1. Przerwania ogólnie

1.1. Obsługa zdarzeń, odpytywanie i przerwania

Obsługa zdarzeń jest jedną z kluczowych funkcji w prawie każdym systemie czasu rzeczywistego.

Istnieją dwie metody pozyskania informacji o zdarzeniach:

1. Cykliczne odpytywanie urządzenia czy zdarzenie zaszło (*ang. polling*).
2. Wykorzystanie przerwań generowanych przez zdarzenia (*ang. interrupts*).

Istnieją dwie metody informowania o zdarzeniach:

- metoda odpytywania rejestru zdarzeń
- wyzwalane zdarzeniami przerwania.

1.2. Metoda obsługi zdarzeń poprzez odpytywanie

```
...
do {
      Czytaj rejestr urządzenia
      Sprawdź czy zaszło zdarzenie
      Czekaj(T)
} while(1)
...
```

Rys. 1-1 Ilustracja metody odpytywania

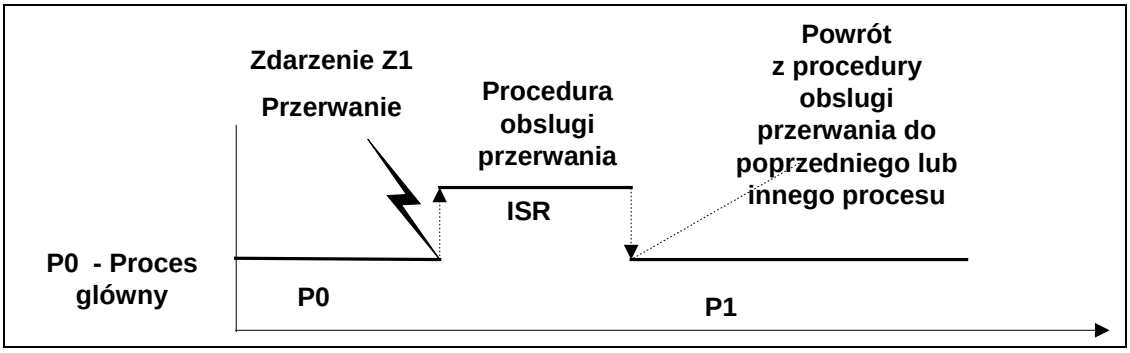
Metoda odpytywania – własności:

1. Długi czas reakcji na zdarzenie
2. Trudności w uszeregowaniu obsługi zdarzeń według priorytetów.
3. Utrata czasu procesora na wykonanie jałowych czynności
4. Prostota implementacji – nie jest wymagany specjalny sprzęt.
5. Przewidywalność – nic nie dzieje się niespodziewanie a tylko wtedy gdy nadejdzie na to zaplanowany czas.

1.3. Obsługa zdarzeń poprzez przerwania

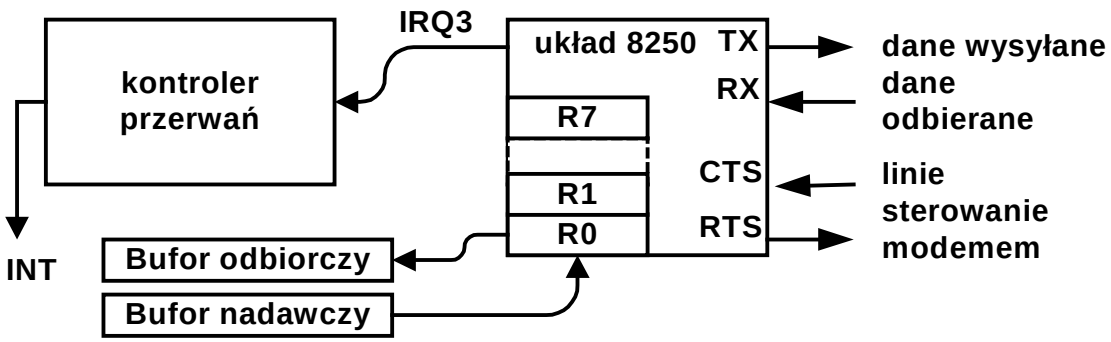
Obsługa przerwania polega na przerwaniu aktualnie wykonywanego procesu i wykonaniu procedury przypisanej danemu zdarzeniu gdy takie zdarzenie zajdzie.

Procedura nazywa się procedurą obsługi przerwania (*ang. interrupt handler*). Często używany jest też skrót ISR (*ang. Interrupt Service Routine*).



Rys. 1-2 Obsługa zdarzenia Z1 poprzez procedurę obsługi przerwania ISR

Przykładem urządzenia zgłaszającego przerwania może być układ transmisji szeregowej typu 8250.



Rys. 1-3 Układ transmisji szeregowej 8250

Może się zdarzyć że do systemu zgłosi się więcej przerwań niż może być w danym czasie obsłużone. Zachodzi wtedy potrzeba rozstrzygnięcia które zdarzenie ma obsługiwane gdy wiele z nich wystąpi naraz. Istnieją tu dwie podstawowe strategie postępowania:

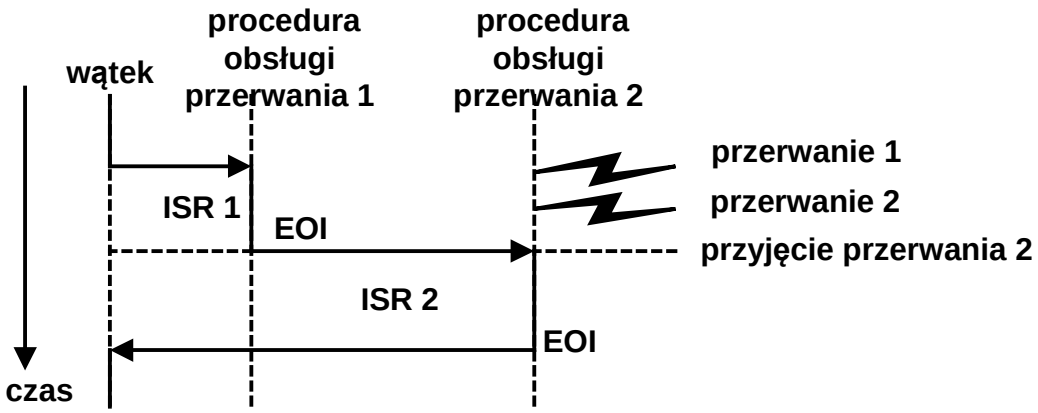
1. Jednopoziomowy system przerwań.
2. Priorytetowy system przerwań.

Gdy za obsługę oczekuje więcej nie obsłużonych przerwań wybór przerwania do obsługi może być dokonany według różnych zasad.

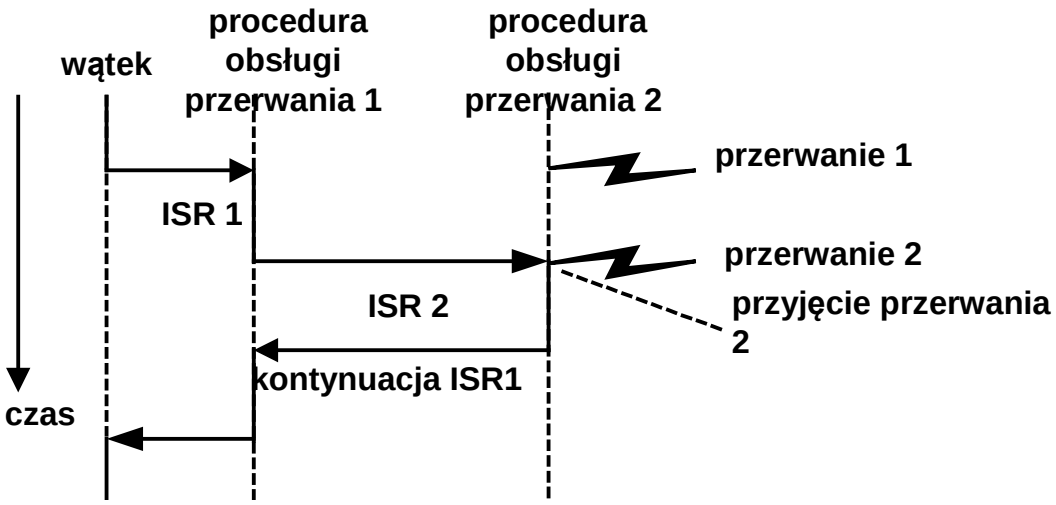
1. Poszczególnym przerwaniom mogą być przypisane priorytety.
2. Przerwania mogą być obsługiwane według kolejności zgłoszeń.

Strategia jednopoziomowa - polega wstrzymaniu się z obsługą nowego przerwania do czasu zakończenia obsługi przerwania.

Strategia wielopoziomowa - gdy w czasie obsługi przerwania o niższym priorytecie pojawi się zgłoszenie przerwania o priorytecie wyższym, to system przyjmie przerwanie o wyższym priorytecie.



Rys. 1-4 Dwa przerwania obsługiwane w systemie jednopoziomowym



Rys. 1-5 Obsługa przerwań w trybie wielopoziomowym

☞ Nie należy mylić priorytetów przerwań wynikających z działania kontrolera przerwań z priorytetami wątków.

Obsługa przerwań w systemie komputerowym jest czasami blokowana przez system operacyjny. Maksymalny czas zablokowania przerwań *Tdmax* jest podstawową miarą jakości systemu czasu rzeczywistego.

Czas Tdmax powinien być jak najkrótszy.

Obsługa zdarzeń poprzez przerwania ma dla systemu komputerowego daleko idące konsekwencje.

2. Obsługa przerwań w systemie QNX6 Neutrino

Funkcje konieczne do obsługiwania przerwań:

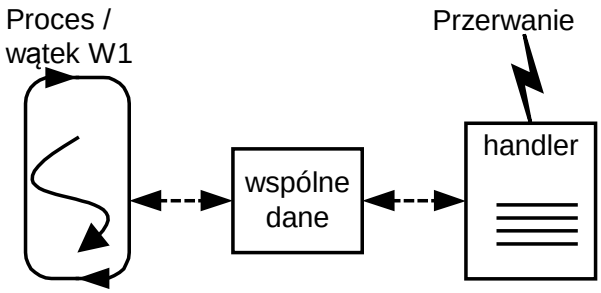
- funkcje blokowania przerwań (wszystkich lub pojedynczych) w systemie jedno i wieloprocesorowym.
- funkcja **`InterruptAttach()`** instalujące handler obsługi przerwania i funkcja **`InterruptAttachEvent()`** transformacji przerwanie w zdarzenie.
- funkcja blokującą wątek bieżący w oczekiwaniu na przerwanie.

Opis funkcji	Nazwa funkcji
Blokowanie przerwań	**`InterruptDisable()`**
Odblokowanie przerwań	**`InterruptEnable()`**
Założenie wirującej blokady przerwań	**`InterruptLock()`**
Zdjęcie wirującej blokady przerwań	**`InterruptUnlock()`**
Zamaskowanie przerwania	**`InterruptMask()`**
Odmaskowanie przerwania	**`InterruptUnmask()`**
Instalacja procedury obsługi przerwania	**`InterruptAttach()`**
Transformacja przerwania w zdarzenie	**`InterruptAttachEvent()`**
Oczekiwanie na przerwanie	**`InterruptWait()`**

Tabela 2-1 Funkcje systemowe dotyczące obsługi przerwań

1.1 Blokowanie i maskowanie przerwań

Procedury obsługi przerwania często muszą modyfikować struktury danych z których korzystają także inne jeszcze procesy lub wątki – należy zapewnić wzajemne wykluczanie.



Rys. 2-1 Wątek W1 i procedura obsługi przerwania wykorzystują wspólne dane

Blokowanie przerwań (*ang. interrupt disabling*)

Blokowanie przerwań jest to niedopuszczenie do zgłoszenia wszystkich przerwań i realizowane jest w procesorze.

Maskowanie przerwań (*ang. interrupt mask*)

Maskowanie przerwań jest to niedopuszczenie do zgłoszenia określonego przerwania i realizowane jest w kontrolerze przerwań.

`InterruptDisable` – blokowanie przerwań

```
int InterruptDisable(void)
```

Wykonanie funkcji spowoduje zablokowanie wszystkich przerwań zewnętrznych.

☞

Czas zablokowania przerwań należy ograniczyć do niezbędnego minimum.

Po zablokowaniu przerwań komputer staje „głuchy” i przestaje reagować na jakiekolwiek zewnętrzne zdarzenia z wyjątkiem zdarzeń obsługiwanych w trybie odpytywania.

Funkcję blokowania przerwań mogą wykonywać tylko procesy będące własnością administratora.

Wątek zamierzający zablokować przerwania powinien wcześniej zażądać przywileju wykonania sprzętowej operacji wejścia wyjścia poprzez wykonanie funkcji **`ThreadCtl(_NTO_TCTL_IO,0)`**

`InterruptEnable` – odblokowanie przerwań

`int InterruptEnable(void)`

☞ W systemach wieloprocesorowych do zapewnienia ochrony sekcji krytycznej w procedurach obsługi przerwań blokowanie przerwań jest niewystarczające. Należy użyć mechanizmu wirującej blokady.

`InterruptLock` – założenie blokady

`int InterruptLock(intrspin_t *spinlock)`

`spinlock -` Zmienna blokady - wspólna dla handlera i wątku

Funkcja **`InterruptLock()`** sprawdza zawartość zmiennej **`spinlock()`.**

Działanie:
Gdy wartość zmiennej **`spinlock`** wskazuje że nie jest ona zajęta, to ją zajmuje wpisując do niej odpowiednią wartość i blokuje przerwania.

Gdy zmienna **`spinlock`** jest zajęta, wykonywane jest ponowne sprawdzenie wartości tej zmiennej. Sprawdzanie odbywa się tak długo aż zmienna **`spinlock`** nie zostanie zwolniona.

Sprawdzenie i ustawienie zmiennej odbywa się jako nieprzerywalna operacja atomowa.

InterruptUnlock – zdjęcie blokady

int InterruptUnlock(intrspin_t *spinlock)

spinlock - zmienna blokady - wspólna dla handlera i wątku

Funkcja **InterruptUnlock()** powoduje zdjęcie blokady i odblokowanie przerwań.

```
intrspin_t zm_blok;
....
InterruptLock(&zm_blok);
/* Sekcja krytyczna   */
...
InterruptUnlock(&zm_blok);
```

Indywidualne przerwania można blokować używając rejestru maski w kontrolerze przerwań. Jedno określone przerwanie zablokowane może być przy użyciu funkcji:

InterruptMask – zamaskowanie przerwania

int InterruptMask(int intr, int id)

intr - Numer przerwania które ma być zamaskowane
id - Identyfikator handlera zwracany przez funkcje **InterruptAttach()**, **InterruptAttachEvent()** lub –1 gdy przerwanie ma być zamaskowane dla wszystkich handlerów.

Wykonanie funkcji powoduje zamaskowanie przerwania sprzętowego podanego jako pierwszy parametr, dla handlera o identyfikatorze podanym jako drugi parametr.

Zamaskowane przerwanie można dozwolić (odmaskować) używając funkcji:

`InterruptUnmask` – dozwolenie przerwania

```
int InterruptUnmask(int intr, int id)
```

`intr` - **n**umer przerwania które ma być dozwolone
`id` - **i**dentyfikator handlera zwracany przez funkcje **`InterruptAttach`**, **`InterruptAttachEvent`** lub –1 gdy przerwanie ma być dozwolone dla wszystkich handlerów.

1.2 Instalacja procedur obsługi przerwań.

Typowe czynności realizowane poprzez procedurę obsługi przerwania:

1. Stwierdzenie które z urządzeń wymaga obsługi (gdy więcej urządzeń dzieli jedno przerwanie).
2. Obsługa urządzenia – zwykle sprowadza się ona do odczytu i zapisu pewnych rejestrów urządzenia.
3. Aktualizacja wspólnych struktur danych (dostępnych także dla wątków aplikacji).
4. Zasygnalizowanie aplikacji wystąpienia zdarzenia.

Instalacji handlera obsługi przerwania następuje poprzez wykonanie funkcji: **`InterruptAttach –`** instalacja obsługi przerwania

```
int InterruptAttach(int itnr,
const struct sigevent *(* handler)(void*, int),
const void *area, int size, unsigned flags)
```

Gdzie:

`intr`	Numer przerwania
`handler`	Wskaźnik na procedurę obsługi przerwania (handler)
`area`	Adres obszaru komunikacji handlera z programem
`size`	Wielkość obszaru komunikacji handlera z programem
`flags`	Flagi

Wykonanie funkcji spowoduje zainstalowanie funkcji **`handler()`** określonej jako drugi parametr, do obsługi przerwania o numerze **`intr`**, podanej jako pierwszy parametr funkcji.

Parametr trzeci i czwarty dotyczą obszaru komunikacyjnego pomiędzy wątkiem a handlerem.

Przed wykonaniem funkcji należy zażądać prawa wykonania operacji wejścia wyjścia poprzez wykonanie funkcji:

`ThreadCtl(_NTO_TCTL_IO,0)`

Działanie procedury obsługi przerwań jest modyfikowane przez flagi:

- **`_NTO_INTR_FLAGS_END -`** nowy handler dopisany będzie na końcu łańcucha i wykona się jako ostatni.
- **`_NTO_INTR_FLAGS_PROCESS`** system kojarzy handler z procesem a nie z wątkiem. Handler będzie deinstalowany gdy kończy się proces a nie wątek.

Procedura obsługi przerwania
Handler jest funkcją o następującym prototypie:

`const struct sigevent* handler(void* area, int id)`

Handler może zwracać albo stałą NULL albo wskaźnik do prawidłowo zadeklarowanej i zainicjowanej struktury typu **`sigvent`**.

1. Gdy procedura obsługi przerwania zwraca NULL to nie powoduje to dalszych działań.
2. Gdy procedura obsługi przerwania zwraca wskaźnik do struktury typu **`sigvent`** to generowane jest zdarzenie wyspecyfikowane w strukturze **`sigevent`**.

Zapoczątkowana może być jedna z trzech akcji:

- Zdarzenie powodujące odblokowanie wątku
- Wygenerowanie impulsu
- Wygenerowanie sygnału

Gdy handler przerwania generuje zdarzenie, to zdarzenie to powinno prowadzić do odblokowania pewnego procesu lub wątku. Żądany typ zdarzenia zależy od zainicjowania struktury **`sigevent`** .

1. Gdy przerwanie ma odblokować wątek zawieszony na funkcji **`InterruptWait()`** to zdarzenie powinno być typu **`SIGEV_INTR`**.
2. Gdy przerwanie ma odblokować wątek zablokowany na funkcji **`MsgReceive()`** to zdarzenie powinno być typu **`SIGEV_PULSE`**. W tym przypadku wątek może odbierać także komunikaty.
3. Możliwe jest także zainicjowanie struktury **`sigevent`** na zdarzenia typu SIGEV_SIGNAL, SIGEV_SIGNAL_CODE, SIGEV_SIGNAL_THREAD, SIGEV_THREAD. Ze względu na mniejszą efektywność nie jest to zalecane.

Przy tworzeniu procedur obsługi przerwań należy przestrzegać zasad:

1. Rozmiar stosu którym dysponuje procedura obsługi przerwania jest ograniczony. Stąd nie powinna ona zawierać dużych tablic czy innych struktur danych . Bezpiecznie jest przyjąć że dostępny rozmiar stosu wynosi około 200 bajtów.
2. Procedura obsługi przerwań wykonywana jest asynchronicznie z wątkami należącymi do pewnego procesu i używa wspólnych z nimi danych. Wszystkie zmienne modyfikowane przez handler powinny być poprzedzone słowem kluczowym **`volatile (`**aby kompilator nie umieszczał ich w rejestrach**`)`**, a ich modyfikacja wewnątrz wątków zabezpieczona przez zablokowanie przerwań lub spinlock.
3. Procedura obsługi przerwania wykonywana jest poza normalnym szeregowaniem więc powinna być tak krótka jak to tyko możliwe. Jeżeli wymagane jest wykonanie czasochłonnych czynności to powinny być one wykonane w procesie lub wątku który zostanie prze handler odblokowany.
4. Procedura obsługi przerwania nie może wywoływać żadnych funkcji systemowych z wyjątkiem niektórych wyraźnie dozwolonych funkcji.

Oczekiwanie na przerwanie
Kończący się handler może wygenerować zdarzenie **SIGEV_INTR**. Zdarzenie to może odblokować oczekujący na przerwanie wątek. Funkcją która blokuje wątek w oczekiwaniu na przerwanie jest funkcja **InterruptWait().**

InterruptWait – oczekiwanie na przerwanie

int InterruptWait(int flags, iunt_64 *timeout)

flags - Flagi – należy przyjąć 0
timeout - Maksymalny okres oczekiwania – obecnie należy przyjąć NULL

Funkcja powoduje zablokowanie wątku bieżącego w oczekiwaniu na przerwanie. Funkcja zwraca –1 gdy błąd.

InterruptDetach – deinstalacja handlera przerwania

int InterruptDetach(int id)

id - Identyfikator zwracany przez funkcję **InterruptAttach() i InterruptAttachEvent()**

```
// Obsluga przerwania zegarowego
#include <sys/neutrino.h>
#define TIME_INT 0
struct sigevent event;
int icnt = 0;
int intid = 0;
int sec = 0;

const struct sigevent *handler(void *arg, int id) {
  icnt++;
  if(icnt == 1000) {
    icnt = 0;
    intid = id;
    sec++;
    return(&event);
  } else return(NULL);
}

main() {
   int res,i,sec = 0;
   i = 0;
   printf("Program startuje \n");
   event.sigev_notify = SIGEV_INTR;
   ThreadCtl( _NTO_TCTL_IO, 0 );
   res = InterruptAttach(TIME_INT,
         &handler,NULL,0,0);
   printf("Handler zaininstalowany: %d \n",res);
   do {
     InterruptWait(0,NULL);
     printf("Przerwanie: %d %d %d\n",sec,i,intid);
     i++;
   } while(sec<60);
   InterruptDetach(res);
}
```

Przykład 2-1 Obsługa przerwania zegarowego

1.3 Transformacja przerwania w zdarzenie

Czynności wykonywane w ramach ISR nie podlegają szeregowaniu. W kodzie handlera wykonać można tylko niezbędny zakres czynności a następnie powiadomić pewien wątek o zaistnieniu przerwania a wątek ten wykona resztę pracy.

☞ W kodzie procedury obsługi przerwania wykonać należy tylko niezbędne czynności a następnie powiadomić pewien wątek o wystąpieniu przerwania. Wątek ten wykona resztę pracy.

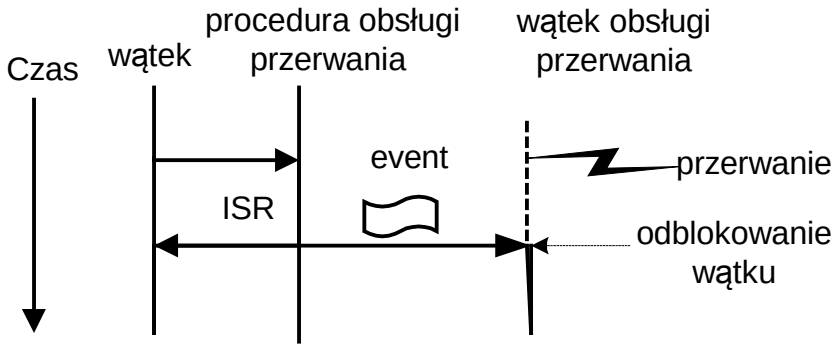
Obsługa zdarzeń w QNX6 Neutrino:

1. Wszystkie czynności wykonywane są przez procedurę obsługi przerwania.
2. Wewnątrz procedury obsługi przerwania wykonane będą najważniejsze czynności a resztę pracy wykona odblokowany specjalnie wątek.
3. Wewnątrz procedurę obsługi przerwania nie są wykonywane żadne czynności a jedyną jego funkcją jest odblokowanie pewnego wątku.

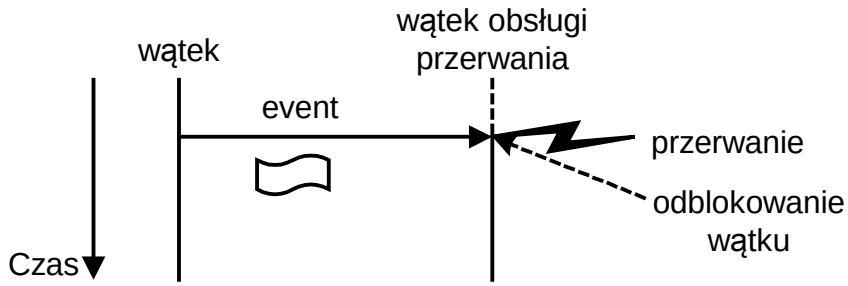
W pierwszym przypadku programista pisze procedurę obsługi przerwania i kojarzy ją z przerwaniem wykorzystując funkcję **`InterruptAttach().`**

W drugim przypadku postępowanie jest analogiczne jak w przypadku pierwszym z ta różnicą że na zakończenie handler zwraca zainicjowane odpowiednio zdarzenie w postaci struktury typu **`sigevent`** . Gdy handler zwróci takie zdarzenie to skojarzony z nim wątek będzie odblokowany.

W trzecim przypadku nie ma potrzeby w ogóle potrzeby pisania handlera. W funkcji **`InterruptAttachEvent()`** specyfikuje się numer przerwania i rodzaj zdarzenia na które to przerwanie ma być transformowane.



Rys. 2-2 Procedura obsługi przerwania wykonuje część pracy a następnie odblokowuje wątek



Rys. 2-3 Przerwanie zamieniane w zdarzenie które **`event`** odblokowuje wątek.

`InterruptAttachEvent` – transformacja przerwania w zdarzenie

```
int InterruptAttachEvent(int itnr,const struct
sigevent *event,unsigned flags)
```

Gdzie:

`intr` **-** Numer przerwania

`event` **-** Wskaźnik na strukturę opisu zdarzenia które ma być wygenerowane gdy nadejdzie przerwanie

`Flags` - Flagi

```
#include <sys/neutrino.h>
#define TIME_INT 0
struct sigevent event;

main() {
   int res, i = 0;
   i = 0;
   printf("Program startuje \n");
   ThreadCtl(_NTO_TCTL_IO,0);
   SIGEV_INTR_INIT(&event);
   res = InterruptAttachEvent(TIME_INT, &event,
                              _NTO_INTR_FLAGS_END);
   if(res < 0) {
         perror("install"); exit(0);
   }
   printf("Handler zaininstalowany: %d \n",res);
   InterruptUnmask (TIME_INT, res);
   do {
     printf("Czekam\n");
     InterruptWait(0,NULL);
     printf("Przerwanie: %d \n",i);
     InterruptUnmask (TIME_INT, res);
     i++;
  } while(i <10);
  InterruptDetach(res);
  printf("Koniec\n");
}
```

Przykład 2-2 Obsługa przerwania zegarowego za pomocą zdarzenia

3. Obsługa przetwornika AD/DA karty PCM 3718 w trybie przerwań

Obsługa przetwornika AD w trybie odpytywania posiada wady:

- odpytywanie statusu przetwornika powoduje utratę czasu procesora.
- trudno jest uzyskać precyzyjnie określony moment wyzwolenia przetwornika.

Jeżeli chcemy odczytywać wartości z przetwornika AD w ściśle określonych momentach czasu do wyzwalania przetwornika należy użyć liczników układu 8254 które generują impulsy wyzwalające konwersję. Zakończenie konwersji sygnalizowane jest przerwaniem.

Programowanie karty:

- Ustalenie trybu sygnalizowania końca konwersji i wyzwalania
- Ustalenie współczynnika podziału liczników

Ustawienie rejestrów sterujących:

	B7	B6	B5	B4	B3	B2	B1	B0
BASE+9	INTE	I2	I1	I0	-	DMAE	ST1	ST0
	1	1	0	1	-	0	1	1

Tabela 3-1 Zawartość rejestru sterującego karty PCM-3718 w trybie przerwań

INTE = 1 zakończenie konwersji sygnalizowane przerwaniem
Bity B6-B4 numer przerwania
Bity B1 i B0 wyzwalanie konwersji z liczników układu 8254

	B7	B6	B5	B4	B3	B2	B1	B0
BASE+10	-	-	-	-	-	-	TC1	TC0

Tabela 3-2 Rejestr TIMR konfiguracji liczników

TC0=0 układ wyzwalania jest stale włączony
TC0=1 włączony jest wtedy, gdy wejście **TRIG0** ma poziom wysoki.

TC1=0 licznik 0 zlicza impulsy podawane z zewnętrznego źródła
TC1=1 to podłączony jest do wewnętrznego źródła 1000 KHz
Programowanie liczników układu 8254:

BASE+12	Licznik 0 (odczyt/zapis)
BASE+13	Licznik 1 (odczyt/zapis)
BASE+14	Licznik 2 (odczyt/zapis)
BASE+15	Słowo sterujące

Tabela 3-3 Rejestry układu licznikowego 8254

Proces główny:

- Inicjuje kartę poprzez wykonanie funkcji **`card_init()`,**
- Ustawia stopień podziału liczników funkcją **`pcl_counter(20,10),`**
- Ustawia zakres przemiatanych kanałów - funkcja **`pcl_mux(0,0`**). Inicjuje zdarzenie **`event`**
- Wykonuje funkcję **`InterruptAttach(ADC_INT,handler,NULL,0,0`**). Funkcja ta instaluje handler przerwania **`ADC_INT`**
- Odmaskowuje przerwania **`InterruptUnmask(ADC_INT,id);`**

Obsługa przerwania – funkcja **`handler`**:

1. Odczyt młodszego i starszego bajtu wyniku z rejestrów **BASE** i **BASE+1**. Złożenie wartości razem.
2. Wpis uzyskane z przetwornika AD wartości do bufora cyklicznego.
3. Przekazanie odczytanej wartości na przetwornik DA **`pisz_da(val)`**
4. Wyświetlenie wartości na linijce diodowej **`led_disp(val);`**
5. Skasowanie przerwania przez zapis do rejestru **BASE+8.**

Program odczytuje wartość analogową z przetwornika AD i przekazuje ją na przetwornik DA. Jest to podstawa do realizacji algorytmów DSP.

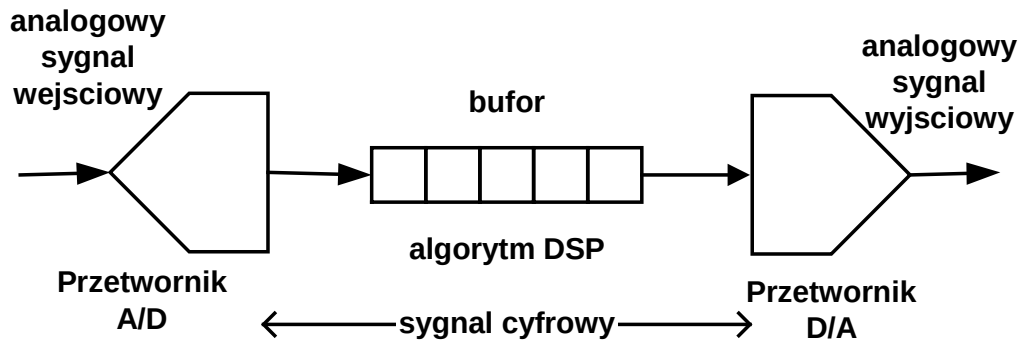
```
// System QNX Neutrino ------------------------------------
// Karta PCM 3718  -  przetwornik  AD - tryb przerwan
// (C) Jedrzej Ulasiewicz 2010
// Przerwanie AD - 5 Odblokowac w BIOS plyty gdy zablokowane
// Przetwornik DA na plycie PCM3718HO, pin 19 złącza P1 analog
#include <sys/neutrino.h>
#include <hw/inout.h>
#include <sys/mman.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#define ADRB 0x300 // Adres bazowy karty
#define ADL    0     // Mlodszy bajt AD + kanal
#define ADH    1     // Starszy
#define RANGE  1     // Wzmocnienie kanalu
#define MUXR   2     // Kanal konc i pocz
#define DALO   4     // Mlodszy bajt DA
#define DAHI   5     // Starszy  bajt DA
#define STATR  8     // Rejestr statusu
#define CONTR  9     // Rejestr sterujacy
#define TIMR  10     // Start timera
#define DIOH  11     // Wyjscia cyfrowe
#define COUNT0 12    // Licznik 0
#define COUNT1 13    // Licznik 1
#define COUNT2 14    // Licznik 3
#define COUNTC 15    // Rej. ster. licznikow

// Adres bazowy karty
static int base = ADRB;

#define BSIZE 1000
#define ADC_INT 5  // Przerwanie karty

static short int buf[BSIZE];
volatile int head,tail,count, id, cnt = 0;
// struct sigevent event;
uintptr_t port;

void card_init(void ) {
// Inicjalizacja karty
// from - kanal poczatkowy, to kanal koncowy
   unsigned char val1,val2 ;
   // INT, IRQ5, DMAE = 0, wyzw z 8 MUXR     54
   val1 = 0xD3;
   out8(base +  CONTR    ,  val1);
   val2  =  in8(base +  CONTR    );
   if(val2 != val1) {
     printf("Blad inicjalizacji karty\n");
```

```
    exit(0);
  }
  printf("Status:%x  Control: %x \n",val1,val2);
  // Pacer enable
  out8(base +  TIMR      , 0x01);
  // INTE I2 I1 I0 X DMA ST1 ST0
  //   1    1   0 1 0  0      1     1
  out8(base +  CONTR,  0xD3);
}


void set_range(int from, int to, unsigned char zakres)
// Ustawienie wzmocnienia kanalow karty
// from - kanal poczatkowy, to kanal koncowy
// zakr pom  0-10 V -> 4, 0-5V -> 5, 0-2.5 -> 6, 0-1.25 -> 7
{
   int i;
   for(i = from; i<= to; i++) {
     out8(base + MUXR,  i);
     out8(base + RANGE,zakres);
  }
}

void pcl_counter(int l1, int l2) {
   // Programowanie  licznikow
   //  licznik 1
   out8(base +  COUNTC , 0x74) ;
   out8(base + COUNT1  , l1 & 0xFF);
   out8(base + COUNT1  , l1 >> 8);
    //  licznik 2
   out8(base +  COUNTC , 0xB4) ;
   out8(base + COUNT2  , l2 & 0xFF);
   out8(base + COUNT2  , l2 >> 8);
}

void pcl_mux(int first, int last) {
 out8(base + MUXR,  (last << 4) | (first & 0x0F));
}
```

```
void pisz_da(unsigned short int x)
// Zapis kanalu DA
{ unsigned short int yh,yl;
   unsigned short int y;
   y = x;
   y = y & 0x0FFF;
   yl  = (y & 0x0F);
   yl = yl << 4;
   yh = y >> 4;
   out8(base + DALO,yl);
   out8(base + DAHI, yh);
}

void led_disp(unsigned short int val)
{ unsigned short int y;
  y = val >> 9;
  out8(base + DIOH,0xFF>>(7-y));
}

// struct sigevent event;
volatile int icnt = 0;
int intid = 0;
int sec = 0;

const struct sigevent *handler(void *arg, int id)
// Handler obslugi przerwania
{   unsigned short int chn, val,ah,al;
    unsigned int xh,xl;
    // Odczyt wart. pomiarowych
    al =  in8(base + ADL);
    ah =  in8(base + ADH);
    chn = al & 0x0F;
    xh  = ah << 4;
    xl  = al  >> 4;
    val = xh + xl;
    icnt++;
   // Zapis do bufora cykliczneho
    buf[head] = val;
    head = (head+1) %BSIZE;
    cnt++;
    // Wyswietlenie na linijce diodowej
    led_disp(val);
    pisz_da(val);
    // Skasowanie przerwania
    out8(base  +  STATR      ,  0x00);
    return(NULL);
}
```

```
int main() {
   int id;
   time_t t1,t2;
   printf("Start watku odczyt\n");
   ThreadCtl( _NTO_TCTL_IO, 0 );
   port = mmap_device_io(16,base);
   printf("port %x\n",port);
   // Inicjacja trybu pracy karty ------------
   card_init();
   // Ustawienie zakresu kanalow --------
   pcl_mux(0,0);
   // Ustawienie zakresu pomiarowego --
   set_range(0,0,5);
   // Ustawienie czestotliwosci licznikow-
   pcl_counter(10,100);
   id = InterruptAttach(ADC_INT,handler,NULL,0,0);
   if(id < 0) {
         perror("install"); exit(0);
   }
   printf("Handler zaininstalowany: %d \n",id);
   t1 = time(NULL);
   // Skasowanie przerwania
   out8(base + STATR,0x00);
   // Odmaskowanie przerwania
   InterruptUnmask(ADC_INT,id);
   do {
       printf("icnt: %d cnt: %d\n",icnt,cnt);
      sleep(1);
   } while(icnt < 10000);
   InterruptMask(ADC_INT,id);
   InterruptDetach(id);
   t2 = time(NULL); t2 = t2 -t1;
   printf("pomiarow %d czas %d sek \n",icnt,t2);
   printf("Koniec\n");
   return 0;
}
```

Przykład 3-1 Obsługa przetwornika AD w trybie przerwań



Rys. 3-1 Zasada cyfrowego przetwarzania sygnałów DSP

Drugi program demonstruje podejście z wykorzystaniem transformacji przerwania w zdarzenie.

Proces główny:

- Inicjuje kartę poprzez wykonanie funkcji **`card_init(),`**
- Ustawia stopień podziału liczników funkcją **`pcl_counter(20,10),`**
- Ustawia zakres przemiatanych kanałów - funkcja **`pcl_mux(0,0`**). Inicjuje zdarzenie **`event`**
- Wykonuje funkcję **`InterruptAttachEvent(ADC_INT,&event,0`**). Funkcja ta transformuje przerwanie **`ADC_INT`** w zdarzenie **`event`**.

Obsługa przerwania:

1. Skasowanie przerwania przez zapis do rejestru **BASE+8.**
2. Odczyt młodszego i starszego bajtu wyniku z rejestrów **BASE** i **BASE+1**. Złożenie wartości razem.
3. Wpis uzyskane z przetwornika AD wartości do bufora cyklicznego.
4. Przekazanie odczytanej wartości na przetwornik DA **`pisz_da(val)`**
5. Wyświetlenie wartości na linijce diodowej **`led_disp(val);`**
6. Odmaskowanie przerwania **`ADC_INT`**.
7. Oczekiwanie na kolejne przerwanie.

```
// System QNX Neutrino ---------------------------------------
// Karta PCM 3718  -  przetwornik  AD - tryb przerwan
// (C) Jedrzej Ulasiewicz 2010
// Przerwanie AD - 5 Odblokowac w BIOS plyty gdy zablokowane
// Przetwornik DA na plycie PCM3718HO, pin 19 złącza P1 analog
#include <sys/neutrino.h>
#include <hw/inout.h>
#include <sys/mman.h>
#include <pthread.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#define ADRB 0x300 // Adres bazowy karty
#define ADL    0   // Mlodszy bajt AD + kanal
#define ADH    1   // Starszy bajt
#define RANGE  1   // Wzmocnienie kanalu
#define MUXR   2   // Kanal konc i pocz
#define DALO   4   // Mlodszy bajt DA
#define DAHI   5   // Starszy  bajt DA
#define STATR  8   // Rejestr statusu
#define CONTR  9   // Rejestr sterujacy
#define TIMR  10   // Start timera
#define DIOH  11   // Wyjscia cyfrowe
#define COUNT0 12  // Licznik 0
#define COUNT1 13  // Licznik 1
#define COUNT2 14  // Licznik 3
#define COUNTC 15  // Rej. ster. licznikow
#define ADC_INT 5  // Przerwanie karty PCM3718
#define BSIZE 1000

static int base = ADRB; // Adres bazowy karty

static short int buf[BSIZE];
int head,tail,count, id, cnt = 0;
struct sigevent event;
uintptr_t port;
```

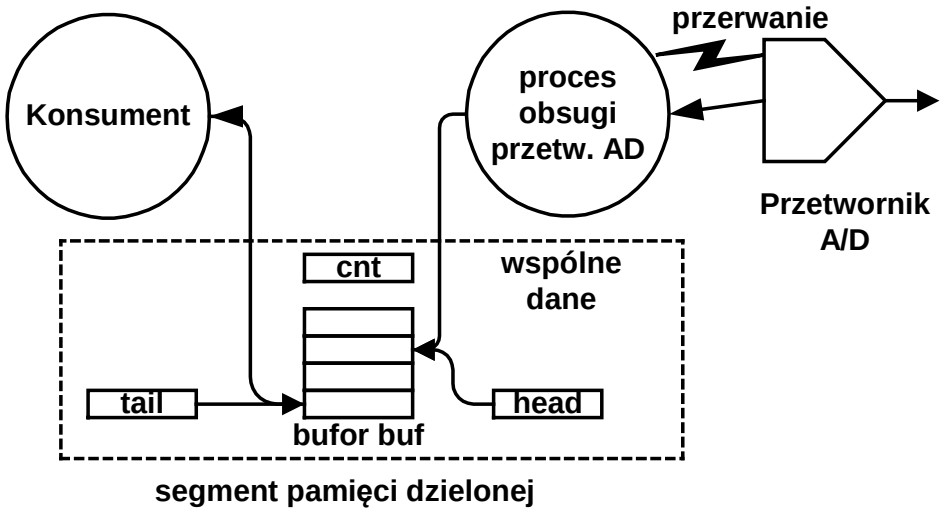
```
int card_init(void ) {
// Inicjalizacja karty
// from - kanal poczatkowy, to kanal koncowy
   unsigned char val1,val2 ;
   // INT, IRQ5, DMAE = 0, wyzw z 8 MUXR     54
   val1 = 0xD3;
   out8(base +  CONTR    ,  val1);
   val2  =  in8(base +  CONTR    );
   if(val2 != val1) {
     printf("Blad inicjalizacji karty\n");
     exit(0);
   }
   printf("Status:%x  Control: %x \n",val1,val2);
   // Pacer enable
   out8(base +  TIMR      , 0x01);
   // INTE I2 I1 I0 X DMA ST1 ST0
   //   1    1   0 1 0  0      1     1
   out8(base +  CONTR,  0xD3);
 }

void set_range(int from, int to, unsigned char zakres)
 // Ustawienie wzmocnienia kanalow karty
 // from - kanal poczatkowy, to kanal koncowy
 // zakr. pom  0-10 V -> 4, 0-5V -> 5, 0-2.5 -> 6, 0-1.25 -> 7
 {
    int i;
    for(i = from; i<= to; i++) {
      out8(base + MUXR,  i);
      out8(base + RANGE,zakres);
   }
 }

 int pcl_counter(int l1, int l2)
 // Programowanie  licznikow
 // Czestotliwosc przerwan = 1000000 /(l1 * l2) Hz
{
    //  licznik 1
    out8(base + COUNTC , 0x74) ;
    out8(base + COUNT1  , l1 & 0xFF);
    out8(base + COUNT1  , l1 >> 8);
    //  licznik 2
    out8(base + COUNTC , 0xB4) ;
    out8(base + COUNT2  , l2 & 0xFF);
    out8(base + COUNT2  , l2 >> 8);
}

int pcl_mux(int first, int last)
// Kanal poczatkowy I koncowy
```

```
{
 out8(base + MUXR,  (last << 4) | (first & 0x0F));
}


void pisz_da(unsigned short int x)
// Zapis kanalu DA
{ unsigned short int yh,yl;
   unsigned short int y;
   y = x;
   y = y & 0x0FFF;
   yl  = (y & 0x0F);
   yl = yl << 4;
   yh = y >> 4;
   out8(base + DALO,yl);
   out8(base + DAHI, yh);
}

led_disp(unsigned short int val)
// Wyswietlanie linijka LED
{ unsigned short int y;
  y = val >> 9;
  out8(base + DIOH,0xFF>>(7-y));
}

main() {
   int id, tid,i = 0;
   unsigned short int x, chn, val,ah,al,y;
   unsigned int xh,xl;
   printf("Start odczyt\n");
   ThreadCtl( _NTO_TCTL_IO, 0 );
   port = mmap_device_io(16,base);
   printf("port %x\n",port);
   // Inicjacja trybu pracy karty ------------
   card_init();
   // Ustawienie zakresu kanalow --------
   pcl_mux(0,0);
   // Ustawienie zakresu pomiarowego --
   set_range(0,0,5);
   // Ustawienie czestotliwosci licznikow-
   pcl_counter(200,100);
   SIGEV_INTR_INIT(&event);
   id = InterruptAttachEvent(ADC_INT, &event,0);
   if(id < 0) {
         perror("install");
         exit(0);
   }
   printf("Handler zaininstalowany: %d \n",id);
```

```
  do {
    // Skasowanie przerwania
    out8(base + STATR,  0x00);
    // Odczyt wart. pomiarowych
    al =  in8(base + ADL);
    ah =  in8(base + ADH);
    chn = al & 0x0F;
    xh = ah << 4;
    xl  = al  >> 4;
    val = xh + xl;
    printf("  cnt: %3d kan: %d val: %d  \n",cnt, chn,val);
    // Zapis do przetwornika DA
    pisz_da(val);
    // Zapis do bufora cykliczneho
    buf[head] = val;
    head = (head+1) %BSIZE;
    cnt++;

    // Wyswietlenie na linijce diodowej
    led_disp(val);
    InterruptUnmask(ADC_INT,id);
    // Oczekiwanie na przerwanie ----
    InterruptWait(NULL,NULL);
  } while(cnt < 10000);

}
```

Przykład 3-2 Obsługa przetwornika AD karty PCM-3718 w trybie transformacji przerwań w zdarzenie

Program można ulepszyć składując odczytane dane w pamięci dzielonej w buforze cyklicznym



Rys. 3-1 Aplikacja obsługi przetwornika AD w trybie przerwań, wyniki w pamięci dzielonej