ROZDZIAL SIEDEMNASTY: **PRZERWANIA, PRZERWANIA KONTROLOWANE I WYJATKI**

Koncepcja przerwan jest czyms, co rozwijalo sie w ciagu lat. Rodzina 80x86 powiekszyla tylko zamieszanie wokól przerwan poprzez wprowadzenie instrukcji int (przerwania programowe). Istotnie, rózni producenci uzywaja terminów takich jak wyjatki, bledy, zawieszenia, pulapki i przerwania do opisania zjawiska, który omawia ten rozdzial. Niestety nie ma wyraznej jednomyslnosci, co do dokladnego znaczenia tych terminów. Rózni autorzy róznie adoptuja te terminy na swój wlasny uzytek. Chociaz jest kuszace unikanie uzywania takich naduzywanych generalnie terminów, dla celów tego omówienia byloby milo miec zbiór dobrze zdefiniowanych terminów, jakie mozemy uzyc w tym rozdziale. Dlatego tez, wybierzemy trzy z powyzszych terminów, przerwania, pulapki i wyjatki i je zdefiniujemy. Rozdzial ten spróbuje uzyc najpowszechniejszego znaczenia dla tych terminów, ale nie bedzie niespodzianka, jesli znajdziemy inne teksty uzywajace ich w innych kontekstach.

W 80x86 sa trzy typy popularnie znane przerwania: pulapki, wyjatki i przerwania (przerwania sprzetowe). Rozdziale ten bedzie opisywal kazda z tych form i opisze ich wsparcie dla CPU 80x86 i kompatybilne maszyny PC.

Chociaz terminy przerwanie kontrolowane i wyjatki sa czesto uzywane zamiennie, bedziemy uzywali terminu przerwanie kontrolowane do oznaczania oczekiwania przekazania sterowania do specjalnego podprogramu obslugi. Pod wieloma wzgledami przerwanie kontrolowane jest niczym wiecej niz. wywolanym wyspecjalizowanym podprogramem. Wiele tekstów odnosi sie do przerwan kontrolowanych jako przerwan programowych. Instrukcja int 80x86 jest glównym narzedziem dla wykonania przerwania kontrolowanego. Zauwazmy, ze przerwania kontrolowane sa zazwyczaj bezwarunkowe; to znaczy, kiedy wykonujemy instrukcje int, sterowanie zawsze jest przekazywane do procedury powiazanej z przerwaniem kontrolowanym. Poniewaz przerwania kontrolowane wykonuja sie przez wyrazne instrukcje, latwo jest okreslic dokladnie, które instrukcje w programie beda wywolywaly podprogram obslugi przerwan kontrolowanych.

Wyjatek jest automatycznie generowana pulapka (wymuszenie zamiast prosby), która wystepuje w odpowiedzi na warunek wyjatku. Generalnie, nie ma okreslonej powiazanej z wyjatkiem, zamiast tego wyjatek wystepuje w odpowiedzi na niewlasciwe zachowanie wykonania zwyklego programu 80x86. Przyklady warunków, które moga zglaszac (powodowac) wyjatki obejmuja wykonywanie instrukcji dzielenia przez zero, wykonywanie niedozwolonych opcodów i bledy ochrony pamieci. Obojetnie, kiedy taki warunek wystapi, CPU natychmiast zawiesza wykonywanie biezacej instrukcji i przekazuje sterowanie do podprogramu obslugi wyjatków. Ten podprogram moze zdecydowac jak obsluzyc warunek wyjatku; moze próbowac naprawiac problem lub przerwac program i wydrukowac wlasciwy komunikat bledu. Chociaz generalnie nie wykonujemy okreslonych instrukcji powodujacych wyjatek, podobnie przerwan sprzetowych (pulapek), wykonywanie instrukcji jest czyms, co powoduje wyjatki. Na przyklad, dostaniemy blad dzielenia, kiedy wykonujemy instrukcje dzielenia gdzies w programie.

Przerwania sprzetowe, trzecia kategoria, do której bedziemy sie odnosic po prostu jako do przerwan, sa przerwaniami sterowanymi opartymi na zewnetrznych zdarzeniach sprzetowych (zewnetrzne dla CPU) Generalnie te przerwania nie maja nic do roboty z biezaco wykonywanymi instrukcjami; zamiast tego, niektóre zdarzenia, takie jak nacisniecie klawisza na klawiaturze lub limit czasu na tajmerze chipa, informuja CPU, ze urzadzenie potrzebuje jego uwagi, a potem zwraca sterowanie ponownie do programu.

Podprogram obslugi przerwan jest procedura napisana specjalnie do dzialania z pulapkami, wyjatkami i przerwaniami. Chociaz rózne zjawiska powoduja pulapki, wyjatki i przerwania, struktura podprogramu obslugi przerwan lub ISR jest w przyblizeniu taka sama dla kazdego z nich.

17.1 STRUKTURA PRZERWAN 80X86 I PODPROGRAM OBSLUGI PRZERWAN (ISR)

Pomimo róznych powodów wystapienia przerwan kontrolowanych, wyjatków i przerwan, dziela one wspólny format dla swoich podprogramów obslugi. Oczywiscie, te podprogramy obslugi przerwan beda wykonywaly rózne dzialania w zaleznosci od zródla wywolania., Ale jest calkiem mozliwe napisanie pojedynczego podprogramu obslugi przerwan, który przetwarza przerwania kontrolowane, wyjatki i przerwania sprzetowe. Jest to rzadko robione, ale struktura systemu przerwan 80x86 pozwala na to. Sekcja ta bedzie opisywala strukture przerwan 80x86 i to jak napisac podstawowy podprogram obslugi przerwan dla przerwan trybu rzeczywistego 80x86.

Chipy 80x86 pozwalaja na 256 wektorów przerwan. To znaczy, ze mamy do 256 róznych zródel przerwan, a 80x86 bezposrednio wywola podprogram obslugi dla tego przerwania bez przetwarzania programowego. Kontrastuje to z niewektorowymi przerwaniami, które przekazuja sterowanie bezposrednio do pojedynczego podprogramu obslugi przerwan, bez wzgledu na zródlo przerwania.

80x86 dostarcza 256 wejsc do tablicy wektorów przerwan poczynajac od adresu 0:0 w pamieci. Jest to 1KB tablica zawierajaca 256 4 bajtowych wejsc. Kazde wejscie w tej tablicy zawiera adres segmentowany, który wskazuje podprogram obslugi przerwan w pamieci. Generalnie, bedziemy sie odnosili do przerwan poprzez ich indeks w tej tablicy, tak wiec zerowy adres przerwania (wektor) jest w komórce pamieci 0: 0, wektorze przerwania jeden jest pod adresem 0: 4, wektor przerwania dwa jest pod adresem 0: 8 itd.

Kiedy wystapi przerwanie, bez wzgledu na zródlo, 80x86 robi, co nastepuje?

1) CPU odklada rejestr flag na stos
2) CPU odklada daleki adres powrotny (segment: offset) na stos, najpierw wartosc segmentu
3) CPU okresla powód przerwania (tj. numer przerwania) i pobiera cztero bajtowy wektor przerwania spod adresu 0: wektor*4
4) CPU przekazuje sterowanie do podprogramu okreslonego przez wejscie tablicy wektorów przerwan

Po ukończeniu tych kroków, sterownie ma podprogram obsługi przerwań. Kiedy podprogram obsługi przerwań chce zwrócić sterowanie musi wykonać instrukcję iret (interrupt return – powrót z przerwania). Zdejmuje ona daleki adres powrotny i flagi ze stosu. Zauważmy, że wykonanie dalekiego powrotu jest niewystarczające, ponieważ na stosie pozostaną flagi.

Jest jedna ważna różnica pomiędzy tym jak 80x86 przetwarza przerwania sprzętowe a innymi typami przerwań – na wejściu do podprogramu obsługi przerwań sprzętowych, 80x86 blokuje dalsze przerwania sprzętowe przez wyzerowanie flagi przerwania. Przerwania kontrolowane i wyjątki nie robią tego. Jeśli chcemy odrzucić dalsze przerwania sprzętowe wewnątrz procedury przerwania kontrolowanego lub wyjątku, musimy wyraźnie wyzerować flagę przerwania instrukcją cli. Odwrotnie, jeśli chcemy zezwolić na przerwania wewnątrz podprogramu obsługi przerwań sprzętowych, musimy wyraźnie włączyć ją ponownie instrukcją sti. Zauważmy, że na blokowanie flagi przerwania 80x86 wpływa tylko przerwanie sprzętowe.. Wyzerowanie flagi przerwania nie będzie zapobiegało wykonaniu przerwania kontrolowanego lub wyjątku.

ISR'y są napisane podobnie jak prawie każda inna procedura w języku asemblera z wyjątkiem tego, że wracają one instrukcją iret a nie ret. Chociaż odległość procedury ISR (near kontra far) nie ma zazwyczaj znaczenia, powinniśmy uczynić wszystkie procedury ISR far. Uczyni programowanie łatwiejszym, jeśli zdecydujemy się wywołać ISR bezpośrednio zamiast używać normalnych mechanizmów procedur przerwania.

ISR'y wyjątków i przerwań sprzętowych mają bardzo specjalne ograniczenia: muszą zachować stan CPU. W szczególności, te ISR'y muszą zachować wszystkie rejestry, które modyfikują. Rozważmy następujący ekstremalnie prosty ISR:

```
SimpleISR       proc    far
                mov     ax, 0
                iret
SimpleISR       endp
```

Ten ISR oczywiście nie zachowuje stanu maszynowego; wyraźnie narusza wartość w ax a potem zwraca z przerwania. Przypuśćmy, że wykonaliśmy poniższy fragment kodu, kiedy przerwanie sprzętowe przekazało sterowanie do powyższego ISR'a:

```
                mov     ax, 5
                add     ax, 2
;Przypuśćmy, że tu wystąpiło przerwanie

                puti
                -
                -
                -
```

Podprogram obsługi przerwań, ustawi rejestr ax na zero a nasz program wydrukuje zero zamiast wartości pięć. Gorzej, przerwania sprzętowe są generalnie asynchroniczne, w znaczeniu, że mogą wystąpić w każdym czasie i rzadko występują w tym samym miejscu programu. Dlatego też, powyższa sekwencja kodu drukowałaby siedem większość czasu; inaczej będzie drukował zero lub dwa (będzie drukował dwa, jeśli przerwanie wystąpi pomiędzy instrukcjami, mov ax, 5 a add ax, 2) Błędy w podprogramach obsługi przerwań sprzętowych są bardzo trudne do odnalezienia, ponieważ takie błędy często wpływają na wykonywanie nie powiązanego kodu.

Rozwiązaniem tego problemu oczywiście jest upewnienie się, że zachowaliśmy wszystkie rejestry, jakich używamy w podprogramie obsługi przerwań dla przerwań sprzętowych i wyjątków. Ponieważ pułapki są wywoływane jasno, zasady zachowywania stanów maszynowych w takich programach są identyczne jak dla procedur.

Napisanie ISR'a jest tylko pierwszym krokiem do implementacji programu obsługi przerwania. Musimy również zainicjalizować wejście tablicy wektorów przerwań adresem naszego ISR'a. Są dwa popularne sposoby wykonania tego – przechowanie adresu bezpośrednio w tablicy wektorów przerwań lub wywołanie DOS'a i pozwolenie, aby DOS wykonał to za nas.

Przechowanie samego adresu jest łatwym zadaniem, Wszystko, co musimy zrobić to załadować rejestr segmentowy zerem, (ponieważ tablica wektorów przerwań jest w segmencie zero) i przechować cztery bajty adresu pod właściwym offsetem wewnątrz segmentu. Następująca sekwencja kodu inicjalizuje wejście do przerwania 255 adresem podprogramu SimpleISR przedstawionego wcześniej:

```
mov     ax, 0
mov     es, ax
pushf
cli
mov     word  ptr  es:[0ffh*4], offset  SimpleISR
mov     word   ptr  es:[0ffh*4 +2], seg SimpleISR
popf
```

Odnotuj jak ten kod wyłącza przerwania podczas zmiany tablicy wektorów przerwań. Jest to ważne, jeśli poprawiamy wektor przerwań sprzętowych, ponieważ nie robi tego dla przerwań występujących pomiędzy ostatnimi dwoma powyższymi instrukcjami mov; w tym punkcie, wektor przerwań jest w wewnętrznie sprzecznym stanie i wywołując przerwanie w tym punkcie przekażemy sterowanie do offsetu SimpleISR i segmentu poprzedniego programu obsługi przerwania 0FFh. To oczywiście będzie katastrofa Instrukcje, które wyłączają przerwania podczas poprawiania wektora są zbyteczne, jeśli poprawiamy adres programu obsługi pułapki lub wyjątku.

Być może lepszym sposobem inicjalizacji wektora przerwań jest użycie wywołania DOS'owskiego Zbioru Wektorów Przerwań. Wywołanie DOS (zobacz „MS-DOS, PC-BIOS i I/O Plików) z ah równym 25h dostarcza tej funkcji. To wywołanie oczekuje numeru przerwania w rejestrze al. I adresu podprogramu obsługi przerwań w ds:dx. Wywołanie MS-DOS, które wykonuje tą samą rzecz jak powyższa to

```
mov     ax, 25ffh               ;AH=25h, AL = 0FFh
mov     dx, seg  SimpleISR      ;£aduje DS:DX adresem ISR
mov     ds., dx
lea     dx, SimpleISR
int     21h                     ;Wywołanie DOS
mov     ax, dseg                ;Przywrócenie DS, więc ponownie wskazuje DSEG
mov     ds., ax
```

Chociaż ta sekwencja kodu jest trochę bardziej złożona niż włożenie danych bezpośrednio do tablicy wektora przerwań, jest bezpieczniejsza. Wiele programów monitoruje zmiany robione na tablicy wektorów przerwań przez DOS., Jeśli wywołujemy DOS, który zmienia wejście tablicy wektora przerwań, programy te uświadomią sobie swoje zmiany. Jeśli pominiemy DOS, programy te mogą nie odkryć, że poprawiono ich własne przerwania i mogą nie działać.

Ogólnie, jest to bardzo zły pomysł poprawianie tablicy wektora przerwań i nie przywracanie oryginalnego wejścia po zakończeniu naszego programu. Cóż programy zawsze zachowują poprzednią wartość wejścia tablicy wektora przerwań i przywracają tą wartość przed zakończeniem. Poniższa sekwencja kodu demonstruje jak to zrobić. Po pierwsze przez poprawianie tablicy bezpośrednio:

```
        mov     ax, 0
        mov     es, ax

;Zachowanie bieżącego wejścia w zmiennej dword IntVectSave:

        mov     ax, es:[IntNumber*4]
        mov     word  ptr IntVectSave, ax
        mov     ax, es:[IntVect*4 +2]
        mov     word  ptr IntVectSave+2, ax

;Poprawienie tablicy wektora przerwań adresem naszego ISR'a
        pushf                                   ;wymagane, jeśli jest to przerwanie hw
        cli                                     ; „     „  „    „   „    „ „      „

        mov     word  ptr es:[IntNumber*4], offset OurISR
        mov     word  ptr  es:[IntNumber*4+2], seg OurISR

        popf                                    ;wymagane jeśli jest to przerwania hw

        -
        -
        -
;Przywrócenie wejścia wektora przerwań przed opuszczeniem:

        mov     ax, 0
        mov     es, ax

        pushf                                   ;wymagane, jeśli jest to przerwanie hw
        cli                                     ;"   „  „  „  „     „

        mov     ax, word  ptr  IntVectSave
        mov     es:[IntNumber*4[, ax
        mov     ax, word  ptr  IntVectSave+2
        mov     es:[IntNumber*4+2], ax

        popf                                            ;wymagane, jeśli jest to przerwania hw
        -
        -
        -
```

Jeśli wolelibyśmy wywołanie DOS do zachowania i przywrócenia wejścia tablicy wektora przerwań, możemy uzyskać adres istniejącego wejścia tablicy przerwań używając wywołania DOS Pobranie Wektora Przerwań. Wywołanie to z ah = 35h, oczekuje numeru przerwania w al.; zwraca istniejący wektor dla tego przerwania w rejestrach es:bx. Próbka kodu, który zachowuje wektor przerwań używając DOS to

```
;Zachowanie bieżącego wejścia w zmiennej dword IntVectSave:
```

```
                mov     ax, 3500h + IntNumber                   ;AH=35h, AL = Int #
                int     21h
                mov     word  ptr IntVectSave,  bx
                mov     word  ptr  IntVectSave+2, es

;Poprawa tablicy wektora przerwań adresem naszego ISR'a

                mov     dx, seg OurISR
                mov     ds, dx
                lea     dx, OurISR
                mov     ax, 2500h + IntNumber           ;AH=25, AL=Int #
                int     21h

                -
                -
                -
;Przywrócenie wejścia wektora przerwań przed opuszczeniem:

                lds     bx, IntVectSave
                mov     ax, 2500h+IntNumber             ;AH=25, AL=Int #
                int     21h
                -
                -
                -
```

17.2 PRZERWANIA KONTROLOWANE

Przerwanie kontrolowane jest przerwaniem wywoływanym programowo. Wykonując przerwanie kontrolowane używamy instrukcji int 80x86 (przerwanie programowe). Są tylko dwie podstawowe różnice pomiędzy przerwaniem kontrolowanym a dowolnym wywołaniem procedury far: instrukcja, jakiej używamy do wywołania podprogramu (int kontra call) i fakt, że przerwanie kontrolowane odkłada flagi na stos, więc musimy użyć instrukcji iret do powrotu z niej. W przeciwnym razie, rzeczywiście nie ma różnicy pomiędzy kodem programu obsługi przerwania kontrolowanego a ciałem typowej procedury far.

Głównym celem przerwania kontrolowanego jest dostarczenie stałego podprogramu, który różne programy mogą wywoływać bez znajomości aktualnego adresu i czasu wykonania. MS-DOS jest doskonałym przykładem. Instrukcja int 21h jest przykładem wywołania przerwania kontrolowanego Nasze programy nie muszą znać aktualnego adresu pamięci punktu wejścia DOS'a do wywołania DOS. Zamiast tego, DOS poprawia wektor przerwanie 21h kiedy ładuje go do pamięci.Kiedy wykonujemy int 21h, 80x86 automatycznie przekazuje sterowanie do punktu wejścia DOS gdziekolwiek w pamięci się to wydarzy.

Jest duża lista podprogramów wspierających, które używają mechanizmu przerwań kontrolowanych do połączenia aplikacji z nią samą. DOS, BIOS, sterownik myszy i Netware oto kilka przykładów. Ogólnie, używamy przerwań kontrolowanych do wywołania funkcji rezydentnych. Programy rezydentne ładują się same do pamięci i pozostają w pamięci dopóki się nie zakończą Poprze poprawę wektora przerwań wskazujemy podprogram wewnątrz kodu rezydentnego, inne programy, które działają po zakończeniu programu rezydentnego mogą wywołać rezydentne podprogramy poprzez wykonanie właściwej instrukcji int.

Większość programów rezydentnych nie używa oddzielnych wejść do wektorów przerwań dla każdej funkcji jaką dostarczają. Zamiast tego, zazwyczaj poprawiają pojedynczy wektor przerwań i przekazują sterowanie do właściwego podprogramu używając numeru funkcji, który kod wywołujący przekazuje w rejestrze Poprzez konwencję większość programów rezydentnych oczekuje numeru funkcji w rejestrze ah typowy podprogram obsługi przerwań kontrolowanych. Typowy podprogram obsługi przerwań kontrolowanych będzie wykonywał instrukcję wyboru na wartości z rejestru ah i przekaże sterowanie do właściwego podprogramu obsługi funkcji.

Ponieważ program obsługi przerwań kontrolowanych są praktycznie identyczne z procedurami far pod względem zastosowania, nie będziemy tu omawiać przerwań kontrolowanych bardziej szczegółowo. Jednakże, tekst tego rozdziału będzie zgłębiał ten temat bardziej, kiedy omawiać będzie programy rezydentne.

17.3 WYJĄTKI

Wyjątki występują (lub są wywoływane0 kiedy wystąpi anormalny warunek podczas wykonywania. Jest mniej niż osiem możliwych wyjątków na maszynie pracującej w trybie rzeczywistym. Wykonywanie w trybie chronionym dostarcza wielu innych, ale nie będziemy ich rozpatrywać tutaj, będziemy tylko rozważać te wyjątki, które działają w trybie rzeczywistym.

Chociaż procedury obsługi wyjątków są zdefiniowane dla użytkownika, sprzęt 80x86 definiuje wyjątki, które mogą wystąpić 80x86 również przypisuje stałą liczbę przerwań do każdego wyjątku. Poniższe sekcje opisuje każdy z tych wyjątków szczegółowo.

Generalnie podprogram obsługi wyjątków powinien zachować wszystkie rejestry. Jednakże, jest kilka specjalnych przypadków, gdzie możemy chcieć wyciągnąć wartość rejestru przed zwróceniem. Na przykład, jeśli wychodzimy poza granice zakresu, możemy chcieć zmodyfikować wartość w rejestrze określonym przez instrukcję bound przed zwróceniem. Niemniej jednak, nie powinniśmy dowolnie modyfikować rejestrów w podprogramie obsługi wyjątków chyba ,że zamierzamy natychmiast przerwać wykonywanie naszego programu.

17.3.1 WYJĄTEK BŁĘDU DZIELENIA (INT 0)

Wyjątek ten występuje wtedy kiedy próbujemy dzielić wartość przez zero lub iloraz nie mieści się w rejestrze przeznaczenia kiedy używamy instrukcji div lub idiv. Zauważmy, że instrukcje FPU fdiv i fdivr , nie wywołują tego wyjątku.

MS-DOS dostarcza ogólnego programu obsługi wyjątku dzielenia, który drukuje informację taką jak „divide error" i zwraca sterowanie do MS-DOS. Jeśli chcemy obsłużyć błąd dzielenia sami, musimy napisać swój własny program obsługi wyjątku i poprawić adres tego podprogramu pod lokacją 0:0.

W procesorach 8086, 8088, 80186 i 80188 adres powrotny na stosie wskazywał następną instrukcję po instrukcji dzielenia. Na 80286 9 późniejszych procesorach, adres powrotny wskazuje początek instrukcji dzielenia (wliczając w to bajt przedrostka, który się pojawia) . Kiedy wystąpi wyjątek dzielenia, rejestry 80x86 nie są modyfikowane; to znaczy zawierają wartości jakie przechowywały, kiedy 80x86 pierwszy raz wykonywał instrukcje div lub idiv.

Kiedy wystąpi wyjątek dzielenia, są trzy sensowne rzeczy jakich możemy próbować: przerwać program (najłatwiejsze wyjście) , skok do sekcji, kodu , który próbuje kontynuować wykonywanie programu zważywszy na błąd , lub próbujemy dojść dlaczego wystąpił błąd, poprawić go i ponownie wykonać instrukcję dzielenia. Kilu ludzi wybierze tą ostatnią alternatywę ponieważ jest taka trudna.

17.3.2 WYJĄTEK POJEDYNCZEGO KROKU (ŚLEDZENIA) (INT1)

Wyjątek pojedynczego kroku wystąpi po każdej instrukcji jeśli bit trace w rejestrze flag jest równy jeden. Debbugery i inne programy często będą ustawiały tą flagę ponieważ mogą one śledzić wykonywanie się programu.

Kiedy wystąpi ten wyjątek, adres powrotny na stosie jest adresem następnej instrukcji do wykonania. Program obsługi wyjątku śledzenia może zdekodować ten opcod i zdecydować jak postąpić dalej. Większość debbugerów używa wyjątku śledzenia do sprawdzania punktów kontrolnych i innych zdarzeń, które zmieniają się dynamicznie podczas wykonywania programu. Debuggery, które używają wyjątku śledzenia dla pojedynczych kroków często disasemblują kolejną instrukcję używając adresu powrotnego na stosie jako wskaźnika do tego bajtu opcodu instrukcji.

Generalnie, program obsługi wyjątku pojedynczego kroku powinien zachować wszystkie rejestry 80x86 i inne informacje o stanie. Jednak, jak zobaczymy interesujące zastosowanie wyjątku śledzenia później w tym tekście, gdzie będziemy celowo modyfikować wartości rejestrów czyniąc zachowanie jednej instrukcji zachowaniem innej (zobacz „Klawiatura PC")

Przerwanie jeden jest również dzielone przez możliwości uruchomienia wyjątków na 80386 i późniejszych procesorów. Procesory te dostarczają wsparcia zintegrowanego z układem poprzez rejestry uruchomieniowe. Jeśli wystąpi jakiś warunek, który dopasuje wartość w jednym z rejestrów uruchomieniowych, 80386 i późniejsze procesory wygenerują wyjątek uruchomieniowy, który używa wektora przerwania jeden.

17.3.3 WYJĄTEK PUNKTU ZATRZYMANIA (INT 3)

Wyjątek punktu zatrzymania jest w rzeczywistości przerwaniem kontrolowanym, nie wyjątkiem. Występuje kiedy CPU wykonuje instrukcję int 3. Jednakże, będziemy rozpatrywać to jako wyjątek ponieważ programiści rzadko wkładają instrukcje int 3 bezpośrednio do swoich programów. Zamiast tego debugger taki jak CodeView często dają sobie radę z rozmieszczeniem i usunięciem instrukcji int 3.

Kiedy 80x86 wywołuje podprogram obsługi wyjątku punktu zatrzymania, adres powrotny na stosie jest adresem następnej instrukcji po opcodzie punktu zatrzymania. Odnotujmy jednak, że są dwie instrukcje int, które przekazują sterowanie do tego wektora. Ogólnie, jest jednobajtowa instrukcja int 3, której opcod to 0cch; w przeciwnym razie jest dwubajtowy odpowiednik; 0cdh, 03h.

17.3.4 WYJĄTEK PRZEPEŁNIENIA (INT 4 / INTO)

Wyjątek przepełnienia, podobnie jak int 3, jest technicznie przerwaniem kontrolowanym. CPU wywołuje ten wyjątek tylko, kiedy wykonuje instrukcję into a flaga przepełnienia jest ustawiona. Jeśli flaga ta jest wyzerowana, instrukcja into jest faktycznie nop, jeśli flaga przepełnienia jest ustawiona, into zachowuje się jak instrukcja int 4, Programista może wprowadzić instrukcję into po obliczeniu całkowitym dla sprawdzenia przepełnienia arytmetycznego. Użycie into jest odpowiednikiem następującej sekwencji kodu:

```
                <Jakiś kod arytmetyki całkowitej>
                        jno     GoodCode
                        int     4
GoodCode:
```

Jedna dużą zaletą instrukcji into jest to ,że nie opróżnia potoku lub kolejki wstępnego pobrania jeśli flaga przeniesienia nie jest ustawiona. Dlatego też użycie instrukcji into jest dobrą techniką jeśli dostarczamy podprogramu obsługi pojedynczego przepełnienia (to znaczy nie mamy jakiegoś określonego kodu dla każdej sekwencji gdzie może wystąpić przepełnienie)

Adres powrotny na stosie jest adresem kolejnej instrukcji po into. Generalnie, program obsługi przepełnienia nie zwraca tego adresu. Zamiast tego zazwyczaj przerywa program lub zdejmuje adres i flagi ze stosu i próbuje obliczeń w inny sposób.

17.3.5 WYJĄTEK GRANICZNY (INT 5 / BOUND)

Podobnie jak into, instrukcja bound (zobacz „Instrukcje INT, INTO, BOUND i IRET") powoduje wyjątek warunkowy. Jeśli określony rejestr jest poza określoną granicą, instrukcja bound jest odpowiednikiem instrukcji int 5; jeśli rejestr jest wewnątrz określonej granicy , instrukcja bound jest faktycznie nop.

Adres powrotny , który odkłada bound jest adresem samej instrukcji bound, a nie instrukcji następującej po bound. Jeśli wracamy z wyjątku bez modyfikacji wartości w rejestrze (lub modyfikacji granic) wygenerujemy pętlę nieskończoną ponieważ kod ponownie będzie wykonywał instrukcję bound i powtarzał ten proces ciągle i ciągle.

Jedną sprytną sztuczką z instrukcją bound jest generowanie globalnego maksimum i minimum dla tablicy liczb całkowitych ze znakiem. Poniższy kod demonstruje jak możemy to zrobić:

```
;Ten program demonstruje jak obliczyć minimalną i maksymalną wartość dla tablicy liczb całkowitych ze znakiem
; używając instrukcji bound.

                        .xlist
                        .286
                        include         stdlib.a
                        includelib      stdlib.lib
                        .list

dseg                    segment para public 'data'

;Poniższe dwie wartości zawierają granice dla instrukcji BOUND

LowerBound              word    ?
```

```
UpperBound              word    ?

; Tu  zachowamy adres INT 5

OldInt5                 dword   ?

;Tu mamy tablicę dla której chcemy obliczyć minimum i maksimum:

Array                   word    1, 2, -5, 345, -26, 23 ,200 ,35, -100 ,20, 45
                        word    62, -30, -1, 21, 85, 400, -265, 3, 74, 24, -2
                        word    1024, -7, 1000, 100, -1000, 29, 78, -87, 60

ArraySize               =       ($ - Array) / 2

dseg                    ends

cseg                    segment  para  public  'code'
                        assume  cs:cseg, ds:dseg

; Nasz ISR przerwania 5. Oblicza wartość w AX największej i najmniejszej granicy i przechowuje AX w jednej z
; nich (wiemy że AX jest  poza zakresem z racji tego faktu ,że jesteśmy w tym ISR).
;
; Odnotujmy: w tym szczególnym przypadku wiemy ,że DS. wskazuje dseg, więc nie będziemy się martwić
;przeładowaniem tego ISR'a.
;
;Uwaga: kod ten nie obsługuje konfliktów pomiędzy bound / int 5 a klawiszem print screen. Wciskając prtsc
podczas
; wykonywania tego kodu możemy wytworzyć niepoprawny wynik (zobacz tekst)

BoundISR                proc    near
                        cmp     ax, LowerBound
                        jl      NewLower

; Musi być naruszona górna granica
                        mov     UpperBound, ax
                        iret

NewLower:               mov     LowerBound, ax
                        Iret

BoundISR                endp

Main                    proc
                        mov     ax, dseg
                        mov     ds, ax
                        meminit

; Zaczynamy od poprawienia adresu naszego ISR'a  w wektorze 5

                        mov     ax, 0
                        mov     es, ax
                        mov     ax, es:[5*4]
                        mov     word  ptr  OldInt5, ax
                        mov     ax, es:[5*4+2]
                        mov     word  ptr OldInt5 + 2, ax
```

```
                mov     word  ptr  es:[5*4],  offset BoundISR
                mov     es:[5*4 +2], cs

;Okay, przetwarzamy elementy tablicy. Zaczynamy inicjalizacją górnej i dolnej wartości granicy  pierwszym
; elementem tablicy

                mov     ax, Array
                mov     LowerBound, ax
                mov     UpperBound, ax

;Teraz przetwarzamy każdy element tablicy

                mov     bx, 2                   .;zaczynamy od drugiego elementu
                mov     cx, ArraySize
GetMinMax:      mov     ax, Array[bx]
                bound   ax, LowerBound
                add     bx, 2                   ;przejście do kolejnego elementu
                loop    GetMinMax

                printf
                byte    „Minimalna wartość to %d\n"
                byte    „Maksymalna wartość to %d\n", 0
                dword   LowerBound, Upper Bound

;Okay, przywracamy wektor przerwań:

                mov     ax, 0
                mov     es, ax
                mov     ax, word  ptr  OldInt5
                mov     es:[5*4], ax
                mov     ax, word  ptr OldInt+2
                mov     es:[5*4+2], ax

Quit:           ExitPgm                         ;makro DOS do wyjścia z programu
Main            endp

cseg            ends

sseg            segment  para  stack 'stack'
stk             db      1024  dup {"stack")
sseg            ends

zzzzzzseg       segment  para  public 'zzzzzz'
LastBytes       db      16  dup (?)
zzzzzzseg       ends
                end     Main
```

Jeśli tablica jest duża a wartości pojawiającej się w niej są względnie losowe, kod ten demonstruje szybki sposób określania wartości minimalnej i maksymalnej w tablicy. Alternatywne, porównanie każdego elementu z górną i dolną granicą i przechowanie wartości jeśli jest poza zakresem , jest generalnie wolniejszym podejściem. Prawda, jeśli instrukcja bound powoduje przerwanie kontrolowane, jest to dużo wolniejsze niż metoda porównania i przechowania. Jednakże przy dużej tablicy z wartościami losowymi naruszenie granicy będzie występowało rzadko.

Większość czasu instrukcja bound będzie wykonywała się w 7 – 13 cykli i nie będzie opróżniała potoku i kolejki wstępnego pobrania.
Uwaga: IBM , w swojej nieskończonej mądrości, zdecydował użyć int 5 do operacji print screen. Domyślnie obsługując int 5 będziemy zrzucać zawartość ekranu do drukarki. Z tego wynikają dwie implikacje dla tego kto będzie stosował instrukcję bound w swoich programach. Po pierwsze, nie zainstalujesz sobie swojego własnego programu obsługi int 5 i wykonasz instrukcję bound, która wygeneruje wyjątek graniczny, powodując wydruk zawartości ekranu. Po drugie, jeśli naciśniesz klawisz PrtSc z zainstalowanym programem obsługi int 5, BIOS wywoła twój program. Ten pierwszy przypadek jest błędem programistycznym, ale ten drugi przypadek znaczy, że musisz uczynić program obsługi wyjątku granicznego trochę sprytniejszym. Powinien poszukać bajtu wskazującego na adres powrotny. Jeśli jest w opcodzie instrukcji int 5 (0cdh), wtedy musi wywołać oryginalny program obsługi int 5, lub po prostu wrócić z przerwania. Jeśli nie ma opcodu int 5, wtedy ten wyjątek został wywołany prawdopodobnie przez instrukcję bound. Zauważmy ,że kiedy wykonujemy instrukcje bound adres powrotny może nie być wskazywany bezpośrednio w opcodzie bound (0c2h). Może wskazywać bajt przedrostka instrukcji bound (np. segment, tryb adresowania lub rozmiar przesłonięcia). Dlatego też lepiej jest sprawdzić opcod int 5.

17.3.6 WYJĄTEK NIEPRAWIDŁOWEGO OPCODU (INT 6)

80286 i późniejsze procesory wywołują ten wyjątek jeśli próbujemy wykonać opcod, który nie odpowiada poprawnej instrukcji 80x86. Procesory te również wywołują ten wyjątek jeśli próbujemy wykonać bound, lds, les, lidt lub inne instrukcje . które wymagają operandu pamięci ale wyszczególniamy operand rejestru w polu mod/rm bajtu mod/reg/rm.

Adres powrotny na stosie wskazuje niepoprawny opcod. Przez analizę tego opcodu możemy rozszerzyć zbiór instrukcji 80x86. na przykład możemy uruchomić kod 80486 na procesorze 80386 przez dostarczenie podprogramu, który imituje dodatkowe instrukcje 80486 (takie jak bswap, cmpxchg, itp.).

17.3.7 NIEDOSTĘPNY KOPROCESOR (INT 7)

80286 i późniejsze procesory wywołują ten wyjątek jeśli próbujemy wykonać instrukcje FPU (lub innego koprocesora) bez zainstalowanego koprocesora. Możemy użyć tego wyjątku do symulowania koprocesora w oprogramowaniu.

Na wejściu do programu obsługi wyjątku, adres powrotny wskazuje opcod koprocesora który generuje wyjątek.

17.4 PRZERWANIA SPRZĘTOWE

Przerwania sprzętowe są formą bardziej inżynierską (jako przeciwieństwo programistów PC) powiązaną z terminem przerwanie. Zaadoptujemy taką samą strategię i odtąd będziemy używali niezmodyfikowanego terminu „przerwanie" w znaczeniu przerwania sprzętowego.

Na PC, przerwania pochodzą z wielu różnych źródeł. Podstawowymi źródłami przerwań jednakże są chip zegarowy PC, klawiatura , port szeregowy, port równoległy, stacje dyskowe, zegar czasu rzeczywistego CMOS, mysz, karta dźwiękowa i inne urządzenia peryferyjne. Urządzenia te są podłączone do programowalnego sterownika przerwań (PIC) Intel 8259A, który ustawia priorytet przerwań i łączy z CPU 80x86. Chip 8259A dodaje znaczną złożoność do programu , który przetwarza przerwania, więc ma sporo sensu omówienie najpierw PIC, przed próbą opisania jak podprogramy obsługi przerwań działa z nim. Później w tej sekcji opiszemy krótko każde urządzenie i warunki pod jakimi przerywają pracę CPU. Tekst ten w pełni opisze wiele z tych urządzeń w późniejszych rozdziałach, wiec ten rozdział nie będzie wnikał w szczegóły, z wyjątkiem tego kiedy omówimy przerwania zegarowego.

17.4.1 PROGRAMOWALNY STEROWNIK PRZERWAŃ (PIC) 8259A

Chip programowalnego sterownika przerwań 8259A (odtąd 8259 lub PIC) akceptuje przerwania z ośmiu różnych urządzeń. Jeśli jedno z tych urządzeń żąda obsługi, 8259 przełączy linię wyjściowa przerwania (przyłączoną do CPU) i przekaże programowalny wektor przerwań do CPU. Możemy kaskadować urządzenia wspierając do 64 urządzeń przez połączenie dziewięciu 8259 razem: osiem urządzeń z ośmioma wejściami każde,

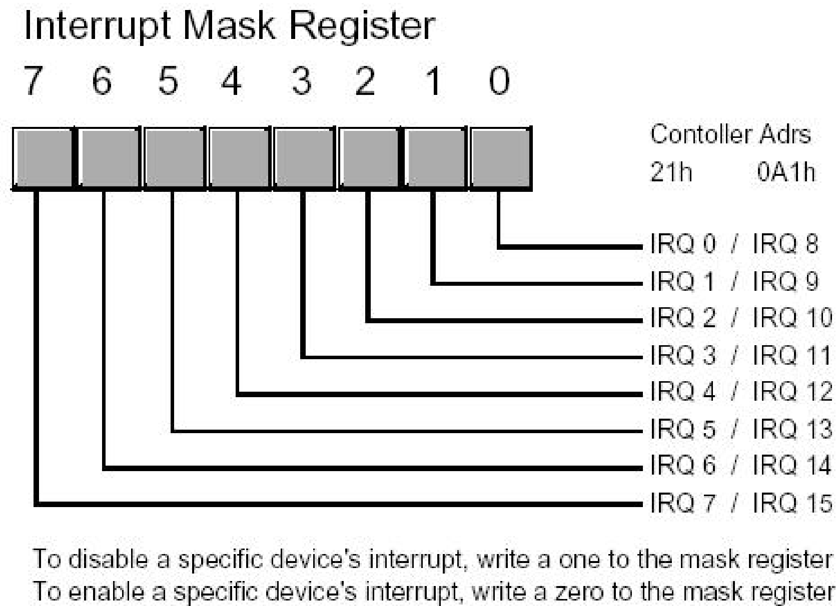
których wyjścia stają się ośmioma wejściami dziewiątego urządzenia. Typowy PC używa dwóch z tych urządzeń dostarczając 15 wejść przerwań (siedem na PIC master z ośmioma wejściami pochodzącymi z PIC slave przetwarzającymi osiem wejść) Następująca sekcja po tej opisze urządzenia połączone do każdego z tych wejść, a teraz skoncentrujemy się na tym co 8259 robi z tymi wejściami. Pomimo to ze względu na to omówienie, poniższa tablica listuje źródła przerwań na PC:

Wejście 8259	INT 80x86	Urządzenie
IRQ 0	8	Chip zegara
IRQ 1	9	Klawiatura
IRQ 2	0Ah	Kaskada dla sterownika 2 (IRQ 8 – 15)
IRQ 3	0Bh	Port szeregowy 2
IRQ 4	0Ch	Port szeregowy 1
IRQ 5	0DH	Port równoległy 2 w AT, zarezerwowany w systemie PS/2
IRQ 6	0Eh	Stacja dyskietek
IRQ 7	0Fh	Port równoległy 1
IRQ 8 / 0	70h	Zegar czasu rzeczywistego
IRQ 9 / 1	71h	Powrót pionowy CGA (i inne urządzenia IRQ 2)
IRQ 10 / 2	72h	Zarezerwowane
IRQ 11 / 3	73h	Zarezerwowane
IRQ 12 / 4	74h	Zarezerwowane w AT, urządzenie pomocnicze w systemie PS/2
IRQ 13 / 5	75h	Przerwanie FPU
IRQ 14 / 6	76h	Sterownik dysku twardego
IRQ 15 / 7	77h	Zarezerwowane

Tablica 66: Wejścia programowalnego sterownika przerwań

PIC 8259 jest bardzo złożonym chipem do oprogramowania. Na szczęście, całe to trudne zadanie zostaje zrobione przez BIOS kiedy jest ładowany system. Nie będziemy omawiali tu jak zainicjalizować 8259, w tym tekście, ponieważ taka informacja jest użyteczna tylko dla piszących systemy operacyjne takie jak Linux, Windows lub OS/2, Jeśli chcesz uruchomić swój podprogram obsługi przerwań poprawnie pod DOS'em lub innym OS'em, nie musisz reinicjalizować PIC'a.

Sprzęgamy PIC z systemem poprzez cztery lokacje I/O; port 20h / 0A0h i 21h / 0A1h. Pierwszy adres w każdej parze jest adresem master'a PIC'a (IRQ 0-7)



Rysunek 17.1 Rejestr maskowania przerwań 8259

drugi adres w każdej parze odpowiada slave'owi PIC (IRQ 8-15). Port 20h / 0A0h jest lokacją odczyt / zapis, z której zapisujemy polecenia PIC i odczytujemy status PIC, będziemy się do tego odnosili jako rejestrów poleceń lub rejestrów stanu. Rejestr poleceń jest tylko do zapisu, rejestr statusu jest tylko do odczytu. W zasadzie dzielą one tą samą lokację I/O linia odczyt/zapis w PIC określa który rejestr CPU jest dostępny. Port 21h / 0A1h jest lokacją odczyt / zapis, która zawiera rejestr maskowania przerwań, do którego będziemy się odnosili jako rejestru maski. Wybieramy właściwy adres w zależności od tego, jakiego sterownika przerwań chcemy użyć.

Rejestr maskowania przerwań jest ośmiobitowym rejestrem, który pozwala nam pojedynczo blokować i odblokowywać przerwania z urządzenia do systemu. Jest to podobne do działań instrukcji cli i sti .Zapisanie zera do odpowiedniego bitu aktywuje dane przerwanie urządzenia. Zapis jedynki blokuje przerwanie z urządzenia. Zauważmy, że nie jest to intuicyjne. Rysunek 17.1 pokazuje rejestr maskowania przerwań.

Kiedy zmieniamy bity w rejestrze maski, ważne jest aby po prostu nie załadować al wartością i wyprowadzić jej bezpośrednio do portu rejestru maski.. Zamiast tego powinniśmy odczytać rejestr maski a potem logicznym or wprowadzić lub and wyprowadzić bity jakie chcemy zmienić. Następująca sekwencja kodu aktywuje COM1: przerwanie bez wpływania na inne:

```
in      al., 21h        ;odczyt istniejących bitów
and     al., 0efh       ;włączenie IRQ4 (COM1)
```

Rejestr poleceń dostarcza więcej opcji, ale są tylko trzy polecenia, jakie chcielibyśmy wykonać na tym chipie, jakie są kompatybilne z inicjalizacja BIOS'a 8259; wysłanie polecenia końca przerwania i wysłanie jednego z dwóch poleceń rejestru statusu.

Przy wystąpieniu określonego przerwania, 8259 maskuje dalsze przerwania z tego urządzenia dopóki nie otrzyma sygnału końca przerwania z podprogramu obsługi przerwań. W DOS'ie wykonujemy to poprzez wpisanie wartości 20h do rejestru poleceń.. Robi to poniższy kod:

```
mov     al., 20h
out     20h, al.                        ;Port 0A0h jeœli IRQ 8-15
```

Musimy wysłać dokładnie jedno polecenie końca przerwania do PIC dla każdego przerwania jakie obsługujemy. Jeśli nie wyślemy polecenia końca przerwania, PIC nie będzie honorował żadnego innego przerwania z tego urządzenia; jeśli wyślemy dwa lub więcej poleceń końca przerwania, możliwe jest , że przypadkowo potwierdzimy nowe przerwanie , które może być w toku i zgubimy to przerwanie.

Dla pewnych programów obsługi przerwań nasz ISR nie będzie jedynie ISR'em, który wywołuje przerwania. Na przykład, BIOS PC dostarcza ISR dla przerwania zegarowego , który zajmuje się czasem. Jeśli

pogrzebiemy w tym przerwaniu, będziemy musieli wywołać ISR BIOS'a PC aby system działał poprawnie z czasem i obsłużyć pokrewne czasowo zadania (zobacz „Podprogramu obsługi przerwań wiązań łańcuchowych") . Jednakże ISR zegara BIOS'a wyprowadza polecenie końca przerwania. Dlatego też nie powinniśmy wyprowadzać polecenia końca przerwania sami, w przeciwnym razie BIOS wyprowadzi drugie polecenie końca przerwania i możemy zgubić przerwanie w procesie.

Inne dwa polecenie jakie możemy wysłać do 8259 pozwalają nam wybrać odczyt z rejestru obsługiwanego przerwania (ISR) lub rejestru zgłaszającego przerwanie (IRR). Rejestr obsługiwanego przerwania zawiera zbiór bitów dla każdego aktywnego ISR'a (ponieważ 8259 zezwala na priorytetowość ISR, jest całkiem możliwe ,że jeden ISR będzie przerwany przez ISR o wyższym priorytecie). Rejestr zgłaszający przerwanie zawiera zbiór bitów na odpowiednich pozycjach dla przerwania, które nie było jeszcze obsłużone (prawdopodobnie przerwanie ma mniejszy priorytet niż przerwanie aktualnie obsługiwane przez system). Odczytując rejestr obsługiwanego przerwania wykonamy następujące instrukcje:

```
;Odczyt rejestru obsługiwanego przerwania w PIC #1 (pod adresem I/O 20h)
                mov     al., 0bh
                out     20h, al.
                in      al, 20h
```

Odczytując rejestr zgłaszający przerwania użyjemy poniższego kodu:

```
;Odczyt rejestru zgłaszającego przerwanie w PIC #1 (pod adresem I/O 20h)
                mov     al., 0ah
                out     20h, al
                in      al, 29h
```

Zapisanie innych wartości poleceń portów może spowodować niepoprawne działanie systemu.

17.4.2 PRZERWANIE ZEGAROWE (INT 8)

Płyta główna PC zawiera kompatybilny chip zegarowy 8254. Chip ten zawiera trzy kanały zegarowe, każdy generujący przerwania (w przybliżeniu) co 55 ms. Jest to około 1/18.2 sekundy. Często słyszymy, że to przerwanie odnosi się do „zegara osiemnastosekundowego". Po prostu będziemy wywoływać to przerwanie zegarowe.

Wektor przerwania zegarowego jest prawdopodobnie jest najpowszechniej poprawianym przerwaniem w systemie. Okazuje się, że są dwa wektory przerwań zegarowych w systemie. Int 8 jest wektorem sprzętowym powiązanym z przerwaniem zegarowym (ponieważ przychodzi od IRQ 0 w PIC). Generalnie nie powinniśmy łatać tego przerwania jeśli chcemy napisać zegarowy ISR. Zamiast tego powinniśmy poprawić drugie przerwanie zegarowe, przerwanie 1ch. Podprogram obsługi przerwania zegarowego BIOS (int 8) wykonuje instrukcję 1ch zanim wraca. Daje to użytkownikowi poprawionego podprogramu dostęp do przerwania zegarowego. Chyba ,że chętnie zduplikujemy kod zegarowy D|BIOS i DOS, chociaż nigdy nie powinniśmy całkowicie zamieniać istniejących zegarowych ISR'ów na jeden ze swoich własnych, powinniśmy zawsze zakładać, że ISR'y BIOS lub DOS wykonają dodatkowo nasz ISR. Poprawka w wektorze 1ch jest najłatwiejszym sposobem zrobienia tego.

Ale nawet zamiana wektora 1ch na wskaźnik do naszego ISR'a jest bardzo niebezpieczna. Podprogram obsługi przerwań zegarowych jest jednym z najczęściej poprawianych przez różne programy rezydentne (zobacz „Programy rezydentne") Prze proste zapisanie adresu naszego ISR'a w wektorze przerwania zegarowego możemy zablokować taki program rezydentny i spowodować, że nasz system będzie źle funkcjonował. Do rozwiązania tego problemu musimy stworzyć łańcuch przerwań. Po więcej szczegółów zajrzyjmy do :Podprogramy obsługi przerwań wiązań łańcuchowych".

Domyślnie przerwanie zegarowe jest zawsze włączone w chipie sterownika przerwań. Faktycznie, zablokowanie tego przerwania może spowodować krach naszego systemu lub co najmniej złe funkcjonowanie. Albo co najmniej system nie będzie wskazywał poprawnie czasu jeśli zablokujemy przerwanie zegarowe.

17.4.3 PRZERWANIE KLAWIATURY (INT 9)

Mikrokontroler klawiatury na płycie głównej PC generuje dwa przerwania przy każdym naciśnięciu klawiszy – jeden kiedy naciskamy klawisz i jeden kiedy go zwalniamy. Jest to IRQ 1 na master PIC'u. BIOS

odpowiada na to przerwanie poprzez odczyt kodu klawisza klawiatury, konwertując go do znaku ASCII i przechowuje kod i kod ASCII w systemowym buforze klawiatury.

Domyślnie to przerwanie jest zawsze włączone. Jeśli zablokujemy to przerwanie, system nie będzie mógł odpowiedzieć na wciskanie klawiszy, wliczając w to ctrl-alt-del. Dlatego też nasze programy powinny zawsze włączać to przerwanie, jeśli zostało ono zablokowane.

Po więcej informacji o przerwaniu klawiatury zajrzymy do „Klawiatura PC".

17.4.4 PRZERWANIA PORTU SZEREGOWEGO (INT 0Bh i INT 0Ch)

PC używa dwóch przerwań IRQ 3 i IRQ 4 do wsparcia przerwania komunikacji szeregowej. Chip sterownika komunikacji szeregowej (SCC) 8250 (lub kompatybilny) generuje przerwanie w jednej z czterech sytuacji: pojawia się znak na lini szeregowej, SCC kończy transmisję znaku i mamy żądanie innego, pojawił się błąd lub wystąpiła zmiana statusu. SCC aktywuje taka samą linię przerwania (IRQ 3 lub 4) dla wszystkich czterech źródeł przerwań. Podprogram obsługi przerwań jest odpowiedzialny za dokładne określenie natury przerwania poprzez zapytanie SCC.

Domyślnie, system blokuje IRQ 3 i IRQ 4. Jeśli instalujemy szeregowy ISR, będziemy musieli wyzerować bit maski przerwania w 8259 PIC przed tym nim będziemy odpowiadać na przerwania z SCC. Co więcej, projekt SCC zawiera własną maskę przerwania. Będziemy musieli również odblokować maskę przerwania na chipie SCC.

17.4.5 PRZERWANIA PORTU RÓWNOLEGŁEGO (INT 0Dh I 0Fh)

Przerwania portu równoległego są zagadką IBM zaprojektował oryginalny system pozwalający na dwa przerwania portu równoległego a potem natychmiast zaprojektował kartę interfejsu drukarki, która nie wspierała zastosowania tych przerwań. W wyniku, jedynie oprogramowanie oparte nie o DOS używa przerwań portu równoległego (IRQ 5 i IRQ 7). Istotnie w systemie PS/2 IBM zarezerwował IRQ 5 , które uprzednio używane było dla LPT2.

Jednakże, przerwania te nie marnują się. Wiele urządzeń, których inżynierowie IBM nie mogli przewidzieć, kiedy projektowali pierwsze PC, mogą znaleźć dobre zastosowanie dla tych przerwań. Przykładami są karty SCSI i karty dźwiękowe.. Wiele dzisiejszych urządzeń zawiera „zworki przerwań", które pozwalają nam wybierać IRQ 5 lub IRQ 7 kiedy instalujemy urządzenie.

Ponieważ IRQ 5 i IRQ 7 mają takie małe zastosowanie dla przerwań portu równoległego, zignorujemy „przerwania portu równoległego" w tym tekście.

17.4.6 PRZERWANIA DYSKIETKI I DYSKU TWARDEGO (INT 0Eh I INT 76H)

Dyskietka i dysk twardy generują przerwania przy finalizowaniu operacji dyskowych. Jest to bardzo użyteczna cecha dla systemów wielozadaniowych, takich jak OS/2, Linux czy Windows. Podczas gdy dysk odczytuje lub zapisuje dane, CPU może wykonywać instrukcje dla innego procesu. Kiedy dysk kończy operację odczytu lub zapisu, przerywa CPU więc może on wznowić oryginalne zadanie.

Gdybyśmy zajęli się urządzeniami dyskowymi jako interesującym tematem w tym tekście, książka ta musiałaby by ć znacznie dłuższa. Dlatego też, tekst ten unika omawiania przerwań urządzeń dyskowych (IRQ 6 i IRQ 14). Jest wiele tekstów, które omawiają nisko poziomowe I/O w asemblerze.

Domyślnie przerwania dyskietki i twardego dysku są zawsze włączone. Nie powinniśmy zmieniać tego stanu jeśli zamierzamy używać urządzeń dyskowych w systemie.

17.4.7 PRZERWANIE ZEGARA CZASU RZECZYWISTEGO (INT 70h)

PC/AT i późniejsze maszyny zawierają zegar czasu rzeczywistego CMOS. Urządzenie to jest zdolne do generowania przerwania zegarowego w wielokrotności 976 mikrosekund (wywołanie 1ms). Domyślnie przerwanie zegara czasu rzeczywistego jest zablokowane. Powinniśmy włączać tylko to przerwanie jeśli mamy zainstalowany ISR int 70h.

17.4.8 PRZERWANIE FPU (INT 75h)

FPU 80x87 generuje przerwanie jeśli kiedykolwiek wystąpi wyjątek zmienno przecinkowy. W CPU z wbudowanym FPU (80486DX i lepszych) jest bit w jednym z rejestrów sterujących, jaki możemy ustawić do symulowania przerwania wektorowego BIOS generalnie inicjalizuje takie bity dla zgodności z istniejącym systemem.

Domyślnie BIOS blokuje przerwanie FPU. Większość programów, które używają FPU wyraźnie testuje rejestr stanu FPU do określenia czy wystąpił błąd. Jeśli chcemy pozwolić na przerwanie FPU, musimy włączyć przerwania w 8259 i FPU 80x87.

17.4.9 PRZERWANIA NIEMASKOWALNE (INT 2)

Chipy 80x86 w rzeczywistości dostarczają dwóch rodzajów końcówek przerwań. Pierwsze to przerwania maskowalne. Jest to końcówka do której jest dołączony PIC 8259. Przerwanie to jest maskowalne ponieważ możemy włączyć lub wyłączyć go instrukcjami cli i sti. Przerwanie niemaskowalne, jak wskazuje nazwa, nie może być zablokowane programowo. Generalnie, PC używa tego przerwania do zasygnalizowania błędu parzystości pamięci, chociaż pewne systemy używają tego przerwania również do innych celów Wiele starczych systemów PC przyłącza FPU do tego przerwania.

To przerwanie nie może być zamaskowane, więc domyślnie zawsze jest włączone.

17.4.10 INNE PRZERWANIA

Jak wspomniano w sekcji o PIC 8259, jest kilka przerwań zarezerwowanych przez IBM. Wiele systemów używa tych zarezerwowanych przerwań dla myszki i innych celów. Ponieważ takie przerwania są z natury zależne od systemu, nie będziemy ich tu omawiać.

17.5 PODPROGRMAY OBSŁUGI PRZERWAŃ WIĄZAŃ ŁAŃCUCHOWYCH

Podprogramy obsługi przerwań dzielą się na dwie podstawowe grupy – te , które potrzebują zastrzeżonego dostępu do wektora przerwań i te, które musza dzielić wektor przerwań z kilkoma innymi ISR'ami. Te z pierwszej kategorii wliczają w to obsługę błędów ISR (np. błąd dzielenia lub przepełnienia) i pewne sterowniki urządzeń. Port szeregowy jest dobrym przykładem urządzenia, które rzadko ma więcej niż jeden ISR powiązany ze sobą w danym czasie. ISR'y zegara, zegara czasu rzeczywistego i klawiatury generalnie podpadają pod drugą kategorię. Nie jest wcale niezwykłe znaleźć kilka ISR'ów w pamięci, dzielących każde z tych przerwań.

Dzielenie wektora przerwań jest raczej łatwe. Wszystko co ISR musi zrobić przy dzieleniu wektora przerwań to zachować stary wektor przerwań kiedy instalowany jest ISR (czasami musimy zrobić to tak i tak, więc możemy przywrócić wektor przerwań kiedy nasz kod się skończy) a potem wywołać oryginalny ISR przed lub po tym jak przetworzymy nasz własny ISR. Jeśli zachowamy adres oryginalnego ISR'a w dseg, w zmiennej podwójnego słowa OldIntVect, możemy wywołać oryginalny ISR kodem takim jak ten:

```
; Przypuszczalnie DS. wskazuje DSEG w tym punkcie
                pushf                       ;symulowanie instrukcji INT przez odłożenie flag i wykonanie
                call    OldIntVect          ;dalekiego wywołania
```

Ponieważ OldIntVect jest zmienną dword, kod ten generuje dalekie wywołanie do podprogramu, którego adres segmentowy pojawia się w zmiennej OldIntVect. Kod ten nie skacze do lokacji zmiennej OldIntVect.

Wiele programów obsługi przerwań nie modyfikuje rejestru ds. wskazującego lokalny segment danych. Faktycznie, niektóre proste ISR'y nie zmieniają żadnego rejestru segmentowego. W takich przypadkach jest popularne wstawienie koniecznej zmiennej (zwłaszcza wartości starego segmentu) bezpośrednio w segmencie kodu. Jeśli to zrobimy nasz kod może skoczyć bezpośrednio do oryginalnego ISR'a zamiast go wywoływać. Możemy użyć takiego kodu:

```
MyISR           proc    near
                -
                -
                -
                jmp     cs: OldIntVect
```

```
MyISR           endp
OldIntVect      dword   ?
```

Ta sekwencja kodu przekazuje flagi naszego ISR'a, adres powrotny flag i wartość adresu powrotnego do oryginalnego ISR'a. Świetnie, kiedy oryginalny ISR wykonuje instrukcję iret, będzie wracał bezpośrednio do kodu przerywającego (zakładając, że nie przekazał sterowania do jakiegoś innego ISR w łańcuchu).

Zmienna OldIntVect musi być w segmencie kodu jeśli używamy tej techniki do przekazania sterowania do oryginalnego ISR'a. W końcu kiedy wykonujemy powyższą instrukcję jmp, musimy mieć już przywrócony stan CPU, wliczając w to rejestr ds. Dlatego też, nie wiemy jaki segment wskazuje ds. a jest prawdopodobne, że nie wskazuje naszego segmentu lokalnego. Istotnie, jedyny rejestr segmentowy jakiego wartość jest znana do cs, więc musimy przechować adres wektora w segmencie kodu.

Poniższy prosty program demonstruje przerwania łańcuchowe. Ten krótki program poprawia wektor 1ch. ISR zlicza sekundy i powiadamia program główny o każdej mijającej sekundzie . Program główny drukuje krótką wiadomość co sekundę. Kiedy minie 10 sekund, program usuwa ISR z łańcucha przerwań i kończy się

```
;TIMER.ASM
;Program ten demonstruje jak poprawić wektor przerwania zegarowego 1Ch i stworzyć łańcuch przerwań

                .xlist
                .286
                include         stdlib.a
                includelib      stdlib.lib
                .list

dseg            segment  para  public  'data'

;TIMERISR będzie uaktualniał poniższe dwie zmienne
;Uaktualni zmienną MSEC co 55 ms
;Uaktualni zmienną TIMER co sekundę

MSEC            word    0
TIMER           word    0

dseg            ends

cseg            segment  para  public  'code'
                assume  cs:cseg, ds:dseg

; Zmienna OldIntVect musi być w segmencie kodu z powodu sposobu w jaki TimerISR przekazuje sterownie do
; następnego ISR'a w łańcuchu int 1Ch

OldIntVect      dword   ?

; Podprogram obsługi przerwania zegarowego
; Zwiększa zmienną MSEC o 55 przy każdym przerwaniu.
; Ponieważ to przerwanie wywoływane jest co 55 ms (w przybliżeniu)
; zmienna MSEC zawiera bieżąca liczbę milisekund.
; Kiedy wartość ta  przekroczy 1000 (jedna sekunda), ISR odejmie
; 1000 od zmiennej MSEC i zwiększy TIMER o jeden

TimerISR        proc    near
                push    ds.
                push    ax
                mov     ax, dseg
                mov     ds., ax
```

```
                mov     ax, MSEC
                add     ax, 55                  ;przerwania co 55 ms
                cmp     ax, 1000
                jb      SetMSEC
                inc     Timer                   ;właśnie przekazano sekundę
                sub     ax, 1000                ;modyfikacja wartości MSEC
SetMSEC:        mov     MSEC, ax
                pop     ax
                pop     ds
                jmp     cseg: OldIntVect        ;przekazanie do oryginalnego ISR'a
TimerISR:       endp

Main            proc
                mov     ax, dseg
                mov     ds, ax
                meminit

;Zaczynamy od  podstawienie adresu naszego ISR'a  do wektora 1Ch. Zauważmy, że musimy wyłączyć
; przerwania podczas poprawiania wektora przerwań i musimy założyć, że przerwania są później przywrócone;
; instrukcje cli i sti. Jest to wymagane  ponieważ przerwanie zegarowe może nadejść pomiędzy tymi dwoma
; instrukcjami, które zapisują do wektora przerwania 1Ch. Może to nieźle namieszać

                mov     ax, 0
                mov     es, ax
                mov     ax, es:[1ch*4]
                mov     word  ptr OldInt1C, ax
                mov     ax, es:[1Ch*4+2]
                mov     word  ptr  OldInt1C+2, ax

                cli
                mov     word  ptr es:[1Ch*4], offset TimerISR
                mov     es:[1Ch*4+2],cs
                sti

;Okay, ISR uaktualni³ zmienn¹ TIMER  co sekundê
; stale drukując tą wartość dopóki nie minie 10 sekund. Potem kończy

                mov     Timer, 0
TimerLoop:      printf
                byte    "Timer = %d\n", 0
                dword   Timer
                cmp     Timer, 10
                jbe     TimerLoop

;Okay, przywracamy wektor przerwań. Musimy wyłączyć przerwania z powodów jak powyżej

                mov     ax, 0
                mov     es, ax
                cli
                mov     ax, word  ptr  OldInt1C
                mov     es:[1Ch*4], ax
                mov     ax, word ptr OldInt1C+2
                mov     es:[1Ch*4+2], ax
                sti
```

```
Quit:           ExitPgm                     ;makro DOS do wyjścia z programu
Main            endp
cseg            ends

sseg            segment  para stack ‘stack’
stk             db      1024 dup (“stack”)
sseg            ends

zzzzzzseg       segment  para public  ‘zzzzzz’
LastBytes       db      16 dup (?)
Zzzzzzseg       ends
                end     Main
```

17.6 PROBLEMY WSPÓŁBIEŻNOŚCI

Drobnym problemem konstrukcyjnym przy tworzeniu ISR'a jest to co zdarzy się jeśli włączymy przerwanie w ISR a nadejdzie drugie przerwanie z tego samego urządzenia? To przerwie ISR i potem wejdzie ponownie od samego początku ISR'a. Wiele aplikacji nie zajmuje się właściwie tymi warunkami. Aplikacja, która może właściwie obsłużyć taką sytuację jest nazywana współbieżną . Segment kodu, który nie działa poprawnie przy współbieżności jest nazywany niewspółbieżnym.

Rozpatrzmy program TIMER.ASM z poprzedniej sekcji. Jest to przykład programu nie współbieżnego. Przypuśćmy, że podczas wykonywania ISR'a, mamy przerwanie w następującym punkcie:

```
TimerISR        proc    near
                push    ds.
                push    ax
                mov     ax, dseg
                mov     ds., ax

                mov     ax, MSEC
                add     ax, 55              ;przerwanie co 55 ms
                cmp     ax, 1000
                jb      SetMSEC

;<<<<<<Przypuśćmy, że przerwanie wystąpi w tym miejscu>>>>>>>

                inc     Timer               ;sekunda przekazana
                sub     ax, 1000            ;modyfikacja wartości MSEC
SetMSEC:        mov     MSEC, ax
                pop     ax
                pop     ds
                jmp     cseg:OldInt1C       ;przekazanie do oryginalnego ISR'a
TimerISR        endp
```

Przypuśćmy, że przy pierwszym wywołaniu przerwania, MSEC zawiera 950 a Timer zawiera trzy. Jeśli wystąpi drugie przerwanie w powyższym określonym miejscu, ax będzie zawierało 1005. Więc przerwanie zawiesi ISR i powróci do jego początku. Zauważmy, że TimerISR jest wystarczający dość dla zachowania rejestru ax zawierającego wartość 1005. Kiedy wykonuje się drugie wywołanie przerwania TimerISR, znajduje w MSEC jeszcze 950 ponieważ pierwsze wywołanie nie uaktualniło jeszcze MSEC. Dlatego też, dodaje 55 do tej wartości, określając, że przekroczono 1000, zwiększa Timer (ma teraz cztery) a potem przechowuje pięć w MSEC. Potem wraca (przez skok do następnego ISR'a w łańcuchu int 1Ch). Ewentualnie przekaże sterowanie do pierwszego wywołania podprogramu TimerISR. W tym czasie (mniejszym niż 55 ms po uaktualnieniu Timer przez drugie wywołanie) kod TimerISR zwiększa ponownie zmienną Timer i uaktualnia MSEC do pięć. Problem z tą sekwencją jest taki, że zwiększa zmienną Timer dwukrotnie w czasie mniejszym niż 55 ms.

Teraz możemy roztrząsać, że przerwania sprzętowe zawsze zerują flagę blokowania przerwania, więc nie byłoby możliwe dla tego przerwania aby było współbieżne. Co więcej, możemy roztrząsać, że ten podprogram jest zbyt krótki, więc nigdy nie osiagnie55ms do znanego punktu w powyższym kodzie. Jednakże zapominamy o czymś; w systemie mogą być jakieś inne zegarowe ISR'y, które wywołują nasz kod po tym jak się wykona. Taki kod osiąg a 55 ms i zdarzy się włączenie przerwania, czyniąc doskonałą możliwość aby nasz kod mógł stać się współbieżnym.

Kod pomiędzy instrukcjami mov ax, MSEC a mov MSEC, ax jest nazywany obszarem krytycznym lub krytyczną sekcją. Program nie może być współbieżny podczas wykonywania w rejonie krytycznym. Posiadanie rejonu krytycznego nie oznacza, ze program nie jest współbieżny. Większość programów, nawet te, które są współbieżne mają różne rejony krytyczne. Kluczem jest zapobieżenie przerwaniom, które powodują rejony krytyczne, będące współbieżnymi, w tych rejonach krytycznych . Najłatwiejszym sposobem zapobieżenia takiemu wystąpieniu jest wyłączenie przerwań podczas wykonywania kodu w krytycznej sekcji. Możemy łatwo zmodyfikować TimerISR do tego celu:

```
TimerISR        proc    near
                push    ds.
                push    ax
                mov     ax, dseg
                mov     ds., ax
;Zaczynamy sekcję krytyczna, wyłączamy przerwania

                pushf                   ; zachowanie bieżącego stanu flagi I
                cli                     ;upewnienie czy przerwania wyłączone

                mov     ax, MSEC
                add     ax, 55          ;przerwanie co 55 ms
                cmp     ax, 1000
                jb      SetMSEC

                inc     Timer           ;przekazana sekunda
                sub     ax, 1000        ;modyfikacja wartości MSEC
SetMSEC:        mov     MSEC, ax

;Koniec  rejonu krytycznego,  przywrócenie flagi I do jej poprzedniego stanu
                popf
                pop     ax
                pop     ds.
                jmp     cseg:OldInt1C   ;przekazanie do oryginalnego ISR'a
TimerISR        endp
```

Powrócimy do problemu współbieżności i rejonów krytycznych w następnych dwóch rozdziałach tego tekstu.

17.7 SPRAWNOSĆ SYSTEMU STEROWANEGO PRZERWANIAMI.

Przerwania wprowadzają znaczną ilość złożoności do systemu oprogramowania (zobacz „Debuggowanie ISR'a"). Można spytać czy używanie przerwań jest rzeczywiście warte tych problemów. Odpowiedź to oczywiście tak. Czy ludzie używaliby przerwań gdyby udowodniono, że nie są warte zachodu? Jednakże, przerwania są jak wiele innych świetnych rzeczy w informatyce - mają swoje miejsce; jeśli spróbujemy użyć przerwań w niewłaściwy sposób sprawy mogą zrobić się gorsze.

Następująca sekcja zgłębia aspekty wydajności zastosowania przerwań. Jak wkrótce odkryjemy, system sterownia przerwaniami jest zazwyczaj lepszy pomimo złożoności. Jednakże, nie zawsze. Dla wielu systemów metody alternatywne dostarczają lepszej wydajności.

17.7.1 UKŁAD WE/WY STEROWANY PRZERWANIAMI KONTRA ODPYTYWANIE

Celem systemu sterowania przerwaniami jest zezwolenie CPU na kontynuowanie przetwarzania instrukcji podczas gdy wystąpi aktywność I/O. Jest to bezpośredni kontrast z systemem odpytywania gdzie CPU nieustannie testuje urządzenia I/O aby zobaczyć czy operacje I/O są zakończone. W systemie sterowania przerwaniami, CPU zajmuje się swoimi sprawami, a przerwaniami urządzeń I/O zajmuje się kiedy wymagają obsługi. Generalnie jest to bardziej wydajne niż marnowanie cykli CPU na odpytywanie urządzeń kiedy nie są gotowe.

Port szeregowy jest doskonałym przykładem urządzenia , które działa zupełnie dobrze ze sterowanymi przerwaniami I/O. Możemy uruchomić program komunikacyjny, który zaczyna ściąganie pliku przez modem. Przy każdym nadejściu znaku generuje przerwanie a program komunikacyjny uruchamia się, buforuje znaki a potem wracamy z przerwania. Tymczasem inny program (jak word procesor) może być uruchomiony z prawie znikomym pogorszeniem wydajności ponieważ zabiera tak mało czasu przy przetwarzaniu przerwań portu szeregowego.

Scenariusz ten kontrastuje z tym gdzie program do komunikacji szeregowej ciągle odpytuje chip komunikacji szeregowej aby zobaczyć czy przyszedł jakiś znak. W tym przypadku CPU cały czas zajmuje się szukaniem znaku wejściowego pomimo, że rzadko (w terminologii CPU) nadchodzi. Dlatego też żadne cykle CPU nie mogą być użyte do przetwarzania takiego jak uruchomienie word procesora.

Przypuśćmy, że przerwania nie są dostępne a chcemy pozwolić na ściąganie w tle podczas używania programu word procesora. Program word procesora będzie musiał testować dane wejściowe portu szeregowego co kilka milisekund zapobiegając utracie jakiejś danej. Czy możemy sobie wyobrazić jak trudno byłoby napisać taki word procesor? System przerwań jest w tym przypadku wyborem oczywistym.

Jeśli pobieranie danych podczas przetwarzania tekstu wydaje się zbyt skomplikowane, rozważmy prostszy przypadek – klawiaturę PC. Jeśli wystąpi przerwanie naciśnięcia klawiatury, ISR klawiatury odczyta naciśnięty klawisz i zachowa go w systemowym buforze klawiatury do chwili kiedy aplikacja będzie chciała odczytać daną klawiatury. Czy możemy sobie wyobrazić jak trudno będzie napisać taką aplikację jeśli musimy stale odpytywać port klawiatury zachowując zagubione znaki? Nawet w środku długiego obliczenia? Ponownie przerwania dostarczają łatwego rozwiązania.

17.7.2 CZAS OBSŁUGI PRZERWANIA

Oczywiście, właśnie omówiony system komunikacji szeregowej jest przykładem najlepszego scenariusza. Program komunikacyjny pobiera tak mało czasu na wykonanie swojej pracy, że większość czasu pozostaje poza programem przetwarzania tekstu. Jednakże uruchamiając różne systemy I/O sterowane przerwaniami, na przykład kopiowanie jednego dysku do innego, podprogram obsługi przerwań będzie miał zauważalny wpływ na wydajność systemu przetwarzania tekstu.

Dwa czynniki sterują wpływem ISR'a na system komputerowy: częstotliwość przerwań i czas obsługi przerwania. Częstotliwość to, jak wiele razy na sekundę (lub inny pomiar czasu) poszczególnego wystąpienia przerwania. Czas obsługi przerwań to, ile czasu potrzeba ISR'owi na obsługę przerwania.

Właściwość częstotliwości różni się w zależności od źródła przerwania. Na przykład, chip zegarowy generuje równomierne przerwania około 18 razy na sekundę, podobnie, port szeregowy odbierający 9600 bps generuje więcej niż 100 przerwań na sekundę. Z drugiej strony, klawiatura rzadko generuje więcej niż około 20 przerwań na sekundę i nie są one bardzo regularne.

Czas obsługi przerwania jest oczywiście uzależniony od liczby instrukcji, które musi wykonać ISR. Czas obsługi przerwania jest również zależny od określonego CPU i częstotliwości zegara. Taki sam ISR wykonujący identyczne instrukcje na dwóch CPU, będzie wykonywał się mniej razy na szybszej maszynie.

Ilość czasu potrzebna programowi obsługi przerwania do obsłużenia przerwania pomnożona przez częstotliwość przerwania określa wpływ jaki będzie miało przerwanie na wydajność systemu. Pamiętajmy, każdy cykl CPU zużyty przez ISR jest jednym cyklem mniej dostępnym programom użytkowym. Rozważmy przerwanie zegarowe. Przypuśćmy ,że ISR zegarowy potrzebuje 100µs do zakończenia swojego zadania. To znaczy, że przerwanie zegarowe zużywa 1,8 ms co każdą sekundę lub około 0,18% całkowitego czasu komputera. Używając szybszego CPU zredukujemy te procenta (poprzez redukcję czasu zużywanego przez ISR); używając wolniejszego CPU zwiększamy te procenta. Niemniej jednak zauważmy, że taki krótki ISR jak ten nie będzie miał znaczącego wpływu na całkowitą wydajność systemu.

Sto mikrosekund to szybko dla typowego ISR'a, zwłaszcza kiedy nasz system ma kilka zegarowych ISR'ów połączonych razem. Jednakże, nawet jeśli ISR zegarowy pobrał 10 razy tyle przy wykonaniu, pozbawiłby tylko system mniej niż 2% dostępnych cykli CPU. Nawet jeśli pobrałby 100 razy więcej (10ms), wystąpiłoby pogorszenie wydajności tylko o 18%; większość ludzi ledwie zauważyłoby takie pogorszenie .

Oczywiście nie można pozwolić aby ISR pobierał tyle czasu ile chce. Ponieważ przerwanie zegarowe występuje co 55 ms, maksymalny jaki może użyć ISR jest poniżej 55 ms. Jeśli ISR wymaga więcej czasu niż jest pomiędzy przerwaniami, system może ostatecznie zgubić przerwanie. Co więcej, system zużyje cały swój czas na obsługę przerwań zamiast zajmować się czymś innym. Wiele systemów mających ISR'y zużywające 10% całkowitych cykli CPU nie dostarcza problemu. Jednakże, zanim przestaniemy go lubić i zaczniemy projektować wolniejszy podprogram obsługi przerwań, powinniśmy zapamiętać, że nasz ISR nie jest prawdopodobnie jedynym ISR'em w systemie. Jeśli nasz ISR zajmuje 25% cykli CPU, może być inny ISR, który robi to samo; i inny, i inny i... Co więcej mogą być ISR'y, które wymagają szybszej obsługi. Na przykład ISR portu szeregowego może musieć odczytać znak z chipu komunikacji szeregowej co każdą milisekundę. Jeśli nasz ISR zegarowy wymaga 4 ms dla wykonania i robi to z wyłączonym przerwaniem, ISR portu szeregowego starci pewne znaki.

Ostatecznie oczywiście chcemy napisać ISR'y, które byłyby tak szybkie jak to tylko możliwe, więc muszą mieć mniejszy wpływ na wydajność systemu. Jest to jeden z głównych powodów dla których większość ISR'ów pod DOS jest pisana w języku asemblera. Chyba ,że projektujemy system wbudowany, na którym działa tylko Twoja aplikacja, wtedy musimy zdać sobie sprawę, ż nasze ISR'y muszą koegzystować z innymi ISR'ami i aplikacjami; nie chcemy aby wydajność naszego ISR'a niekorzystnie wpływał na wydajność innego kodu w systemie.

17.7.3 WSTRZYMANIE OBSŁUGI PRZERWANIA

Wstrzymanie obsługi przerwania jest to czas pomiędzy punktem w którym urządzenie sygnalizuje ,że potrzebuje obsługi a punktem w którym ISR dostarcza potrzebnej obsługi. Nie jest to natychmiastowe! PIC 8259 musi zasygnalizować CPU, CPU musi przerwać bieżący program, odłożyć flagi i adres powrotu, uzyskać adres ISR i przekazać sterowanie do ISR'a. ISR może wymagać odłożenia różnych rejestrów, ustawienia pewnych zmiennych, sprawdzenia stanu urządzenia dla określenia źródła przerwania i tak dalej. Ponadto mogą być inne ISR'y połączone w wektor przerwań przed naszym i wykonują się one całkowicie przed przekazaniem sterowania do naszego ISR'a, który w rzeczywistości obsługuje to urządzenie. Ostatecznie ISR w rzeczywistości robi to co urządzenie uważa ,ze powinno być zrobione. W najlepszym przypadku na najszybszym mikroprocesorze z prostym ISR'em, opóźnienie może być rzędu mikrosekund. W wolniejszych systemach z kilkoma ISR'ami w łańcuchu, opóźnienie może sięgać kilku milisekund.

Dla pewnych urządzeń, wstrzymanie obsługi przerwania jest ważniejsze niż faktyczny czas obsługi. Na przykład, urządzenie wejściowe może przerwać CPU co 10 sekund. Jednakże, to urządzenie może nie potrafić przechować dane na swoim porcie wejściowym przez więcej niż milisekundę. Teoretycznie czas obsługi przerwania mniejszy niż 10 sekund jest dobry; ale CPU musi odczytać dane w przeciągu jednej milisekundy swojego wejścia lub system zgubi dane.

Niskie wstrzymanie obsługi przerwania (to znaczy, odpowiadające szybko) jest bardzo ważne w wielu aplikacjach. Istotnie, w pewnych aplikacjach wymagania co do opóźnienia są ścisłe jeśli musimy użyć bardzo szybkiego CPU lub musimy porzucić całkowicie przerwania i powrócić do odpytywania. Momencik! Czy odpytywanie nie jest mniej wydajne niż system sterowany przerwaniami? Jak odpytywanie może coś poprawić?

System sterowany przerwaniami I/O poprawia wydajność systemu poprzez zezwolenie CPU na działanie z innymi zadaniami pomiędzy operacjami I/O. W zasadzie obsługiwanie przerwań zabiera bardzo mało czasu CPU w porównaniu do nadchodzących przerwań w systemie. Poprzez zastosowanie I/O sterowanych przerwaniami możemy użyć wszystkich tych innych cykli CPU dla jakichś innych celów. Jednak przypuśćmy, że urządzenie I/O tworzy żądanie obsługi z taką szybkością, że nie ma wolnych cykli CPU. I/O sterowane przerwaniami dostarczy kilku korzyści w takim przypadku.

Na przykład przypuśćmy, że mamy ośmiobitowe urządzenie I/O połączone z dwoma portami I/O. Przypuśćmy, że bit zero portu 310h zawiera jeden jeśli dana jest dostępna i zero w przeciwnym wypadku. Jeśli dana jest dostępna, CPU musi odczytać osiem bitów z portu 311h. Odczytanie portu 311h zeruje bit zero portu 310h dopóki nie nadejdzie kolejny bajt. Jeśli chcemy odczytać 8192 bajty z tego portu możemy zrobić to z następującym fragmentem kodu:

```
                mov     cx, 8192
                mov     dx, 310h
                lea     bx, Array               ;wskazuje bx jako bufor pamięci
DataAvailLP:    in      al., dx                 ;odczyt statusu portu
                shr     al, 1                   ;test bitu zero
                jnc     DataAvailLp             ;czekaj dopóki jest dana dostępna
                inc     dx                      ;wskazuje port danych
```

```
        in      al., dx                         ;odczyt danych
        mov     [bx], al.                       ;przechowanie danych w buforze
        inc     bx                              ;przesunięcie na następny element tablicy
        dec     dx                              ;pokazuje ponownie status portu
        loop    DataAvailLp                     ;powtarzane 8192 razy
        -
        -
        -
```

Kod ten używa klasycznej pętli odpytywania (DataAvailLp) do oczekiwania na każdy dostępny znak. Ponieważ są tylko trzy instrukcje w pętli odpytującej, pętla ta może prawdopodobnie wykonywać się mniej niż w mikrosekundę. Więc może on zająć jedną mikrosekundę do określenia czy dana jest dostępna, w którym przypadku kod nie dochodzi do skutku i przez sekundę instrukcje w tej sekwencji odczytamy dane z urządzenia. Bądźmy szczerzy i powiedzmy, że pobiera następną mikrosekundę. Przypuśćmy, zamiast tego, że używamy podprogramu obsługi przerwań. Dobrze napisany ISR połączony z dobrze zaprojektowanym systemem sprzętowym będzie prawdopodobnie miał opóźnienia mierzone w mikrosekundach.

Mierząc najlepsze opóźnienia możemy mieć nadzieję, że osiągniemy wymaganą część zegara sprzętowego, który zaczyna odliczać, kiedy wystąpi przerwanie. Na wejściu do naszego programu obsługi przerwania możemy odczytać ten licznik do określenia ile czasu minęło pomiędzy przerwaniem a jego obsługą. Na szczęście, takie urządzenie istnieje na PC – chip zegarowy 8254, który dostarcza źródła przerwania 55 ms.

Chip zegarowy 8254 w rzeczywistości zawiera trzy oddzielne zegary: zegar #0, zegar #1 i zegar #2. Pierwszy zegar (zegar #0) dostarcza przerwania zegarowego, więc skupimy się na nim w naszym omówieniu.. Zegar zawiera 16 bitowy rejestr, który 8254 zmniejsza w regularnych odstępach (1,193,180 razy na sekundę). Kiedy zegar osiąga zero, generuje przerwanie na lini IRQ 0 8259 a potem przechodzi cyklicznie do 0FFFFh i kontynuuje odliczanie w dół od tego punktu. Ponieważ licznik automatycznie resetuje do 0FFFFh po wygenerowaniu każdego przerwania, to znaczy, że zegar 8254 generuje przerwania co 65,536 / 1,193,180 sekund lub co każde 54,9254932198 ms, czyli 18.2064819336 razy na sekundę. Będziemy wywoływać to co 55 ms lub 18 (lub 18,2) razy na sekundę odpowiednio. Innym sposobem przedstawienia tego jest to ,że 8254 zmniejsza licznik co 838 nanosekund (lub 0,838 μs).

Poniższy krótki program asemblerowy odmierza opóźnienie przerwania poprzez poprawianie wektora 8. Kiedykolwiek chip zegarowy odlicza w dół do zera, generuje przerwanie, które bezpośrednio wywołuje ten ISR programu. ISR szybko odczytuje rejestr licznika chipu zegarowego, negując wartość (więc 0FFFFh staje się jeden, 0FFFEh staje się dwa itd.) a potem dodaje do całości. ISR również zwiększa licznik aby można było prześledzić liczbę razy jaką trzeba dodać wartość licznika do wartości całkowitej. Potem ISR skacze do oryginalnego programu obsługi int 8. Program główny, w między czasie, po prostu oblicza i wyświetla bieżący średni odczyt z licznika. Kiedy użytkownik naciska jakiś klawisz, program się kończy.

```
;Program ten  mierzy  opóźnienie ISR Int 08.
;Działa przez odczyt chipu zegarowego bezpośrednio na wejściu do ISR INT 08. Przez uśrednienie  tej wartości
; dla jakiejś liczby wykonań,  możemy określić średnie opóźnienie dla tego kodu.

                .xlist
                .386
                option          segment:use16
                include         stdlib.a
                includelib      stdlib.lib
                .list

cseg            segment   para public  ‘code’
                assume  cs:cseg, ds: nothing

;Wszystkie zmienne są w segmencie kodu żeby zredukować opóźnienie ISR’a (nie musimy odkładać i ustawiać
; DS., zachowując kilka instrukcji na początku ISR’a)

OldInt8         dword   ?
```

```
SumLatency      dword   0
Executions      dword   0
Average         dword   0
```

; Program ten odczytuje chip zegarowy 8254. Ten chip zlicza od 0FFFFh w dół do zera a potem generuje przerwanie. Zawija od 0 do 0FFFFh i kontynuuje odliczanie w dół generując przerwanie
;
; Adres portu chipu zegarowego 8254 :

```
Timer0_8254     equ     40h
Cntrl_8254      equ     43h
```

;Następujący ISR odczytuje chip zegarowy 8254 , neguje wynik (ponieważ zegar odlicza w tył), dodaje wynik
; do zmiennej SumLatency, a potem zwiększa zmienną Executions, która liczy liczbę wykonań tego kodu.
;Tymczasem program główny jest zajęty obliczaniem i wyświetlaniem średniego opóźnienia dla tego ISR'a
;
;Odczytując 16 bitową wartość licznika 8254, kod ten musi zapisać zero do portu sterującego 8254 a potem odczytać
;dwukrotnie port zegarowy (odczytuje najmniej i najbardziej znaczący bajt). Musi być krótkie opóźnienie pomiędzy
; odczytem dwóch bajtów z tego samego adresu portu

```
TimerISR        proc    near
                push    ax
                mov     eax, 0                  ;Ch 0, zatrzask i odczyt danych
                out     Cntrl_8254, al.         ;Wyprowadzenie  do lini poleceń rejestru  8253
                in      al., Timer0_8254        ;Odczyt  zatrzasku #0 (LSB) & ignoruje
                mov     ah, al.
                jmp     SettleDelay             ;osadzenie opóźnienia dla chipu 8254
SettleDelay:    in      al., Timer0_8254        ;Odczytuje zatrzask #0 (MSB)
                xchg    ah, al
                neg     ax
                add     cseg: SumLatency, eax
                inc     cseg:Executions
                pop     ax
                jmp     cseg: oldInt8
TimerISR        endp

Main            proc
                Meminit
```

; Zaczynamy od poprawienia adresu naszego ISR'a w wektorze int 8. Zauważmy ,że musimy wyłączyć przerwania
;podczas rzeczywistego poprawiania wektora przerwań i musimy założyć, że przerwania są ponownie włączane;
; stąd instrukcje cli i sti. Są wymagane ponieważ przerwania zegarowe mogą nadejść między dwoma instrukcjami
;które zapisują do wektora int 8. Ponieważ wektor przerwań jest w tym punkcie w niespójnym stanie, o może
;powodować krach systemu

```
                mov     ax, 0
                mov     es, ax
                mov     ax, es:[8*4]
                mov     word  ptr OldInt8, ax
                mov     ax, es:[8*4+2]
                mov     word  ptr OldInt8+2, ax

                cli
                mov     word  ptr es:[8*4], offset TimerISR
                mov     es:[8*4+2], cs
```

```
                sti

;Najpierw czekamy na pierwsze wywołanie powyższego ISR'a. Ponieważ będziemy dzielili przez wartość w
; zmiennej Executions, musimy upewnić się, że jest większa niż zero  przed wykonaniem

Wait4Non0:      cmp     cseg: Executions, 0
                je      Wait4Non0

; Okay, zaczynamy wyświetlanie dobrej wartości dopóki użytkownik naciśnie klawisz na klawiaturze do
;zatrzymania wszystkiego:

DisplayLp:      mov     eax, SumLAtency
                cdq                                     ;rozszerzamy eax -> edx
                div     Executions
                mov     Average, eax
                printf
                byte    „Count: %ld, average: %ld\n", 0
                dword   Executions, Average

                mov     ah, 1                           ;Test dla  nacicniêcia klawisza
                int     16h
                je      DisplayLp
                mov     ah, 0                           ;Odczyt tego naciśnięcia klawisza
                int     16h

;Okay, przywracamy wektor przerwań. Musimy tu wyłączyć przerwania  z tego samego powodu co powyżej

                mov     ax, 0
                mov     es, ax
                cli
                mov     ax, word  ptr OldInt8
                mov     es:[8*4], ax
                mov     ax, word ptr OldInt8+2
                mov     es:[8*4+2], ax
                sti

Quit:           ExitPgm                                 ;Makro DOS dla wyjścia z programu
Main            endp

cseg            ends

sseg            segment   para stack ' stack'
stk             db      1024  dup ("stack")
sseg            ends

zzzzzzseg       segment   para public 'zzzzzz'
LastBytes       db      16   dup (?)
Zzzzzzseg       ends
                end     Main
```

Na procesorze 66 MHz 80486DX/2 powyższy kod raportuje średnią wartość 44 po jego uruchomieniu około 10000 iteracji. Wyniesie to około 37 µs pomiędzy urządzeniem sygnalizującym przerwanie a ISR'em, który może go przetworzyć. Opóźnienie odpytywania I/O byłoby prawdopodobnie mniejszego rozmiaru niż ten!

Generalnie, jeśli mamy jakąś aplikację o dużej szybkości, taką jak nagrywanie audio lub video lub odgrywanie, prawdopodobnie nie możemy pozwolić na opóźnienia związane z przerwaniami I/O. Z drugiej strony taka aplikacja żąda takiej wysokiej wydajności z systemu, której prawdopodobnie nie będziemy mieli cykli CPU potrzebnych do innych procesów podczas oczekiwania na I/O.

Inną sprawą w związku z opóźnieniem ISR jest zgodność opóźnienia. To znaczy, czy jest taka sama ilość opóźnienia od przerwania do przerwania? Pewne ISR'y mogą tolerować znaczne opóźnienia tak długo jak jest ono spójne (to znaczy, opóźnienie jest mniej więcej takie samo od przerwania do przerwania). Na przykład przypuśćmy, że chcemy poprawić przerwanie zegarowe aby można było odczytać port wejściowy co 55 ms i przechować dane. Później, kiedy przetwarzamy dane, nasz kod może działać przy założeniu, że dane są odczytywane co 55 ms (lub 54.9). Może to nie być prawdą jeśli są inne ISR'y w łańcuchu przerwania zegarowego przed naszym ISR'em/ Na przykład, może być ISR, który odlicza 18 przerwań a potem wykonuje jakąś sekwencję kodu, która wymaga 10 ms.. To znaczy, że 16 z każdych 18 przerwań naszego programu gromadzenia danych będzie gromadził dane z 55 ms przerwami . Ale kiedy wystąpi osiemnaste przerwanie, inny ISR zegarowy będzie opóźniony o 10 ms przed przekazaniem sterowania do naszego podprogramu. To znaczy, że nasz siedemnasty odczyt to będzie 65 ms od ostatniego odczytu. Nie zapomnijmy, że chip zegara jeszcze odlicza w dół podczas wszystkiego tego, co oznacza, że teraz są tylko 45 ms do następnego przerwania. Dlatego też nasz osiemnasty odczyt wystąpiłby 45 ms po siedemnastym. Ledwie zgodnie ze wzorcem. Jeśli nasz ISR potrzebuje zgodnego opóźnienia, powinniśmy spróbować zainstalować nasz ISR najwcześniej jak to możliwe w łańcuchu przerwań.

17.7.4 PRZERWANIA PRIORYTETOWE

Przypuśćmy ,że mamy wyłączone przerwania (być może przetwarzamy jakieś przerwanie) i dwa żądania przerwań przyszłe podczas gdy przerwania są wyłączone. Co się zdarzy kiedy ponownie włączymy przerwania? Które przerwanie najpierw obsłuży CPU? Oczywistą odpowiedzią byłoby „przerwanie które wystąpiło jako pierwsze". Jednakże, przypuśćmy, że oba wystąpiły w dokładnie tym samym czasie (lub przynajmniej wewnątrz dość krótkiego odcinka czasu, który nie pozwala nam określić, które wystąpiło jako pierwsze), lub być może, jeśli rzeczywiście wystąpi taki przypadek, PIC 8259 nie może wyśledzić , które przerwanie wystąpiło pierwsze? Co więcej, co jeśli jedno z przerwań jest ważniejsze niż drugie? Na przykład przypuśćmy, że jedno przerwanie mówi, że użytkownik właśnie nacisnął klawisz na klawiaturze a drugie przerwanie mówi, że reaktor jądrowy stopi się, jeśli nie zrobimy czegoś w ciągu następnych 100 μs. Czy chcielibyśmy najpierw przetwarzać naciśnięcie klawisza, nawet jeśli nadeszło pierwsze? Prawdopodobnie nie. Zamiast tego chcielibyśmy ustalić priorytety przerwań na podstawie ich ważności; przerwanie z reaktora nuklearnego jest prawdopodobnie trochę ważniejsze niż przerwanie naciśnięcia klawisza i obsłużylibyśmy go jako pierwsze.

PIC 8259 dostarcza kilka schematów priorytetów, ale BIOS PC inicjalizuje 8259 do używania stałych priorytetów. Kiedy używamy stałych priorytetów, urządzenie na IRQ 0 (zegar) ma najwyższy priorytet a urządzenie na IRQ 7 ma najniższy priorytet. Dlatego też, 8259 w PC (działający DOS) zawsze rozwiązuje konflikty w ten sposób. Jeśli mamy zamiar podłączyć reaktor nuklearny do PC, prawdopodobnie użyjemy przerwania niemaskowalnego ponieważ ma wyższy priorytet niż inne dostarczone przez 8259 (a nie możemy zamaskować go instrukcją CLI)

17.8 DEBUGGOWANIE ISR'ÓW

Chociaż napisanie ISR'ów może uprościć zaprojektowanie wielu typów programów, ISR'y są prawie zawsze bardzo trudne do zdebuggowania. Są dwa główne powody dla których ISR'y są trudniejsze niż zwykłe aplikacje do zdebuggowania. Po pierwsze, jak wspomniano wcześniej, niesforny ISR może modyfikować wartości używane przez program główny (lub jeszcze gorzej, przez jakiś inny program w pamięci) i jest trudno sprecyzować źródło błędu. Po drugie, większość debuggerów protestuje kiedy próbujemy ustawić punkt przerwania wewnątrz ISR.

Jeśli nasz kod zawiera jakieś ISR'y a program wydaje się źle zachowywać i nie możemy bezpośrednio zobaczyć powodu, powinniśmy bezpośrednio podejrzewać ingerencję ISR'a. Wielu programistów zapomina o ISR'ach pojawiających się w ich kodzie i spędzają tygodnie próbując zlokalizować błąd w swoich nie – ISR'owych kodach, tylko odkrywają problem jaki był z ISR'em. Zawsze najpierw podejrzewaj ISR. Generalnie, ISR'y są krótkie i możemy szybko wyeliminować ISR jako przyczynę problemu zanim spróbujemy prześledzić błąd.

Debuggery często mają problemy ponieważ nie są one współbieżne lub wywołują BIOS lub DOS (które nie są współbieżne) więc jeśli ustawimy punkt przerwania w ISR'ze , który przerywa BIOS lub DOS a debugger

wywołuje BIOS lub DOS, system może nie funkcjonować z powodu problemów współbieżności. Na szczęście większość nowoczesnych debuggerów ma tryb zdalnej korekty, który pozwala nam połączyć terminal lub inny PC do portu szeregowego i wykonać polecenia debuggujące na drugim monitorze i klawiaturze. Ponieważ debugger porozumiewa się bezpośrednio z chipem szeregowym, unikajmy wywoływania BIOS lub DOS i unikajmy problemu współbieżności. Oczywiście, nie pomoże wiele jeśli napiszemy ISR szeregowy, ale działa dobrze z większością innych programów.

Dużym problemem kiedy debuggujemy podprogramy obsługi przerwań jest to ,że nastąpi awaria systemu bezpośrednio po tym jak poprawimy wektor przerwań. Jeśli nie mamy udogodnienia zdalnej kontroli najlepszym podejściem do debuggowania tego kodu jest rozłożenie ISR na jego podstawowe elementy. Może to być kod, który po prostu przekazuje sterowanie do kolejnego ISR'a w łańcuchu przerwań (w stosownych przypadkach) Potem dodaje jedną sekcję kodu po kolei do naszego ISR'a dopóki ISR nie zawiedzie.

Oczywiście, najlepszą strategią debuggowania podprogramów obsługi przerwań jest napisanie kodu, który nie ma błędów. Jednak nie jest to praktyczne rozwiązanie, jedyną rzeczą jaką możemy zrobić to próbować zrobić to jak najbardziej jest to możliwe w ISR'ze. Im mniejszy ISR tym mniej złożony i wyższe prawdopodobieństwo, że nie będzie zawierał żadnego błędu.

Debuggowanie ISR'ów, niestety, nie jest łatwe a nie jest to coś czego nie możemy nauczyć się z książki. Potrzeba wiele doświadczeń i popełnienia wielu błędów. Niestety tak jest , ale nie ma niczego zastępczego dla doświadczenia kiedy debuggujemy ISR'y.

17.9 PODSUMOWANIE

Rozdział ten omawia trzy zjawiska występujące w systemie PC: przerwania (sprzęt), przerwania kontrolowane i wyjątki. Przerwanie jest to procedura asynchroniczna wywołująca generowanie CPU w odpowiedzi na zewnętrzne sygnał sprzętowy. Przerwanie kontrolowane jest wywołaniem programowym i jest specjalną formą procedury wywołania. Wyjątki występują kiedy program wykonuje się a instrukcja generuje jakiś rodzaj błędu. Po dodatkowe szczegóły zajrzyj

*"Przerwania, Przerwania kontrolowane i wyjątki"

Kiedy wystąpi przerwanie, przerwanie kontrolowane lub wyjątek, CPU 80x86 odkłada flagi i przekazuje sterowanie do podprogramu obsługi przerwań (ISR). 80x86 wspiera tablicę wektorów przerwań, która dostarcza adresów segmentowych dla 256 różnych przerwań. Kiedy piszemy swój własny ISR, musimy przechować adres naszego ISR'a we właściwej lokacji w tablicy wektorów przerwań do uruchomienia tego ISR'a. Dobrze wychowany program również zachowuje wartość oryginalnego wektora przerwań aby można było przywrócić ją kiedy będzie koniec. Po szczegóły zajrzyj:

*"Struktura przerwań 80x86 i podprogramy obsługi przerwań (ISR)"

Przerwanie kontrolowane lub przerwanie programowe jest niczym więcej jak wykonywanie instrukcji 80x86 „int n". Takie instrukcje przekazują sterowanie do ISR'a, którego wektor pojawia się w n-tym wejściu w tablicy wektorów przerwań. Generalnie będziemy używali przerwań kontrolowanych do wywoływania podprogramów w programach rezydentnych pojawiających się gdzieś w pamięci (podobnie jak DOS lub BIOS)

*"Przerwania kontrolowane"

Wyjątek wystąpi jeśli tylko CPU wykonuje instrukcje a instrukcja ta jest niedozwolona lub wykonanie tej instrukcji generuje jakiś rodzaj błędu (jak dzielenie przez zero). 80x86 dostarcza kilku wbudowanych wyjątków, chociaż ten tekst tylko operuje wyjątkami dostępnymi w trybie rzeczywistym.

*"Wyjątki"
*"Wyjątek błędu dzielenia (INT 0)"
*"Wyjątek pojedynczego kroku (śledzenia) (INT 1)
*"Wyjątek punktu zatrzymania (INT 3)"
*"Wyjątek przepełnienia (INT 4/INTO)"
*"Wyjątek graniczny (INT 5/BOUND")

*”Wyjątek niepoprawnego opcodu (INT 6)”
*”Koprocesor nie dostępny (INT 7)”

PC dostarcza sprzętowego wsparcia dla 15 przerwań wektorowych używając pary chipów programowalnych sterowników przerwań (PIC) 8259A. Urządzenia, które zwykle generują przerwania sprzętowe wliczając w to zegar, klawiaturę, porty szeregowe, porty równoległe, urządzenia dyskowe, karty dźwiękowe, zegar czasu rzeczywistego i FPU. 80x86 pozwala nam włączyć lub zablokować wszystkie przerwania maskowalne instrukcjami cli i sti . PIC pozwala również maskować indywidualnie urządzenia, które mogą przerywać system. Jednak, 80x86 dostarcza specjalnych przerwań niemaskowalnych, które mają wyższy priorytet niż pozostałe przerwania sprzętowe i nie może być zablokowane przez program.

*”Przerwania sprzętowe”
*”Programowalny sterownik przerwań (PIC) 8259A”
*”Przerwanie zegarowe (INT 8)”
*”Przerwanie klawiatury (INT 9)”
*”Przerwanie portu szeregowego (INT 0Bh i INT 0Ch)”
*”Przerwanie portu równoległego (INT 0Dh i INT 0Fh”)
*”Przerwanie dyskietki i dysku twardego (INT 0Eh i INT 76h)”
*”Przerwanie zegara czasu rzeczywistego (INT 70h)”
*”Przerwanie FPU (INT 70h)”
*”Przerwania niemaskowalne (INT 2)”
*”Inne przerwania”

Podprogramy obsługi przerwań które napiszemy mogą koegzystować z innymi ISR’ami w pamięci. W szczególności nie będziemy mogli po prostu zamienić wektora przerwań adresem naszego ISR’a i pozwolić działać ISR’owi stamtąd. Często, będziemy musieli stworzyć łańcuch przerwania i wywołać poprzedni ISR w łańcuchu przerwania, kiedy wykonujemy przetwarzanie przerwania. Aby zobaczyć dlaczego tworzymy łańcuch przerwania i nauczyć się jak je tworzyć zobacz

*”Podprogramy obsługi przerwań wiązań łańcuchowych”

Wraz z przerwaniami nadchodzi możliwość współbieżności, to znaczy taka możliwość, że podprogram może być przerwany i wywołany ponownie przed pierwszym końcowym wykonaniem . Rozdział ten wprowadził koncepcję współbieżności i daje kilka przykładów, które demonstrują problemy z kodem niewspółbieżnym

*”Problemy współbieżności”

Głównym celem systemu sterowanego przerwaniami jest poprawienie wydajności tego systemu. Dlatego też, nie powinno nas zaskoczyć, że ISR’y powinny być tak wydajne jak tylko to możliwe. Rozdział ten omawia dlaczego system I/O sterowany przerwaniami może być bardziej wydajny i porównuje I/O sterowane przerwaniami z odpytywaniem I/O. Jednakże przerwania mogą powodować problemy jeśli odpowiedni ISR jest zbyt wolny. Dlatego programiści którzy piszą ISR’y muszą być świadomi takich parametrów jak czas obsługi przerwania, częstotliwość przerwania i opóźnienie przerwania.

*”Sprawność systemu sterowania przerwaniami”
*”Układ We/Wy sterowany przerwaniami kontra odpytywanie”
*”Czas obsługi przerwania”
*”Opóźnienie przerwania”

Jeśli wielokrotne przerwania występują równocześnie, CPU musi zdecydować które przerwanie obsłużyć najpierw. PIC 8259 i PC używają schematu przerwań priorytetowych , który przydziela najwyższy priorytet dla zegara.80x86 zawsze przetwarza przerwanie , najpierw z najwyższym priorytetem.

*”Przerwania priorytetowe”