KoNaR

KOŁO NAUKOWE ROBOTYKÓW

I³BOT

Control Center.

Jan Kędzierski

Edgar Ostrowski

Wrocław, 18.06.2007

Spis treści

1 Wstęp

Celem projektu było stworzenie panelu do sterowania oraz wizualizacji danych sensorycznych dla platformy mobilnej I^3BOOT. Oprogramowanie stworzono za pomocą biblioteki QT. Niniejszy raport zawiera opis przygotowanej części sprzętowo-kominikacyjnej dla robota. Zawarto w nim opis instrukcji, komend oraz listing kodu realizujący kolejkę FIFO.

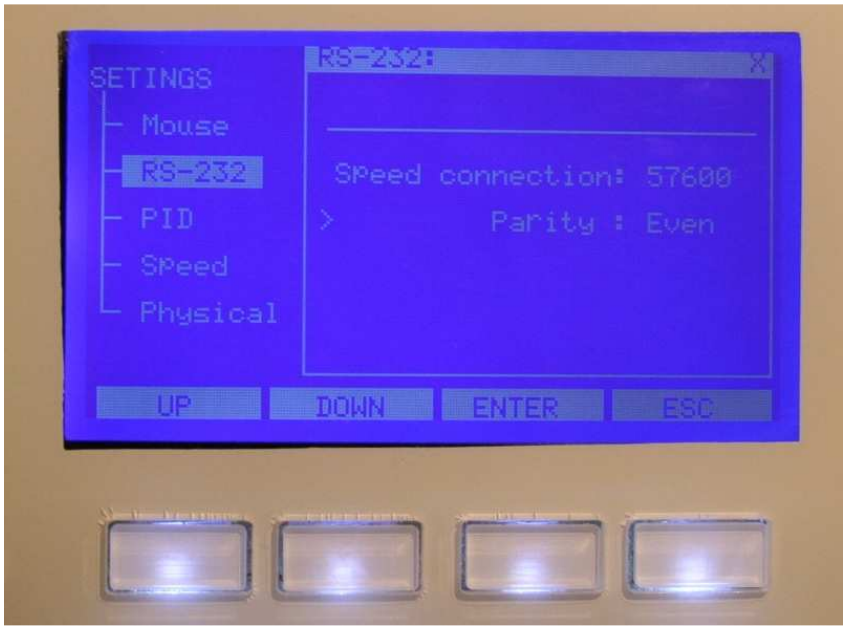
W robocie zrealizowano redundantny system pomiarowy. Wyposażono go w pomiar prędkości za pomocą akcelerometrów, optycznego czujnika przyspieszeń oraz enkoderów. Pomiar przemieszczenia wykonywany jest za pomocą optycznego czujnika. Do wizualizacji ruchu robota wykorzystano pomiary przesunięcia w globalnym układzie odniesienia. Informacje o użytym w robocie mikrokontrolerze, czujnikach i pozostałych peryferiach można znaleźć na stronie [2]:

http://www.konar.ict.pwr.wroc.pl/infopage.php?id=13

2 Komunikacja

Komunikacja z robotem odbywa się przy pomocy interfejsu RS232. Wykorzystano zintegrowany w mikrokontrolerze interfejs komunikacyjny SCI [1]. W ramach projektu zrealizowano możliwość konfiguracji transmisji po stronie nadajnika jak i odbiornika [Fot. 1] [Fot. 2]. Konstrukcję wyposażona była w graficzny wyświetlacz 240x128 pkt dzięki temu możliwe było napisanie łatwego w obsłudze programu konfigurującego łącze komunikacyjne. Zmian prędkości, bitu stopu, bitu parzystości można łatwo dokonywać dzięki przyjaznemu interfejsowi na wyświetlaczu robota.

W menu robota wybieramy poszczególne opcje MAIN MENU>SETTINGS>RS-232



Fot. 1 Interfejs konfiguracyjny

Fot. 2 Konfiguracja po stronie komputera PC.

W robocie zrealizowano także prostą konsole, która wyświetla i potwierdza zrozumiałe komendy [Fot. 2]. Dzięki konsoli można w łatwy sposób sprawdzić czy robot otrzymuje odebrane informacje.

Fot. 3 Konsola robota

3 Lista komend

Poniżej w tabeli poniżej przedstawiono wszystkie obecnie obsługiwane komendy przez robota.

Lp.	*Nazwa komendy*	*wartość zwracana*	*Uwagi*
1	go(V_{lin},V_{kat})	MC68k: OK. (potwierdzenie)	zadanie prędkości liniowej i kątowej (-40<=V_{lin}<=40, -40<=V_{kat}<=40) [integer]
2	stop	MC68k: OK. (potwierdzenie)	zatrzymanie napędów
3	AA	aa(A_x,A_y)	wartości przyspieszeń platformy
4	AV	aa(V_{lin},V_{kat})	prędkości robota zmierzone przy pomocy akcelerometrów
5	OV	ov(V_{lin},V_{kat})	prędkości robota zmierzone przy pomocy czujnika optycznego
6	OP	op(x,y,theta)	pozycja robota w globalnym układzie odniesienia
7	op(x,y,theta)	MC68k: OK. (potwierdzenie)	ustawienie pozycji robota w globalnym układzie odniesienia [float]
8	EV	ev(V_{lin},V_{kat})	Prędkości robota zmierzone przy pomocy enkoderów
9	OEV	oev(V_{lin},V_{kat}, V_{lin},V_{kat})	prędkości robota zmierzone przy pomocy enkoderów i czujnika optycznego
10	WV	wv(V_L,V_P)	prędkości kół zmierzone przy pomocy enkoderów
11	WP	wp(P_L,P_P)	pozycje kół zmierzone przy pomocy enkoderów
12	wp(P_L,P_P)	MC68k: OK. (potwierdzenie)	ustawienie pozycji kół [float]
13	PID	pid(Kp,Kd,Ki)	zwraca nastawy regulatorów PID
14	pid(Kp,Kd,Ki)	MC68k: OK. (potwierdzenie)	ustawia nastawy regulatorów PID (0<=Kp,Kd,Ki<=100) [integer]
15	buz(time,tone)	MC68k: OK. (potwierdzenie)	użycie funkcji buzzer wydającej dzwięk przez platformę o zadanym czasie i zadanej tonacji (1<=time<=255, 0<=$tone_t$<=80) [integer]

Tabela 1. Zestaw komend.

Przykładowe zapytania:

```
<< go(5,0)
>> MC68k: OK.
<< pid(40,12,10)
>> MC68k: OK.
<< WV
```

```
>> wv(5,4)
```

4 Kolejka FIFO

Kolejka (ang. **FIFO**, *First In, First Out*; *pierwszy na wejściu, pierwszy na wyjściu*) – liniowa struktura danych, w której nowe dane dopisywane są na końcu kolejki, a z początku kolejki pobierane są dane do dalszego przetwarzania. W robocie zaimplementowaną na potrzeby projektu kolejkę wykorzystano do przesyłania tekstów (komend) znak po znaku. Kod kolejki dla mikrokontrolera MC68332 pobrano ze strony domowej formy Freescale. Napisano go z pomocą mgr Mariusza Janiaka na podstawie dokumentu AN1724/D .

Zawartość pliku **scibuf.h**

```
#ifndef LWORD
#define LWORD unsigned long
#endif

#ifndef WORD
#define WORD unsigned int
#endif

#ifndef BYTE
#define BYTE unsigned char
#endif

#define QSIZE 200                   /*rozmiar kolejki*/

/*Definicje struktur*/
typedef struct {
      unsigned int in;              /*Wzkaznik na poczatek kolejki*/
      unsigned int full;            /*Flaga pelnej kolejki*/
      unsigned int out;             /*Wskaznik na koniec kolejki*/
      unsigned char q[QSIZE];       /*Tablica kolejki*/
      }queue_struct;
/*Prototypy funkcji*/

void qinit(queue_struct *);  /*Inicjalizuje wybrany bufor */
WORD qstat(queue_struct *);   /* Zwaraca info. o stanie wybranego bufora */
//sciinit(void);              /* SCI module initialization */
interrupt void sciint(void);  /* wskaznik przerwania SCI (TX i RX) */
char rx_byte(BYTE *);         /* Czyta bajt z bufora RX */
char tx_byte(BYTE);           /* Zapisuje bajt do bufora TX */
BYTE SCI_PutText(char *, BYTE);     /*Wysyla stringa przez RS-a*/
void komunikacja();
void test_screen();

/*UWAGA: Wymaga wczesniejszego skonfigurowania SCI*/
```

Zawartość pliku **scibuf.c**

```
#include "scibuf.h"
#include "qsm.h"
#include "lcd.h"
#include <stdio.h>
#include <string.h>


char rx_byte(BYTE *rxbyte)
/* Read a received byte from the RX buffer */
/* Return 0 if no byte available, or non zero if byte read OK */
```

```
{
	LWORD rxq_full_out;
	WORD rxin, rxfull, rxout;
	/* Read rx buffer variables into locals */
	rxin = rxbuff.in;
	rxfull = rxbuff.full;
	rxout = rxbuff.out;
	if ((rxin!=rxout)||rxfull)	/* IF (rx data avaialable) */
	{
		*rxbyte = rxbuff.q[rxout];	/* read data byte */
		rxout++;				/* update pointer..*/
		if (rxout>(QSIZE-1)) rxout = 0;	/* check for wraparound */
		rxq_full_out = (LWORD)rxout;  /* prepare for coherent write..*/
		*(LWORD *)(&rxbuff.full) = rxq_full_out; /* of rxqout with
rxqfull zero */
		return 1;		/* return 1 for successful read */
	}
	else
	{
		return 0;		/* return 0 as no byte available */
	}
}
void qinit (queue_struct *qvar)
/* Initialize specified queue */
{
	qvar->in = 0x0002;	/* Set in=out, and full=0, ie. buffer empty */
	qvar->full = 0x0000;
	qvar->out = 0x0002;
}
WORD qstat(queue_struct *qvar)
/* Return the number of valid bytes in a specified buffer */
{
	WORD qin, qfull, qout;
/* Read q buffer variables into locals */
	qin = qvar->in;
	qfull = qvar->full;
	qout = qvar->out;
	if (qin>qout)
	{
		/* Data in buffer, no pointer wrap */
		return (qin-qout);
	}
	if (qin<qout)
	{
		/* Data in buffer with pointer wrap */
		return (QSIZE-qout+qin);
	}
	/* qin must be = qout */
	if (qfull)
	{
		/* Buffer full */
		return (QSIZE);
	}
	/* Buffer empty */
	return (0);
}
char tx_byte(BYTE txbyte)
/* Queue a byte into the TX buffer for transmission */
/* Return 0 if buffer already full, or non zero if byte queued OK */
{
```

```
      WORD txin, txfull, txout;
      LWORD txq_in_full;

      /* Read rx buffer variables into locals */
      txin = txbuff.in;
      txfull = txbuff.full;
      txout = txbuff.out;

      if ((txin!=txout)||!txfull) /* IF (tx buffer not full) */
      /* Add data to tx buffer as space is available */
      {
            txbuff.q[txin] = txbyte;
            txin++;
            if (txin>(QSIZE-1)) txin = 0; /* check for wraparound */
            if (txin==txout)/* Buff is now full, need to set FULL flag
coherently with upd. of txin */
            /* Also generate warning that buffer is now full */
            {
                  txq_in_full = (((LWORD)txout)<<16)|1; /* prepare for
coherent write..*/
                  *(LWORD *)(&txbuff.in) = txq_in_full;/* of txin with
txqfull non-zero */
            }
            else
            {
            /* Buffer not full yet, so don't set FULL */
                  txbuff.in = txin;
            }
            /* Enable transmitter interrupt (TIE bit set) */
            SCCR1 = 0x00ac;
            return (1);
      }
      else
      /* Return error code as no buffer space to queue data */
      {
            return (0);
      }
}

interrupt void sciint()
/* SCI RX interrupt handler */
/* Note : The SCI interrupt handler will only clear a single
      interrupt source each time it is executed, by using an else-if
structure.
      This ensures that the transmitter routine is not called unnecessarily
      if the SCI transmitter is idle (TDRE set) but with no transmit buffer
      data pending, and the interrupt source therefore disabled */
{
      WORD status, qin, qfull, qout;
      WORD scidata;
      LWORD q_long;

      /* Read status reg to determine source of exception */
      status=SCSR;

      if (status & 0x0040)/* if RDRF (rx data register full) flag set */
      {
            scidata = SCDR;   /* Read received data (+ clear RDRF) */

            if (status & 0x0008)    /* if OR (receiver OverRun) flag set */
```

```
            /* Note : typically RDRF will be set at the same time as OR, in
which case
            the RDRF handler will clear OR.
            OR may be set on its own if an overrun is generated in between
QSM_SCSR being read with RDRF set (normal receive), and RDRF being cleared
by reading SCDR. Because of this possibility, we need an independent OR
flag test and clear mechanism */
            {
/* Any additional receiver overrun handling should be added here and in the
separate OR handler. */
            }
            else
            {
/* Note : This s/w flow discards any received data which has caused a
receiver overrun, as it may be regarded as a non-recoverable error. The s/w
could be modofied by removing the previous 'else' statement so that no data
is discarded, although this may result in missing data from the receive
buffer when overrun occurs. */
/* Read rx buffer variables into locals */
                  qin = rxbuff.in;
                  qfull = rxbuff.full;
                  qout = rxbuff.out;

                  if ((qin!=qout)||!qfull)/* IF (rx buffer not full) */
                  /* data received, but OK. as space in rx buffer */
                  {
                        rxbuff.q[qin] = (BYTE)scidata;
                        qin++;
                        if (qin>(QSIZE-1)) qin = 0;/* check for wraparound
*/
                        if (qin==qout)/* Buffer now full, so set FULL flag
coherently with update of qin */
/* Also generate warning that buffer is now full */
                        {
                              q_long = (((LWORD)qout)<<16)|1;/* prepare for
coherent write..*/
                              *(LWORD *)(&rxbuff.in) = q_long;/*of qin with
rxqfull non-zero*/
                        }
                        else
                        {
/* Buffer not full yet, so don't set FULL */
                              rxbuff.in = qin;
                        }
                  }
                  else
/* Data received, but buffer full (buffer overrun) so flag error */
/* Any additional handler for buffer overrun should be added here */
                  {

                  }
            }
      }
      else if (status & 0x0008) /* if OR (receiver OverRun) flag set */
/* Note : typically RDRF will be set at the same time as OR, in which case
the RDRF handler will clear OR.OR may be set on its own if an overrun is
generated in between SCSR being read with RDRF set (normal receive), and
RDRF being cleared by reading SCDR. Because of this possibility, we need an
independent OR flag test and  clear mechanism */
      {
            scidata = SCDR;
```

```
 /* Read received data to clear OR */
/* Any additional receiver overrun handling should be added here and in the
RDRF + OR handler */
      }

      else if (status & 0x0100)
/* if TDRE (tx data register empty) flag set */
      {
            qin = txbuff.in;
            qfull = txbuff.full;
            qout = txbuff.out;

            if ((qin!=qout)||qfull) /* IF (tx data avaialable) */
            {
                  scidata = txbuff.q[qout];/* read data byte */
                  qout++;           /* update pointer..*/
                  if (qout>(QSIZE-1)) qout = 0;/* check for wraparound */
                  q_long = (LWORD)qout;   /* prepare for coherent write..*/
                  *(LWORD *)(&txbuff.full) = q_long; /* of txqout with
txqfull zero */

                  SCDR = scidata;         /* Write tx data to SCI (+ clear
TDRE)*/
            }
            else                          /* no more tx data pending.. */
            {
                  /* ..so disable transmitter interrupt (clear TIE bit) */
                  SCCR1 = 0x002c;
            }
      }
      /* else   /* some other interrupt source */
      /* {  */
      /*      sciinit();      /* something is wrong - re-initialize sci */
      /* } */
}
```

5 Przesyłanie tekstu

Przesyłanie tekstu odbywało się znak po znaku. W argumencie funkcji przesyłającej podawano ciąg znaków w postaci tablicy. Funkcja ta przesyłała kolejno znak po znaku do kolejki FIFO.

```
BYTE SCI_PutText(char *str, BYTE mode)
/*Dodaje stringa do kolejki*/
/* mode = 0 - program czeka az uda mu sie umiescic wszystkie znaki w
kolejce*/
/* mode = 1 - program nie czeka gdy nie uda mu sie umiescic znaku w kolejce
*/
/*   zapamietuje polozenie znaku ostatnio nadanego.                */
/*Zwraca 0 jesli nie udalo mu sie umiescic calego stringa w kolejce, 1 w
*/
/*przeciwnym razie.*/
{
      static WORD offset=0;

      switch (mode)
      {
      case 0:{
```

```
            while(*str)
                  if(tx_byte(*str)) str++;
            offset=0;
      }
      return 1;
      case 1:{
            str+=offset;
            while(*str){
                  if(tx_byte(*str)){
                        str++;
                        offset++;
                  }
                  else return 0;
            }
            offset=0;
      }
      return 1;
      }
}
```

6 Przesyłanie i rozpoznawanie komend.

Komendy rozpoznawano porównując pierwsze trzy znaki. Znaki te dla każdej funkcji były unikalne. Funkcja rozpoznająca komendy sprawdzała z możliwie dużą prędkością czy w kolejce przechowującej odebrane informacje znajdują się jakieś znaki. Jeśli tak wystarczyło sprawdzić czy komenda jest kompletna. Kompletna komenda zakończona powinna być znakiem końca linii '\n'. Pojawienie się tego znaku zezwalało w kolejnym kroku rozpoznać jaka to komenda.

```
void komunikacja(){
      unsigned char buf[40];
      unsigned char buf2[80];
      unsigned char rxchar;
      unsigned int i=0;
      unsigned int znaki=0;
      signed int WL=0;
      signed int WK=0;
      float kat=0;

znaki= qstat(&rxbuff);
if (rxbuff.in>0) i=rxbuff.in-1;
rxchar=rxbuff.q[i];
if ((znaki>0)&&(rxchar=='\n'))
{
for (i=0;i<40;i++) buf[i]=' ';
for (i=0;i<znaki;i++)
            {
            rx_byte(&rxchar);
            buf[i]=rxchar;
            }
////////////////////////////////////////
///////   go(x,x)    ////////////////////
if (strncmp(buf,"go(0,0)\n",3)==0)
{
      SCI_PutText("MC68k: OK.\n", 0);
      sscanf(buf,"go(%d,%d)\n",&WL,&WK);
      if (stan.state==2)
```

```
            {
            ster.V_lin=WL;
            ster.V_kat=WK;
            ster.V_L=(WL-WK);
            ster.V_P=(WL+WK);
            }
return;
}
////////////////////////////////////////
////////////////////////////////////////
///////     stop     ///////////////////

if (strncmp(buf,"stop\n",4)==0)
{
      SCI_PutText("MC68k: OK.\n", 0);
      if (stan.state==2)
            {
            ster.V_lin=0;
            ster.V_kat=0;
            ster.V_L=0;
            ster.V_P=0;
            }
      motors_stop();
return;
}
////////////////////////////////////////
////////////////////////////////////////
///////      AA - acc acc    ///////////

if (strncmp(buf,"AA\n",2)==0)
{
sprintf(buf2,"aa(%d,%d)\n",stan.A_lin,stan.A_kat);
SCI_PutText(buf2, 0);

return;
}
////////////////////////////////////////
////////////////////////////////////////
///////     AV - acc vel    ////////////

if (strncmp(buf,"AV\n",2)==0)
{
sprintf(buf2,"aa(%d,%d)\n",stan.V_A_lin,stan.V_A_kat);
SCI_PutText(buf2, 0);
return;
}
////////////////////////////////////////
////////////////////////////////////////
///////     OV - optic vel   ///////////

if (strncmp(buf,"OV\n",3)==0)
{
sprintf(buf2,"ov(%.1f,%.1f)\n",stan.V_M_lin,stan.V_M_kat);
SCI_PutText(buf2, 0);
return;
}
////////////////////////////////////////
////////////////////////////////////////
///////   OP - optic pos    ////////////

if (strncmp(buf,"OP\n",2)==0)
```

```
{
sprintf(buf2,"op(%.1f,%.1f,%.2f)\n",stan.poz_x,stan.poz_y,stan.theta);
SCI_PutText(buf2, 0);
return;
}
//////////////////////////////////////
//////////////////////////////////////
///////  op - set optic pos   //////////

if (strncmp(buf,"op(0,0,0)\n",3)==0)
{
      SCI_PutText("MC68k: OK.\n", 0);
      sscanf(buf,"op(%d,%d,%f)\n",&WL,&WK,&kat);
      stan.poz_x=WL;
      stan.poz_y=WK;
      stan.theta=kat;
return;
}
//////////////////////////////////////
//////////////////////////////////////
///////  EV - encoder vel  //////////

if (strncmp(buf,"EV\n",2)==0)
{
sprintf(buf2,"ev(%d,%d)\n",stan.V_E_lin,stan.V_E_kat);
SCI_PutText(buf2, 0);
return;
}

//////////////////////////////////////
//////////////////////////////////////
/////// OEV - optic encoder vel  ///////

if (strncmp(buf,"OEV\n",3)==0)
{
sprintf(buf2,"oev(%.1f,%.1f,%d,%d)\n",stan.V_M_lin,stan.V_M_kat,stan.V_E_li
n,stan.V_E_kat);
SCI_PutText(buf2, 0);
return;
}

//////////////////////////////////////
//////////////////////////////////////
///////  WV - wheel encoder  //////////

if (strncmp(buf,"WV\n",3)==0)
{
sprintf(buf2,"wv(%d,%d)\n",stan.V_W_L,stan.V_W_P);
SCI_PutText(buf2, 0);
return;
}

//////////////////////////////////////
//////////////////////////////////////
///////  PID - PID settings  //////////

if (strncmp(buf,"PID\n",3)==0)
{
sprintf(buf2,"pid(%d,%d,%d)\n",settings.Kp,settings.Kd,settings.Ki);
SCI_PutText(buf2, 0);
return;
```

```
}
////////////////////////////////////////
////////////////////////////////////////
///////  pid - set pid   //////////

if (strncmp(buf,"pid(0,0,0)\n",3)==0)
{
SCI_PutText("MC68k: OK.\n", 0);
sscanf(buf,"pid(%d,%d,%d)\n",&settings.Kp,&settings.Kd,&settings.Ki);
return;
}

}
}
////////////////////////////////////////
////////////////////////////////////////
///////  buz   (0-1000,0-99) //////////

if (strncmp(buf,"buz(0,0)\n",3)==0)
{
SCI_PutText("MC68k: OK.\n", 0);
sscanf(buf,"buz(%d,%d)\n",&WL,&WK);
buzer((unsigned char)WL,(unsigned char)WK);
return;
}
```

7 Literatura

[1] *QSM Queued Serial Module Reference Manual, Freescale Semiconductors Inc.*

[2] Kędzierski J. Ostrowski E. *Robot mobilny klasy (2,0) „Blue Screen”.* http://www.konar.ict.pwr.wroc.pl/infopage.php?id=13