Der klassische Entwicklungsgang

Er kommt mit einfachsten Mitteln aus, ist aber wirklich mühevoll:

1. Programm erfassen. Hierzu dient ein Texteditor (oder jede x-beliebige Textverarbeitung). Das Programm wird Anweisung für Anweisung eingetippt. Dabei gibt es typischerweise einige Narrenfreiheit – man kann die Zeilen beliebig einrücken, beliebig viele Leerzeichen zwischen den einzelnen Ausdrücken lassen usw. Es entsteht der Quelltext (meist eine einfache ASCII-Textdatei).
2. Programm übersetzen (assemblieren/compilieren). Hierzu ist der Assembler oder Compiler aufzurufen (natürlich über die Kommandozeile: Name des Übersetzungsprogramms, Dateiname des Quelltextes und - bei Bedarf - diverse Steuerzeichen (die die Einzelheiten des Ablaufs regeln). Im Ergebnis entsteht eine binär codierte sog. Objektdatei.
3. Programm ausführungsfertig machen (linken = Programmverbindungen herstellen). Das ablauffähige Programm wird ggf. aus mehreren für sich compilierten Teilen (den sog. Modulen) zusammengebaut, und es werden die Bezüge zu anderen Programmen und Datenbereichen eingefügt. Hierzu ist der Linker aufzurufen (wieder über die Kommandozeile). Im Ergebnis entsteht eine ausführbare Programmdatei (z. B. eine .EXE-Datei).
4. Programm ausführen. Hierzu den Programmnamen über die Kommandozeile eingeben.

Wenn's nicht funktioniert:

- nachdenken, woran es liegen könnte,
- Quelltext ändern (vgl. Schritt 1) und alles andere nochmal von vorn...

Elementare Formen der Ablaufautomatisierung
Die einzelnen Eingaben in die Kommandozeile kann man in einer Stapelverarbeitungsdatei (Batch-Datei) zusammenfassen. In der Programmentwicklung sind vor allem die sog. Make-Dateien von Bedeutung. Das sind Textdateien, die den gesamten Programmfertigungsablauf steuern (Compilieren, Linken usw.). Der Entwicklungszyklus verkürzt sich so zu: Ausdenken und Eintippen - MAKE - laufenlassen und zusehen, ob's funktioniert - Ändern - wieder MAKE usw.).

Typische Schwierigkeiten beim Entwickeln:

- die Oberfläche,
- die Systemschnittstellen.

Natürlich gibt es Programmieraufgaben, bei denen es größte Schwierigkeiten bereitet, überhaupt einen Lösungsansatz zu finden. Die weitaus meisten Anwendungsprobleme lassen sich aber auf Grundlage des gängigen Fachwissens in vergleichsweise kurzer Zeit lösen. Nicht selten ist das, was die eigentliche Anwendung darstellt, schnell ausprogrammiert. Die weitaus meiste Zeit verbringt der Programmierer hingegen damit, eine komfortable Bedienoberfläche zu schaffen, die Dateifunktionen bereitzustellen usw. Das ist auch dann der Fall, wenn solche Funktionen im Betriebssystem vorhanden sind. Die Schwierigkeiten: man muß (1) wissen, wie man sie aufruft, und man muß (2) jeden Aufruf einzeln hinschreiben.

Integrierte Entwicklungsumgebungen (IDEs)
IDE = Integrated Development Environment. Der Grundgedanke: den Programmierkomfort zu erhöhen, indem man eine einheitliche Oberfläche vorsieht, über die alle zur Programmfertigung erforderlichen Abläufe ausgelöst und gesteuert werden können. Die ersten IDEs des PC-Bereichs liefen unter DOS und hatten noch keine echten graphischen Darstellungen (Abb. 1). Typische Beispiele: die Turbo-Sprachen von Borland und die DOS-Versionen von Power Basic.

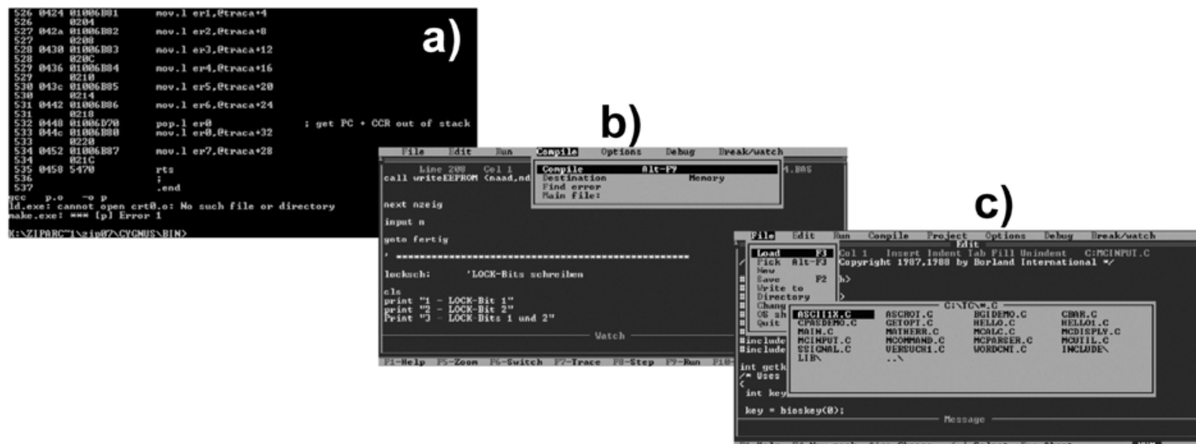


Abb. 1 Zur Geschichte der Entwicklungsumgebungen

a) ein klassisches Entwicklungswerkzeug. Hier: ein GNU-Programmpaket zur Mikrocontrollerprogrammierung (freie Software). Textbildschirm. Steuerung über Kommandozeile. Ablaufautomatisierung über MAKE-Dateien. Die herkömmliche Unix/Linux-Programmentwicklung läuft ähnlich ab. Man kommt zurecht, die Einarbeitung dauert aber ihre Zeit.

b), c) integrierte Entwicklungsumgebungen unter DOS (hier: Power Basic und Turbo C). Zeichenorientierter Bildschirm. Menüprinzip. Neuere Versionen mit Mausunterstützung. Wesentlich bequemer. Um das eingetippte Programm laufen zu lassen, genügt es, eine Funktionstaste zu drücken.

Moderne Entwicklungsumgebungen nutzen selbstverständlich die Möglichkeiten der graphischen Bedienoberflächen aus. Es gibt zwei Grundtypen (Abb. 2):

a) Entwicklungsumgebungen zum herkömmlichen Programmieren (durch Eintippen von Anweisungen). Ein komfortabler Editor ist mit dem betreffenden Compiler (oder Assembler), Linker usw. und mit dem Fehlersuchunterstützungsprogrammen (Debugger) verbunden. Der Komfort betrifft u. a. die Möglichkeit, mit mehreren Programmtexten gleichzeitig zu arbeiten, die Syntaxprüfung beim Eintippen (die etliche Trivialfehler schon im Moment des Entstehens erkennt), umfassende Such- und Hilfefunktionen (ersparen oft das Blättern in Handbüchern) und eine Versionsverwaltung (bewahrt automatisch die älteren Entwicklungsstände auf und ermöglicht es, darauf zuzugreifen). Der typische Entwickungsgang: Eintippen - Laufenlassen - ggf. Abläufe im einzelnen verfolgen (mittels Debugger) - Programm verbessern usw.

b) Entwicklungsumgebungen zum Entwickeln von Anwendungen mit graphischer Oberfläche. Die Anwendungsoberfläche (wie sie sich später dem Nutzer darbietet) muß man nicht mehr mühevoll mit (einzutippenden) Funktionsaufrufen erzeugen. Sie wird vielmehr durch Aufrufen und Verändern der jeweiligen Bildelemente, Steuerelemente und Dialoge direkt auf dem Bildschirm entworfen. Dabei entsteht eine Art Skelett des Programms, das später nur noch in den Einzelheiten auszufüllen ist. Die Oberfläche ist schon fertig; man muß lediglich die eigentlichen Anwendungsfunktionen ausprogrammieren (z. B. das, was beim Anklicken eines bestimmten Steuerelements tatsächlich geschehen soll). Der typische Entwicklungsgang: Bedienoberfläche aufbauen - solange probieren, bis sie brauchbar erscheint - das Programmskelett ausfüllen - ggf. Abläufe im einzelnen verfolgen (mittels Debugger) - Programm verbessern - auf die Kritik der Nutzer reagieren - Oberfläche verbessern usw. Das erste derartige System, das sich am Markt durchgesetzt hat: Visual Basic von Microsoft.

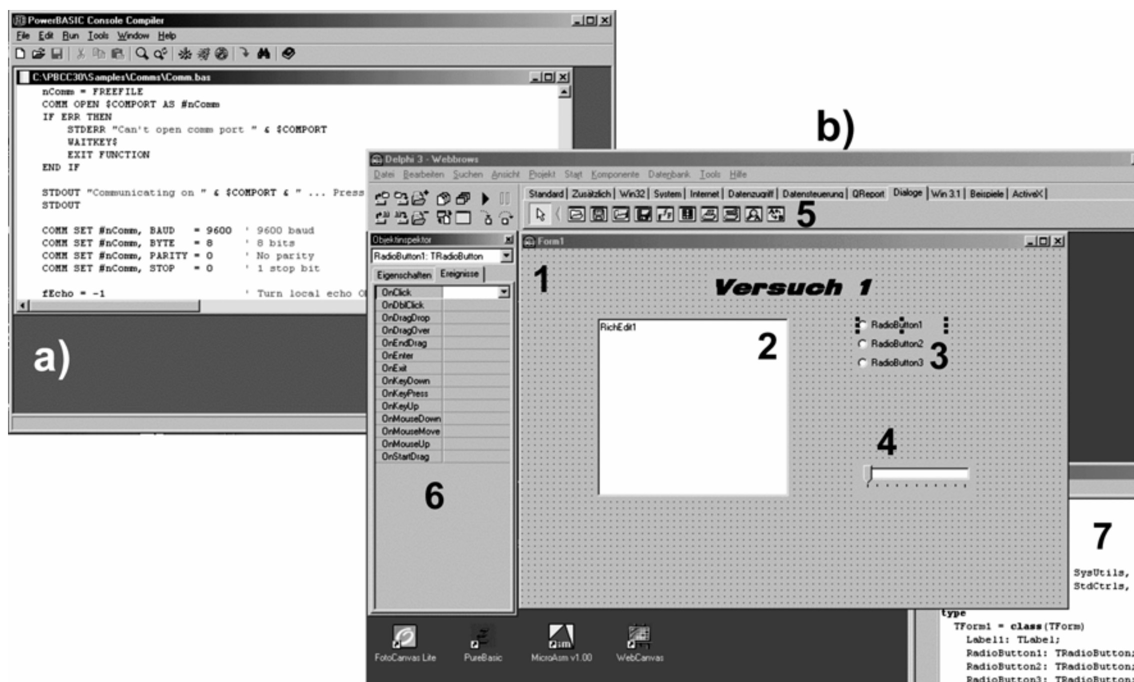


Abb. 2 Entwicklungsumgebungen

Erklärung:

a) Power Basic Console Compiler. Herkömmliche Programmentwicklung. Programmeingabe über Editor.

b) Borland Delphi. 1- Formular zum Entwickeln der Bedienoberfläche; 2, 3, 4- verschiedene Steuerelemente; 5 - über diese Leiste können die einzubauenden steuerelemente abgeruifen werden; 6 - hierüber lassen sich jedem Steuerelement die verschiedenen Ereignisse (Tastenbetätigungen, Mausbewegungen usw.) zuordnen. Ein ähnlicher Dialog ermöglicht es, die Eigenschaften der einzelnen Steuerelemente zu ändern. 7 - der Programmtext. Die Programmstücke, die zu den einzelnen Steuerelementen gehören, werden automatisch eingetragen. Die Steuerelemente können mittels Maus auf dem Formular hin- und hergeschoben sowie in ihrer Größe verändert werden.

Debugging: Fehlersuchen in der Software

Das Fehlersuchen (Debugging) beruht letzten Endes darauf, zu beobachten, was wirklich abläuft. Einschlägige, als Soft- oder Hardware angebotene Hilfsmittel (Debugger, In-Circuit-Emulatoren usw.) leisten nichts anderes: sie stellen dar, was abgelaufen ist, und geben dem wandelnden bzw. vor dem Bildschirm sitzenden, eigentlichen "Debugger" Stoff zum Nachdenken. Einschlägige Vorkehrungen können als Hardware oder als Software realisiert werden (Tabelle 1).

Ablaufaufzeichnung
Signalbelegungen und Maschinenzustände werden aufgezeichnet und später ausgewertet.

Ablaufverfolgung
Wir lassen das Programm so langsam laufen, daß wir die uns interessierenden Vorgänge in Ruhe beobachten können. Je nachdem, um welche Vorgänge es geht, gibt es verschiedene Betriebsarten

(Beobachten des Eintritts in bestimmte Funktionen, Beobachten bestimmter Befehle (Adreßvergleichsstop), Beobachten aller Befehle (Schrittbetrieb) usw.).

Prinzip	Hardware	Software
Ablaufaufzeichnung	Logikanalysator, In-Circuit-Emulator (ICE), eingebaute Ablaufspeicher	Trace- und Checkpoint-Funktionen
Ablaufverfolgung	Vergleichsstop- und Trigger-Hardware (eingebaut oder extern (z. B. im Logikanalysator oder im ICE))	Breakpoint-Funktionen

Tabelle 1 Vorkehrungen zum Beobachten von Abläufen in der Software (Übersicht)

Die Grenzen der Ablaufaufzeichnung:

- wir können typischerweise nicht alles aufzeichnen, sondern nur Ausschnitte des jeweiligen Ablaufs. Das erfordert eine entsprechende Auswahl.
- die aufgezeichneten Daten müssen nach den Problemstellen durchsucht werden, die uns interessieren. (Aus unvollständigen Aufzeichnungen ist der Zusammenhang mit dem Gesamtablauf nicht immer erkennbar.)

Der Vorteil der Ablaufverfolgung:
Wir können die Abläufe tatsächlich in allen Einzelheiten beobachten.

Die Nachteile der Ablaufverfolgung:

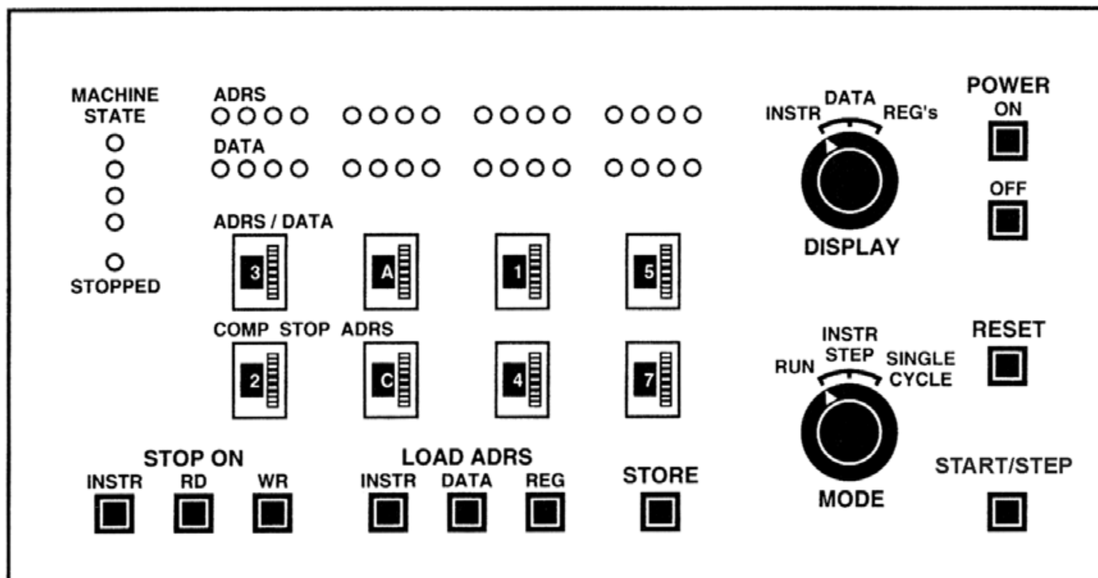
- es handelt sich um eine statische Betriebsweise; das Realzeitverhalten ist nicht mehr gewährleistet. Man kann also nicht *alle* Abläufe so prüfen.
- es ist kaum praktikabel, Tausende oder gar Millionen von Funktionaufrufen oder Befehlen so abzuarbeiten (Fehlersuchzeit).

Stichprobenhafte Ablaufbeobachtung
Der übliche Ausweg: wir sehen uns den Programmablauf nur an bestimmten Stellen an. Solche Stellen sind u. a. Eintrittspunkte in die Interruptbehandlung, der Wechsel zwischen Anwender- und Systemzustand, die Taskumschaltung usw. Von besonderer Bedeutung ist die Möglichkeit, beliebige Zugriffsadressen auswählen zu können (Adreßvergleichsstop). Solche Stopadressen ergeben sich oftmals aus Annahmen über den Programmablauf bzw. aus Fehlerhypothesen. Das heißt, wir fragen uns, ob eine bestimmte Stelle im Programm überhaupt erreicht wird, ob ein bestimmtes Unterprogramm aufgerufen wird usw. und setzen demgemäß die Vergleichsstopadresse. Dann lassen wir das Programm laufen. Tritt der Vergleichsstop ein, so hat sich unsere Annahme bestätigt.

Die herkömmliche Fehlersuchunterstützung
Es werden entsprechende Schaltmittel in der Hardware vorgesehen (Adreßvergleicher, umschaltbare Taktgeneratoren, Bedien- und Anzeigeeinrichtungen). Die herkömmliche Benutzerschnittstelle: das Bedienfeld (Abb. 3). Folgende Funktionen haben sich als zweckmäßig erwiesen:

- Wahl zwischen Normalbetrieb, befehlsweisem Betrieb (Instruction Step), taktweisem Betrieb (Single Cycle) und verschiedenen Vergleichsstop-Betriebsarten (Compare Stop),
- Einstellung einer Programmstartadresse,
- Auslösung der Programmausführung von der eingestellten Adresse an (der nächste Befehl wird von dieser Adresse gelesen),
- Einstellung wenigstens einer Vergleichsstopadresse,

- wenigstens eine Vergleichsstop-Betriebsart (am wichtigsten ist "Stop bei Befehlszugriff"; weiterhin sinnvoll ist "Stop bei Datenzugriff", wobei es vorteilhaft ist, zwischen Lese- und Schreibzugriffen unterscheiden zu können),
- die Anzeige des Befehlszählers und der wichtigsten Prozessor-Register,
- Funktionen zum Anzeigen und Ändern der Inhalte von Registern und auswählbaren Speicherplätzen,
- Auslösen des nächsten Taktzyklus, Weiterschalten zum nächsten Befehl oder zum Übergang in den Normalbetrieb (Starttaste).



POWER - Netz ein/aus; RESET - Rücksetzen; START/STEP - Programmstart oder Ausführung eines Befehls oder eines Taktzyklus (je nach Betriebsart); MODE - Betriebsartenwahl[1)]; STORE - an den Schaltern ADRS/DATA eingestelltes Datenwort speichern[2)]; LOAD ADRS - an den Schaltern ADRS/DATA eingestellte Adresse laden[3)]; STOP ON - rastende Tasten zum Einstellen der Stopbetriebsart[4)]; COMP STOP ADRS - Rändelschalter zum Einstellen der Vergleichsstopadresse; ADRS/DATA - Rändelschalter zur Adreß- und Dateneingabe[5)]; DATA - Datenanzeige; ADRS - Adreßanzeige (hier wird bei befehlsweisem Betrieb der Befehlszählerinhalt angezeigt); MACHINE STATE - zeigt an, in welchem internen Zustand sich der Prozessor befindet; STOPPED - zeigt Stopzustand an; DISPLAY - Anzeigeauswahl[6)].

1) RUN = Normalbetrieb, INSTR STEP = befehlsweise, SINGLE CYCLE = taktweise,

2) unter der zuvor geladenen Adresse,

3) INSTR = Befehlsadresse (Befehlszähler wird überladen = erzwungene Verzweigung), DATA = Datenadresse, REG = Registeradresse (Adreßeingabe zwecks Anzeigen und Ändern von Speicher- oder Registerinhalten),

4) INSTR = Stop bei Befehlslesen, RD = Stop beim Datenlesen, WR = Stop beim Datenschreiben. Die Stopbedingung wird wirksam, wenn die Zugriffsadresse der eingestellten Adresse (COMP STOP ADRS) entspricht.

5) Beispiel: Ändern eines Registerinhaltes: Registeradresse einstellen, Taste LOAD ADRS REG drücken, Datenbelegung einstellen, Taste STORE drücken.

6) INSTR = Anzeige der aktuellen Befehlsadresse (Befehlszähler), DATA - Datenanzeige, REG’s - Registeranzeige (jeweils Daten und Adresse)

Abb. 3 Das Bedienfeld eines (hypothetischen) Computers

Mit derartigen Bedien- und Anzeigevorkehrungen kann man Programme jeglicher Art Befehl für Befehl verfolgen (manchmal auch den einzelnen Befehl Taktzyklus für Taktzyklus) und so versuchen, der Fehlerursache auf die Spur zu kommen.

Die Bedientafel der modernen Computer
Anordnungen aus Lampen, Schaltern, Tasten usw. sind altmodisch - sie wurden durch Bildschirm, Tastatur und Maus ersetzt. Die Fehlersuchunterstützung ist zumeist Sache der Software.

Simulatoren und Emulatoren
Die Befehlsausführung wird mit Software gleichsam nachgespielt. Hier stehen praktisch die gleichen Funktionen zur Verfügung wie auf einer Bedientafel ähnlich Abb. 3 – nur mit viel mehr Komfort (Abb. 4). Beispielsweise kann man nicht nur einzelne Speicherworte sehen, sondern ganze Speicherbereiche. Vor allem kann man auch den Programmtext auf dem Bildschirm betrachten – früher mußte man die Programm-Listings ausdrucken und sich mit dem Papierstapel an das Bedienfeld setzen...

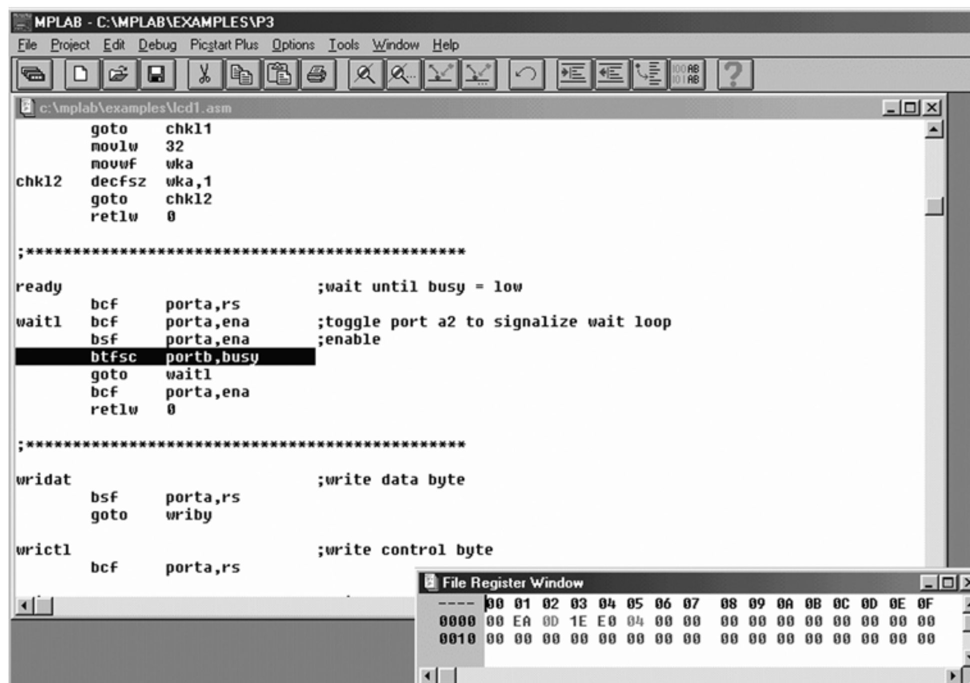
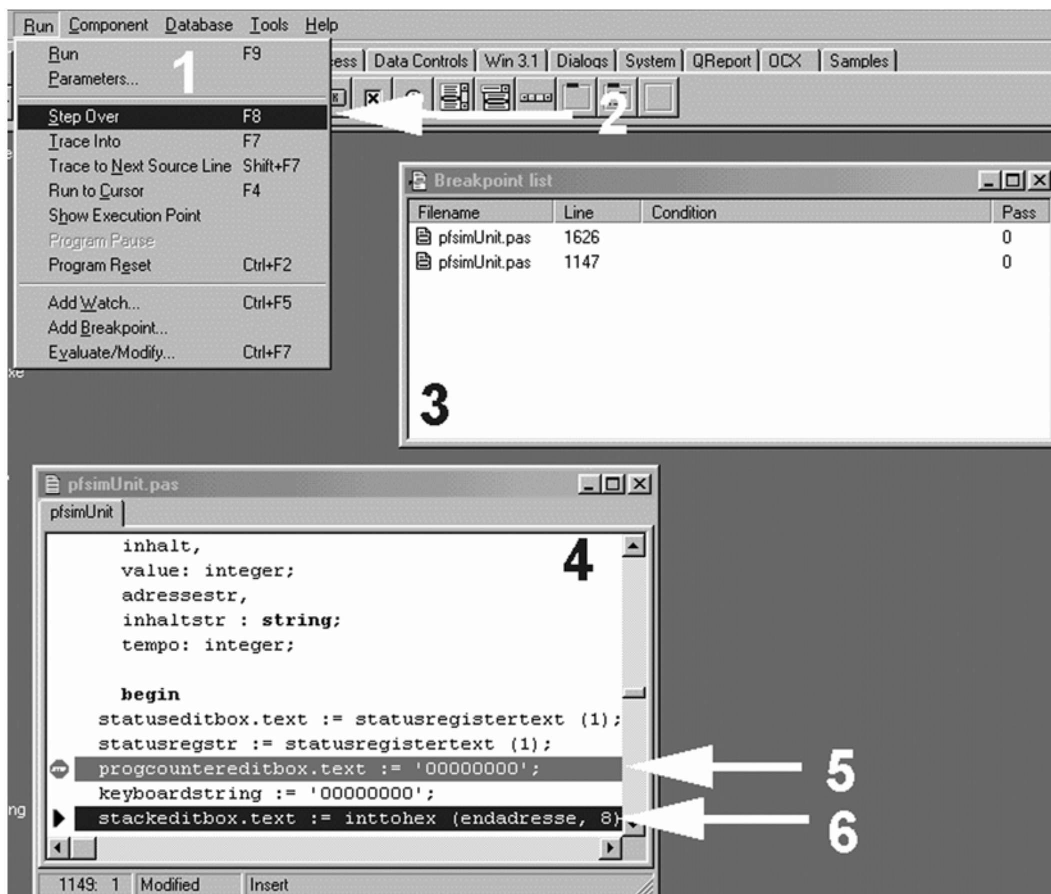


Abb. 4 Bildschirmdarstellung eines Simulators (Microchip)

Der Bildschirm zeigt einen Ausschnitt des eingegebenen Programmtextes. In zusätzlichen Fenstern können Belegungen von Registern, Ports, Speicherbereichen usw. eingeblendet werden. Der Simulationsablauf ist auf vielfältige Weise steuerbar: befehlsweise Ausführung, Durchlauf mit voller Simulationsgeschwindigkeit, Anhalten nach Erreichen einer bestimmten Adresse usw. Des weiteren können Verarbeitungszustände zwecks späterer Auswertung abgespeichert werden (Trace-Funktion). Die in der Abbildung dunkel hervorgehobene Zeile kennzeichnet den Befehl, der als nächster ausgeführt wird.

Fehlersuchprogramme (Debugger)
Diese Programme laufen zusammen mit der fehlerbehafteten Software auf derselben Hardware und nutzen auch dieselben Bedien- und Anzeigemittel. Ein Debugger ermöglicht es, das Programm, in dem Fehler gesucht werden sollen (Zielprogramm), Befehl für Befehl auszuführen. Hierzu gibt es verschiedene Prinzipien, z. B. das Umschalten des Prozessors auf Schrittbetrieb,das Einschieben von Breakpoint-Befehlen oder die Nutzung eingebauter Adreßvergleichsvorkehrungen (vgl. Tabelle 2).

Source-Level Debugger
Der herkömmliche Debugger unterstützt die Ablaufverfolgung in Maschinenprogrammen. Die weitaus meisten Programme werden aber heutzutage in höheren Programmmiersprachen geschrieben. Damit ein herkömmlicher Debugger hier etwas nützt, müßte man stets den compilierten Programmcode (mit anderen Worten: ein Assembler-Listing) vorliegen haben – und selbst dann wäre es eine mühsame Angelegenheit. Source-Level Debugger stellen die Beziehung zum Quellcode – also zu den Anweisungen des urnsprünglichen Programmtextes – her, so daß wir unseren Programmablauf Anweisung für Anweisung verfolgen können (Abb. 5).



1 - Bedien-Menü der Run-Funktion; 2 - die aktuelle Auswahl: Step Over bedeutet, daß Anweisung für Anweisung abgearbeitet wird (“Schrittbetrieb”); 3 - Fenster mit Verzeichnis der gesetzten Breakpoints (“Stopbedingungen”); 4 - Fenster mit Quellcode; 5 - auf diese Programmzeile wurde einer der Breakpoints gesetzt; 6 - die Programmzeile, die als nächstes ausgeführt wird.

Abb. 5 Anzeigen eines Source-Level Debuggers (Borland Delphi)

Wie kommt das zustande? – Durch einen Trick: beim Compilieren werden dem ausführbarem Programm spezielle Debugging-Angaben hinzugemischt. Dadurch wird das Programm länger (und langsamer). Deshalb wird meist das wirklich fertige, endgültige Programm ohne Debugging-Unterstützung compiliert.

Das Prinzip der programmseitigen Debugging-Unterstützung
Es wird stichprobenhaft signalisiert, was gerade abläuft, z. B. daß das Programm die Marke X erreicht hat, daß das Unterprogramm A aufgerufen worden ist usw. Wenn wir keine Unterstützung dafür haben, müssen wir solche Funktionen ausprogrammieren.

Breakpoints
Ein Breakpoint ist die softwareseitige Entsprechung des Vergleichsstops. Unter der Adresse, die den "Vergleichsstop" bewirken soll, speichern wir einen Breakpoint-Befehl. Wird diese Adresse erreicht, so wird der Breakpoint-Befehl ausgeführt.

Die Wirkungsweise des Breakpoint-Befehls
Der Befehl löst einen Software-Interrupt aus (Breakpoint Exception). Hierdurch wird das Debugging-Programm aufgerufen. Dieses rettet zunächst die Register auf den Stack. Im Stack steht somit der gesamte Prozessorzustand einschließlich der Adresse des Folgebefehls. Diese Angaben können ausgegeben bzw. angezeigt werden. Schließlich fragt das Debugging-Programm ab, wann das ursprüngliche Programm fortgesetzt werden soll (z. B. nach Betätigen einer entsprechenden Taste).

Nicht vergessen: den ursprünglichen Befehl. Wurde er einfach mit dem Breakpoint-Befehl überschrieben, so fehlt er, wenn das Programm fortgesetzt wird. *Auswege:*

- Breakpoint dazwischenschieben. Der Breakpoint-Befehl wird einfach in den Quelltext des Programms eingefügt. Dann wird erneut assembliert.
- von vornherein NOPs vorsehen. Gewiefte Programmierer bauen an markanten Stellen des Programms NOPs ein, die bei Bedarf mit einem Breakpoint-Befehl überschrieben werden können.
- komfortable Debugging-Software. Diese fügt Breakpoints in das ausführungsfertige Programm ein, rettet aber zuvor die ursprünglichen Befehle und merkt sich die jeweiligen Adressen. Vor der Rückkehr wird dann der ursprüngliche Befehl wieder eingefügt.

Der Prozessor hat keinen Breakpoint-Befehl. Ein naheliegender Ausweg: Wir schreiben die entsprechend Behandlungsabäufe selbst und rufen sie als Unterprogramm auf (der Aufruf – z. B. CALL BREAK – ersetzt den Breakpoint-Befehl). Beim Gestalten solcher Unterprogramme haben wir weitgehende Narrenfreiheit. Wir können z. B. veranlassen, daß die Breakpoint-Anzeige nur dann erscheint, wenn eine bestimmte Variable einen bestimmten Wert hat, wir können Eingabefunktionen zum Ändern bestimmter Variablenwerte vorsehen usw.

Checkpoints
Als Checkpoints (Prüfpunkte) wollen wir in den Programmablauf eingefügte Kontroll-Ausgaben bezeichnen, die keine Rückantwort (z. B. Tastenbetätigungen) erwarten. Anhand solcher Checkpoints können wir erkennen, ob das Programm überhaupt läuft oder ob es hängengeblieben ist, und wir können u. a. die Schleife identifizieren, in der sich das Programm gerade befindet.

Selbstgeschmiedete Breakpoints und Checkpoints als universelle Debugging-Hilfsmittel
Der Grundgedanke: die heutigen Entwicklungsumgebungen sind so schnell, daß man es sich leisten kann, das Programm immer wieder mit veränderten Breakpoint- oder Checkpoint- Funktionen zu compilieren und auszuführen. Mit Kopieren und Einfügen, Suchen und Ersetzen ist auch das Einbauen, Abändern und Entfernen solcher Hilfsfunktionen kein Problem. Es gibt gewiefte Programmierer, die diese Arbeitsweise bevorzugen und die eingebauten Debugging-Vorkehrungen kaum verwenden.

Besonders einfache Checkpoint-Ausgaben
Bildschirmausgaben sind vergleichsweise langsam (Millisekunden). Sie können das Realzeitverhalten unseres Programms soweit verfälschen, daß es gar nicht mehr lauffähig ist. Abhilfe: wir versuchen, auf schnellstem Wege entsprechende Signale auszugeben. Brauchbare Signalwege:

- die Bussysteme und Interfaces des PCs,
- anwendungsseitige Anschlüsse (E-A-Ports o. dergl.).

Signaldarstellung:

- Anzeige über entsprechende Diagnoseadapter (Abb. 6),
- Signalisierung von Impulsmustern, die z. B. mittels Ozilloskop oder Logikanalysator ausgewertet werden können

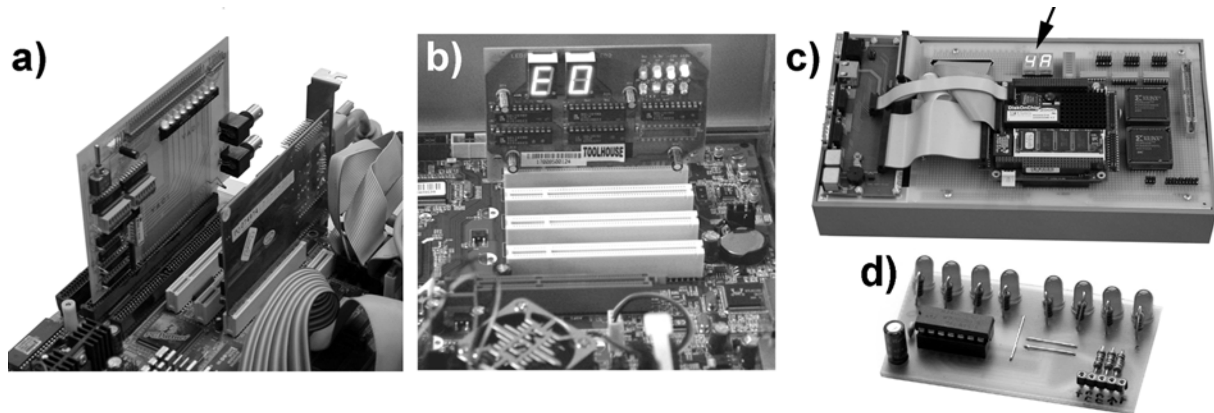


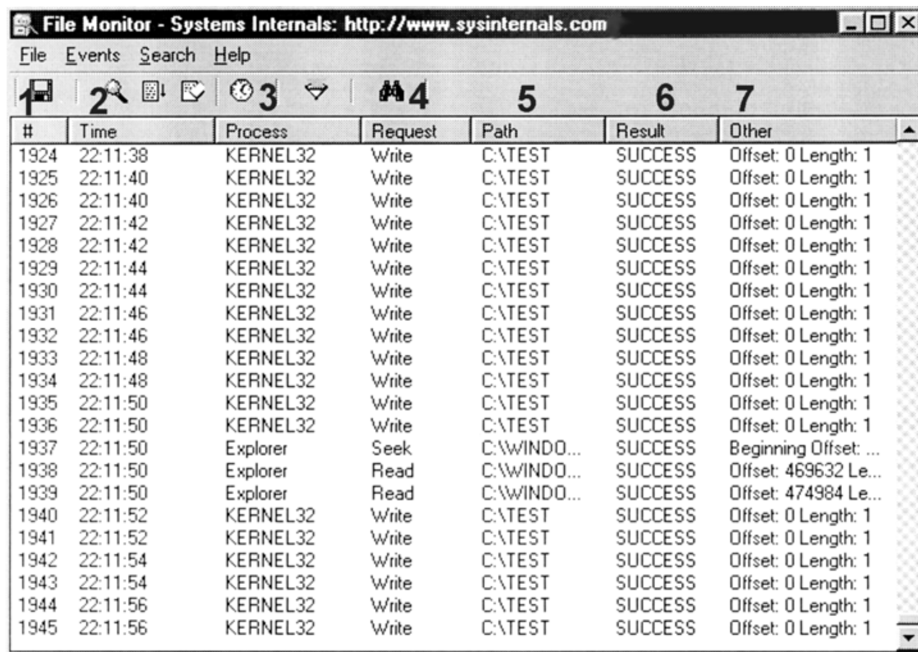
Abb. 6 Diagnoseadapter.a) am ISA-Bus; b) am PCI-Bus; c) PC/104-Versuchsplattform mit eingebautem Diagnoseadapter (Pfeil); d) zum Anschließen an Mikrocontroller (Schieberegisterschnittstelle)

Der Anfangstest (POST) der PCs gibt typischerweise Testfortschritts- und Fehlermeldungen in Form einzelner Bytes aus. Eine übliche Ausgabeadresse: 80H. Es werden Steckkarten angeboten, die diese Ausgaben anzeigen können (a, b). Die entsprechende Hardware kann man auch fest einbauen (c). Es liegt nahe, diese Ausgabe- und Anzeigemöglichkeit zum Debugging auszunutzen. Im Bereich der Mikrontroller sind dem Einfallsreichtum keine Grenzen gesetzt. Hier kommt es oft darauf an, mit möglichst wenigen Anschlüssen auszukommen. Unser Beispiel (d) zeigt acht LEDs, die an ein Schieberegister angeschlossen sind. Zur Informationsausgabe genügen zwei Signale (Taktausgang und Datenausgang).

Monitorprogramme
Ein Monitor ist hier ein Programm, das bestimmte Verarbeitungszustände, Systemaufrufe usw. anzeigt und protokolliert (Abb. 7).

In-Circuit-Emulatoren
Ein In-Circuit-Emulator (ICE) ist ein Prüfgerät, das typischerweise auf einer Sonderausführung (Bondout-Version) des jeweiligen Prozessorschaltkreises beruht. Hierbei sind ansonsten nicht zugängliche, aber für die Ablaufbeobachtung und Triggerung wichtige Signale auf zusätzliche Anschlüsse geführt. Dieser Prozessor wird in das zu prüfende System (Zielsystem) eingesetzt. Er ist zudem mit Trigger- und Ablaufspeichereinrichtungen sowie einem Bedienrechner verbunden.

Eingebaute Fehlersuchunterstützung
Tabelle 2 nennt die einschlägigen Vorkehrungen in den IA-32-Prozessoren (sie sind zwar in den meisten Typen nahezu gleichartig implementiert, aber in den Einzelheiten modellspezifisch). Sie wirken so, daß in bestimmten, programmseitig wählbaren Situationen ein Fehlersuchprogramm (Debugger) gerufen werden kann. Moderne Prozessoren stellen zudem Angaben zur Vorgeschichte des aktuellen Ablaufs bereit, um die Ablaufverfolgung zu unterstützen.



#	Time	Process	Request	Path	Result	Other
1924	22:11:38	KERNEL32	Write	C:\TEST	SUCCESS	Offset: 0 Length: 1
1925	22:11:40	KERNEL32	Write	C:\TEST	SUCCESS	Offset: 0 Length: 1
1926	22:11:40	KERNEL32	Write	C:\TEST	SUCCESS	Offset: 0 Length: 1
1927	22:11:42	KERNEL32	Write	C:\TEST	SUCCESS	Offset: 0 Length: 1
1928	22:11:42	KERNEL32	Write	C:\TEST	SUCCESS	Offset: 0 Length: 1
1929	22:11:44	KERNEL32	Write	C:\TEST	SUCCESS	Offset: 0 Length: 1
1930	22:11:44	KERNEL32	Write	C:\TEST	SUCCESS	Offset: 0 Length: 1
1931	22:11:46	KERNEL32	Write	C:\TEST	SUCCESS	Offset: 0 Length: 1
1932	22:11:46	KERNEL32	Write	C:\TEST	SUCCESS	Offset: 0 Length: 1
1933	22:11:48	KERNEL32	Write	C:\TEST	SUCCESS	Offset: 0 Length: 1
1934	22:11:48	KERNEL32	Write	C:\TEST	SUCCESS	Offset: 0 Length: 1
1935	22:11:50	KERNEL32	Write	C:\TEST	SUCCESS	Offset: 0 Length: 1
1936	22:11:50	KERNEL32	Write	C:\TEST	SUCCESS	Offset: 0 Length: 1
1937	22:11:50	Explorer	Seek	C:\WINDO...	SUCCESS	Beginning Offset: ...
1938	22:11:50	Explorer	Read	C:\WINDO...	SUCCESS	Offset: 469632 Le...
1939	22:11:50	Explorer	Read	C:\WINDO...	SUCCESS	Offset: 474984 Le...
1940	22:11:52	KERNEL32	Write	C:\TEST	SUCCESS	Offset: 0 Length: 1
1941	22:11:52	KERNEL32	Write	C:\TEST	SUCCESS	Offset: 0 Length: 1
1942	22:11:54	KERNEL32	Write	C:\TEST	SUCCESS	Offset: 0 Length: 1
1943	22:11:54	KERNEL32	Write	C:\TEST	SUCCESS	Offset: 0 Length: 1
1944	22:11:56	KERNEL32	Write	C:\TEST	SUCCESS	Offset: 0 Length: 1
1945	22:11:56	KERNEL32	Write	C:\TEST	SUCCESS	Offset: 0 Length: 1

1 - laufende Nummer; 2 - Uhrzeit; 3 - Programm, das den Zugriff ausgeführt hat; 4 - Art des Zugriffs; 5 - Pfad; 6 - Ergebnis (in Ordnung oder fehlerhaft); 7 - Aufrufparameter.

Abb. 7 Anzeige eines Windows-Monitorprogramms. Dieses Programm zeichnet alle Dateizugriffe auf

Vorkehrung	Wirkungsweise
Schrittbetrieb	bei gesetztem Flagbit TF (Trap Flag) wird nach Ausführung eines jede Befehls eine *Debug Exception**) wirksam
Taskumschaltung	ist nach einer Taskumschaltung im Taskzustandssegment (TSS) der neuen Task das Trap-Bit (T) gesetzt, so wird eine *Debug Exception**) wirksam
Adreßvergleich	bis zu 4 Vergleichsadressen sowie Angaben, die die gewünschte Stopbedingung kennzeichnen (Befehlslesen, Daten Lesen, Daten Schreiben usw.), können in Fehlersuchregister (Breakpoint Address Registers) geladen werden. Tritt eine solche Vergleichsbedingung ein, so wird eine *Debug Exception**) wirksam
Breakpoint-Befehl	die Ausführung dieses Befehls bewirkt die Auslösung einer *Breakpoint Exception**)
Adreßrettung zwecks Ablaufverfolgung	verschiedene maschinenspezifische Register (MSRs) enthalten die Befehls - und die Zieladresse der letzten Verzweigung sowie die aktuelle Befehlsadresse vor der letzten Unterbrechungsauslösung

*): d. h., es wird jeweils ein bestimmter Interrupt ausgelöst

Tabelle 2 Softwareseitig zugängliche Debugging-Vorkehrungen der IA-32-Prozessoren (nach Intel)

Ein besonderer Computer zum Fehlersuchen

Läuft alles – die Fehlersuch- und die (fehlerbehaftete) Anwendungssoftware – auf demselben Computer, so ist offensichtlich mit Problemen zu rechnen:

- die Anzeigen des Programms, in dem Fehler gesucht werden sollen, werden von den Anzeigen des Debuggers überdeckt,

- der Debugger belegt Ressourcen der Systems, die somit andern Programmen nicht mehr zur Verfügung stehen,
- Abstürze während der Fehlersuche sind Systemabstürze – mit der Konsequenz, daß das gesamte System wieder hochzufahren ist,
- Probleme mit der Hardware oder schwere Programmierfehler verhindern oft, daß die Debugging-Software überhaupt laufen kann.

Es liegt deshalb nahe, Programmentwicklung und Fehlersuchen auf einen weiteren Computer auszulagern. Somit ergibt sich eine Konfiguration aus Entwicklungssystem und Zielsystem.

Debug Port und Test Access Port (TAP)
Diese Fachbegriffe bezeichnen Schnittstellen, über die das Zielsystem an das Entwicklungssystem angeschlossen wird. Es gibt zwei grundsätzliche Auslegungen:

- der softwareseitige Zugang. Im Zielsystem läuft nach wie vor ein Debugger, der sich aber zwecks Bedienung und Anzeige mit dem Entwicklungssystem in Verbindung setzt. Der typische Debug Port hat eine serielle Schnittstelle (vgl. Abbildung 3.29). Zudem wird meist noch eine Netzwerkschnittstelle unterstützt.
- der hardwareseitige Zugang. Manche Prozesoren haben eingebaute Fehlersuchvorkehrungen, die auf einem Weg erreicht werden können, der von der üblichen Programmausführung unabhängig ist. Hierzu gibt es eine besondere Schnittstelle: den Test Access Port TAP. Sie ist typischerweise als Boundary-Scan-Interface gemäß Standard IEEE1149.1 ausgelegt.

In-Target Probe ITP
Das ist ein Fehlersuchhilfsmittel, das aus Software und entsprechender Koppelhardware besteht (Abb. 8). Die Software läuft auf dem Entwicklungsrechner (einem üblichen PC), der mit einer speziellen Steckkarte auszurüsten ist. Diese wird über ein Kabel mit dem TAP-Anschluß des Zielsystems verbunden

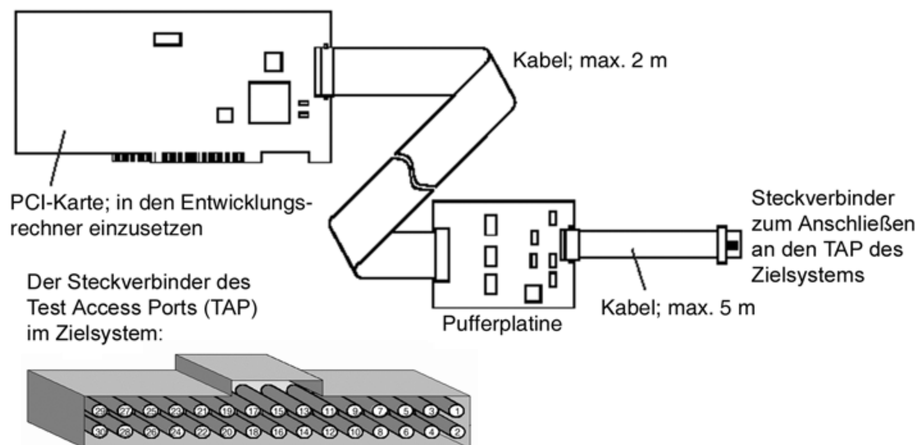


Abb. 8 In-Target-Probe ITP (nach Intel)

Programmentwicklung in Windows-Umgebungen
Typische Anforderungen an die Entwicklungshilfsmittel (zuzüglich zur ohnehin erforderlichen Entwicklungssoftware) seien anhand von vier Fallbeispielen kurz aufgezählt.

1. Anwendungen auf Grundlage integrierter Entwicklungsumgebungen
Typischerweise genügen die Debugging-Vorkehrungen der Entwicklungsumgebung (vgl. Abb. 5), ergänzt um altbewährte Tricks.

2. Ehrgeizigere Anwendungen (bis hin zum direkten Aufsetzen auf der Win32-API)
Monitorprogramme, ggf. ein spezieller Windows-Debugger. Nahezu unentbehrlich: Microsoft SDK (Software Development Kit).

3. Gerätetreiber für übliche Hardware (Geräte mit bekannten Kommandosätzen o. dergl.)
Wie Fallbeispiel 2. Zuzüglich Microsoft DDK (Driver Development Kit). Ggf. Sonderversion des Betriebssystems installieren (in solchen Versionen werden die Systemaufrufe mit Debug-Informationen angereichert, die von einem Windows-Debugger ausgewertet werden können). Zur Programmentwicklung passenden Compiler wählen (soll heißen: typischerweise einen von Microsoft – nicht alle Compiler anderer Anbieter erzeugen korrekten Code für Gerätetreiber).

4. Besonders ehrgeizige Vorhaben
(Betrifft die Ansteuerung komplizierter Sonderhardware, Realzeitanwendungen usw.) Wie Fallbeispiel 3. Ggf. zu ergänzen durch einen ITP-Debugger (erfordert Zielsystem mit TAP-Anschluß).

Hinweise:

1. In Fallbeispiel 1 kommt man meist mit einem einzigen PC aus. Ansonsten ist es sinnvoll, einen Arbeitsplatz aus Entwicklungssystem und Zielsystem aufzubauen.
2. Wird das Zielsystem entsprechend konfiguriert, ist es sogar möglich, Unterstützung aus der Ferne anzufordern (z. B. vom Lieferanten des Betriebssystems; Abb. 9).
3. Bei weitem nicht alle Motherboard haben einen TAP-Anschluß (oder eine Freifläche zum Bestücken eines entsprechenden Steckverbinders). Es ist also gezielt nachzufragen, wenn ein entsprechender PC benötigt wird.

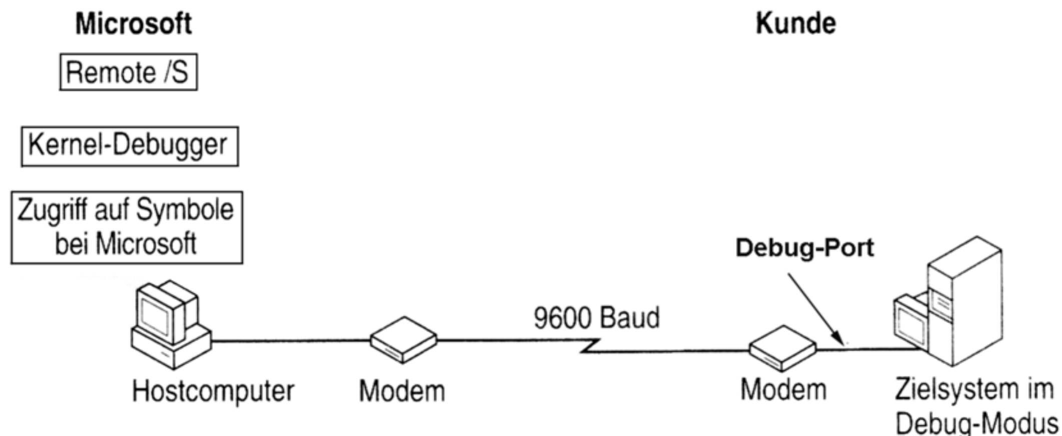


Abb. 9 Aus der Ferne Fehler suchen: Windows-Debugging als Remote-Sitzung (nach Microsoft)

PCs wie Mikrocontroller programmieren

Wir nehmen den PC, wie er ist, nutzen ihn aber gleichsam mikrocontrollermäßig (hardware-nahe, auf Verarbeitungsgeschwindigkeit ausgerichtete Programmierweise). Typische Entwicklungsumgebungen: C oder Basic oder Pascal (Delphi/Kylix), ggf. Inline-Assembler (für E-A-Abläufe usw.).

Die Vorteile (im Vergleich zum Emulatorbetrieb):

- es gibt keine Einbußen an Verarbeitungsleistung,
- wir können alle im PC-Bereich verfügbaren Programmentwicklungswerkzeuge ausnutzen.

Achtung, wenn es auf Geschwindigkeit ankommt: keine Systemaufrufe in den zeitkritischen Abläufen, also keine Tastaturabfragen, Bildschirmausgaben usw. (solche Systemaufrufe sind schnell hingeschrieben, dauern aber Millisekunden).

In die Anwendung eingebaute Debugging-Unterstützung
Es ist sinnvoll, entsprechende Debugging-Unterstützung von Anfang an einzuplanen. In unseren Einsatzfällen betrifft das vor allem die E-A-Zugriffe. Typische Debugging-Funktionen: Schrittbetrieb (von Zugriff zu Zugriff), Vergleichsstop, Zugriffsaufzeichnung (Protokollierung) und Zugriffssimulation (zum Erproben ohne angeschlossene E-A-Hardware). Eine naheliegene Auslegung: wir schreiben zunächst ein Rahmenprogramm, das nur die zum Debugging erforderliche Oberfläche sowie die entsprechend unterstützten E-A-Unterprogramme umfaßt (Abb. 10, 11). Ist eine Anwendung zu programmieren, so nehmen wir das Rahmenprogramm gleichsam als Schablone und füllen es mit den anwendungsspezifischen Abläufen aus (vgl. die Vorgehensweise bei der Nutzung von Visual Basic, Visual C++ usw.).

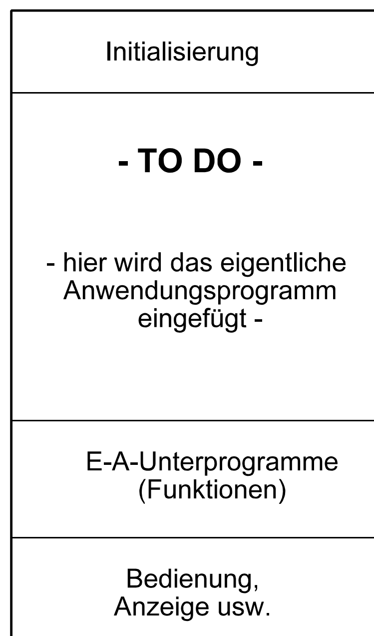


Abb. 10 Rahmenprogramm mit eingebauter Debugging-Unterstützung

```
 1   2   3        4                              5
=========== Microcontroller Program Template / IDE/ATA 0.9 ===========

  0   0  DIR  Port A:   1100 0011              S - Simulate
  1   7  DAT  Port C:   0000 0011              E - Execute
  2   8  DAT  Port D:   0000 1100
  3   9  DAT  Port E:   0100 0010              P - Single Step
  4   4  DIR  Port E:   0000 0100              R - RUN
  5   0  DIR  Port A:   1001 0010
  6   0       -------   0000 0000              I - Stop on input:  -
  7   0       -------   0000 0000              O - Stop on output: -   6
  8   0       -------   0000 0000              D - Stop on dir:    -
  9   0       -------   0000 0000
 10   0       -------   0000 0000              L - Logout
 11   0       -------   0000 0000
 12   0       -------   0000 0000              A - Run again
 13   0       -------   0000 0000              X - Exit
 14   0       -------   0000 0000         7
 15   0       -------   0000 0000       Port C: 2F

            A           B           C           D           E
  DAT:  0000 0000   0000 0000   0000 0011   0000 1100   0100 0010
8 DIR:  1001 0010   0000 0000   0000 0000   0000 0000   0000 0100
  PORT: 0000 0000   0000 0000   0000 0000   0000 0000   0000 0000
```

1 - laufende Nummer; 2 - interne E-A-Adresse; 3 - symbolische Bezeichnung; 4 - Bitmuster; 5 - Bedienmenü; 6 - Stopfunktionen; 7 - Beispiel einer simulierten Eingabe; 8 - Portanzeige. Die Positionen 1 bis 4 bilden die sog. Logout-Anzeige.

Abb. 11 Eine Debugging-Anzeige

Typische Betriebszustände und Funktionen:

- Simulationszustand (Simulate). Es läuft alles intern. E-A-Hardware wird nicht angesprochen. Ausgaben werden aufgezeichnet und angezeigt, Eingaben zum Bildschirm umgeleitet (manuelle Eingabe; vgl. Position 7 in der Abbildung). In dieser Betriebsart ist das Programm auch unter Windows lauffähig.
- Ausführungszustand (Execute). Läuft mit Zugriff auf die angeschlossen E-A-Hardware (im Beispiel nur unter DOS).
- Schrittbetrieb (Single Step). Weiterschalten von E-A-Zugriff zu E-A-Zugriff. Zugriffe werden angezeigt.
- Normalbetrieb (Run). Programm läuft mit voller Geschwindigket. Bildschirmanzeige wird nicht aufgefrischt.
- Stopfunktionen (Stop on Input/Output/Dir). Es ist der jeweils gewünschte Portbezeichner (A, B usw.) einzugeben. Wird ein entsprechender Zugriff ausgeführt, so wechselt das Programm aus dem Normalbetrieb in den Schrittbetrieb.
- Logout-Aufzeichnung. Ist diese Funktion aktiv, so werden die E-A-Zugriffe in einem Logout-Puffer aufgezeichnet (im Beispiel kann der Puffer 8000 Zugriffe aufnehmen). Der Pufferinhalt kann auf dem Bildschirm betrachtet werden (vgl. die Positionen 1 bis 4 in der Abbildung).

Typische E-A-Funktionen:

- PORTINIT: alle Ports initialisieren.
- OUTPORT (Portregister, Datenbyte). Datenbyte ausgeben.
- SETBIT (Portregister, Bitposition, Bitwert). Ein einzelnes Bit in einem Portregister gemäß Bitwert (0 oder 1) setzen.
- MODIFY (Portregister, Maskenbyte, Datenbyte). Portregisterinhalt ändern. Die Positionen, an denen das Maskenbyte eine Eins hat, werden mit der Belegung des Datenbytes überschrieben.

- INPORT (Portregister). Byte vom Port einlesen.
- INPORTIN (Portregister). Byte vom Port einlesen. Es werden nur jene Bitpositionen zurückgegeben, die im Richtungsregister auf Eingabe gestellt sind (zum Aufbau entsprechender E-A-Ports vgl. Abschnitt 5.4.3). Die verbleibenden Bitpositionen werden mit Nullen belegt.

Wie funktioniert ein Emulator?

Ein Emulator ist ein Programm, das die Funktionsweise einer bestimmten Rechnerarchitektur – vor allem: die Wirkungen der Maschinenbefehle – mittels Software nachbildet.

Ein typischer Einsatzfall: wir haben ein Program, das zur Ausführung auf Maschinen einer bestimmten Rechnerarchitektur X (Zielarchitektur) vorgesehen ist und als Maschinencode vorliegt, es gibt aber keinen entsprechenden Computer, der dieses Programm ausführen könnte. Der Ausweg: wir nehmen eine Maschine einer anderen Architektur (Plattform-Architektur) und schreiben ein Programm, das die Funktionsweise der Zielarchitektur nachbildet (emuliert).

An sich ist ein Emulator nicht besonders kompliziert. Wir müssen für alle Speichermittel der Zielmaschine entsprechende Speicherbereiche vorsehen, also für den Programmspeicher, für den Datenspeicher, für den Registersatz usw. (Abb. 12). Diese Speichermittel werden zweckmäßigerweise mit Feldstrukturen (Arrays) nachgebildet. Das zu emulierende Programm wird in das Befehls-Array geladen. Der Emulator holt Maschinenbefehl für Maschinenbefehl aus diesem Array und bildet dessen Wirkungsweise nach. Es ergibt sich eine recht einfache Programmschleife (Abb. 13).

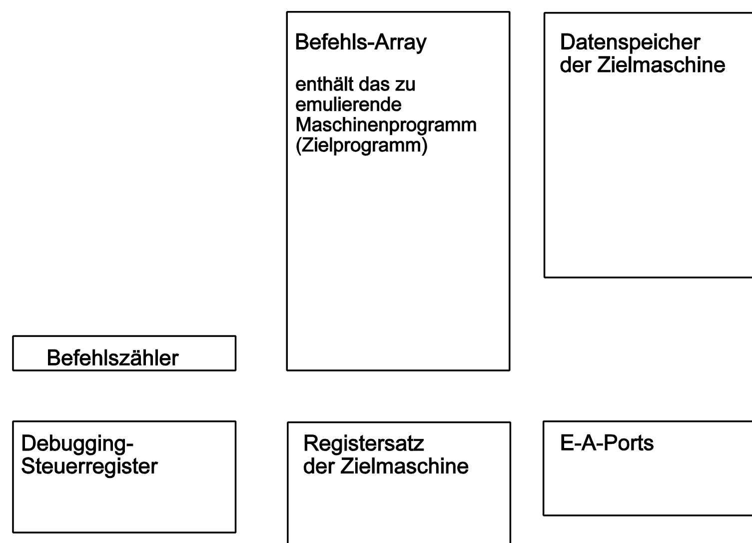


Abb. 12 Datenstrukturen eines typischer Emulators

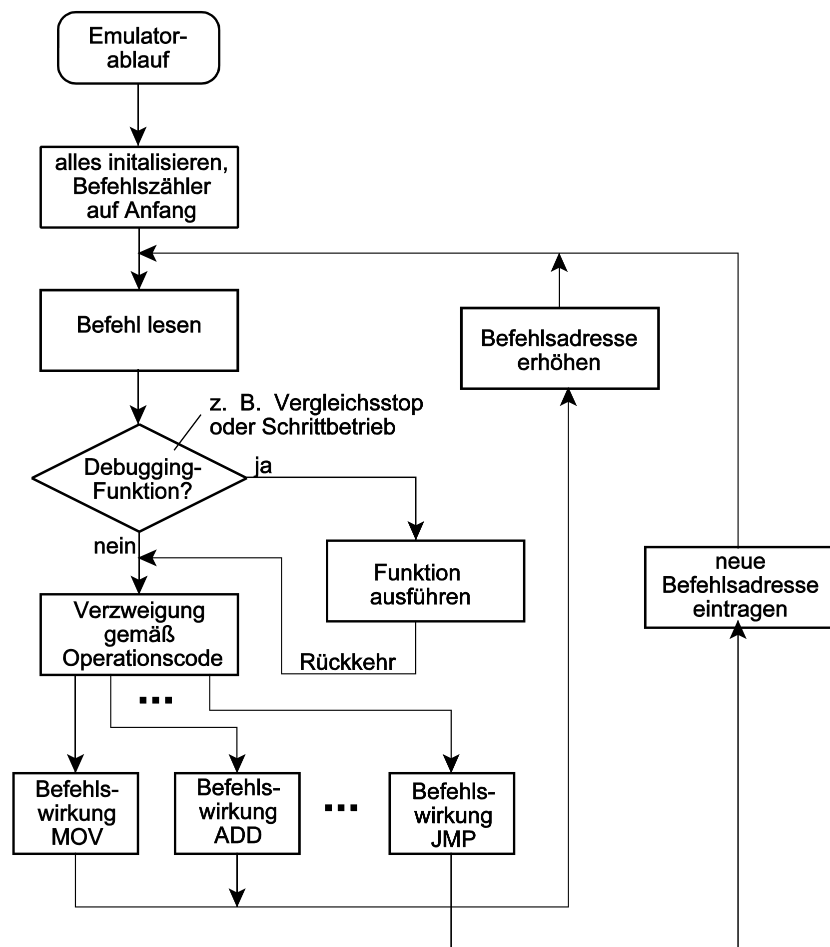


Abb. 13 Ein typischer Emulator im Ablaufdiagramm

Verfeinerungen:

- Nachbildung des Interruptmechanismus. Vor der Ausführung eines jeden Befehls sind Unterbrechungsbedingungen abzufragen und ggf. zu behandeln.
- Debugging-Funktionen. Die eigentliche Stärke eines guten Emulators. Da alles über Software läuft, sind auch alle Einzelheiten der programmseitigenAuswertung zugänglich. Als Beispiel einer entsprechenden Bedienoberfläche vgl. Abbildung 3.10.

Emulator und Simulator
Die Übergänge sind fließend. Eine Unterscheidung für den Hausgebrauch: der Emulator soll von Grund auf Verarbeitungsleistung bringen (und dabei mit echter E-A-Hardware zusammenwirken), der Simulator dient lediglich zum Verfolgen und Debuggen von Programmabläufen. Manche Simulatoren befassen sich gar nicht mit der Ein- und Ausgabe, manche bilden eine elementare E-A-Hardware softwaremäßig nach. Simulatoren sind vor allem zum Lernen gedacht und dazu, den Entwicklern Gelegenheit zu geben, ihre Programme am PC zu erproben – also ohne Zugriff auf die jeweilige Zielhardware.

Typische Anwendungsbeispiele:

1. das gleichzeitige Entwickeln von Hard- und Software, wobei es die Zielhardware in der ersten Zeit noch gar nicht gibt).

2. Fehlersuche. Auch schwerste Fehler im Zielprogramm bringen einen Emulator nicht zum Absturz; es ist auch in solchen Fällen möglich, den Programmablauf in allen Einzelheiten zu beobachten. Allein aus diesem Grunde kann es zweckmäßig sein, auch weit verbreitete und längst bekannte Architekturen zu emulieren.
3. Virensuche. Verdächtige Befehlsfolgen werden emuliert, um zu erkennen, ob sie wirklich in der Lage sind, Schaden anzurichten.

Auf Geschwindigkeit programmieren
Wenn der Emulator so schnell laufen soll, daß der nachgebildete Mikrocontroller mit einem richtigen Schaltkreis mithalten kann, haben wir nichts zu verschenken. Andererseits ist ein Emulator nur dann wirklich von Nutzen, wenn er eine umfassende Debugging-Unterstützung bereitstellt. Somit liegt es nahe, mehrere Geschwindigkeitsstufen vorzusehen:

- Emulation in reiner Form (ohne Debugging und Anzeige). Keine Verlangsamung zulässig.
- Emulation mit Analysefunktionen zwecks Debugging (z. B. Adreßvergleich). Verlangsamung darf kaum merklich sein.
- Emulation mit Bedienung und Anzeige (Schrittbetrieb usw.). Wenn wir auf Bildschirmanzeige, Tastenabfrage usw. umgeschaltet haben, können wir uns Zeit lassen.

Aus diesen Anforderungen heraus ergeben sich zwei Programmierweisen:

- hohe Geschwindigkeit, aber keine Bildschirmanzeigen usw.:
  - Emulatorschleife nicht verlassen,
  - keine Unterprogrammaufrufe in der Emulatorschleife (und schon gar keine Systemaufrufe),
  - schnelle Entscheidungen über Debugging-Verzweigung (Vergleichsstop usw.),
  - kritische Abläufe in Assembler codieren (vor allem die Verzweigung gemäß Operationscode[1)]).
- mit Bedienung und Anzeige (Schrittbetrieb, Simulatorbetrieb). Es kommt nicht auf Geschwindigkeit an. Somit können wir es uns leisten, den Compiler für uns arbeiten zu lassen und die Unterstützung des Systems in Anspruch zu nehmen (übliche Hochsprachen-Programmierung).

1): wer weiß schon, was der Compiler aus einer SWITCH- oder SELECT CASE-Anweisung macht? Die sicherste Lösung: eine zu Fuß programmierte Verzweigung über eine Sprungtabelle...