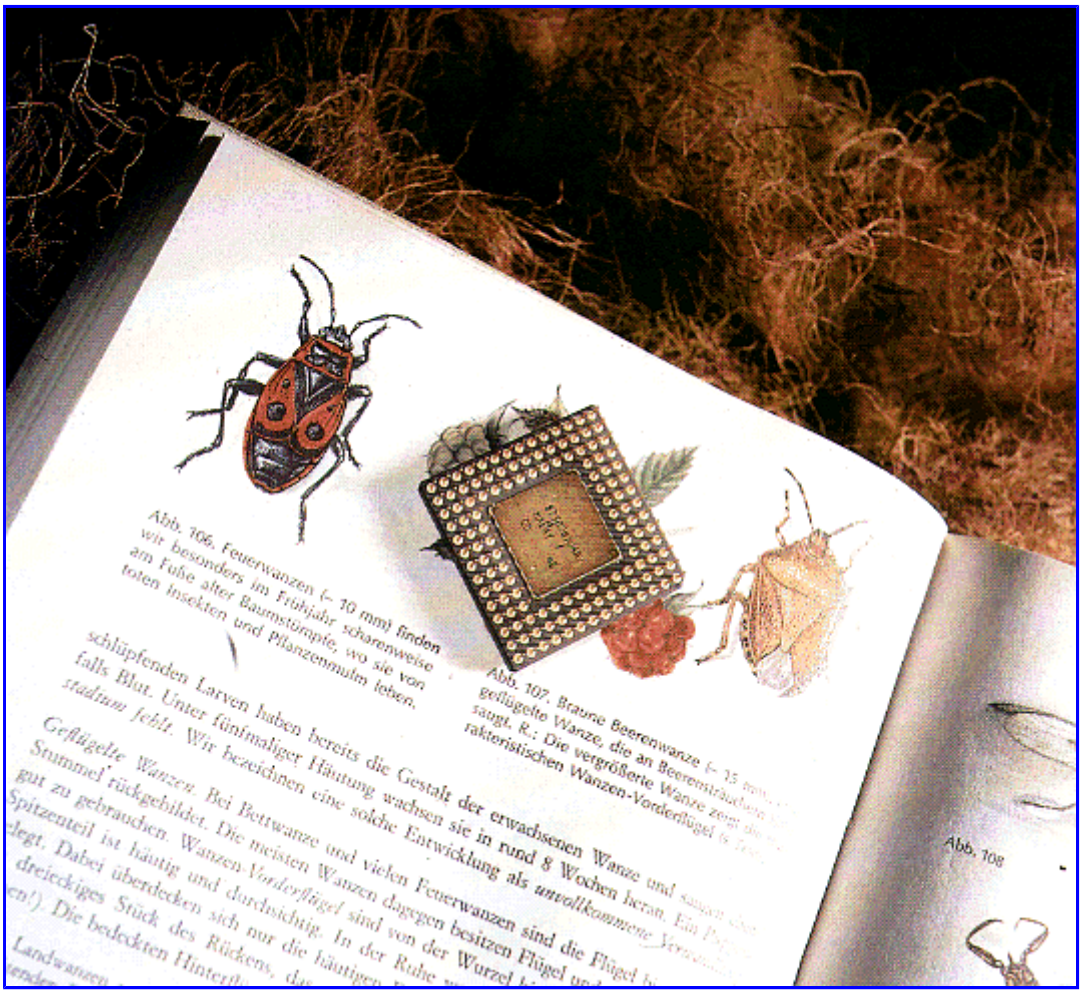

Von Wanzen und ihren Jägern

Die Programmierung der Debug-Register des 80386 - Teil II

Christoph Ascherl

Das Ausreizen der Debug-Register bis auf das letzte Bit ist Sache eines Debug-Exception-Handlers. Beim Setzen von Breakpoints sowie beim Erreichen derselben durch eine Applikations-Software wird dieser aktiviert. Ein Konzept für die Realisierung im Protected Mode mit der Intel-Entwicklungsumgebung sowie ein Erfahrungsbericht vom Testen eines Debug-Exception-Handlers sind Thema dieser Folge über die Debug-Register. Wie die Sache unter DOS (in Turbo-Pascal) aussehen kann, folgt dann im nächsten Teil.

Mit den Kenntnissen, die im ersten Teil vermittelt wurden, dürfte nun der Einsatz der Debug-Register keine größeren Probleme mehr bereiten, sofern nur eine Applikationstask vorhanden ist und die Breakpoints nur global verwendet werden oder auch bei Programmierung im Real Mode. Bei einem Multitasking-System ist jedoch ein Eingriff in das Betriebssystem unvermeidbar. Dieser Fall wird später im Beispiel dargestellt. Er ist der

komplizierteste und zugleich leistungsfähigste Einsatz der Debug-Register. Weniger versierte Programmierer sollten sich dennoch nicht abschrecken lassen, einzelne Teile dürften sicher auch für sie von Interesse sein.

Ansonsten müssen sie auf den Debug-Exception-Handler im Real Mode warten (Turbo-Pascal 5...6), der aus Platzgründen - er ist doch viel größer und leistungsfähiger geworden als ursprünglich vorgesehen - in die nächste Ausgabe verschoben wurde.

Bei der Entwicklung des Debug-Exception-Handlers für den Protected Mode standen die Software-Tools von Intel zur Verfügung: C386, ASM386, BND386 sowie BLD386. Diese Tools werden zunächst einmal vorgestellt.

Mit Intel in den Protected Mode

C-Compiler und Assembler müssen Code für den 80386 erzeugen können. Der BND386 ist der Binder. Er ist im Prinzip nichts anderes als der herkömmliche Linker, ist aber im speziellen für Protected-Mode-Software gedacht.

Den Abschluß bildet der BLD386, der Builder. Dieser enthält quasi eine eigene Programmiersprache und ist daher ein äußerst komplexes, wenngleich auch sehr leistungsfähiges Tool. Er entspricht im weiteren Sinne dem Locater LOC86 aus dem Real Mode. Ein Locater weist einem Programm feste Speicheradressen zu - dies geht nicht unter MSDOS, da dieses den Freispeicher selbst verwalten will und die Speicheradresse eines Programms daher nicht von vornherein festgelegt ist.

Die Testumgebungen waren ICE386 und MON386. Das ICE386 ist ein In-Circuit-Emulator, welcher den Prozessor ersetzt. Ein Vorteil dieses Gerätes ist die Möglichkeit, den Programmfluß aufzeichnen und anschließend analysieren zu können, der Nachteil ist sein enorm hoher Preis. Er ist daher ausschließlich im professionellen Bereich zu finden.

Der MON386 ist ein ganz gewöhnlicher Monitor wie zum Beispiel CodeView oder Turbo-Debugger, er ist jedoch auch im Protected Mode einsetzbar. Im Gegensatz zu einem In-Circuit-Emulator läuft die Software eines Monitors immer auf der Zielhardware ab und bedarf daher eigener Speicherbereiche.

Bevor das Konzept des Debug-Exception-Handlers nun näher zu erläutern ist, muß zuerst noch angemerkt werden, daß dieser im Protected Mode unbedingt auf Privilege Level 0 zu laufen hat, da die speziellen `MOV´-Assembler-Anweisungen (`MOV reg, DRn´ und `MOV DRn, reg´) nur auf diesem Level gültige Anweisungen sind. Dies muß entweder schon bei der Produktion mit dem BLD386 bedacht werden, oder aber das Betriebssystem zieht generell Interrupt-Handler dynamisch auf den Level 0 herunter.

Bei Erreichen eines Breakpoints soll nicht, wie es beim ICE386 der Fall ist, das ganze System stehenbleiben, sondern lediglich eine Meldung an die Debug-Task erfolgen und/oder die entsprechende Task suspendiert (= vertagt, oder einfacher: angehalten) werden. Das Gesamtsystem läuft ansonsten weiter. Dies bedeutet, daß Breakpoints taskspezifisch angegeben und daher die Debug-Register lokal betrieben werden können. Sie werden nur aktiviert, wenn die entsprechende Task, für welche vorher ein Breakpoint angegeben wurde, am Laufen ist. Um dies zu erreichen, ist nach der Spezifizierung eines Breakpoints für die betroffene Task das Debug-Trap-Bit im TSS der Task zu setzen (jede Task hat ein eigenes TSS).

Erfolgt dann zu einem späteren Zeitpunkt ein Task-Switch zu dieser Task, wird zuerst der

Debug-Exception-Handler aktiviert. Dieser ermittelt den Task-Identifikator, welcher eine eindeutige Identifizierung einer jeden Task ermöglicht, belegt die Debug-Register DR0 bis maximal DR3 mit den zu dieser Task spezifizierten Breakpoints und gibt sie in DR7 lokal frei. Der Handler wird beendet, die Task läuft an, trifft irgendwann auf eine der in den Debug-Registern angegebenen Adressen, und der Debug-Exception-Handler wird wieder aktiviert.

Anhand der Statusbits in DR6 erkennt der Handler, daß es sich bei der Ursache um einen tatsächlichen Breakpoint handelt, und leitet die diesbezügliche Behandlung ein. Die Debug-Task kann mit einem vom Betriebssystem aus zur Verfügung zu stellenden Mechanismus signalisiert (= benachrichtigt) werden, und alles läuft dann normal weiter, oder aber die Applikations-Task kann suspendiert werden. Dazu muß dann aber vom Betriebssystem aus eine entsprechende Breakpoint-Liste realisiert sein.

Ein Problem tritt auf, wenn ein Breakpoint einer Task zugeordnet werden soll, die zum Zeitpunkt der Eingabe noch gar nicht existiert (und damit auch ihr TSS nicht). Hier ist zusätzlich ein Eingriff in den `Create-Task-Mechanismus´ des Betriebssystems erforderlich, damit dann beim Kreieren das Debug-Trap-Bit gesetzt werden kann.

Wie sich zeigt, ist eine Realisierung eines solchen Konzeptes mit größeren Eingriffen in das Betriebssystem verbunden und daher nur im Zusammenspiel mit diesem zu entwickeln. Das Konzept mit dem Debug-Trap-Bit im TSS bietet aber den Vorteil, wenigstens nicht in das Herz eines Multitasking-Betriebssystems, den Scheduler (dieser ist für den eigentlichen Task-Switch zuständig), eingreifen zu müssen.

Große Hürden: ICE386 und MON386

Spätestens beim Testen zeigt sich dann, daß die Debug-Register wirklich eine hervorragende Einrichtung sind ... so gut nämlich, daß der In-Circuit-Emulator ICE386 und der Monitor MON386 diese selbst `mißbrauchen´. Wie schön waren die Zeiten am I^2ICE (für 8086- bis 80286-Prozessoren zuständig), als ein ICE noch wirklich ein ICE war, mit dem man alles testen konnte, auch Debugger! Mit einer neuen Release will Intel wenigstens den Single-Step freigeben (Ankündigung).

Im folgenden soll nun erläutert werden, wie man ICE386 und MON386 halbwegs überlisten kann, um wenigstens einigermaßen kontrolliert einen Debug-Exception-Handler austesten zu können.

Wird der Handler in Assembler geschrieben, sollten drei Register spendiert werden, eines für Status aus DR6, Control aus DR7 und Address aus DR0/3. Die benötigten Debug-Register werden nur jeweils einmal an einer Stelle gelesen beziehungsweise beschrieben. Alle weiteren Zugriffe erfolgen dann nur noch über die ausgewählten Standardregister. Dies bietet den Vorteil, daß insbesondere der Schreibzugriff auf ein Debug-Register während der Testphase geNOPt (ein Patch mit der Assembler-Anweisung NOP) werden kann.

Wegen der vielfältigen, möglichen Ursachen für eine Debug-Exception kann der Handler sehr umfangreich werden, und daher sollte man, sofern es die Programmablaufzeit zuläßt, auch in einer höheren Programmiersprache schreiben. In C sind dazu auf jeden Fall die verfügbaren Registervariablen zu deklarieren (beim C386 sind es derer drei). Sie realisieren mittels Inline-Code praktisch den einzig sinnvollen Zugriff auf die Debug-Register. Der vorher aufgeführte Vorteil wird somit auch in C automatisch miterkauft. Man muß aber beachten, daß auch ein Handler in C mit `IRETD´ abzuschließen und das bereits beschriebene RF-Flag (zur Erinnerung: Restart- oder Resume-Flag) zu behandeln ist.

Beim Testen mit dem ICE386 ist es zunächst einmal wichtig zu wissen, wie sich dieses bei den verschiedenen Ursachen für eine Debug-Exception verhält. Es zeigt sich, daß ein Erreichen eines Breakpoints (durch Adresse im DR0 bis DR3 spezifiziert) hardwaremäßig abgefangen wird und somit kein Eintritt in den Debug-Exception-Handler erfolgt. Dieses auf den ersten Blick vielleicht nachteilig erscheinende Verhalten wandelt sich jedoch schon bald in einen großen Vorteil, wenn innerhalb des Handlers Breakpoints zum Testen notwendig sein könnten.

Setzt eine Applikation zu irgend einem Zeitpunkt einmal das Trap-Flag (aus EFLAGS), so wird der Debug-Exception-Handler ebenfalls nicht aktiviert. Vielmehr wird die Applikationssoftware angehalten, und das ICE386 wartet wie nach dem ICE-Kommando `ISTEP´ auf weitere Eingaben seitens des Testers. `ISTEP´ selbst ist wohl über diesen Weg realisiert. Abgesehen vom Handler kann das Programm weitergeführt werden.

Völlig unbrauchbar wird das ICE386, wenn bei einem Taskwechsel das Trap-Bit im TSS der neuen Task gesetzt ist. Die ermunternde Meldung `Unknown Break at address xxx´ besagt eigentlich alles, der Task-Switch wird natürlich nicht ausgeführt und der Debug-Exception-Handler erst recht nicht erreicht. Ein Fortfahren an dieser Stelle im Programm hat dann keinen Sinn mehr.

Den eigentlichen Dreh- und Angelpunkt für einen erfolgreichen Test mit dem ICE386 stellt die INT-1-Anweisung dar. Mit ihr wird tatsächlich über die IDT (Interrupt Descriptor Table; äquivalent dazu im Real Mode: IVT = Interrupt Vector Table) in den Debug-Exception-Handler verzweigt. Es empfiehlt sich daher, irgendwo im Debugger oder einer Applikation eine Patcharea (= Spielwiese des Programmierers, in der mal kurz was ausprobiert werden kann) einzurichten, in welche dann diese `INT-1´-Anweisung geschrieben und bei Bedarf ausgeführt werden kann.

Da nun, wie bereits dargestellt, im Debug-Exception-Handler vom ICE386 aus Breakpoints gesetzt und auch erreicht werden können, sind die Grundvoraussetzungen für einen Test desselben erst einmal geschaffen. Beim eigentlichen Testlauf muß dann aber auf jeden Fall unmittelbar nach dem Lesen des Statusregisters DR6 ein Breakpoint gesetzt und der ermittelte Status für den vorgesehenen Testfall manipuliert werden. Ein Lesen der Debug-Register ist ungefährlich, ein Beschreiben kann mitunter tödlich enden, da die Debug-Register fest in der Hand des ICE386 liegen. Beim anschließenden Durchlauf sind dann gegebenenfalls noch weitere `Randbedingungen´ wie etwa die Breakpoint-Adresse, der Task-Identifikator und vieles andere mehr zu berücksichtigen, doch das ist im wesentlichen systemabhängig.

Erheblich ungünstiger sieht die Sache natürlich aus, wenn als Testumgebung nur der MON386 zur Verfügung steht. Dieser stellt zwar im fortgeschrittenen Teststadium die einzige Möglichkeit dar, den Debugger voll auszutesten, da der Interrupt 1 sich durch Überschreiben des Eintrags in der IDT 1 problemlos umlenken läßt, doch beißt sich damit der Hund in den Schwanz: ein vorher mit dem MON386 spezifizierter Breakpoint kann nach dem Umlenken des IDT-Eintrages eben nicht mehr von diesem behandelt werden; man landet im eigenen Debug-Exception-Handler.

Probleme mit dem Monitor

Um dennoch ein bißchen testen zu können, teilt man sich die Debug-Register. Der MON386 bekommt DR0 und DR1, der Debugger während der Testphase nur DR2 und DR3. Besondere Beachtung ist dem Kontroll-Register DR7 zu widmen; dies wird geteilt, eine gegenseitige Beeinflussung ist möglich.

Zur Realisierung dieses Mechanismus ist ein Prolog im Debug-Exception-Handler von Nöten, der das Statusregister DR6 interpretiert und dann den entsprechenden Handler aktiviert. Der MON386-Handler muß dabei unbedingt mit unverändertem Stack erreicht werden, da ansonsten seine interne Listenverwaltung durcheinanderkommt - schließlich rechnet dieser ja nicht damit, daß ihm so einfach die Kontrolle entzogen wird.

Ein Kapitel für sich ist der MON386-Befehl `SWBREAK´. Dieser funktioniert in etwa nach der in Teil I erwähnten, herkömmlichen Debugging-Methode ohne die Debug-Register. Anstelle eines `INT 3´-Opcodes findet sich hier aber der nicht dokumentierte Einbyte-Opcode 0F1h. Intel hüllt sich darüber in Schweigen, doch kann es sein, daß er in ähnlicher Art und Weise realisiert ist.

Auch eine INT-1-Assembler-Anweisung muß in den MON386 geleitet werden, da erst diese ihn wieder zum Leben erwecken kann, falls man einmal mit `GO´ ohne Angabe einer Breakpoint-Adresse gestartet hat. Erst dann wären nämlich wieder weitere Eingaben in der Kommandozeile möglich.

Für ein manchmal gleichwohl recht brauchbares `Primitiv-Debuggen´ stellt der MON386 zwei Funktionsaufrufe in einer Library zur Verfügung, um wenigstens einen Character ein- beziehungsweise auslesen zu können. Einen Trace-Buffer, mit dem man den zuletzt durchlaufenen Code analysieren könnte, gibt es generell bei keinem Monitor - hier müßte ein Logic Analyzer ran.

Es bleibt anzumerken, daß der MON386 zwar eine sehr eingeschränkte Testmöglichkeit bietet, andererseits aber, abgesehen vielleicht vom Brennen ins EPROM, die einzige Möglichkeit darstellt, den Debugger effektiv ablaufen zu lassen und seine volle Funktionsfähigkeit auszutesten.

In Assembler und C

Das abgedruckte Beispielprogramm stellt keinen `ausgereiften´ Debug-Exception-Handler dar, vielmehr zeigt es die gerade erwähnten Fragmente Prolog (in Assembler) und den eigentlichen Handler (in C). Tiefergreifende Details sind vor allem betriebssystemabhängig und daher für ein Beispiel nicht geeignet.

Die Voraussetzung für den dargestellten Prolog ist, daß beim Hochfahren des Systems die globale Variable `original_int1_entry´ belegt wird. Wird mit einem Monitor gearbeitet, so enthält sie den Pointer (Selector:Offset) auf den ursprünglichen Debug-Exception-Handler des Monitors, andernfalls muß sie einen Nullpointer (0:0) enthalten. Mittels des Assemblerbefehls `LAR´ (Load Access Rights Byte, nur im Protected Mode möglich) läßt sich einfach ermitteln, ob ein Monitor-Handler gegebenenfalls angesprungen werden kann. Im Real Mode muß explizit ein `CMP´ (Compare Two Operands) mit einem Nullpointer erfolgen, um diese Unterscheidung zu treffen.

Durch den Zugriff auf die Variable `original_int1_entry´ ergibt sich jedoch das Problem, daß auch das entsprechende Datensegment in DS geladen, der Monitor-Handler aber auf jeden Fall mit unverändertem Stack und Registern aufgerufen werden muß. Ein indirekter Sprung (`JMP DS:[original_int1_entry]´) kann daher nicht eingesetzt werden. Die Lösung liegt im Aufteilen des Sprungs in mehrere Anweisungen.

RET einmal anders

Wenn man weiß, daß ein Far Return die obersten zwei Einträge auf dem Stack als Return-

Adresse interpretiert, läßt sich ein `JMP´ leicht nachbilden: zuerst also gewünschten CS:EIP auf den Stack pushen, dann das Datensegment restaurieren, und mit dem `RET´ geht´s ab zum Debug-Exception-Handler des Monitors. Bleibt anzumerken, daß DS (und im Beispiel auch AX, um DS laden zu können) mittels `MOV´ auf dem Stack zwischengespeichert werden, da dies der einzige Datenbereich ist, der in jedem Kontext freien Speicherplatz bietet und kein neuer Selektor geladen werden muß.

Der eigentliche Debug-Exception-Handler wurde in C geschrieben, dadurch sind komplexere Mechanismen wesentlich einfacher zu realisieren.

Um einen Zugriff auf die Debug-Register des 80386 zu erhalten, sind zu Beginn einige Makros definiert, die als Inline-Code die entsprechenden `MOV´-Operationen nachbilden. Als Übergabeparameter von C zu den Assemblerbefehlen werden Registervariablen vereinbart. Da sowohl die Darstellung des Inline-Codes als auch die Zuweisung von Registern zu Registervariablen compilerspezifisch sind, müssen gegebenenfalls Anpassungen vorgenommen werden. Besonders ist darauf zu achten, in welchen Registern die Registervariablen abgelegt werden.

Im Programmablauf erfolgt zuerst die Ermittlung der Ursache durch Lesen von DR6. Das Makro `search_reason´ liefert den Index des niedrigsten gesetzten Bits und löscht dieses (wie beschrieben können ja mehrere Ursachen gleichzeitig auftreten). Realisiert wird dies durch die Assemblerbefehle `BSF´ (Bit Scan Forward, sucht nach dem ersten gesetzten Bit innerhalb eines Speicherbereiches oder Registers) und `BTR´ (Bit Test and Reset). Diese Assemblersequenz kommt dabei mit einem einzigen Register aus.

Anschließend erfolgt die Aufgliederung der Ursachen. Bei einem Task-Switch wird im Beispielprogramm eine Tabelle durchsucht (es wird angenommen, daß für jeden gesetzten Breakpoint ein Eintrag in eine Breakpoint-Table erfolgt, ähnlich wie sie hier in breakpoint_table_type spezifiziert ist) und die lineare Adresse in die Debug-Register DR0 bis DR3 eingetragen. Die nicht belegten Register werden aus in Teil I aufgeführtem Grunde mit 0xffffffff belegt. Das Makro `control_it´ übernimmt das Setzen der entsprechenden Bits in DR7. Aufgrund eines fatalen Fehlers im Optimierer von älteren Versionen des C386 darf man dieses Makro (und damit das gesamte File) gegebenenfalls nur mit der Option `ot(1)´ (gibt die Optimierungsstufe an) kompilieren.

In der Routine für die Breakpoint-Behandlung durch Erreichen einer entsprechenden Adresse (DR0 bis DR3) kann entweder die lineare Adresse aus dem entsprechenden Debug-Register oder aber die virtuelle Adresse (Returnadresse nach dem IRETD) vom Stack verarbeitet werden. Im Beispiel wird letztere vom Prolog in einer globalen Variable abgespeichert. Eine Konvertierung in eine lineare Adresse muß durch das Betriebssystem erfolgen. In einem Multitasking-System hat die Routine auch den Task-Identifikator zu ermitteln, welcher anschließend in die Auswertung mit einfließen kann. Bei Daten-Breakpoints muß in jedem Fall die Returnadresse vom Stack genommen werden, da im entsprechenden Debug-Register ja die Adresse des Datums eingetragen wurde, jedoch die Adresse des darauf zugreifenden Codes von Interesse sein dürfte.

Als Fazit der Programmierung der Debug-Register bleibt zu sagen: arm dran ist, wem sich für Protected-Mode-Programmierung nur eine der beiden vorgestellten Testumgebungen bietet. Intel hat die Debug-Register anscheinend vorwiegend aus Selbstzweck ins Leben gerufen. Dies macht ihre Nutzung für eigene Zwecke gleichwohl viel interessanter. Der Einfallsreichtum des Testers ist auf jeden Fall gefragt; wie sich die Sache im Real Mode darstellt - und welche Konflikte sich zum Beispiel mit Turbo-Debug auftun können - erfahren Sie im nächsten Teil. (as)

Kasten 1

Will man anderen Monitoren den INT 1 nicht vorenthalten, ist dieser Prolog nötig, der `stackneutral´ DR6 vorab auswertet.

```
; Beispiel für den Prolog eines Debug-Exception-Handlers;
;
; Autor: Christoph Ascherl - 1990
;
;
; Dies ist der Prolog für einen Debug-Exception-Handler. Er
; interpretiert das Debug-Statusregister DR6 und verzweigt
; daraufhin entweder in den eigentlichen INT 1-Handler des
; Debuggers oder, falls vorhanden, in den Handler des MON386.
;
; Die entsprechenden Handler müssen mit unveränderten Registern
; und Stack aufgerufen werden.
;
; Beim Hochfahren des Debuggers muß bereits der Originaleintrag
; aus der IDT (1) ermittelt und abgespeichert werden, bevor auf
; diesen Interrupt-Level ein assign (= Eintragen dieses Prologs in
; die IDT) erfolgt.
;
; Dieses Programm kann mit dem Intel-Assembler ASM386 assembliert
; werden und ist für 32-Bit Code im Protected-Mode geschrieben.
; (Aufruf: ASM386 prolog.asm)
;
; Zur Verdeutlichung sei hier noch der Stack aufgezeigt (SS:ESP
; wie er direkt nach Eintritt in dises Programm steht).
;
;
;             |     .      |
;             |------------|
; SS:ESP+8    |   EFLAGS   |  EFLAGS (durch Interrupt gepusht)
;             |------------|
; SS:ESP+4    |     CS     |  Return-Selektor (durch Interrupt
;             |------------|                    gepusht)
; SS:ESP   -> |    EIP     |  Return-Offset   (durch Interrupt
;             |------------|                    gepusht)
; SS:ESP-4    | MON386-CS  |  Selektor des MON386-Handlers
;             |------------|
; SS:ESP-8    | MON386-EIP |  Offset   des MON386-Handlers
;             |------------|
; SS:ESP-12   |     DS     |  Zwischenspeicher für das Register DS
;             |------------|
; SS:ESP-16   |    EAX     |  Zwischenspeicher für das Register
;             |------------|                                   EAX
;             |     .      |
;
;
;
;
    NAME prolog

    PUBLIC   int_1_prolog
    EXTRN    int_1_handler: NEAR

; Das folgende Datensegment dient nur der ASSUME-Anweisung, die
; Daten selbst können irgendwo anders im System stehen.

DATA SEGMENT RW PUBLIC
```

```
    EXTRN    return_address:        PWORD
    EXTRN    original_int1_entry:   PWORD
DATA ENDS

$EJECT
CODE32 SEGMENT ER PUBLIC USE32
    ASSUME DS: DATA

; Die Adresse der Prozedur int_1_prolog muß in der IDT eingetragen
; sein

int_1_prolog PROC FAR

; Zuerst DS und EAX merken - auf dem Stack ist gerade ein bißchen
; Platz frei

    MOV  [ESP - 12], EAX
    MOV  AX, DS
    MOV  [ESP - 16], AX

; Dann eigenes Datensegment laden

    MOV  AX, DATA
    MOV  DS, AX

; Die Returnadresse in das unterbrochene Programm kann für den
; Debug-Exception-Handler ganz interessant sein, besonders wenn
; es sich um einen Datenbreakpoint handelt - also auch abspeichern

    MOV  EAX, [ESP]
    MOV  DWORD PTR return_address, EAX  ; das ist der return - EIP
    MOV  AX, [ESP + 4]
    MOV  WORD PTR return_address + 4, AX ; das ist der return - CS

; Jetzt eine Überprüfung des Selektors aus dem Originaleintrag in
; der IDT (1). Falls gar nicht mit dem MON386 gearbeitet wird,
; würde ein Aufruf desselben wohl in die Pampa gehen.

    LAR  AX, WORD PTR original_int1_entry + 4 ;CS des far-pointers
    JNZ  debugger            ; Sprung, falls kein MON386 vorhanden

; Debug-Statusregister lesen und überprüfen: die reserved-bits
; ausmaskieren und dabei gleich erkennen, ob eine
; INT 1-Instruction vorlag

    MOV  EAX, DR6
    AND  EAX, 0E00FH
    JZ   mon386                        ; Sprung bei INT 1-Instr.

; An dieser Stelle überprüfen, ob ein Single-Step-Interrupt
; vorliegt. Weitere Checks könnten folgen - etwa wenn sich MON386
; und Debugger die Debug-Addressregister teilen.

    BT   EAX, 14
    JNC  debugger                ; Sprung, falls kein Single-Step

; Hierher gelangen wir, wenn es tatsächlich einen MON386-Handler
; gibt und dieser auch die Kontrolle übernehmen soll -
; restaurieren wir die Register und preparieren den Stack. Mit
; einem einfachen FAR JMP Befehl ist es nicht getan, da ohne
```

```
; gültiges Datensegment original_int1_entry nicht adressierbar
; ist.

mon386:

    PUSH DWORD PTR original_int1_entry + 4  ; Push CS (dword ptr!)
    PUSH DWORD PTR original_int1_entry                  ; Push EIP
    MOV  AX, [ESP - 4]                       ; DS restaurieren ...
    MOV  DS, AX
    MOV  EAX, [ESP - 8]                         ; ... und auch EAX
    RET                                       ; FAR JMP zum MON386

; Hier ist dann Schluß, vom MON386 geht´s direkt in die
; Applikation zurück.

; Vorbereitungen für einen Einstieg in den eigentlichen Debug-
; Exception-Handler treffen. Zuerst auch hier die Register
; restaurieren.

debugger:

    MOV  AX, [ESP - 12]                      ; DS restaurieren ...
    MOV  DS, AX
    MOV  EAX, [ESP - 16]                        ; ... und auch EAX

; Dann die eigentliche Befehlsfolge wie in jedem Interrupt-
; Handler: Register retten, irgendwas ausführen und schließlich
; die Register wieder in den Ursprungszustand versetzen. Wegen der
; Komplexität der Debug-Exceptions kann hier ruhigen Gewissens
; eine C-Routine aufgerufen werden (unter Realzeitgesichtspunkten
; ist diese Aussage natürlich zu verifizieren).

    PUSHAD
    CALL int_1_handler                                ; near call
    POPAD

; Zur Krönung setzen wir jetzt noch das Resume-Flag
; (Restart-Flag).

    BTS  DWORD PTR [ESP + 8], 16

; Das war´s - bis zum nächsten Mal (Breakpoint).

    IRETD

int_1_prolog ENDP
CODE32       ENDS
             END
```

Kasten 2

Mit dem Intel-C386-Compiler ist das Programmieren im Protected Mode kein Problem. Allerdings sind an einigen Stellen Betriebsystemfunktionen zu integrieren.

```
 1  *
 2 /*
 3  * Beispiel für einen Debug-Exception-Handler.
 4  *
 5  * Autor: Christoph Ascherl - 1990
```

```
 6  *
 7  *
 8  * Dieses Programm stellt lediglich die Behandlungsroutinen zur
 9  * Verwaltung der Debugregister dar, nicht jedoch ein bereits
10  * fertig ausgearbeitetes Programm.
11  *
12  * Zur besseren Übersichtlichkeit wird ein Array of Structs
13  * angenommen, welches eine Tabelle darstellt, in der für jeden
14  * spezifizierten Breakpoint seine charakteristischen Werte
15  * festgehalten werden. Die Tabelle im Beispiel heißt
16  * ´breakpoint_table´ und hat eine Größe von NUM_OF_BREAKPOINTS.
17  * Für ungültige Einträge sei der ´table_index´ mit INVALID_ENTRY
18  * belegt, ansonsten mit einer von 1 an aufsteigenden Nummer. Die
19  * Definitionen für die Einträge ´access´ und ´alignment´ sind
20  * entsprechend den von der Prozessorseite vorgegebenen
21  * Zahlenwerten.
22  *
23  * Der Zugriff auf die Debugregister des 80386 erfolgt über
24  * Registervariablen und Inlinecodierung, die über Makros
25  * realisiert ist.
26  *
27  * Ein Integer besteht hier übrigens aus 32 Bit.
28  *
29  * Compilierbar mit dem Intel C-Compiler C386:
30  * c386 int1.c cp ot(2) ds(data)
31  */
32
33 #define NUM_OF_BREAKPOINTS      10
34 #define INVALID_ENTRY       0xffff
35
36 /*
37  * Mögliche Werte für die Felder R/Wn in DR7 (access).
38  */
39
40 #define EXECUTE                  0
41 #define DATA_WRITE               1
42 #define DATA_ACCESS              3
43
44 /*
45  * Mögliche Werte für die Felder LENn in DR7 (alignment).
46  */
47
48 #define BYTE                     0
49 #define WORD                     1
50 #define DWORD                    3
51
52 /*
53  * Indizes für die durch DR6 angezeigten Ursachen eines INT 1.
54  */
55
56 #define DR0_BREAKPOINT           0
57 #define DR1_BREAKPOINT           1
58 #define DR2_BREAKPOINT           2
59 #define DR3_BREAKPOINT           3
60 #define GENERAL_DETECT          13
61 #define SINGLE_STEP             14
62 #define TASK_SWITCH             15
63 #define INT_1_INSTRUCTION     0xff
64
65 /*
66  * Deklaration von drei Registervariablen. Sie werden die Register
67  * EDI, ESI und EBX belegen.
68  */
69
```

```
 70 #define  special_parameters                    \
 71          register long  dr_addr,       /* edi */ \
 72                         dr_status,     /* esi */ \
 73                         dr_control     /* ebx */
 74
 75 /*
 76  * Makros zum Lesen und Beschreiben der Adreßregister DR0 bis
 77  * DR3. Die entsprechende Adresse ist im Register EDI enthalten.
 78  */
 79
 80 #define  set_DR0   __hex__(0x0F,0x23,0xC7)      /* mov DR0, edi */
 81 #define  get_DR0   __hex__(0x0F,0x21,0xC7)      /* mov edi, DR0 */
 82 #define  set_DR1   __hex__(0x0F,0x23,0xCF)      /* mov DR1, edi */
 83 #define  get_DR1   __hex__(0x0F,0x21,0xCF)      /* mov edi, DR1 */
 84 #define  set_DR2   __hex__(0x0F,0x23,0xD7)      /* mov DR2, edi */
 85 #define  get_DR2   __hex__(0x0F,0x21,0xD7)      /* mov edi, DR2 */
 86 #define  set_DR3   __hex__(0x0F,0x23,0xDF)      /* mov DR3, edi */
 87 #define  get_DR3   __hex__(0x0F,0x21,0xDF)      /* mov edi, DR3 */
 88
 89 /*
 90  * Makros zum Lesen und Beschreiben des Statusregisters DR6.
 91  * Den Status enthält das Register ESI.
 92  */
 93
 94 #define  set_DR6   __hex__(0x0F,0x23,0xF6)      /* mov DR6, esi */
 95 #define  get_DR6   __hex__(0x0F,0x21,0xF6)      /* mov esi, DR6 */
 96
 97 /*
 98  * Makros zum Lesen und Beschreiben des Controlregisters DR7.
 99  * Das Controlwort ist im Register EBX enthalten.
100  */
101
102 #define  set_DR7   __hex__(0x0F,0x23,0xFB)      /* mov DR7, ebx */
103 #define  get_DR7   __hex__(0x0F,0x21,0xFB)      /* mov ebx, DR7 */
104
105 /*
106  * Komfortablerer Zugriff auf die Debug-Register des 80386.
107  * Ermittlung und Zuweisung an eine der Registervariablen erfolgen
108  * in einem Zuge. Sollte als Parameter einer der
109  * ´special_parameters´ angegeben werden und dabei ein Statement
110  * wie a = a entstehen, wird dies durch den Compiler wegoptimiert
111  */
112
113 #define  set_bp_0(addr__)       dr_addr = addr__; set_DR0
114 #define  set_bp_1(addr__)       dr_addr = addr__; set_DR1
115 #define  set_bp_2(addr__)       dr_addr = addr__; set_DR2
116 #define  set_bp_3(addr__)       dr_addr = addr__; set_DR3
117 #define  get_bp_0(addr__)       get_DR0; addr__ = dr_addr
118 #define  get_bp_1(addr__)       get_DR1; addr__ = dr_addr
119 #define  get_bp_2(addr__)       get_DR2; addr__ = dr_addr
120 #define  get_bp_3(addr__)       get_DR3; addr__ = dr_addr
121 #define  get_status(status__)   get_DR6; status__ = dr_status
122 #define  set_status(status__)   dr_status = status__ ; set_DR6
123 #define  get_control(control__) get_DR7; control__ = dr_control
124 #define  set_control(control__) dr_control = control__ ; set_DR7
125
126
127 /*
128  * Das Makro ´search_reason´ ist eine Assemblerroutine, welche im
129  * Übergabeparameter den Index des niedrigsten gesetzten
130  * Statusbits übermittelt. Dieses Bit wird anschließend in der
131  * Registervariablen gelöscht.
132  */
133
```

```
134 #define  push_esi __hex__(0x56)               /* push esi      */
135 #define  bsf      __hex__(0x0F,0xBC,0xF6)     /* bsf  esi, esi */
136 #define  btr      __hex__(0x0F,0xB3,0x34,0x24) /* btr  [esp],esi*/
137 #define  pop_esi  __hex__(0x5E)               /* pop  esi      */
138
139 #define  search_reason(reason__)                                  \
140          push_esi;              /* push esi;                     \
141                                  = push status */                 \
142          bsf;                   /* bsf  esi, esi;                \
143                                  = bsf  index, status */          \
144          reason__ = dr_status;  /* mov  mem, esi;                \
145                                  =  mov  memory, index */         \
146          btr;                   /* btr  dword ptr [esp], esi;    \
147                                  = btr  indexbit im status */     \
148          pop_esi                /* pop  esi;                     \
149                                  = pop  geänderter status */
150
151 /*
152  * Makro zum Setzen der Bits im Controlwort (Registervariable).
153  * Übergabeparmeter sind die Nummer des Adreßregisters, access und
154  * alignment. Eine Überprüfung dieser Werte findet nicht statt.
155  * Bei Datenbreakpoints wird Pipelining ausgeschaltet.
156  */
157
158 #define  control_it(dr_nr__,access__,alignment__)                 \
159          dr_control |= (((long)1 << (dr_nr__ * 2)) |              \
160                        ((long)access__ << (dr_nr__ * 4 + 16)) |   \
161                        ((long)alignment__ << (dr_nr__ * 4 +18))); \
162          if (access__) dr_control |= 0x100
163
164
165 /*
166  * Beispiel für den Inhalt eines Eintrags in der breakpoint_table.
167  * Der entsprechende Speicherplatz wird hier nicht geordert.
168  */
169
170 typedef struct {
171     unsigned short table_index;
172     unsigned long  task_id;
173     unsigned long  linear_address;
174     char far *     virtual_address;
175     unsigned char  access;
176     unsigned char  alignment;
177 } breakpoint_table_type;
178
179 /*
180  * Der Pointer auf folgende Struktur dient als Übergabeparameter
181  * für eine Betriebssystemroutine (hier get_task_info() genannt),
182  * die u.a. die Konvertierung einer virtuellen in eine lineare
183  * Adresse vornimmt.
184  */
185
186 typedef struct {
187     unsigned long linear_address;
188     char far *    virtual_address;
189 } convert_address_type;
190
191 /*
192  * extern-Deklarationen
193  */
194
195 extern unsigned long       get_task_info (convert_address_type *);
196 extern breakpoint_table_type breakpoint_table[NUM_OF_BREAKPOINTS];
197
```

```
198 /*
199  * Globale Variable, welche die im Prolog ermittelte (virtuelle)
200  * Rücksprungadresse des Handlers enthält.
201  */
202
203 char far *  return_address = 0;
204
205 /*
206  ****************************************************************
207  *                                                              *
208  * Eigentliche Prozedur des Debug-Exception-Handlers            *
209  *                                                              *
210  ****************************************************************
211  *
212  * Ermittlung der Ursache für die Debug-Exception anhand von DR6
213  * und Verzweigung in die entsprechenden Unterprozeduren.
214  */
215
216 void int_1_handler(void)
217
218 {
219
220     /*
221      * Deklaration der Registervariablen.
222      * Die Variablen heißen dr_addr, dr_status und dr_control.
223      * Merke: Beim C386 können keine weiteren Registervariablen
224      * vereinbart werden.
225      */
226
227     special_parameters;
228     unsigned long      reason,
229                        lin_addr;
230
231     void               matched_address (unsigned long);
232     void               task_switch (void);
233
234
235     /*
236      * Status der Debug-Register ermitteln und reservierte Bits
237      * löschen.
238      */
239
240     get_status(dr_status);
241     dr_status &= 0xE00F;
242
243     /*
244      * Da mehrere Ursachen gleichzeitig eine Debug-Exception
245      * auslösen können, ist eine Schleife unbedingt erforderlich.
246      */
247
248     do {
249         /*
250          * Falls kein Bit gesetzt ist (bei gewöhnlicher INT 1-
251          * Anweisung), würde ein undefinierter Wert
252          * zurückgegeben. Diesen Fall verhindert die IF-Abfrage.
253          * Ansonsten enthält die Variable ´reason´ das
254          * entsprechende Indexbit.
255          */
256
257         if (dr_status) {
258             search_reason(reason);
259         }
260         else
261             reason = INT_1_INSTRUCTION;
```

```
262
263
264         switch (reason) {
265
266             /*
267              * Instruction Address Breakpoint oder Data Address
268              * Breakpoint. Ermittlung der linearen Adresse aus DR0
269              * bis DR3.
270              */
271
272             case DR0_BREAKPOINT:
273                 get_bp_0(lin_addr);
274                 matched_address (lin_addr);
275                 break;
276
277             case DR1_BREAKPOINT:
278                 get_bp_1(lin_addr);
279                 matched_address (lin_addr);
280                 break;
281
282             case DR2_BREAKPOINT:
283                 get_bp_2(lin_addr);
284                 matched_address (lin_addr);
285                 break;
286
287             case DR3_BREAKPOINT:
288                 get_bp_3(lin_addr);
289                 matched_address (lin_addr);
290                 break;
291
292
293             /*
294              * Task Switch Breakpoint.
295              */
296
297             case TASK_SWITCH:
298                 task_switch();
299                 break;
300
301
302             /*
303              * Single-Step Trap.
304              * General Detect Fault.
305              * INT 1 - Instruction.
306              */
307
308             case SINGLE_STEP:
309             case GENERAL_DETECT:
310             case INT_1_INSTRUCTION:
311             default:
312                 break;
313         }
314     }
315     while (dr_status);
316
317
318     /*
319      * Reset von DR6.
320      */
321
322     set_status(0);
323
324 }
325
```

```
326 /*
327  ****************************************************************
328  *                                                              *
329  * Applikation erreichte einen Breakpoint.                      *
330  *                                                              *
331  ****************************************************************
332  *
333  * Folgende Prozedur kann nur ein Grundgerüst aufzeigen, da die
334  * Behandlung von Breakpoints speziell auf Betriebssystem und
335  * Debugger zugeschnitten sein muß (besonders beim Multitasking
336  * im PVAM).
337  */
338
339 void matched_address (lin_addr)
340 unsigned long lin_addr;
341
342 {
343     unsigned short            i;
344     unsigned long             task_id;
345     char far *                virt_addr;
346     breakpoint_table_type *   table_ptr;
347     convert_address_type      address_info;
348
349     /*
350      * Aufruf einer Betriebssystemroutine, welche den Task-
351      * Identifikator ermittelt und eine Adreßkonvertierung von
352      * virtueller in lineare Adresse vornimmt. Bei
353      * Codebreakpoints würde diese schon im entsprechenden
354      * Adreßregister (DR0 bis DR3) stehen, nicht jedoch bei
355      * Datenbreakpoints.
356      */
357
358     address_info.virtual_address = return_address;
359     task_id = get_task_info (&address_info);
360
361     /*
362      * Durchsuchen der breakpoint_table, um dieser weitere
363      * Charakteristika zu diesem Breakpoint zu entnehmen (s.u.).
364      */
365
366     for (i = 0; i < NUM_OF_BREAKPOINTS; i++) {
367
368         table_ptr = breakpoint_table + i;
369
370         if ( (table_ptr->table_index != INVALID_ENTRY) &&
371              (table_ptr->task_id == task_id) &&
372              (table_ptr->linear_address == lin_addr) ) {
373
374             break;
375         }
376     }
377     if (i == NUM_OF_BREAKPOINTS) return;
378
379     /*
380      * Bei Datenbreakpoints ermittelte Adresse übermitteln.
381      */
382
383     if (table_ptr->access != EXECUTE) {
384         lin_addr = address_info.linear_address;
385     }
386     virt_addr = address_info.virtual_address;
387
388     /*
389      * An dieser Stelle muß ein Beriebssystemmechanismus in Kraft
```

```
390      * treten, welcher die Kommunikation zwischen dem Handler und
391      * der entsprechenden Debuggertask ermölicht. Folgende
392      * Informationen könnten für die Auswertung durch die
393      * Debuggertask von Interesse sein:
394      *
395      * lineare Adresse,
396      * virtuelle Adresse,
397      * ermittelter Index in der Tabelle,
398      * sonstiges (Daten aus der breakpoint_table).
399      *
400      * Außerdem kann an dieser Stelle dann noch ein Vertagen
401      * (suspendieren) der unterbrochenen Task erfolgen.
402      *
403      * Im Beispiel stehen an dieser Stelle folgende Parameter zur
404      * Verfügung:
405      * lin_addr, virt_addr, task_id und table_ptr.
406      */
407
408 }
409
410 /*
411  ****************************************************************
412  *                                                              *
413  * Unterbrechung durch einen Taskwechsel                        *
414  *                                                              *
415  ****************************************************************
416  *
417  * Hier können z.B. tasklokale Breakpoints gesetzt werden. Aber
418  * auch ein Aufzeichnen des Programmflußes wäre denkbar (d.h. in
419  * welcher Reihenfolge laufen die Tasks ab).
420  */
421
422 void task_switch ()
423
424 {
425
426     special_parameters;
427     unsigned long             task_id;
428     unsigned short            i,
429                               breakpoint_nr;
430     breakpoint_table_type *   table_ptr;
431     convert_address_type      address_info;
432
433
434     /*
435      * Aufruf einer Betriebssystemroutine, welche den Task-
436      * Identifikator ermittelt. Da hier die gleiche Funktion wie
437      * oben benutzt wird, zeigt der Null-Pointer an, daß keine
438      * Konvertierung der Adresse zu erfolgen hat.
439      */
440
441     address_info.virtual_address = 0;
442     task_id = get_task_info (&address_info);
443
444     /*
445      * Initialisierung der Registervariablen ´dr_control´.
446      */
447
448     dr_control = 0;
449
450     /*
451      * Ermittlung der dieser Task zugeordneten Breakpoints und
452      * Setzen derselben.
453      */
```

```
454
455     for (i = breakpoint_nr = 0; i < NUM_OF_BREAKPOINTS; i++) {
456
457         table_ptr = breakpoint_table + i;
458
459         if ( (table_ptr->table_index != INVALID_ENTRY) &&
460              (table_ptr->task_id == task_id) ) {
461
462             switch (breakpoint_nr) {
463                 case 0:  set_bp_0(table_ptr->linear_address);
464                          break;
465                 case 1:  set_bp_1(table_ptr->linear_address);
466                          break;
467                 case 2:  set_bp_2(table_ptr->linear_address);
468                          break;
469                 case 3:  set_bp_3(table_ptr->linear_address);
470                          break;
471                 default: return;
472             }
473
474             /*
475              * Setzen der entsprechenden Bits im Controlwort.
476              */
477
478             control_it(breakpoint_nr,
479                        table_ptr->access,
480                        table_ptr->alignment);
481             breakpoint_nr++;
482         }
483
484         /*
485          * Übertragen des Controlworts nach DR7.
486          */
487
488         set_control(dr_control);
489     }
490
491     /*
492      * Wegen des Fehlers im Statusregister DR6 des 80386 werden
493      * die momentan nicht benutzten Adreßregister mit 0xffffffff
494      * beschrieben. dr_addr ist einer der ´special_parameters´.
495      */
496
497     dr_addr = 0xffffffff;
498
499     switch (breakpoint_nr) {
500         case 0 :  set_DR0;
501         case 1 :  set_DR1;
502         case 2 :  set_DR2;
503         case 3 :  set_DR3;
504         default:  break;
505     }
506 }
```

Zu diesem Artikel existieren Programmbeispiele

0391_288.doc

0391_288.zip