abfragbar beziehungsweise steuerbar sein, sind spezielle Zusatzkarten notwendig. Falls der Rechner über einen externen SCSI-Anschluß verfügt, so kann auch dieser herangezogen werden.
Für einfachere Steuerungen, bei denen nicht mehr als 8 Datenausgänge und nicht mehr als 5 Dateneingänge erforderlich sind, kommt man sehr oft mit der Standard-Parallelschnittstelle aus – was den Einbau einer separaten Karte erübrigt.
Neuere Parallelschnittstellen bieten beim 8-Bit-Datenport sogar bidirektionalen Betrieb (diese Anschlüsse sind also auch als Eingänge nutzbar), doch wollen wir uns hier mit dem älteren Typ beschäftigen, damit auf jeden Fall Abwärtskompatibilität besteht und der gute alte 286er ebenfalls für Steuerungszwecke herangezogen werden kann.

Pinbelegung der Schnittstelle

Bild 1 zeigt uns die Bedeutung der Anschlüsse der 25poligen Sub-D-Buchse. Daraus geht auch hervor, ob es sich um einen Eingang oder einen Ausgang handelt. Alle Anschlüsse sind hier unidirektional, das heißt, jeder Anschluß hat entweder Eingangs- *oder* Ausgangsfunktion.
Nach der neuen Norm IEEE 1284 sind jedoch, wie oben bereits erwähnt, auch bidirektionale Betriebsarten möglich, wobei die Interface-Hardware natürlich für diese Norm ausgelegt sein muß.
Wir benötigen für die folgende Beschreibung nur die unidirektionale Betriebsart – also 8 Datenausgänge, 4 Kontrollausgänge und 5 Kontrolleingänge.
Manche Rechner verfügen über mehrere Parallelschnittstellen, welche unter DOS oder Windows bekanntlich mit der Bezeichnung LPT1, LPT2 ... angesprochen werden.

Mehrere Parallel-Ports

Bei der sogenannten hardwarenahen Programmierung über Assembler, C oder Pascal geschieht die Verknüpfung zum Port über seine Adresse.
Jeder Schnittstelle (zum Beispiel Tastatur, serielle und Parallelschnittstelle) ist eine feste Adresse zugeordnet, über die auch die Kommunikation mit dem Prozessor vonstatten geht.

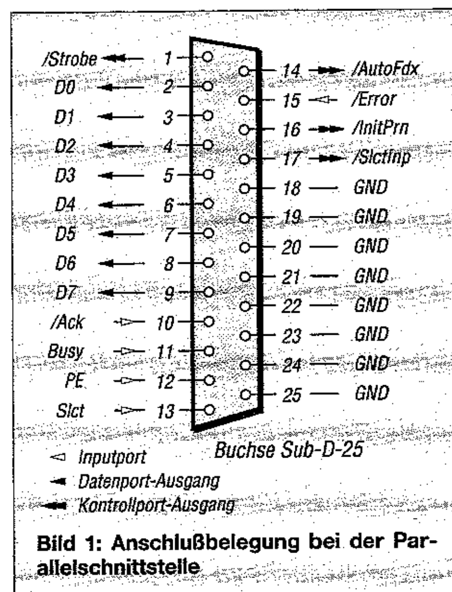

Bild 1: Anschlußbelegung bei der Parallelschnittstelle

Zu jeder der Parallelschnittstellen gehört eine definierte Basisadresse; diese ist jedoch nur für die Ausgabe an den Datenausgängen passend. Die Kontrollausgänge und -eingänge weisen jeweils Offsets zu diesen Basisadressen auf, wie dies aus der Tabelle hervorgeht. Die Konfiguration ist bei den einzelnen Schnittstellenkarten meist durch Jumper wählbar.

Kommunikation mit dem Prozessor in Assembler

Die folgenden beispielhaften Assemblercodes entsprechen der Syntax, wie sie in Turbo-Pascal zu lauten hat. Eine Übertragung in andere Assemblersyntaxen (jedoch stets für 80×86) ist sicher leicht möglich.
Zur Ausgabe von Daten sind in Assembler im Prinzip nur drei Zeilen notwendig, nämlich:

```
MOV  DX, [Portaddr]
MOV  AL, [Data]
OUT  DX, AL
```

Das DX-Register ist immer für die Aufnahme der Port-Adresse zuständig (erster MOV-Befehl), während der Akku AL das zu transferierende Byte aufnehmen muß (zweiter MOV-Befehl). Der OUT-Befehl schließlich schickt den Byte-Wert *Data* an den gewünschten Anschluß.
Die beiden Variablen *Portaddr* und *Data* müssen in eckigen Klammern stehen, da bei der Kompilierung der Wert der Variablen ja noch nicht bekannt sein kann – wohl aber deren Adressen.
In Turbo-Pascal müssen beide Variablen außerdem lokalen Charakter besitzen; stehen die Werte zunächst in globalen Variablen, so sind diese innerhalb der Prozedur jeweils einer lokalen Variablen zuzuweisen. Selbstverständlich lassen sich auch direkte Werte für Port-Adresse bzw. Daten in die Register schreiben.
Beispiel: Soll an den Datenport der Schnittstelle LPT1 der hexadezimale Wert 40h gesendet werden, so kann dies geschehen durch:

```
MOV  DX, 378H  {Port-Adresse für Daten festlegen}
MOV  AL, 40H   {Übertragungswert in Akku}
OUT  DX, AL    {Übertragungswert ausgeben}
```

In der Praxis wird man bei vielen Änderungen oder bei rechnerischer Bestimmung des Übertragungswertes keinen festen Zahlenwert in den Assemblercode schreiben, sondern die Übergabe mit einer Variablen (siehe oben) bewerkstelligen.
Dagegen ist die Port-Adresse besser als Direktwert anzugeben, wenn man sich für eine Schnittstelle entschließt. Das bringt gegenüber einer variablen Übergabe eine Geschwindigkeitssteigerung, besonders wenn die Festlegung der Port-Adresse verhältnismäßig oft geschieht. Eine neuerliche Festlegung der Port-Adresse ist jedoch nur dann notwendig, wenn inzwischen ein anderer Port angesprochen oder wenn die Assembler-Routine verlassen wurde. Wir werden jedoch auf zeitkritische Anwendungen und die Bewältigung deren Probleme später noch eingehen. Der Wert 40h bewirkt nur das Setzen von Bit 6 des Datenports (D6 = Pin 8).
Während der Datenport D0 ... D7 der Schnittstelle die gesetzten Bits des entsprechenden Übertragungswertes unverändert übernimmt, trifft dies für den Steuerport nicht zu – hier werden gesetzte Bits des Übertragungswertes das Pegelresultat L

(angedeutet durch das Wort *NOT* am zugehörigen Anschluß) bewirken. Die einzelnen Bits erhalten dabei folgende Zuordnung:

Not Bit 0 – \Strobe (Pin 1)
Not Bit 1 – \AutoFdx (Pin 14)
Not Bit 2 – \InitPrn (Pin 16)
Not Bit 3 – \SlctInp (Pin 17).

Beispiel: Nehmen wir an, es sollen Pin 16 und Pin 17 auf H gesetzt werden; das bedeutet, Bit 2 und Bit 3 müssen 0 sein (wegen der Invertierung), die beiden unteren Bits müssen 1 sein. Als Schnittstelle diene wieder LPT1. Als Assemblercode ergibt sich dann:

```
MOV DX, 37AH {Adresse für Kontrollport festlegen}
MOV AL, 3    {Übertragungswert in Akku}
OUT DX, AL   {Übertragungswert ausgeben}
```

Der Wert 3 kommt von den gesetzten beiden unteren Bits. Natürlich ist die Wertübertragung auch wieder über eine lokale Variable möglich, also:

```
MOV AL, [Data]
```

Soll das Setzen der Kontrollausgänge direkt durch die gesetzten Bits der Variablen geschehen, so ist diese durch einen NOT-Befehl einfach zu invertieren.

```
MOV AL, [Data]
NOT AL
```

Dies ist jedoch nur bei zu berechnenden Wertausgaben erforderlich – das Steuern diskreter Ausgänge kann durch direkte Werteingabe in den Akku ohne anschließende Invertierung erfolgen. Die vier oberen Bits des Übertragungswertes sind ohne Belang und wirken sich nicht aus. Man kann sie daher zum Beispiel auf 0 setzen.
Die Abfrage der fünf Eingangsanschlüsse geschieht in ähnlicher Weise wie die Steuerung der Ausgänge. Um die Zustände am Inputport der Schnittstelle LPT1 abzufragen, genügen daher folgende Anweisungen:

```
MOV DX, 379H
IN  AL, DX
MOV [Data], AL
```

Zwischen den Bits und den Anschlüssen besteht folgender Zusammenhang:

Bit 3 – \Error (Pin 15)
Bit 4 – Slct (Pin 13)
Bit 5 – PE (Pin 12)
Bit 6 – \Ack (Pin 10)
NOT Bit 7 – Busy (Pin 11).

Hier ist nur Bit 7 (Busy) invertiert. Die untersten drei Bits, für die ja keine physischen Anschlüsse existieren, sind immer auf 1 gesetzt, so daß der Wert im Akku nach dem IN-Befehl mindestens 7 beträgt. Der Wert des Akkus AL, der zunächst die Zustände aufnimmt, kann wiederum auf eine lokale Variable *Data* übermittelt werden, damit man das Ergebnis außerhalb der Assembler-Routine weiterverarbeiten kann.

Port-Adressen zur Kommunikation mit der Parallelschnittstelle

	Daten-port	Kontr.-port	Input-port	Name
bis zu 2	378h	37Ah	379h	LPT1
Schnittstellen	278h	27Ah	279h	LPT2
bis zu 3	3BCh	3BEh	3BDh	LPT1
Schnitt-	378h	37Ah	379h	LPT2
stellen	278h	27Ah	279h	LPT3

Natürlich ist auch eine andere Verarbeitung, zum Beispiel in Assembler, möglich. Man könnte etwa den Rechner stoppen, bis der Anschluß Busy L-Pegel annimmt, und zwar durch:

```
       MOV DX, 379H
@test: IN  AL, DX
       AND AL, 80H
       JZ  @test
```

Der AND-Befehl testet sozusagen das oberste Bit; ist es Null, erfolgt ein Sprung nach *@test*. Diese Schleife wird so lange wiederholt, bis das oberste Akku-Bit gesetzt ist.

Zeitkritische Vorgänge

Zur Bewältigung von sehr schnellen Vorgängen wird man auf eine kompakte Programmierung verzichten, bei der viele Varianten in einer Routine abgearbeitet werden und die Abfrage der Variante innerhalb der Routine stattfindet.
Viel besser eignet sich hierbei die lineare Programmierung, bei der die einzelnen Varianten in separate Routinen aufgelöst sind – die Entscheidung darüber, welche Routine nun zu wählen ist, wird extern gefällt. Das erfordert zwar mehr Speicherplatz, der Geschwindigkeitsgewinn ist jedoch nicht unbeträchtlich.
Ähnliches gilt, wenn eine Routine zum Beispiel dreifach durchlaufen werden soll; zur Geschwindigkeitssteigerung wird sie einfach als dreifacher Code hintereinander geschrieben, anstatt über eine Schleifenkontrollstruktur formal den gleichen Ablauf zu erhalten. Schleifen können in kritischen Fällen zuviel Zeit verbrauchen, weil Bedingungen und Sprünge abzuarbeiten sind.
Wie auch in höheren Programmiersprachen, sollte man in Assembler besondere Sorgfalt darüber walten lassen, welche Befehle sich günstig auf die Geschwindigkeit auswirken. Anstatt immer wieder auf Variablenwerte zuzugreifen (die ja schließlich im Speicher stehen), ist es vorteilhafter, den Variablenwert ein für allemal einem Register zuzuordnen. Wann immer dieser Wert jetzt wieder benötigt wird, ist nur noch ein relativ schneller Registerzugriff erforderlich.

Anpassung an Rechnergeschwindigkeit

Oftmals steht man jedoch auch vor dem Problem, daß nicht Maximalgeschwindigkeit gefordert ist, sondern einheitliche Geschwindigkeit auf jedem Rechner. Wir haben gesehen, wie man einzelne Byte-Werte auf dem Datenport der Schnittstelle ausgeben kann.
Natürlich kann man auch einen ganzen Datenblock ausgeben. Nicht jeder Rechner kann diesen Datenblock gleichschnell verarbeiten. Dabei sind zu unterscheiden die Vorgänge, die prozessorintern ablaufen und solche, bei denen der Prozessor auf den Speicher zugreifen muß (sogenannte E/A-Einheiten).
Alle Blockwerte werden nun in chronologischer Reihenfolge zur Schnittstelle transferiert. Soll dies an jedem Rechner mit gleicher Geschwindigkeit geschehen, muß es eine Möglichkeit geben, die Geschwindigkeit der zum Transfer notwendigen Routine zu drosseln. Dies geschieht mit einer Warteschleife, die sich an geeigneter Stelle in die Routine einfügt. In Bild 2 sehen wir den prinzipiellen Ablauf.
Zur Ausgabe eines Blockwertes muß die große (Block-)Schleife einmal durchlaufen werden – danach erfolgt ein neuer Zyklus mit der Ausgabe des darauffolgenden Wertes. Innerhalb der Schleife sind die Instruktionen 1 bis 3 abzuarbeiten. Doch der Warteschleifenzähler hält den Ablauf auf; er nimmt zunächst einen von außen festzulegenden Wert auf, und bei jedem Zyklus innerhalb der Warteschleifen wird der Zähler dekrementiert. Erst bei Erreichen von Null kann die Warteschleife verlassen werden. Als Zeitmaß für den Gesamtzyklus kann also geschrieben werden:

$$T_{ges} = (T_{instr1} + N_w \cdot T_{instr2} + T^*_{instr2} + T_{instr3}) \cdot Const_r + N_e \cdot Const_b. \quad (1)$$

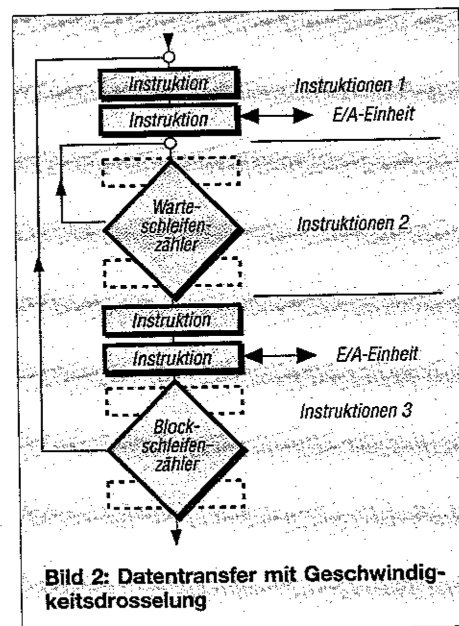

Bild 2: Datentransfer mit Geschwindigkeitsdrosselung

Hierin sind T_{instr} die Summe der Taktzyklen der Einzelinstruktionen, N_w die Anzahl der Warteschleifen, N_e die Anzahl der E/A-Einheiten und die Konstanten $Const_r$ und $Const_b$ sind rechnerspezifische Werte, die einerseits den Prozessortakt und andererseits die Kommunikationsgeschwindigkeit des Prozessors mit der Außenwelt berücksichtigen.
Als Einzelzeiten der Instruktionen nimmt man die Anzahl der jeweils nötigen Taktzyklen, die fast in jedem Assembler-Handbuch zu finden sind. Die Taktzyklen T^*_{instr2} ergeben sich, wenn die Warteschleife zwar angefahren, aber nicht ausgeführt wird.
Genauso muß man bei bedingten Sprungbefehlen darauf achten, daß die Zeiten bei ausgeführtem und nicht ausgeführtem Sprung unterschiedlich sind. Jeder Speicherzugriff erfordert eine E/A-Einheit, deren Bewältigung nicht unmittelbar mit dem Prozessortakt zusammenhängt und somit eine eigene Bewertung – also Konstante – erfordert.

Was nützt uns nun die Kenntnis dieses Zusammenhangs? Setzt man zwei extreme Werte (einen kleinen und einen großen) ein, so erhält man zwei Gleichungen aus Gleichung (1) und natürlich auch zwei unterschiedliche Werte von T_{ges}. Aus diesen beiden Gleichungen lassen sich die zwei noch unbekannten Werte der Rechnerkonstanten ermitteln.
Nun läßt sich Gleichung (1) nach N_w auflösen:

$$N_w = \frac{\frac{T_{ges}}{Const_r} - N_e \cdot Const_b - T_{instr1} - T^*_{instr2} - T_{instr3}}{T_{instr2}} \qquad (2)$$

Es gibt auch eine weitere, jedoch mehr empirische Methode, beide Konstanten zu ermitteln. Anhand von Gleichung (2) erkennen wir, daß für große T_{ges} die restlichen Glieder des Zählers vernachlässigbar werden, also auch das Glied mit $Const_b$.
Legen wir dann einfach $Const_r$ versuchsweise fest, so ergibt sich ein Warteschleifenwert N_w, der unsere Routine vielleicht noch nicht in gewünschter Geschwindigkeit ablaufen läßt, was man ja am tatsächlichen T_{ges} erfährt. Also muß jetzt nur $Const_r$ solange variiert werden, bis man an die Nähe des gewünschten Wertes für T_{ges} herankommt.
Der nächste Schritt ist nun die Vorgabe eines kleinen Wertes für T_{ges}, wobei jetzt beide Konstanten maßgeblich am Ergebnis beteiligt sind. Wir brauchen aber nun nur noch $Const_b$ zu justieren, damit sich T_{ges} auch wirklich auf den gewünschten Wert einstellt. Diese beiden Einstellungen müssen mehrfach wiederholt werden, damit eine hinreichend genaue Näherung erzielbar ist.

■ Anwendung

Als repräsentatives Praxisbeispiel einer Parallelport-Programmierung stellen wir in einem der nächsten Hefte eine R- und C-Messung mit nur zwei Bauelementen (inklusive dem zu messenden) vor.

Universelle HiFi-Anlage (3)

Dr.-Ing. EWALD LENZ

Wie schon erwähnt, kann der Verstärker in mehreren Varianten aufgebaut werden. Die Version A besteht z.B. aus einem IC-Puffer-Verstärker mit ca. 6 bis 10 dB Verstärkung, einem Potentiometer und dem nachfolgenden IC-Verstärker mit einer Verstärkung von 10 bis 16 dB.
Im abschließenden Teil sehen wir uns die verschiedenen Varianten an...

Die einfache Anordnung der Version A (Bild 10 oben) liefert sehr gute Ergebnisse, sofern die zu treibende Last (Eingang des Leistungsverstärkers) nicht zu niederohmig ist. Muß der Ausgang des Vorverstärkers einen größeren Strom liefern können, empfiehlt es sich, eine diskret aufgebaute „Endstufe" mit Verstärkung 1 oder einen geeigneten Operationsverstärker, wie z.B. den BUF634, nachzuschalten (Version B).
Die vorgeschlagene, diskret aufgebaute „Endstufe" ist sehr linear, weist wegen der 100%igen Gegenkopplung einen verschwindend geringen Klirrfaktor auf und kann ohne Probleme auch niederohmige Lasten treiben.
Die Varianten C und D entsprechen den Versionen A bzw. B bis auf die zwischen Puffer-Verstärker und Lautstärkesteller eingeschleifte Klangregelschaltung (TONE). Die Klangregelschaltung kann mittels eines Relais umgangen werden. Für alle, die den Klangsteller häufig oder immer benutzen, kann das Relais entfallen. Der Stromlaufplan in Bild 11 enthält alle Varianten.
Die hier benutzte Klangregelschaltung wurde in den 70er Jahren von Tomlinson-Holman veröffentlicht. Sie hat den Vorteil, daß der 0-dB-Punkt bei 1 kHz nur wenig beeinflußt wird. Diese Beeinflussung des 0-dB-Punktes kann noch weiter verringert werden, wenn man den Höhenregelbereich durch Verkleinern des 270-pF-Kondensators einschränkt (180 bis 220 pF).
Der Regelbereich der Originalschaltung ist aus Bild 12 zu ersehen (15 dB bei 20 Hz und 20 kHz).

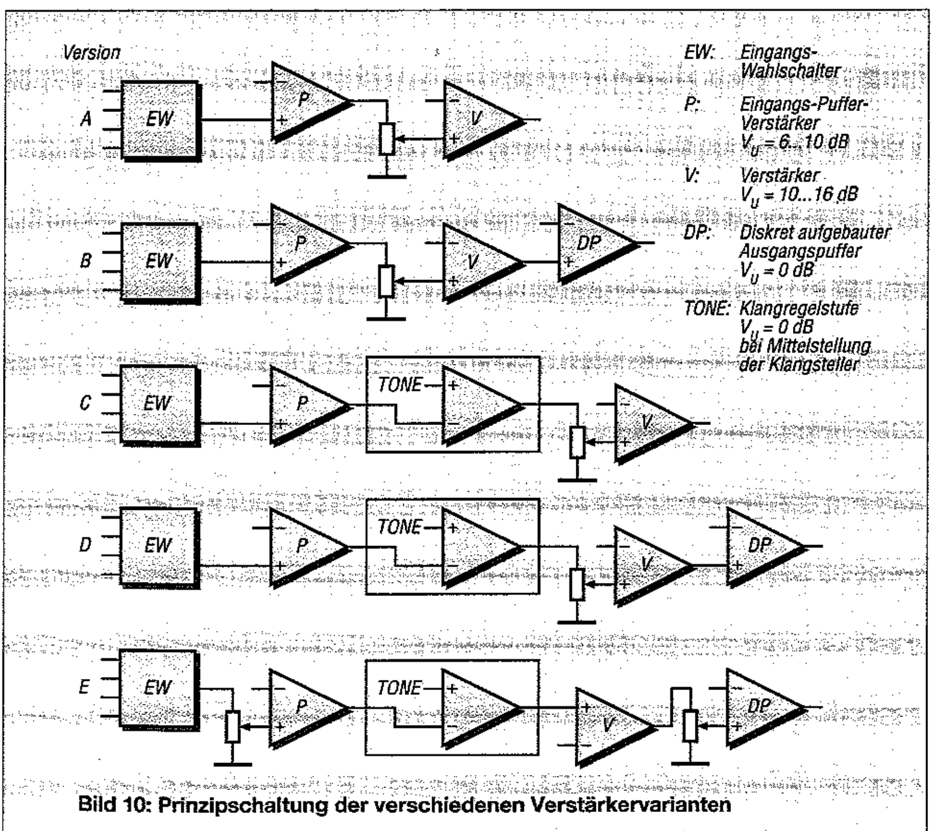

Bild 10: Prinzipschaltung der verschiedenen Verstärkervarianten