

Architektura Systemów Wbudowanych

Laboratorium 9 i 10

Juny 2015

Driver interfejsu szeregowego dla systemu QNX Neutrino

Imię	Nazwisko	Nr indeksu

1.Wstęp

Co to jest manager zasobów?

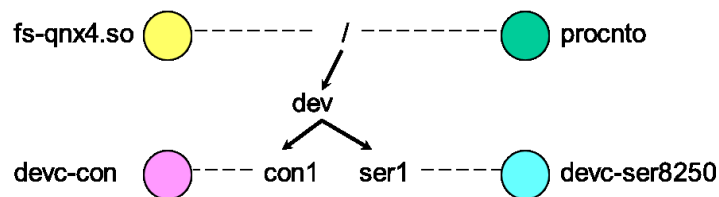
- program rozszerzający system operacyjny poprzez:
 - tworzenie i zarządzanie nazwami w przestrzeni nazw
 - dostarczenie klientom interfejsu zgodnego ze standardem POSIX (np. *open()*, *read()*, *write()*, ...)
- może być połączony ze sprzętem hardware (takim jak port szeregowy albo sterownik dysku, itp.)
- albo może być po prostu encją oprogramowania (jak */dev/mqueue*)

Mapowanie nazw

Dla przykładu, rozwinięcie ścieżki:

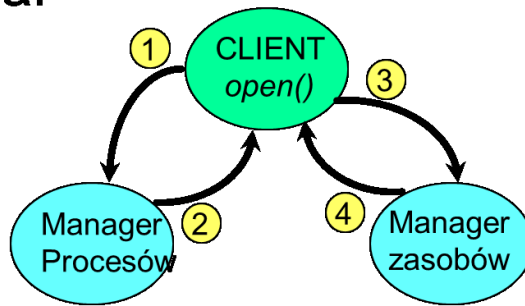
/dev/ser1

```
fd = open("/dev/ser1", ...);
```



QNX – Writing a Resource Manager QNX Software Systems LTD.

Powiązania:

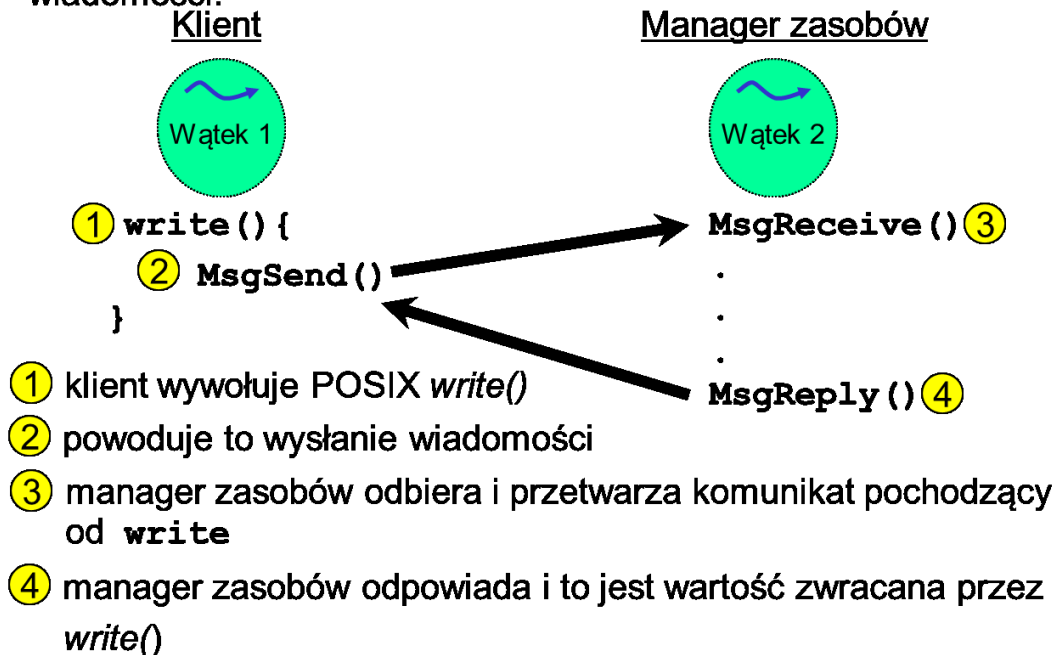


- ① `open()` wysyła wiadomość "zapytanie"
- ② Manager procesów odpowiada kto jest odpowiedzialny (`pid, chid, handle`)
- ③ `open()` powoduje nawiązanie połączenia ze specyficznym managerem (`pid, chid`)
- ④ Manager zasobów odpowiada statusem (`pass/fail`)

Wszystkie kolejne informacje idą bezpośrednio do managera zasobów

QNX – Writing a Resource Manager QNX Software Systems LTD.

Manager zasobów jest zbudowane w oparciu o przekazywanie wiadomości:

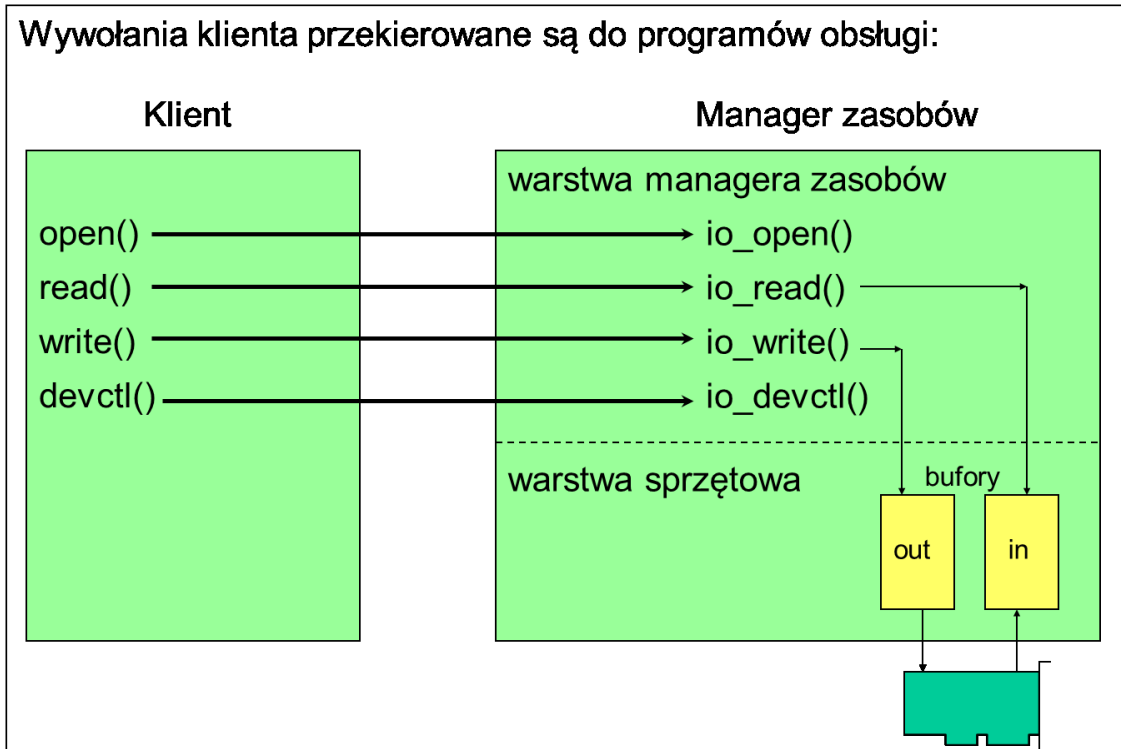


QNX – Writing a Resource Manager QNX Software Systems LTD.

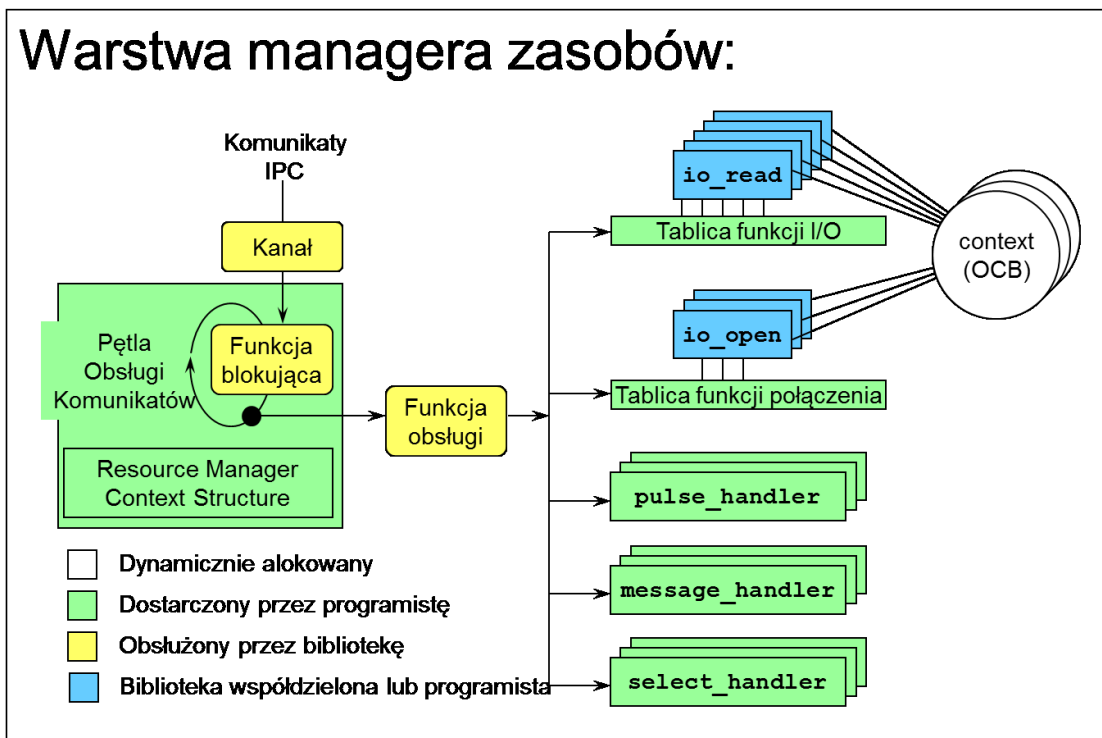
Manager zasobów wykonuje następujące kroki:

- tworzy kanał,
- przejmuje część przestrzeni nazw,
- oczekuje na komunikaty i zdarzenia,
- przetwarza komunikaty i zwraca wyniki.

Spojrzenie na ogólną koncepcję menedżera zasobów.






QNX – Writing a Resource Manager QNX Software Systems LTD. 2005




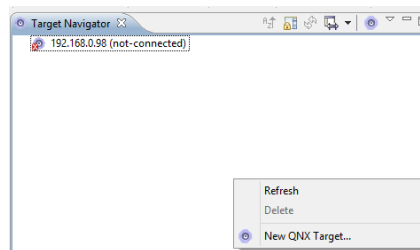
QNX – Writing a Resource Manager QNX Software Systems LTD. 2005

2.Przygotowanie do wykonania zadań.

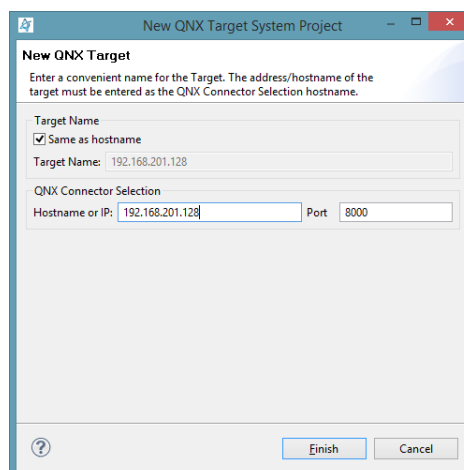
Uruchom środowisko **QNX Momentics IDE** ze wskazaną przez prowadzącego przestrzenią roboczą. Zapoznaj się z podstawowymi widokami środowiska. Sprawdź dostępne perspektywy. Przydatne perspektywy to:

- ❖ C/C++ perspective  ;
- ❖ Debug perspective  ;
- ❖ QNX System Information perspective  ;

Przejdź do perspektywy „QNX System Information”  - klucz na pasku narzędzi w górnej prawej części ekranu. W oknie „Target Navigator” ustaw połączenie z platformą docelową. Wciskając prawy klawisz myszy i wybierając opcję „New QNX Target”.



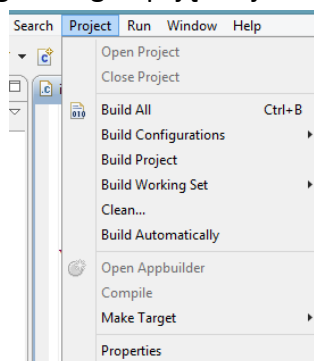
Pojawi się okno:



W polu Hostname or IP wprowadź adres IP platformy docelowej. Upewnij się że na platformie docelowej uruchomiony jest proces **qconn**. Jeśli nie wykonaj polecenie: **qconn #**.

Wróć do perspektywy C/C++  . Otwórz wskazany plik źródłowy.

Kompilacja projektu wybierz z menu głównego opcję Project->Build Project.



Przejdź do realizacji zadań.

3.Zadania do zrealizowania

Zad.3.1. Opracowanie API dla zasobu sprzętowego.

W środowisku w pliku źródłowym uzupełnij kod odpowiedzialny za:

- ❖ Inicjowanie portu szeregowego wyznaczonego przez prowadzącego;
- ❖ Wysyłanie danych łączem szeregowym;
- ❖ Odbieranie danych łączem szeregowym;

Uwaga:

Nadawanie i odbiór powinny być zorganizowane w oparciu o przerwania. Załącznik 3.

Materiały pomocnicze w **załączniku 1**.

Po napisaniu kodu sprawdź działanie opracowanych funkcji prostym programem – przesyłając i odbierając dane z inną grupą laboratoryjną.

Zad.3.2. Opracowanie menedżera zasobu.

Bazując na plikach źródłowych z zadania 3.1 uzupełnij kod, który zamieni program do sprawdzenia komunikacji po łączu szeregowym w menedżera zasobu.

Etapy postępowania:

1. Przygotowanie prototypów funkcji, które będą pełniły rolę funkcji POSIX `open()`, `write()`, `read()`.
2. Zadeklarowanie zmiennych odpowiedzialnych za:
 - ❖ strukturę 'dispatch';
 - ❖ listę komunikatów połączenia;
 - ❖ listę komunikatów I/O;
 - ❖ atrybuty urządzenia;
 - ❖ atrybuty menedżera zasobów;
 - ❖ dispatch context.
3. Alokacja zasobów i inicjalizacja struktury „dispatch”.
4. Inicjowanie funkcji połączenia i tablicy funkcji wejścia-wyjścia.
5. Podmiana standardowych funkcji handlera na własne funkcje.
6. Inicjowanie struktury opisującej urządzenie.
7. Powiadomienie menedżera procesów o sposobie połączenia i podpięcie funkcji wejścia-wyjścia.
8. Alokacja struktury kontekstowej menedżera.
9. Pętla główna programu menedżera zasobu.
 - ❖ rejestrujemy ścieżkę dostępu

w głównej pętli:

- ❖ blokowanie w oczekiwaniu na komunikat

- ❖ wywołanie funkcji obsługi; funkcja obsługi obsługuje nadchodzące zapytania i wykonuje odpowiednie fragmenty kodu, w zależności od zapytania.

10. Przygotowanie kodu własnych funkcji `io_open()`, `io_write()`, `io_read()`.

Poszczególne kroki zaznaczone są w przykładowym kodzie w załączniku 2.

Po napisaniu kodu. Należy skompilować projekt i uruchomić menedżera zasobu. Sprawdź działanie opracowanego menedżera prostym programem – wymieniając dane z inną grupą laboratoryjną, która korzysta z innego portu szeregowego.

Załącznik 1.

Several example functions for serial interface programming

/* Initializing serial interface: bound rate, frame parameters */

```
void SET_RS232(){
    unsigned char temp,licz;

    out8(PORT1+3, 0x80); /* SET DLAB ON */
    out8(PORT1+0, 0x06); /* SET BAUND RATE - 19200 BPS */
    out8(PORT1+1, 0x00); /* */
    out8(PORT1+3, 0x1B); /* PARITY, 1 STOP BIT, 8 BITS */
    out8(PORT1+2, 0xC7); /* FIFO CONTROL */
    out8(PORT1+4, 0x0B); /* TURN ON DTR, RTS and OUT2 */
    licz=16;
    /* just in case */
    do{
        licz--;
        if(!licz) break;
    }while(READ_RS232((unsigned char*)&temp));
}
```

/* Enabling interrupt source */

```
void SET_INT_SOURCE(unsigned char INT_SOURCE){
    out8(PORT1+1, INT_SOURCE);
}
```

/* Disabling interrupt source – interrupt mask*/

```
void CLR_INT_SOURCE(unsigned char INT_SOURCE){
    out8(PORT1+1, INT_SOURCE);
}
```

/* Sending char */

```
void SEND_COMMAND(unsigned char COMM_TRANS){
    out8(PORT1, COMM_TRANS);
}
```

Załącznik 2.

```
/*
resmgr.c
This module contains the source code for the /dev/example device developed as part of the Neutrino
"Writing a Resource Manager" section. If using a shared target, please change EXAMPLE_NAME to
something unique for you, so to avoid testing somebody else's code.
This module contains all of the functions necessary.
*/

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>
#include <sys/neutrino.h>
#include <sys/resmgr.h>

/* default name for this device: /dev/rs232 */
#define EXAMPLE_NAME "/dev/rs232"

/* change to something else if sharing a target */
// #define EXAMPLE_NAME "/dev/dagexample"
void options (int argc, char *argv[]);
```

1. Przygotowanie prototypów funkcji, które będą pełniły rolę funkcji POSIX open(), write(), read().

```
int io_open (resmgr_context_t *ctp, io_open_t *msg, RESMGR_HANDLE_T *handle, void *extra);
int io_read (resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T *ocb);
int io_write(resmgr_context_t *ctp, io_write_t *msg, RESMGR_OCB_T *ocb);
```

2. Zadeklarowanie zmiennych odpowiedzialnych za:

strukturę typu dispatch;

strukturę funkcji połączenia;

strukturę funkcji wejścia-wyjścia;

kontekst menedżera zasobów;

atrybuty menedżera zasobów;

atrybuty funkcji wejścia-wyjścia.

```
/* our connect and I/O functions */
resmgr_connect_funcs_t connect_funcs;
resmgr_io_funcs_t io_funcs;

/* our dispatch, resource manager and iofunc variables */
dispatch_t dpp;
resmgr_attr_t rattr;
dispatch_context_t *ctp;
iofunc_attr_t ioattr;

char *programe = "rs232";
int optv; // -v for verbose operation

main (int argc, char *argv[]){
    int pathID;

    setvbuf (stdout, NULL, _IOLBF, 0);
    printf ("%s: starting...\n", programe);
    options (argc, argv);
    /* allocate and initialize a dispatch structure for use by our main loop */
```

3. Alokacja zasobów i inicjalizacja struktury „dispatch”.

```
dpp = dispatch_create ();
if (dpp == NULL) {
    fprintf (stderr, "%s: couldn't dispatch_create: %s\n",
            programe, strerror (errno));
    exit (1);
}

/*Set up the resource manager attributes structure, we'll use this as a way of passing information to
resmgr_attach(). For now, we just use defaults. */

memset (&rattr, 0, sizeof (rattr)); /* using the defaults for rattr */

/* initialize the connect functions and I/O functions tables to their defaults by calling iofunc_func_init(),
connect_funcs, and io_funcs variables are already declared. */
```

4. Inicjowanie funkcji połączenia i tablicy funkcji wejścia-wyjścia.

```
iofunc_func_init (_RESMGR_CONNECT_NFUNCS, &connect_funcs, _RESMGR_IO_NFUNCS, &io_funcs);

/*over-ride the connect_funcs handler for open with our io_open, and over-ride the io_funcs handlers for
read and write with our io_read and io_write handlers. */
```

5. Podmiana standardowych funkcji handlera na własne funkcje.

```
connect_funcs.open = io_open;  
io_funcs.read = io_read;  
io_funcs.write = io_write;
```

```
/* initialize our device description structure*/
```

6. Inicjowanie struktury opisującej urządzenie.

```
iofunc_attr_init (&ioattr, S_IFCHR | 0666, NULL, NULL);
```

```
/* call resmgr_attach to register our prefix with the process manager, and also to let it know about our  
connect and I/O functions. On error, returns -1 and errno is set. */
```

7. Powiadomienie menedżera procesów o sposobie połączenia i podpięcie funkcji wejścia-wyjścia.

```
pathID = resmgr_attach (dpp, &rattr, EXAMPLE_NAME, _FTYPE_ANY, 0, &connect_funcs, &io_funcs,  
&ioattr);  
if (pathID == -1) {  
    fprintf (stderr, "%s: couldn't attach pathname: %s\n", progname, strerror (errno));  
    exit (1);  
}
```

8. Alokacja struktury kontekstowej menedżera.

```
ctp = dispatch_context_alloc (dpp);
```

9. Pętla główna programu menedżera zasobu.

```
while (1) {  
    if ((ctp = dispatch_block (ctp)) == NULL) {  
        fprintf (stderr, "%s: dispatch_block failed: %s\n", progname, strerror (errno));  
        exit (1);  
    }  
    dispatch_handler (ctp);  
}
```

10. Przygotowanie kodu własnych funkcji io_open(), io_write(), io_read().

```
/*  
 * io_open we are called here when the client does an open. It is up to us to establish a context (in this  
case NULL will do just fine), and return a status code.  
*/
```

```

int io_open (resmgr_context_t *ctp, io_open_t *msg, RESMGR_HANDLE_T *handle, void *extra){
    if (optv) {
        printf ("%s: in io_open\n", progname);
    }

    return (iofunc_open_default (ctp, msg, handle, extra));
}

```

/* io_read

At this point, the client has called their library "read" function, and expects zero or more bytes. Currently our /dev/example resource manager returns zero bytes to indicate EOF -- no more bytes expected. After our exercises, it will return some data.

*/

```

intio_read (resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T *ocb){
    int status;
    static char data[] = "hello";
    int nb;

    if (optv) {
        printf ("%s: in io_read\n", progname);
    }

    if ((status = iofunc_read_verify(ctp, msg, ocb, NULL)) != EOK) {
        if (optv) printf("read failed because of error %d\n", status);
        return (status);
    }
    // No special xtypes
    if ((msg->i.xtype & _IO_XTYPE_MASK) != _IO_XTYPE_NONE) {
        return(ENOSYS);
    }

    nb = strlen(data );
    nb = min( nb, msg->i.nbytes );

    _IO_SET_READ_NBYTES (ctp, nb);
    SETIOV( ctp->iiov, data, nb );

    if (nb > 0) ocb->attr->flags |= IOFUNC_ATTR_ETIME;
    return (_RESMGR_NPARTS (1));
}

```

```

/*
 * io_write
 * At this point, the client has called their library "write" function, and expects that our resource manager
 will write the number of bytes that they have specified to some device. Currently, for /dev/example, all of
 the clients writes always work -- they just go into Deep Outer Space.*/

```

```

int io_write (resmgr_context_t *ctp, io_write_t *msg, RESMGR_OCB_T *ocb){
    int status;
    int nb;

    if (optv) {
        printf ("%s: in io_write, of %d bytes\n", progname, msg->i.nbytes);
    }

    if ((status = iofunc_write_verify(ctp, msg, ocb, NULL)) != EOK) return (status);
    // No special xtypes
    if ((msg->i.xtype & _IO_XTYPE_MASK) != _IO_XTYPE_NONE) {
        return(ENOSYS);
    }

    if( msg->i.nbytes == ctp->info.msglen - (ctp->offset + sizeof(*msg) )){
        /* have all the data */
        char *buf;
        buf = (char *)(msg+1);

        nb = write( STDOUT_FILENO , buf, msg->i.nbytes ); // fd 1 is stdout
        if( -1 == nb )
            return errno;
    }
    else {
#if 0
        char *buf;
        buf = malloc( msg->i.nbytes );
        if (NULL == buf )
            return ENOMEM;
        nb = resmgr_msgread(ctp, buf, msg->i.nbytes, sizeof(*msg));
        nb = write(1, buf, nb ); // fd 1 is stdout
        free(buf);
        if( -1 == nb )
            return errno;
#else
        char buf[1000]; // my hardware buffer
        int count, bytes;
        count = 0;

```

```

        while ( count < msg->i.nbytes ){
            bytes = resmgr_msgread( ctp, buf, 1000, count + sizeof(*msg ));
            if( bytes == -1 ) return errno;
            bytes = write( 1, buf, bytes ); // fd 1 is standard out
            if( bytes == -1 ){
                if (!count ) return errno;
                else break;
            }
            count += bytes;
        }
        nb = count;
#endif
    }
    _IO_SET_WRITE_NBYTES (ctp, nb);

    if (msg->i.nbytes > 0)
        ocb->attr->flags |= IOFUNC_ATTR_MTIME | IOFUNC_ATTR_CTIME;

    return (_RESMGR_NPARTS (0));
}

```

```

/*
options
This routine handles the command line options. For our simple /dev/example, we support:
    -v verbose operation */

```

```

Void options (int argc, char *argv[]){
    int  opt;

    optv = 0;

    while ((opt = getopt (argc, argv, "v")) != -1) {
        switch (opt) {
            case 'v':
                optv++;
                break;
        }
    }
}

```

Załącznik 3.

```
/*intsimple.c
```

This module demonstrates will contain code for handling an interrupt. To test is, simply run it. Note that you may have to do something to cause the interrupts to be generated (e.g. press a key if handling the keyboard interrupt). */

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/neutrino.h>
#include <sys/syspage.h>
```

```
char *programe = "intsimple";
struct sigevent int_event; // the event to wake up the thread
```

```
const struct sigevent *hdr( void *blah, int id ){
    static int count = 0;
    if (++count > 1500 ){
        count = 0;
        return &int_event;
    }
    return NULL;
}
```

```
main (int argc, char **argv){
    int id;
    setvbuf (stdout, NULL, _IOLBF, 0);

    printf ("%s: starting...\n", programe);
```

```
// request I/O privity
ThreadCtl(_NTO_TCTL_IO, 0 );
```

```
/* set up an event for the handler or the kernel to use to wake up this thread. Use whatever type of
event and event handling you want */
```

```
SIGEV_INTR_INIT( &int_event );
```

```
// either register an interrupt handler or the event
```

```
id = InterruptAttach(0, hdr, NULL, 0, _NTO_INTR_FLAGS_TRK_MSK );
```

```
while (1) {  
    // block here waiting for the event  
    InterruptWait(0, NULL );  
    // if using a high frequency interrupt, don't print every interrupt  
    printf ("%s: we got an event after 1500 timer ticks and unblocked\n", progname);  
}  
}
```