Von Wanzen und ihren Jägern

Die Debug-Register des 80386 - Teil 1

Christoph Ascherl

Der 80386-Prozessor bietet seine exzellente Hardware-Unterstützung nicht nur dem Betriebssystem an, vielmehr eröffnet er zugleich auch neue Möglichkeiten für die dazu gehörenden Debugger. Mit seinen acht Debug-Registern hat man einen leistungsfähigen Support für die Behandlung von Breakpoints. Dieser erste Teil des zweiteiligen Artikels [2. Teil] geht zunächst auf die `Theorie´ der Debug-Breakpoints ein.

Unterthema: Die Ursachen für einen Interrupt 1

Debugger (von englisch `bug´: Ungeziefer, Wanze, Käfer, Fehler, Defekt; ein solches Insekt soll einmal in Röhren/Relais-Zeiten den UNIAC-Rechner lahmgelegt haben) dienen dem Aufspüren von Fehlern; bekannte Vertreter dieser Gattung sind zum Beispiel Debug und AFD sowie CodeView, Periscope oder Turbo-Debugger. Jeder Debugger kennt die Möglichkeit, Haltepunkte zu setzen, an denen die Programmausführung gestoppt wird: sogenannte Breakpoints.

Ohne Hardware-Unterstützung ist der Einsatzbereich dieser Breakpoints jedoch arg

eingeschränkt. Die Debug-Register der 386 können hier erhebliche Verbesserungen einbringen, wenngleich sie nicht ganz mit den vielfältigen Debug-Möglichkeiten einer (kostspieligen) Debug-Slotkarte konkurrieren können.

INT 3 jetzt antik

Zum besseren Verständnis seien an dieser Stelle zunächst Prinzip und Einschränkungen der bisherigen Methode dargestellt. Alle folgenden Ausführungen beziehen sich sowohl auf den Real als auch auf den Protected Mode. Die `Realisten´ brauchen nur die `E´s bei EFLAGS, EIP et cetera zu `überlesen´.

Mit Hilfe des Assembler-Befehls INT 3 (Software-Interrupt), welcher als einziger der INT-Befehle einen 1-Byte-Opcode besitzt, wird beim Setzen eines Breakpoints der Applikationscode dahingehend abgeändert, daß man an der entsprechenden Adresse anstelle des ursprünglichen Opcodes einen INT 3 patcht (englisch: to patch = flicken, ausbessern). Bei Ausführung des Codes wird an dieser Stelle dann in den INT-3-Interrupt-Handler verzweigt, welcher den Debugger zu aktivieren hat. Vor Beendigung dieses Interrupt-Handlers muß dann der vor dem Patchen hoffentlich abgespeicherte, ursprüngliche Opcode an seine alte Stelle zurückgeschrieben und der Stack manipuliert werden. Auf ihm steht zuoberst die Rücksprungadresse CS:(E)IP (16-Bit-Segment/Selektor, 16/32-Bit-Offset), die auf die Anweisung nach dem INT 3 zeigt, sowie die (E)FLAGS. Damit korrekt fortgefahren werden kann, muß also auf jeden Fall (E)IP (auf dem Stack!) dekrementiert werden.

Soll der Breakpoint weiterhin Gültigkeit behalten, greift man zu folgendem Trick: neben (E) IP wird auch das Abbild der (E)FLAGS auf dem Stack manipuliert. Das Trap-Flag leistet hierbei gute Dienste, da es nach genau einer Assembler-Anweisung den Interrupt 1 auslöst (man nennt dies auch den Einzelschritt- oder Single-Step-Modus des Prozessors). Der entsprechende Handler muß nun lediglich an die alte Adresse wieder INT 3 patchen.

So sieht die heutzutage mittlerweile etwas antiquierte Methode aus, die schon beim 8086 verfügbar war, aber auch heute noch funktioniert. Ihre Nachteile liegen klar auf der Hand: Breakpoints im ROM sind unmöglich, im RAM sind sie auf Codezugriffe beschränkt und nur global (bezogen auf Task Switches) möglich. Dafür ist ihre Anzahl aber praktisch nicht limitiert.

Der folgende Bericht beschreibt nun Grundlagen und Anwendung der Debug-Register. Die Erläuterungen beziehen sich sowohl auf den Real Mode als auch auf ein Multitasking-System (englisch: task = Prozeß, ein ablauffähiges Programm oder Programmteil) im Protected Mode, wobei letzteres mehr Debug-Möglichkeiten offeriert.

Für beide `Welten´ fällt zu guter Letzt auch ein nützliches Programm ab, wobei - besonders im nächsten Teil - des öfteren auf die Intel-Tool-Kette mit C386, ASM386 ... sowie die Intel Testumgebungen ICE386 und MON386 Bezug genommen wird. Dies sollte jedoch nicht abschrecken; auf die Intel-spezifischen Eigenheiten wird entsprechend hingewiesen. Während der Entwicklungszeit waren Tools und Testumgebungen für den 80386 (besonders für den Protected Mode) noch recht selten.

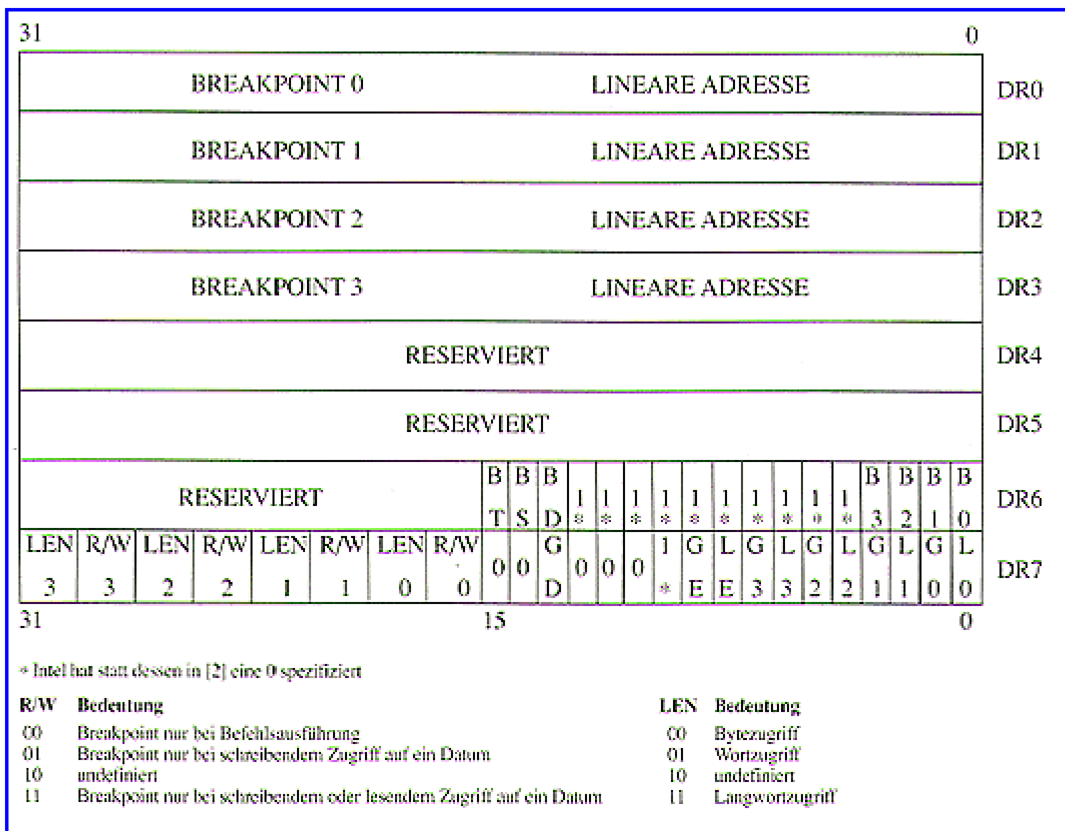
Auf die bei Microsoft so beliebten Übersetzungen der Fachbegriffe wurde bewußt verzichtet, da `Verletzung´, `Ausnahme´, `Debugger-Ausnahmebehandler´, `Task-Umschaltungs-Breakpunkt´ (konsequent wäre Prozeßumschaltungsbruchpunkt - alles klar?) und `nicht geschützter Modus´ [1] mehr Verwirrung stiften, als daß sie dem Leser nützlich wären.

Neue Debugging-Features

Der 80386 bietet acht Debug-Register (sie heißen DR0 bis DR7). DR4 und DR5 sind dabei für zukünftige Erweiterungen von Intel reserviert, Zugriffe darauf spiegeln derzeit DR6 und DR7 wider.

In den Debug-Registern DR0 bis DR3 kann man vier Breakpoint-Adressen deklarieren. Im Real Mode sind hier die physikalischen Adressen einzugeben, also nicht etwa, wie sonst üblich in der Segment:Offset-Form. Im Virtuellen und Protected Mode nehmen die `Debug Address Registers´ lineare Adressen auf, die gegebenenfalls noch über die Paging Unit in die physikalischen umgerechnet werden (siehe vorangegangene Artikel in c't bezüglich des Protected Mode [3, 4]). In Multitasking-Systemen im Protected Mode können auch Breakpoints auf Task Switch (= Prozeßwechsel) gesetzt werden, sofern das Betriebssystem den Hardware Task Switch des Prozessors nutzt.

Breakpoints mit den Debug-Registern sind sowohl bei Code- als auch bei Datenzugriffen möglich und dies sowohl im RAM als auch im ROM. Unterstützt ein Debugger nur globale Breakpoints, könnten lediglich vier gleichzeitig mögliche Breakpoints eine Einschränkung darstellen, und man muß weiterhin zusätzlich auf die eingangs beschriebene INT-3/INT-1-Kombination zurückgreifen. In einem Multitasking-System sind Breakpoints jedoch auch lokal, das heißt taskspezifisch möglich. Hierbei hat man dann vier Breakpoints pro Task. Weiterhin können einzelne Tasks angehalten werden; es muß also nicht unbedingt das ganze System stehenbleiben. Der Debugger selbst ist dabei ebenfalls als eine (oder mehrere) Task(s) zu realisieren.



Die Debug-Register des 80386 bieten eine leistungsfähige Unterstützung für die Breakpoint-Behandlung in Debuggern.

DR0 bis DR3 stehen in direktem Zusammenhang mit dem Debug-Kontrollregister DR7, wo

weitere Spezifizierung und Selektierung erfolgen.

Alles im Griff: DR7

Mit Ausnahme von zwei Bits ist jedes der Steuerbits viermal vorhanden, je einmal für DR0 bis DR3. Damit sind diese vier Register unabhängig voneinander einsetzbar.

Für jedes der Debug-Register DR0 bis DR3 spezifizieren die korrespondierenden Bitfelder R/W0 bis R/W3 (read/write) sowie LEN0 bis LEN3 (length field) in DR7 die Bedingung, welche bei Erreichen einer der eingetragenen Adressen zu einer Debug Exception (INT 1) führt. Die Bits in R/Wn gliedern die Breakpoints: Breakpoint bei Befehlsausführung, Breakpoint bei Schreibzugriff auf ein Datum oder Breakpoint bei allgemeinem Zugriff auf ein Datum (schreibend oder lesend) sind möglich. Bei Breakpoints auf Daten kommen zusätzlich die Bits in LENn ins Spiel: sie bestimmen einen byte-, wort- oder doppelwortweisen Zugriff auf die Daten.

Dabei ist auf folgendes zu achten: legt man den überwachten Bereich als Wort fest (LENn = 1) wird Bit 0 der Adresse in DR0...DR3 ignoriert, bei Doppelwörtern (LENn = 2) entsprechend auch noch Bit 1. Folglich löst jeder Bytezugriff auf den durch Adresse und LENn spezifizierten Breakpoint-Bereich eine Debug Exception aus.

Will man allerdings ein Wort oder Doppelwort komplett überwachen, das im Speicher nicht auf eine Doppelwortgrenze ausgerichtet ist (Misalignment), so muß man dafür mindestens zwei Breakpoints opfern. Legt man auf Eindeutigkeit Wert, so sind mitunter drei Breakpoints vonnöten, zum Beispiel ein Byte auf Adresse xxxx:1, ein Wort auf xxxx:2 und ein Byte auf xxxx:4.

Bei Breakpoints auf Befehlsausführung sind außer der Länge `Byte´ alle anderen Eintragungen in die LEN-Felder unzulässig. Breakpoints dieser Art müssen auf das erste Byte einer Assembler-Anweisung gesetzt werden, andernfalls werden sie nicht erkannt.

Nicht im Reference Manual [2] erwähnt ist das GD-Bit (Guard Debug, Bit 13). Mit diesem `geheimen´ Bit kann sich ein Debugger vor unerlaubtem Zugriff durch die Applikation auf die Debug-Register schützen. Will eine Applikation darauf zugreifen, erfolgt automatisch ein INT 1. Dabei wird vor Eintritt in den Debug-Exception-Handler (= Interrupt-Handler) das GD-Bit gelöscht. Es ist dann vom Debugger bei Bedarf erneut zu setzen. Nebenbei: am ICE386 kann man dieses Bit mit dem Befehl `GDPROT = TRUE/FALSE´ manipulieren. Da sieht man mal, wie wichtig die Intel-Entwicklungsumgebung ist, um an `geheime´ Prozessorstrukturen heranzukommen!

Weitere acht Bits in DR7 (G0 bis G3 und L0 bis L3 - sie stehen für Global/Local Enable) - geben die Adressen in DR0 bis DR3 frei beziehungsweise sperren sie. Jedes Register läßt sich so den globalen oder den lokalen Breakpoints zuschlagen. Bei lokaler Freigabe wird dieses Register mit dem nächsten Task Switch wieder gesperrt, bei globaler Freigabe müssen die Register bei Bedarf explizit gesperrt werden, sind also in allen Tasks aktiv. Im Real Mode sind nur globale Breakpoints möglich; ein Hardware Task Switch ist diesem Modus ja völlig unbekannt.

Die restlichen beiden Bits (GE und LE, Global/Local Exact) gelten für alle Debug-Register gleichermaßen. Bei 386-Prozessor muß unbedingt eines von beiden gesetzt sein, wenn mit Daten-Breakpoints gearbeitet wird. Sonst erfolgt eventuell die Debug Exception erst mehrere Assembler-Anweisungen nach dem Datenzugriff; schlimmstenfalls wird sie gar nicht erkannt. Das liegt daran, daß gewöhnlich die Ausführung einer Anweisung mit darauffolgenden Speicherzugriffen überlappen kann. Während noch der Prozessor einen

Befehl abarbeitet, holt er schon die nächsten Langwörter (Prefetch Queue) und dekodiert sie (beim 486) vor (Instruction Queue).

Weiterhin besitzt jeder 80xxx-Prozessor (also bereits der 8088) einen Write-Cache: Ist nämlich beim Schreiben das Bus-Interface gerade beschäftigt, wartet der Prozessor nicht, sondern legt das Datum in einem Zwischenspeicher ab und setzt seine Arbeit fort.

GE beziehungsweise LE schalten dieses prozessorinterne Pipelining ab. Die Programmausführungszeit wird hierdurch allerdings merklich verringert. Beim 80386 und SX hält sich die Verzögerung noch in Grenzen (Nortons SI V4.5 meldet etwa 3 bis 5 Prozent Performance-Verlust). Relativ beträchtlich wird hingegen der 486 von der Debug-Überwachung gestört. Er verdoppelt jeden 1-Takt-Befehl auf zwei Takte. Außerdem ignoriert er GE und LE völlig: er breakt immer \`exakt´. LE wird wie Ln beim nächsten Task Switch wieder gelöscht.

Das Debug-Statusregister DR6 schließlich dient der Ermittlung einer Debug Exception. Diese kann acht verschiedene Ursachen haben, wobei auch mehrere Ursachen gleichzeitig eine Debug Exception auslösen können. Die Bits B0 bis B3 sind bei Erreichen einer in DR0 bis DR3 eingetragenen Adresse gesetzt, während BT bei einem Task Switch zu einer Task mit gesetztem T-Bit im TSS gesetzt wird (Debug Trap Bit im Task State Segment). Im TSS merken sich Multitasking-Systeme im Protected Mode den kompletten prozessorseitigen Kontext einer Task, wie zum Beispiel alle Registerinhalte, damit diese bei einem Prozeßwechsel nicht verloren gehen.

BS zeigt eine Debug Exception nach einem Einzelschritt an, wenn also im Einzelschrittmodus (gesetztes TF (Trap-Flag) in EFLAGS) ein Befehl ausgeführt wurde. Einen unerlaubten Zugriff auf ein Debug-Register verpetzt das BD-Bit.

Ein Debug-Exception-Handler sollte vorsichtigerweise auch die Möglichkeit einkalkulieren, daß der INT 1 softwaremäßig aufgerufen wird, in diesem Fall ist kein Bit in DR6 gesetzt. Vor dem Eintritt in den Debug-Exception-Handler setzt der Prozessor das entsprechende Statusbit, das Zurücksetzen muß jedoch \`von Hand´ erfolgen. Der Prozessor erklärt sich hierfür als für nicht zuständig. Das Zurücksetzen ist gleichwohl unbedingt erforderlich, da sonst bei der nächsten Debug Exception die Ursache nicht mehr eindeutig identifiziert werden kann. Auch die Breakpoint-Bits B0...B3 werden erst bei einem erneuten Breakpoint \`upgedatet´.

Bug im 80386

An dieser Stelle sei noch auf eine weitere \`Unschönheit´ (ist sehr milde ausgedrückt) hingewiesen, die sich im Laufe der Testphase gezeigt hat, zum Glück beschränkt auf die ältere B1-Maske des 386 (mit DX = 303 nach dem Reset).

Steht in mehreren Adreßregistern DR0 bis DR3 dieselbe, lineare Adresse, ist aber nur eines dieser Register in DR7 mit Ln oder Gn (n = 0...3) freigegeben, so werden bei Erreichen dieser Adresse alle entsprechenden Statusbits Bn gesetzt, also auch die Bits jener Adreßregister, die nicht in DR7 aktiviert wurden.

Dies hat für den Programmierer folgende Konsequenz: Entweder müssen die unbenutzten Adreßregister mit für diese Applikation \`ungültigen´ Werten belegt (prinzipiell gibt es keine ungültige lineare Adresse, da auf Grund des Paging-Mechanismus alle linearen auf physikalisch vorhandenen Adressen abgebildet sein können) oder aber es müssen bei der Auswertung der Statusbits aus DR6 die Kontrollbits aus DR7 mit einbezogen werden. Dies ist nicht nur mühsam, sondern kostet natürlich auch wertvolle Laufzeit im System, die ein

Realzeitverhalten unnötigerweise noch mehr beeinflußt.

Nach so viel Theorie muß nun leider noch weitere folgen, auf welche beim Schreiben eines Debug-Exception-Handlers unbedingt zu achten ist.

Fault oder Trap

Bei einer Debug Exception muß zwischen zwei Typen unterschieden werden: Fault und Trap. Bei einem Fault tritt die Exception vor Ausführung der die Exception verursachenden Anweisung auf, bei einem Trap ist dies erst danach der Fall. Der Instruction Pointer (zur Erinnerung: CS:(E)IP und (E)FLAGS werden bei einem Interrupt auf dem Stack abgelegt - dieser (E)IP ist hier gemeint) zeigt bei einem Fault also noch direkt auf die verursachende Assembler-Anweisung, während er bei einem Trap auf die nachfolgende zeigt. Eine Ermittlung der Adresse, an welcher ein Trap verursacht wurde, ist nicht möglich, da nicht bekannt ist, aus wie vielen Bytes diese Anweisung bestanden hat (dies ist bei Daten-Breakpoints zu beachten).

Dieser Sachverhalt hat nun Konsequenzen für den eigentlichen Debug-Exception-Handler. Die Breakpoints bezüglich Befehlsausführung sind vom Typ \`Fault´. Das bringt den Vorteil mit sich, daß sich ein Breakpoint bequem auf die zu überwachende Adresse setzen läßt, ohne daß bei einem Break die dortige Anweisung bereits abgearbeitet wurde. Nun muß aber nach dem Break eben diese Anweisung noch ausgeführt werden, was erhebliche Probleme aufwirft.

Soll nämlich der Breakpoint weiterhin aktiv bleiben, hat man im Nu die schönste Endlosschleife, da natürlich vor Ausführung dieser Anweisung wieder eine Debug Exception ausgelöst wird. Man kann sich das schön vor Augen führen, wenn man mit Turbo Debug (TD.EXE) einen \`Instruction Fetch´ überwacht (nicht auf JMP, CALL, INT, LOOP oder POPF setzen).

Drei Auswege bieten sich an. Der aufwendigste und weitreichendste bestünde in einer kompletten Disassemblierung des anstehenden Befehls, der dann vollständig zu emulieren wäre - ein Riesenaufwand angesichts der Vielzahl der 386-Befehle und Adressierungsarten. Jeder Debugger kennt jedoch einen wesentlich einfacheren Weg, nämlich den Einzelschrittmodus mit Hilfe des Trap-Flags TF (Bit 8) in (E)FLAGS.

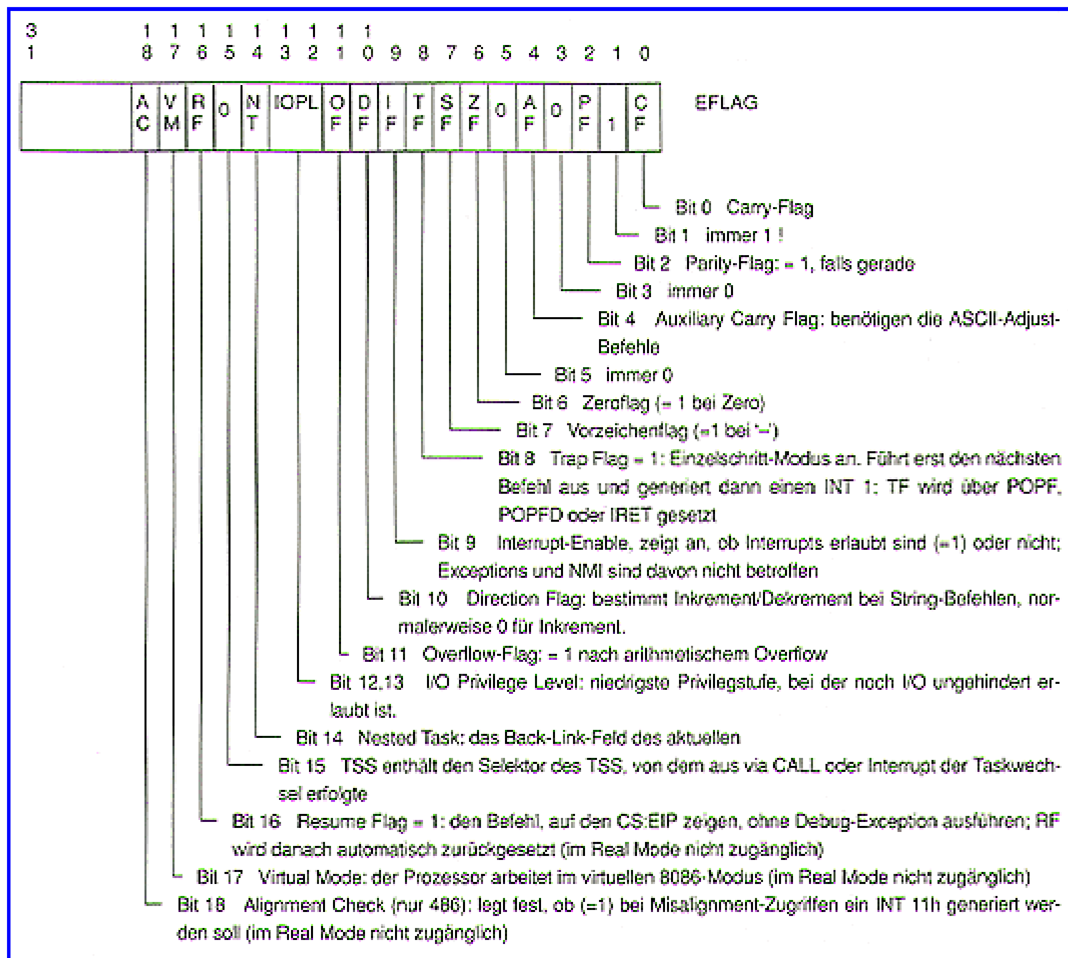
Wie bereits mehrfach erwähnt, \`pusht´ der Prozessor, wie bei jedem anderen Interrupt auch, vor Eintritt in den Debug-Exception-Handler CS:(E)IP und (E)FLAGS auf den Stack. In diesem Image (= Abbild) der (E)FLAGS ist das TF-Flag zu setzen. Beim Verlassen des Handlers mit IRET(D) holt der Prozessor nun CS:(E)IP und (E)FLAGS wieder vom Stack, womit dieses Flag für ihn dann tatsächlich gesetzt ist und er nach Abschluß des nächsten Befehls wieder einen INT 1 veranlaßt, diesmal mit BS in DR6 gesetzt. Vor der Befehlsausführung im Einzelschritt-Modus ist der Breakpoint in DR7 abzuschalten und hernach wieder zu reaktivieren. Außerdem darf man nicht vergessen, den Einzelschrittmodus wieder auszuschalten.

Allerdings klappt das Ganze nicht, wenn man es mit den schon erwähnten Befehlen JMP, CALL, INT, LOOP POPF, RET, IRET zu tun hat. Diese paar Befehle müßte ein guter Debug-Handler dann doch emulieren.

Einfacher per Resume

Im Protected Mode läßt sich der Vorgang noch erheblich vereinfachen dank des neu

eingeführten Bits 16 in EFLAGS. Es ist so neu, daß man sich selbst bei Intel noch nicht auf einen eindeutigen Namen einigen konnte: im Reference Manual [2] findet man für das RF-Flag sowohl die Bezeichnung Restart-Flag als auch Resume-Flag (sprachlich korrekt müßte man von T-Flag und R-Flag sprechen, die doppeltgemoppelte Form TF- und RF-Flag ist aber wesentlich übersichtlicher).



RF und TF in EFLAG dienen der Debug-Arbeit des 386/486-Prozessors.

Statt des TF-Flags setzt man das im Real Mode unerreichbare RF-Flag im EFLAGS-Image auf dem Stack. Die Folge ist, daß der Prozessor bei der nächsten Anweisung eine Debug Exception ignoriert. Bei erfolgreicher(!) Ausführung dieser Anweisung wird das RF-Flag automatisch gelöscht, sofern dies ähnlich wie beim Einzelschritt kein FAR JMP auf ein TSS (bedeutet einen Task Switch), ein CALL, INT, POPFD oder ein IRETD ist. Die letzten beiden Assembler-Anweisungen holen sowieso EFLAGS komplett neu vom Stack.

Bei anderen Exception-Faults (also nicht Interrupt 1) wird dieses RF-Flag automatisch gesetzt. Wichtig ist noch, daß beim Verlassen des Debug-Exception-Handlers mit gesetztem RF-Flag trotzdem noch eine Debug Exception vom Typ Trap sowie jede andere Exception wie zum Beispiel der im Protected Mode allseits beliebte Interrupt 13 (General Protection = verbotener Zugriff auf Daten oder Code) auftreten können.

Ein Breakpoint auf Daten ist nun andererseits vom Typ \`Trap´, was nichts anderes heißt, als daß die Debug Exception erst nach dem Zugriff auf das entsprechende Datum ausgelöst wird. Sollten die dabei möglicherweise überschriebenen Daten noch anderweitig benötigt werden, zum Beispiel um sie dem Anwender des Debuggers anzuzeigen, müssen diese Daten schon beim Setzen des Breakpoints explizit gerettet werden.

Fault und Trap können auch parallel auftreten: in einer Anweisung ist ein Breakpoint auf Daten erfüllt, auf der unmittelbar darauf folgenden Anweisung ist ein Breakpoint auf Befehlsausführung aktiv. Der Handler muß beide Ursachen entsprechend behandeln.

Soviel zur Theorie. Im zweiten Teil des Artikels wird dann die Praxis in Form der Vorstellung eines Konzepts für einen Debug-Exception-Handler folgen. (as)

Literatur

[1] Hansen, Stücklen, Die Debug-Register des Intel 386, Microsoft System Journal, 11/12 1989, S. 178 ff.

[2] 80386 Programmer´s Reference Manual, Intel, 1986

[3] Brett Glass, Protected Mode, c't 2/90, S. 232,

[4] Harald Albrecht, MSDOS in a box, c't 3, 4, 5/90

Kasten 1

Die Ursachen für einen Interrupt 1

Acht mögliche Gründe für eine Debug Exception. Die ersten vier können dabei auch in Kombinationen auftreten.

- Breakpoint-Adresse aus DR0 trifft zu
- Breakpoint-Adresse aus DR1 trifft zu
- Breakpoint-Adresse aus DR2 trifft zu
- Breakpoint-Adresse aus DR3 trifft zu
- Single-Step-Interrupt
- Taskwechsel
- unzulässiger Zugriff auf eines der Debug-Register
- Assemblerbefehl `INT 1´