Programovanie robotov LEGO NXT pomocou NXC

(beta 30 alebo novšie)

(Verzia 2.2, 7. júna 2007)

Daniele Benedettelli

s opravami od Johna Hansena

slovenský preklad

Ing. Slavko Fedorik

(24. marca 2010)

Predslov

Rovnako ako pre staré dobré Mindstorms RIS, CyberMaster a Spybotics, je pre získanie plného výkonu z kocky Mindstorms NXT potrebné programovacie rozhranie, ktoré je praktickejšie ako NXT-G (obdoba grafického jazyka National Instruments Labview), ktorý je dodávaný spolu so súpravou NXT.

NXC je programovací jazyk, ktorý vymyslel John Hansen, a ktorý je špeciálne navrhnutý pre robotov Lego. Ak ste ešte nikdy nepísali žiaden program, nerobte si starosti. NXC sa používa naozaj jednoducho a tento tutoriál vás prevedie prvými krokmi v tomto jazyku .

Aby bolo písanie programov ešte ľahšie, existuje nástroj, zvaný Riadiace centrum Bricx (*BricxCC – Bricx Command Center*). Tento nástroj pomôže pri písaní programu, pri jeho stiahnutí do robota, s jeho spustením i zastavením, pri prechádzaní pamäte flash v NXT, pri konverzii zvukových súborov pre použitie v kocke a veľa ďalšieho. BricxCC pracuje podobne ako textový editor, ale má niekoľko rozšírení. V tomto tutoriáli budeme ako integrované vývojové prostredie (*IDE – integrated development environment*) používať BricxCC (verziu 3.3.7.16 alebo novšiu), ktoré si môžete slobodne stiahnuť z webovej adresy:

http://bricxcc.sourceforge.net/

BricxCC beží na počítačoch s Windows (95, 98, ME, NT, 2K, XP, Vista) (v Linuxe je čiastočne funkčný prostredníctvom wine – pozn. prekladateľa). Jazyk NXC byť tiež použitý aj na iných platformách a stiahnuť ho môžete, rovnako slobodne, z webovej stránky:

http://bricxcc.sourceforge.net/nxc/

Väčšia časť tohto tutoriálu je použiteľná i na iných platformách, ibaže možno prídete o niektoré nástroje, ktoré sú zahrnuté v BricxCC a farebné zvýrazňovanie syntaxe.

Tutoriál bol aktualizovaný tak, aby pracoval s NXC beta 30 a novšími verziami. Niektoré ukážkové programy tak nebudú pracovať s verziami staršími ako beta 30.

Len ako poznámka, moja webová stránka je plná vecí, ktoré súvisia s Lego Mindstorms RCX a NXT, vrátane nástrojov pre komunikáciu PC s NXT:

http://daniele.benedettelli.com

Poďakovanie

Veľká vďaka patrí Johnovi Hansenovi, ktorého práca je neoceniteľná!

Obsah

Predslov....2
Poďakovanie....2
I. Písanie svojho prvého programu....5
1. Postavenie robota....5
2. Spustenie Riadiaceho centra Bricx....5
3. Písanie programu....6
4. Spustenie programu....8
5. Chyby v programe....8
6. Zmena rýchlosti....9
7. Zhrnutie....10
II. Zaujímavejší program....11
1. Zatáčanie....11
2. Opakovanie príkazov....12
3. Pridanie komentárov....13
4. Zhrnutie....14
III. Použitie premenných....15
1. Pohyb po špirále....15
2. Náhodné čísla....17
3. Zhrnutie....17
IV. Riadiace štruktúry....18
1. Príkaz if....18
2. Príkaz do....19
3. Zhrnutie....20
V. Senzory....21
1. Čakanie na senzor....21
2. Akcia pri stlačení senzora....22
3. Svetelný senzor....22
4. Zvukový senzor....24
5. Ultrazvukový senzor....25
6. Zhrnutie....25
VI. Úlohy a podprogramy....27
1. Úlohy....27
2. Podprogramy....28
3. Definícia makier....30

4. Zhrnutie 32
VII. Tvorba hudby....33
1. Prehrávanie zvukových súborov 33
2. Prehrávanie hudby....34
3. Zhrnutie 35
VIII. Viac o motoroch 36
1. Plynulé zastavenie....36
2. Pokročilé príkazy....36
3. Riadenie PID....39
4. Zhrnutie 40
IX. Viac o senzoroch....41
1. Režim a typ senzora....41
2. Rotačný senzor 43
3. Viac senzorov na jednom vstupe....44
4. Zhrnutie 45
X. Paralelné úlohy....47
1. Zlý program 47
2. Kritické sekcie a mutexy 48
3. Použitie semaforov....49
4. Zhrnutie 51
XI. Komunikácia medzi robotmi 52
1.Komunikácia Master – Slave....52
2. Posielanie čísel s potvrdením....54
3. Priame príkazy....56
4. Zhrnutie 57
XII. Viac príkazov....58
1. Časovače....58
2. Bodový maticový displej....59
3. Súborový systém....60
4. Zhrnutie....64
XIII. Záverečné poznámky....65

I. Písanie svojho prvého programu

V tejto kapitole vám ukážem ako napísať úplne jednoduchý program. Naprogramujeme robota tak, aby sa 4 sekundy posúval vpred, potom ďalšie 4 sekundy späť a nakoniec zastane. Nič veľkolepé, ale uvedie vás do základnej myšlienky programovania. A tiež vám ukáže aké je to jednoduché. Avšak, pred tým ako môžeme napísať program, potrebujeme robota

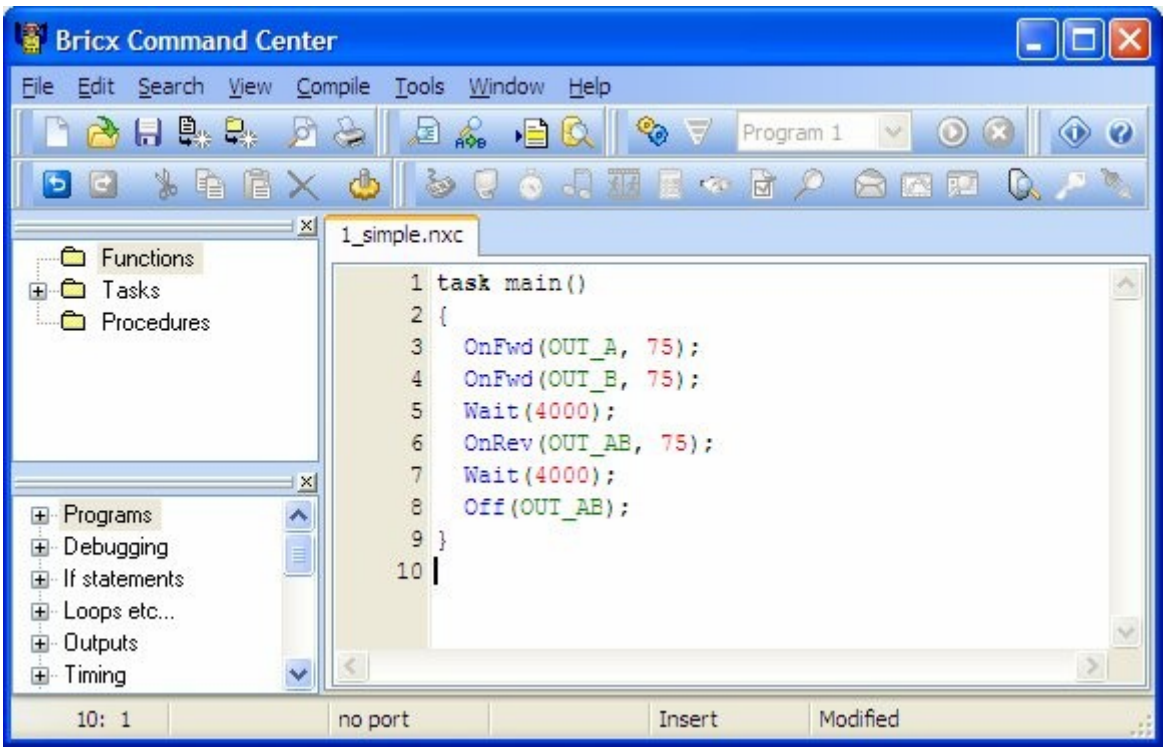
1. Postavenie robota

Robot, ktorého budeme v tomto tutoriáli používať je Tribot, prvý stroj, ktorého stavebný návod sa vám dostane do ruky hneď po otvorení krabice s NXT stavebnicou. Jediný rozdiel je v tom, že pravý motor je pripojený k portu A, ľavý motor k portu C a motor chytača k portu B.

Overte si, že máte správne nainštalované ovládače Mindstorms NXT Fantom, ktoré sú súčasťou súpravy.

2. Spustenie Riadiaceho centra Bricx

Programy budeme písať pomocou Riadiaceho centra Bricx (*Bricx Command Center*). Spustite ho pomocou dvojitého kliknutia na ikonu BricxCC. Predpokladám, že BricxCC už máte nainštalované, ak nie, stiahnite ho z webovej stránky (viz. Predslov) a nainštalujte ho do ľubovoľného adresára. Program sa po sputení spýta, kde máte robota pripojeného. Zapnite robota a stlačte OK. Program (väčšinou) nájde robota automaticky. Potom sa objaví používateľské rozhranie, podobné tomu na obrázku (bez záložky s textom).



Používateľské rozhranie vyzerá podobne ako štandardný textový editor, so zvyčajným menu a tlačidla,i pre otvorenie a uloženie súborov, tlačenie súborov, upravovanie súborov, atď. Program však má i niektoré špeciálne položky menu, pre kompiláciu a sťahovanie programov do robota a pre získanie informácií od robota. Tieto môžete zatiaľ ignorovať.

Ideme napísať nový program, preto stlačte tlačidlo Nový súbor (*New File*), čím sa otvorí nové, prázdne okno.

3. Písanie programu

Teraz napíšte nasledujúci program:

```
task main()
{
    OnFwd(OUT_A, 75);
    OnFwd(OUT_C, 75);
    Wait(4000);
    OnRev(OUT_AC, 75);
    Wait(4000);
    Off(OUT_AC);
}
```

Možno sa vám zdá program trochu komplikovaný, tak si hoďme podrobnejšie rozobrať.

Program v NXC sa skladá z úloh (*tasks*). Náš program má len jednu úlohu, nazvanú **main**. Každý program musí mať úlohu nazvanú **main** – je to úloha, ktorá je vykonávaná robotom. Viac o úlohách sa naučíte v Kapitole VI. Úloha sa skladá z viacerých príkazov. Všetky príkazy úlohy sú uzatvorené do zložených zátvoriek, takže je úplne jasne vidno, že všetky patria do tejto úlohy. Každý príkaz končí bodkočiarkou. Takto možno jednoznačne

určiť kde jeden príkaz končí a kde druhý začína. Takže všeobecný vzhľad úlohy vyzerá takto:

```
task main()
{
    statement1;
    statement2;
    ...
}
```

Náš program má šesť príkazov. Pozrime sa teda na nich po jednom:

```
OnFwd(OUT_A, 75);
```

Tento príkaz vraví robotovi aby zapol výstup A, kde je pripojený motor motor na výstup označený na NXT kocke ako A, pre posun dopredu. Číslo, ktoré nasleduje za označením portu udáva, že motor má byť spustený na 75 % maximálnej rýchlosti.

```
OnFwd(OUT_C, 75);
```

Rovnaký príkaz ako predchádzajúci, len teraz spúšťame motor C. Po týchto dvoch príkazoch sú oba motory spustené a robot sa posúva dopredu.

```
Wait(4000);
```

Nastal čas chvíľu počkať. Tento príkaz vraví, že robot má počkať 4 sekundy. Argument, ktorým je číslo v zátvorke, je počet 1/1000 sekundy: takto možno programu veľmi presne udať, ako dlho má čakať. Takže program 4 sekundy nerobí nič (čaká) a tak robot pokračuje v posúvaní vpred.

```
OnRev(OUT_AC, 75);
```

Po uplynutí čakacej doby sa robot posunul dostatočne ďaleko, takže ho možno nasmerovať v opačnom smere, ktorým je cúvanie. Všimnite si, že možno nastaviť oba motory naraz, pomocou argumentu OUT_AC. Rovnako sme mohli spojiť aj prvé dva príkazy posunu.

```
Wait(4000);
```

Znova čakáme 4 sekundy.

```
Off(OUT_AC);
```

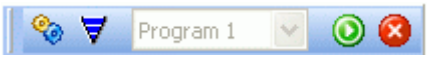
A nakoniec oba motory vypneme.

To je celý program. Posúva oba motory 4 sekundy vpred, potom znova 4 sekundy späť a nakoniec ich vypína.

Pravdepodobne ste si všimli pri písaní programu farby (nie v tomto tutoriáli). Objavujú sa automaticky. Farby a štýly sú použité editorom počas zvýrazňovania syntaxe a sú prispôsobiteľné.

4. Spustenie programu

Po napísaní programu, je treba ho skompilovať (to znamená zmeniť do binárneho kódu, ktorý dokáže robot vykonať) a poslať ho do robota pomocou USB káblu alebo BlueTooth rozhrania. Toto sa nazýva stiahnutie (*downloading*) programu.



Na obrázku môžete vidieť tlačidlá, ktoré umožňujú (zľava doprava) kompilovať, stiahnuť, spustiť a zastaviť program.

Stlačte druhé tlačidlo a (za predpokladu, že ste neurobili pri písaní programu žiadny preklep) bude program správne skompilovaný a stiahnutý (ak program obsahuje chyby, budete na nich upozornený, viz nižšie).

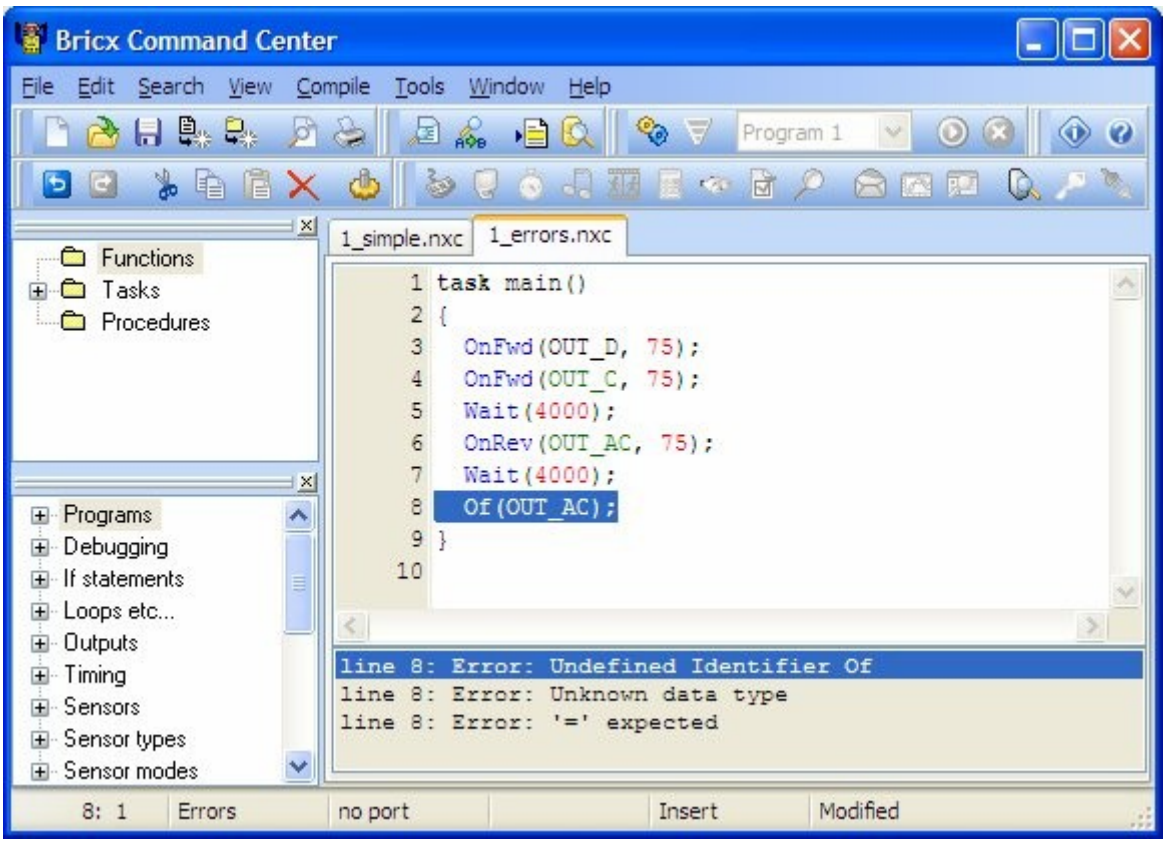
Teraz možno program spustiť. Pre spustenie vojdite do menu v kocke My Files -> Software files a spusťte program 1_simple . Nezabudnite: programové súbory v súborovom systéme NXT budú mať rovnaké meno ako váš súbor NXC.

Aby ste spustili program automaticky, môžete použiť klávesovú skratku CTRL+F5 alebo po stiahnutí programu môžete stlačiť zelené tlačidlo run.

Robí robot to, čo ste čakali? Ak nie, skontrolujte pripojenie.

5. Chyby v programe

Existuje veľká pravdepodobnosť, že pri písaní programu urobíte nejaké chyby. Kompilátor upozorní na chyby a oznámi ich v spodnej časti okna, ako na nasledujúcom obrázku:



BricxCC automaticky vyberie prvú chybu (zle sme napísali meno motora). Keď je v programe viacero chýb, môžete sa kliknutím na chybové hlásenie presunúť na riadok s chybou. Pamätajte, že často chyby na začiatku programu majú za následok i tie ďalšie, preto je lepšie opraviť len niekoľko prvých chýb a potom skúsiť kompiláciu znova. Tiež si všimnite, že zvýrazňovanie syntaxe veľmi pomáha pri predchádzaní chýb. Napríklad, na poslednom riadku sme napísali **Of** namiesto **Off**. Pretože to nie je známy príkaz, nebol zvýraznený.

Existujú však aj chyby, ktoré kompilátor nedokáže odhaliť. Ak sme mali napísať **OUT_B**, spôsobí to spustenie nesprávneho motora. Ak teda robot predvádza neočakávané správanie, najčastejšie je niečo zlé vo vašom programe.

6. Zmena rýchlosti

Ako ste si iste všimli, robot sa posúval dosť rýchlo. Pre zmenu rýchlosti pohybu stačí zmeniť druhý parameter v zátvorke. Výkon motora je číslo v rozmedzí 0 a 100. 100 je najrýchlejšie, 0 znamená stop (NXT servomotory budú držať pozíciu). Tu je nová verzia programu, v ktorom sa robot pohybuje pomalšie:

```
task main()
{
    OnFwd(OUT_AC, 30);
    Wait(4000);
    OnRev(OUT_AC, 30);
```

```
    Wait(4000);
    Off(OUT_AC);
}
```

7. Zhrnutie

V tejto kapitole ste napísali svoj prvý program v NXC, pomocou BricxCC. teraz viete ako napísať program, ako ho stiahnuť do robota a ako nechať robota vykonávať zadaný program. BricxCC môže robiť oveľa viac vecí, ktorých popis je dostupný v dokumentácii BricxCC. Tento turotiál sa bude zaoberať najmä jazykom NXC a spomínať budeme len tie vlastnosti BricxCC, ktoré budeme skutočne potrebovať.

Naučili ste sa aj niektoré dôležité aspekty jazyka NXC. Naučili ste sa najmä, že každý program musí mať aspoň jednu úlohu, nazvanú **main**, ktorá je vždy vykonávaná robotom. Naučili ste sa tiež tri základné príkazy motora: **OnFwd()**, **OnRev()** a **Off()**. A nakoniec ste sa naučili o príkaze **Wait()**.

II. Zaujímavejší program

Náš prvý program nebol veľmi ohurujúci. Tak teraz skúsme vytvoriť niečo oveľa zaujímavejšie. Urobíme to v niekoľkých krokoch, pričom si ukážeme niektoré dôležité vlastnosti programovacieho jazyka NXC.

1. Zatáčanie

Svojho robota môžete zatočiť zastavením alebo obrátením smeru otáčania jedného z dvoch motorov. Tu je príklad. Napíšte ho do editora, skompilujte, stiahnite do svojho robota a spusťte. Robot musí ísť chvíľu rovno a potom urobí 90° otočku vpravo.

```
task main()
{
    OnFwd(OUT_AC, 75);
    Wait(800);
    OnRev(OUT_C, 75);
    Wait(360);
    Off(OUT_AC);
}
```

Možno budete musieť vyskúšať trochu odlišné hodnoty (napr. 500) v druhom príkaze **Wait()**, aby ste zaistili 90 stupňové otočenie. Táto hodnota závisí aj type povrchu, na ktorom sa robot pohybuje. Lepšie však, ako meniť túto hodnotu v programe je použiť pre toto číslo nejaké meno. V NXC možno definovať konštantné hodnoty tak, ako je to ukázané v nasledujúcom príklade:

```
#define MOVE_TIME    1000
#define TURN_TIME    360

task main()
{
    OnFwd(OUT_AC, 75);
    Wait(MOVE_TIME);
    OnRev(OUT_C, 75);
    Wait(TURN_TIME);
    Off(OUT_AC);
}
```

Prvé dva riadky definujú konštanty. Tieto konštanty teraz možno použi v programe. Definovanie konštánt má dve výhody – robia program lepšie čitateľný a je jednoduchšie zmeniť ich hodnoty. Všimnite si, že BricxCC dáva príkazu define vlastnú farbu. Ako uvidíte v Kapitole VI, je možné definovať aj iné veci ako konštanty.

2. Opakovanie príkazov

Poďme vyskúšať napísať program, ktorý bude riadiť robota tak, aby sa pohyboval v štvorci. Chodiť v štvorci znamená: posunúť dopredu, otočiť o 90°, znova posunúť dopredu, otočiť o 90°, atď. Predchádzajúce časti kódu môžeme opakovať štyri krát, ale môžeme to urobiť aj jednoduchšie, pomocou príkazu **repeat**.

```
#define MOVE_TIME    500
#define TURN_TIME    500

task main()
{
    repeat(4)
    {
        OnFwd(OUT_AC, 75);
        Wait(MOVE_TIME);
        OnRev(OUT_C, 75);
        Wait(TURN_TIME);
    }
    Off(OUT_AC);
}
```

Číslo v zátvorkách príkazu **repeat** udáva koľko krát má byť kód v zložených zátvorkách opakovaný. Všimnite si tiež, že vo vyššie uvedenom príklade príkazy odsadzujeme. Toto odsadzovanie nie je potrebné, ale takto je program lepšie čitateľný.

Ako posledný príklad si ukážme ako prikázať robotovi aby urobil 10 štvorcov, teda 10 krát opakoval obídenie štvorca. Tu je program:

```
#define MOVE_TIME    1000
#define TURN_TIME    500

task main()
{
    repeat(10)
    {
            repeat(4)
            {
                OnFwd(OUT_AC, 75);
                Wait(MOVE_TIME);
                OnRev(OUT_C, 75);
                Wait(TURN_TIME);
            }
    }
    Off(OUT_AC);
```

```
}
```

Ako vidíte, použili sme jeden príkaz **repeat** vo vnútri iného. Hovoríme tomu „vnorený“ príkaz **repeat**. Príkazy **repeat** môžete vnárať koľkokrát chcete. Dajte však pozor na na zložené zátvorky a odsadenie, použité vo vnútri programu. Úloha začína na prvej zátvorke a končí na poslednej. Prvý príkaz **repeat** začína na druhej zátvorke a končí na piatej. Vnorený príkaz **repeat** začína na tretej zátvroke a končí na štvrtej. Ako ste si iste všimli, zátvorky sú spracovávané v pároch a časti kódu medzi príslušnými zátvorkami sme odsadili.

3. Pridanie komentárov

Aby bol náš program ešte viac čitateľný, je dobrou praktikou pridať do neho nejaké komentáre. Vždy, keď pridáte do riadku **//**, je zvyšok riadku ignorovaný a môže byť použitý ako komentár. Dlhé komentáre, ktoré sú dlhšie ako jeden riadok, možno pridať medzi **/*** a ***/**. Komentáre sú v BricxCC zvýraznené samostatnou farbou. Celý program by potom mohol vyzerať takto:

```
/*     10 SQUARES
Tento program robí 10 štvorcov
*/

#define MOVE_TIME    500        // Čas pre priamy posun
#define TURN_TIME    360        // Čas pre otočenie o 90°

task main()
{
    repeat(10)                  // Urobiť 10 štvorcov
    {
        repeat(4)
        {
            OnFwd(OUT_AC, 75);
            Wait(MOVE_TIME);
            OnRev(OUT_C, 75);
            Wait(TURN_TIME);
        }
    }
    Off(OUT_AC);                  // Vypnúť motory
}
```

4. Zhrnutie

V tejto kapitole ste sa naučili ako používať príkaz **repeat** a ako používať komentáre. Dozvedeli ste sa tiež o funkcii vnorených zátvoriek a o použití odsadenia. Pomocou týchto vedomostí teraz dokážete urobiť robota, ktorý sa pohybuje po skoro akejkoľvek ceste. Dobrým cvičením, pred pokračovaním alšou kapitolou, je skúsiť a napísať rôzne obmeny programu z tejto kapitoly.

III. Použitie premenných

Premenné sú veľmi dôležitou zložkou každého programovacieho jazyka. Premenné sú miesta v pamäti, kde môžeme uchovávať hodnotu. Túto hodnotu potom môžeme použiť na rôznych miestach programu a môžeme ju podľa potreby meniť. Poďme si popísať použite premenných na príklade.

1. Pohyb po špirále

Povedzme, že chceme upraviť predchádzajúci program takým spôsobom, aby sa robot pohyboval po špirále. Pohyb po špirále môžeme dosiahnuť tak, že pri každom priamom posune zväčšíme čas čakania. Inými slovami, zakaždým chceme zvýšiť hodnotu **MOVE_TIME**. Ale ako dosiahnuť? **MOVE_TIME** je konštanta, a preto sa jej hodnota nemôže meniť. Takže namiesto konštanty potrebujeme premennú. V NXC možno premenné definovať jednoducho. Takto by vyzeral program pre špirálu:

```
#define TURN_TIME    360

int move_time;                  // definícia premennej

task main()
{
    move_time = 200;            // nastavenie východzej hodnoty
    repeat(50)
    {
        OnFwd(OUT_AC, 75);
        Wait(move_time);        // použitie premennej na čakanie
        OnRev(OUT_C, 75);
        Wait(TURN_TIME);
        move_time += 200;       // zvýšenie hodnoty premennej
    }
    Off(OUT_AC);
}
```

Dôležité riadky sú označené komentármi. Najprv definujeme premennú zadaním kľúčového slova **int** , nasledovaného menom, ktoré sme si zvolili (zvyčajne sú používané mená s malými písmenami pre premenné a s veľkými pre konštanty, ale nie je to podmienkou). Meno musí začínať písmenom, ale môže obsahovať číslice a znak podčiarkovníka. Nie sú dovolené žiadne iné znaky (rovnaké pravidlo platí aj pre mená konštánt, mená úloh, atď.) Slovo **int** znamená celé číslo (*integer*). V tejto premennej teda môžu byť uchovávané len celé čísla. V ďalšom riadku priraďujeme do premennej hodnotu 200. Od tohto okamžiku, vždy keď použijeme premennú, jej hodnota bude 200. Nasleduje

cyklus **repeat**, v ktorom používame premennú na nastavenie času čakania v príkaze **wait** a na konci každého cyklu zvýšime hodnotu premennej o 200. Takže po prvý krát robot čaká 200 ms, druhý krát 400 ms, tretí krát 600 ms a tak ďalej.

Okrem pripočítavania hodnôt k premennej, môžeme premennú tiež násobiť číslom pomocou ***=**, odpočítavať pomocou **-=** , či deliť pomocou **/=** (pozor, pri delení je výsledok zaokrúhlený na najbližšie celé číslo). Možno tiež pripočítať jednu premennú k inej, alebo vytvoriť oveľa komplikovanejší vzorec. Nasledujúci príklad nijako nepracuje s hardvérom robota, pretože zatiaľ nevieme ako použiť displej NXT!

```
int aaa;
int bbb,ccc;
int values[];

task main()

{
    aaa = 10;
    bbb = 20 * 5;
    ccc = bbb;
    ccc /= aaa;
    ccc -= 5;
    aaa = 10 * (ccc + 3);     // aaa je teraz rovné 80
    ArrayInit(values, 0, 10); // inicializuje 10 prvkov = 0
    values[0] = aaa;
    values[1] = bbb;
    values[2] = aaa*bbb;
    values[3] = ccc;
}
```

Na druhom riadku si všimnite, že možno definovať viac premenných v jednom riadku. Takže by sme pokojne mohli všetky tri prvé riadky skombinovať do jedného. Premenná pomenovaná **values** je pole, teda premenná, ktorá obsahuje viac čísel: pole môže byť indexované pomocou čísla v lomených zátvorkách. V NXC sú čelo-číselné polia potom deklarované takto:

```
int name[];
```

Nasledujúci riadok potom alokuje (vyhradzuje v pamäti) 10 prvkov a inicializuje ich na hodnotu 0.

```
ArrayInit(values, 0, 10);
```

2. Náhodné čísla

V predchádzajúcich programoch sme presne definovali čo a ako má robot robiť. Ale veci vyzerajú oveľa zaujímavejšie, keď robot robí veci, ktoré nepoznáme. Chceme dostať do jeho pohybu nejakú náhodnosť. V NXC môžeme vytvoriť náhodné čísla. Nasledujúci program používa náhodné čísla, takže sa robot pohybuje náhodným spôsobom. Rovnomerne sa posúva dopredu náhodne dlhý čas a potom robí náhodné otočenie:

```
int move_time, turn_time;

task main()
{
    while(true)
    {
        move_time = Random(600);
        turn_time = Random(400);
        OnFwd(OUT_AC, 75);
        Wait(move_time);
        OnRev(OUT_A, 75);
        Wait(turn_time);
    }
}
```

Program definuje dve premenné a prirauje im náhodné hodnoty. Príkaz **Random(600)** vygeneruje náhodné číslo v rozsahu medzi 0 a 600 (maximálna hodnota nie je vo vrátenej hodnote zahrnutá). Vždy keď zavoláme **Random**, bude vrátené číslo iné.

Všimnite si, že sa môžeme vyhnúť použitiu premenných pomocou priameho zápisu, napr. **Wait(Random(600))**.

V programe si môžete tiež všimnúť nový typ cyklu. namiesto použitia príkazu **repeat** sme použili **while(true)**. Príkaz **while** opakuje príkazy v cykle dovtedy, kým je splnená podmienka v zátvorkách. Špeciálne slovo **true** je vždy **true** (splnené), takže príkazy v zložených zátvorkách sú opakovaný stále dookola (presnejšie dokiaľ nestlačíte tmavo sivé tlačidlo na NXT). O príkaze **while** budeme rozprávať viac v Kapitole IV.

3. Zhrnutie

V tejto kapitole ste sa naučili o používaní premenných a polí. Môžete tiež definovať iné typy dát ako **int**: **short**, **long**, **byte**, **bool** a **string**.

Naučili ste sa aj ako vytvoriť náhodné čísla, takže môžeme robotovi naprogramovať nepredpovedateľné správanie. Nakoniec ste sa dozvedeli o použití príkazu **while** pre vytvorenie nekonečného cyklu, ktorý nikdy nekončí.

IV. Riadiace štruktúry

V predchádzajúcich kapitolách sme spomínali príkazy **repeat** a **while**. Tieto príkazy riadia spôsob akým sú vykonávané ostatné príkazy v programe. Preto sa nazývajú „riadiace štruktúry“. V tejto kapitole si ukážeme niektoré ďalšie riadiace štruktúry.

1. Príkaz if

Niekedy je potrebné, aby bola niektorá časť programu vykonaná iba v určitej situácii. Vtedy je použitý príkaz **if**. Ukážme si to na príklade. Znova budeme meniť program, s ktorým pracujeme už dlhšie, tentokrát však s novou vlastnosťou. Chceme aby robot sledoval priamu čiaru a potom zatočil vľavo alebo vpravo. Na túto úlohu znova využijeme náhodné čísla. Necháme vygenerovať náhodné číslo, ktoré bude buď kladné alebo záporné. Ak bude číslo menšie ako 0 urobíme zatáčku vpravo; inak urobíme zatáčku vľavo. Takto vyzerá program:

```
#define MOVE_TIME            500
#define TURN_TIME            360

task main()
{
    while(true)
    {
        OnFwd(OUT_AC, 75);
        Wait(MOVE_TIME);
        if (Random() >= 0)
        {
            OnRev(OUT_C, 75);
        }
        else
        {
            OnRev(OUT_A, 75);
        }
        Wait(TURN_TIME);
    }
}
```

Príkaz **if** vyzerá podobne ako príkaz **while**. Ak je podmienka v zátvorke splnená (*true*) je vykonaná časť programu medzi zloženými zátvorkami. Ak podmienka splnená nie je, je vykonaná časť programu v zložených zátvorkách za slovom **else**. Pozrime sa lepšie na použitú podmienku. Zápis **Random() >= 0** znamená, že **Random()** musí byť väčšie alebo rovné 0, aby bola podmienka splnená. Hodnoty možno porovnávať rôznymi spôsobmi. Tu sú najčastejšie porovnania:

- **==** rovná sa
- **<** menšie ako
- **<=** menšie alebo rovné
- **>** väčšie ako
- **>=** väčšie alebo rovné
- **!=** nerovná sa

Podmienky možno spájať pomocou **&&**, čo znamená „a zároveň“ (and), alebo pomocou **||**, čo znamená „alebo“ (or). Takto vyzerá niekoľko príkladov podmienok:

- **true** vždy splené (*true*)
- **false** nikdy nesplnené
- **ttt != 3** splnené, ak sa **ttt** nerovná 3
- **(ttt >= 5) && (ttt <= 10)** splnené, keď je **ttt** medzi 5 a 10
- **(aaa == 10) || (bbb == 10)** splnené, ak sa **aaa** alebo **bbb** (alebo obe) rovnajú 10

Všimnite si, že príkaz **if** má dve časti. Časť ihneď za podmienkou, ktorá je vykonaná v prípade splnenia podmienka, a časť za slovom **else**, ktorá je vykonaná v prípade nesplnenia podmienky. Kľúčové slovo **else** a časť príkazu za ním je voliteľná, takže môže byť vynechaná, ak v prípade nesplneia podmienky nie je potrebné nič vykonať.

2. Príkaz do

Ďalšia riadiaca štruktúra, príkaz **do**, má nasledujúcu formu:

```
do
{
    statements;
}
while (condition);
```

Príkazy medzi zloženými zátvorkami za príkazom **do** je vykonávaná tak dlho, kým je splnená podmienka za slovom **while**. Podmienka má rovnaký tvar ako v prípade príkazu **if**. Ako príklad v programe môže poslúžiť prípad, keď sa má robot pohybovať v náhodných kruhoch 20 sekúnd a potom skončiť.

```
int move_time, turn_time, total_time;

task main()
{
    total_time = 0;
    do
    {
```

```
        move_time = Random(1000);
        turn_time = Random(1000);
        OnFwd(OUT_AC, 75);
        Wait(move_time);
        OnRev(OUT_C, 75);
        Wait(turn_time);
        total_time += move_time;
        total_time += turn_time;
    }
    while (total_time < 20000);
    Off(OUT_AC);
    }
```

Všimnite si, príkaz **do** sa správa veľmi podobne ako príkaz **while**. Ale v prípade príkazu **while** je podmienka testovaná pred vykonaním príkazov, ale v príkaze **do** je podmienka testovaná až po ich vykonaní. To má za následok, že pri príkaze **while** nemusia byť príkazy v cykle vykonané ani raz, ale pri príkaze **do** budú príkazy vykonané vždy aspoň raz.

3. Zhrnutie

V tejto kapitole sme si ukázali niekoľko nových riadiacich štruktúr: príkazy **if** a **do**. Spolu s príkazmi **repeat** a **while**, to sú príkazy, ktoré riadia spôsob vykonávania programu. Je veľmi dôležité aby ste pochopili ako pracujú, preto, pred tým ako budete pokračovať, je teda lepšie aby ste si sami vyskúšali niekoľko príkladov.

V. Senzory

Samozrejme môžeme k NXT pripojiť senzory, aby mohol robot reagovať na vonkajšie udalosti. Pred tým, ako si môžeme ukázať ich použitie, je treba zmeniť robota a pridať mu dotykový senzor. Rovnako ako predtým, stačí postupovať podľa inštrukcií pre Tribota, pre vybudovanie predného nárazníka.

Senzor pripojte na vstup 1 NXT.

1. Čakanie na senzor

Začnime celkom jednoduchým programom, v ktorom sa robot posúva dopredu až pokiaľ na niečo nenarazí. Takto vyzerá:

```
task main()
{
    SetSensor(IN_1,SENSOR_TOUCH);
    OnFwd(OUT_AC, 75);
    until (SENSOR_1 == 1);
    Off(OUT_AC);
}
```

V programe sú dva dôležité riadky. Prvý riadok programu oznamuje robotovi, ktorý typ senzora budeme používať. **IN_1** je číslo vstupu, ku ktorému je senzor pripojený. Ostatné senzorové vstupy sú označované **IN_2**, **IN_3** a **IN_4**. **SENSOR_TOUCH** udáva, že pripojený je dotykový senzor. Pre svetelný senzor by sme použili **SENSOR_LIGHT**.

Po nastavení typu senzora program zapína oba motory a robot sa začína pohybovať vpred. Nasledujúci príkaz je veľmi užitočná konštrukcia. Čaká dovtedy, až kým nie je splnená zadaná podmienka. Táto podmienka udáva, že hodnota na senzore **SENSOR_1**

musí byť 1, čo v tomto prípade znamená, že senzor je stlačený. Keď senzor nie je stlačený, hodnota je 0. Takže tento príkaz čaká na stlačenie senzora. Potom vypne motory a úloh končí.

2. Akcia pri stlačení senzora

Poďme teraz vyskúšať ako urobiť aby robot vyhol prekážke. Vždy keď robot narazí na objekt, necháme ho trochu sa vrátiť, otočiť sa a potom pokračovať. Takto vyzerá program:

```
task main()
{
    SetSensorTouch(IN_1);
    OnFwd(OUT_AC, 75);
    while (true)
    {
        if (SENSOR_1 == 1)
        {
            OnRev(OUT_AC, 75); Wait(300);
            OnFwd(OUT_A, 75); Wait(300);
            OnFwd(OUT_AC, 75);
        }
    }
}
```

Ako v predchádzajúcom príklade, najprv nastavíme typ senzora. Potom sa robot začína pohybovať vpred a v nekonečnom cykle **while** pravidelne testujeme či je stlačený senzor, a ak áno, tak sa robot trochu vráti späť, otočí vpravo a potom znova pokračuje vpred.

3. Svetelný senzor

Okrem dotykového senzora, možno v systéme NXT použiť aj svetelný senzor, zvukový senzor a digitálny ultrazvukový senzor. Svetelný senzor môže byť použitý s vlastným svetelným zdrojom alebo bez neho, takže možno merať množstvo odrazeného svetla alebo okolité svetlo v príslušnom smere. Meranie odrazeného svetla užitočné najmä pri vytváraní robota, ktorý sleduje čiaru na podlahe. Presne toto budeme robiť v nasledujúcom príklade. Pre tieto experimenty, dokončime Tribota. Pripojíme svetelný senzor na vstup 3, zvukový senzor na vstup 2 a ultrazvukový senzor na vstup 4, ako je popísané v inštrukciách.

Potrebujeme tiež testovaciu plochu s čiernou stopou, ktorá je súčasťou súpravy NXT. Základným princípom pre sledovanie čiary je, že sa robot pokúša udržať pri pravej hrane čiernej čiary, otočiac sa preč z čiary, ak je svetelná intenzita je príliš nízka (keď je senzor uprostred čiary) a otáčaním sa k čiare, ak je senzor mimo čiary a deteguje vysokú úroveň svetla. Takto vyzerá veľmi jednoduchý program, ktorý zaistí sledovanie pomocou jednej hraničnej úrovne svetla:

```
#define THRESHOLD 40

task main()
{
    SetSensorLight(IN_3);
    OnFwd(OUT_AC, 75);
    while (true)
    {
        if (Sensor(IN_3) > THRESHOLD)
        {
            OnRev(OUT_C, 75);
            Wait(100);
            until(Sensor(IN_3) <= THRESHOLD);
            OnFwd(OUT_AC, 75);
        }
    }
}
```

Program najprv konfiguruje port 3 ako svetelný senzor. Potom nastavuje robota na posun vpred a vstupuje do nekonečnej slučky. Vždy keď je svetelná hodnota vyššia ako 40

(používame konštantu, takže hodnota môže byť jednoducho prispôsobená, pretože dosť závisí na množstve okolitého svetla) zmeníme smer otáčania jedného motora a počkáme kým sa nevrátime znova do stopy.

Ako sami uvidíte po spustení programu, pohyb nie je veľmi plynulý. Skúste pridať príkaz **Wait(100)** pred príkaz **until**, aby sa robot pohyboval lepšie. Všimnite si, že program nepracuje pri pohybe proti smeru hodinových ručičiek. Pre umožnenie pohybu po ľubovoľnej ceste je potrebný oveľa zložitejší program.

Pre čítanie intenzity okolitého svetla s vypnutou LED, nakonfigurujte senzor takto:

```
SetSensorType(IN_3,IN_TYPE_LIGHT_INACTIVE);
SetSensorMode(IN_3,IN_MODE_PCTFULLSCALE);
ResetSensor(IN_3);
```

4. Zvukový senzor

Napíšeme program, ktorý počká na hlasný zvuk, potom sa robot začne pohybovať, až kým nie je zistený ďalší zvuk. Zvukový senzor pripojte na port 2, ako je popísané v inštrukciách pre Tribot.

```
#define THRESHOLD 40
#define MIC SENSOR_2

task main()
{
    SetSensorSound(IN_2);
    while(true){
        until(MIC > THRESHOLD);
        OnFwd(OUT_AC, 75);
        Wait(300);
        until(MIC > THRESHOLD);
        Off(OUT_AC);
        Wait(300);
    }
}
```

V programe najprv definujeme konštantu **THRESHOLD** a alias pre **SENSOR_2**; v úlohe **main** nastavujeme port 2 na čítanie dát zo zvukového senzora a vstupujeme do nekonečnej slučky.

Pomocou príkazu **until** program čaká na zvuk s úrovňou vyššou ako hraničná hodnota, ktorú sme si zvolili. Pozor, **SENSOR_2** nie je len meno, ale makro, ktoré vracia hodnotu úrovne zvuku, prečítanú zo senzora.

Ak nastane hlasný zvuk, robot sa začne pohybovať vpred, až dokiaľ ho ďalší zvuk nezastaví.

Príkaz **wait** musel byť vložený, pretože inak robot a začne a okamžite skočí: v skutočnosti, NXT je také rýchle, že vykonanie riadkov medzi dvomi príkazmi **until** nezaberie skoro žiadny čas. Ak vyskúšate zakomentovať prvý i druhý **wait**, pochopíte to lepšie. Ako alternatívu k príkazu **until**, možno na čakanie na udalosť použiť aj príkaz **while**, stačí len použiť opačnú podmienku, napr.:

```
while(MIC <= THRESHOLD).
```

O analógových senzoroch NXT už nie je na naučenie veľa ďalšieho; len nezabudnite, že oba, svetelný i zvukový senzor vracajú pri čítaní hodnotu v rozsahu od 0 do 100.

5. Ultrazvukový senzor

Ultrazvukový senzor pracuje približne ako sonar, posiela skupinu ultrazvukových vĺn a meria čas, ktorý potrebný na to, aby sa vlny odrazili od predmetu v dohľade a vrátili späť. Je to digitálny senzor, v zmysle, že má integrované zariadenie na analýzu a posielanie dát. S týmto senzorom môžete urobiť robota, ktorý vidí a vyhýba sa prekážkam bez toho by sa ich dotkol (ako to bolo v prípade dotykového senzora).

```
#define NEAR 15 //cm

task main(){
    SetSensorLowspeed(IN_4);
    while(true){
        OnFwd(OUT_AC,50);
        while(SensorUS(IN_4)>NEAR);
        Off(OUT_AC);
        OnRev(OUT_C,100);
        Wait(800);
    }
}
```

Program inicializuje port 4 na čítanie dát z digitálneho ultrazvukového senzora, potom vstupuje do nekonečnej slučky, v ktorej sa robot pohybuje priamo, až kým sa v zornom poli neobjaví niečo bližšie ako **NEAR** cm (v našom príklade 15 cm), potom sa trochu otočí a znova pokračuje v priamom pohybe.

6. Zhrnutie

V tejto kapitole ste videli ako pracovať so všetkými senzormi, ktoré sú zahrnuté v súprave NXT. Ukázali sme si aj, aké užitočné sú pre prácu so senzormi príkazy **until** a **while**.

Odporúčam vám, aby ste si teraz sami napísali viacero programov. Máte všetky potrebné ingrediencie na to, aby ste naučili svojho robota trochu komplikovanejšiemu

správaniu: skúste preložiť do NXC jednoduchšie programy, ktoré sú k dispozícii v programovacej príručke NXT Robo Center.

VI. Úlohy a podprogramy

Až doteraz mali všetky naše programy len jednu úlohu. Ale NXC programy môžu mať viacero úloh. Je tiež možné umiestniť časti programu do takzvaných podprogramov, takže ich možno použiť na viacerých miestach programu. Použitie úloh a podprogramov robí programy lepšie pochopiteľnými a kompaktnejšími. V tejto kapitole sa pozrieme na rôzne možnosti.

1. Úlohy

Program NXC môže mať až 255 úloh, pričom každá musí mať jedinečné meno. Vždy musí existovať úloha s menom **main**, pretože to je úloha, ktorá je vykonávaná ako prvá. Ostatné úlohy budú vykonávané len ak ich bežiaca úloha zavolá alebo ak sú vyslovene naplánované v úlohe **main**. Úloha **main** musí byť pred ich štartom prerušená. Od tohto okamžiku sú obe úlohy bežiace simultánne.

Ukážme si použitie úloh. Chceme urobiť program, v ktorom sa robot pohybuje v štvorcoch, ako predtým. Ale keď narazí na prekážku, musí na ňu reagovať. Je ťažké urobiť to v jednej úlohe, pretože robot musí robiť dve veci naraz: posúvať sa dokola (teda zapínať a vypínať motory) a čakať na senzory. Preto je lepšie na toto použiť dve úlohy, jedna úloha ktorá sa stará o pohyb v štvorcoch a druhá, ktorá reaguje na senzory. Takto vyzerá program:

```
mutex moveMutex;

task move_square()
{
    while (true)
    {
        Acquire(moveMutex);
        OnFwd(OUT_AC, 75); Wait(1000);
        OnRev(OUT_C, 75); Wait(500);
        Release(moveMutex);
    }
}

task check_sensors()
{
    while (true)
    {
        if (SENSOR_1 == 1)
        {
            Acquire(moveMutex);
```

```
            OnRev(OUT_AC, 75); Wait(500);
            OnFwd(OUT_A, 75); Wait(500);
            Release(moveMutex);
        }
    }
}

task main()
{
    Precedes(move_square, check_sensors);
    SetSensorTouch(IN_1);
}
```

Úloha **main** len nastavuje typ senzora a potom spúšťa obe ostatné úlohy, pomocou ich pridania do fronty plánovača; po tomto úloha **main** končí. Úloha **move_square** v nekonečnom cykle presúva robota v štvorcoch. Úloha **check_sensors** kontroluje, či je stlačený dotykový senzor a ak áno, tak riadi robot preč od prekážky.

Je veľmi dôležité man na pamäti, že spustené úlohy bežia naraz, v rovnakom okamihu a to môže viesť k neočakávaným výsledkom, ak sa obe úlohy pokúšajú riadiť motory (a to je ich úlohou).

Aby sme predišli týmto problémom, definovali sme zvláštny typ premennej, **mutex** (čo značí vzájomné vylúčenie – *mutual exclusion*): s týmto typom premenných môžeme pracovať len pomocou funkcií **Acquire** a **Release**. Písanie kritických častí kódu medzi tieto funkcie zaistí, že v jednom okamžiku môže motory ovládať len jedna úloha.

Tieto mutexové premenné sú nazývané aj semafory a táto programovacia technika sa volá konkurenčné programovanie. Paralelné programovanie je podrobnejšie rozobraté v Kapitole X.

2. Podprogramy

Niekedy je potrebné mať rovnaké časti kódu na viacerých miestach programu. V takom prípade možno tieto časti kódu umiestniť do podprogramu a pomenovať ich. Od tejto chvíle je možné tieto časti kódu vykonávať jednoduchým volaním ich mena v úlohe. Pozrime sa na príklad.

```
sub turn_around(int pwr)
{
    OnRev(OUT_C, pwr);
    Wait(900);
    OnFwd(OUT_AC, pwr);
}
```

```
task main()
{
    OnFwd(OUT_AC, 75);
    Wait(1000);
    turn_around(75);
    Wait(2000);
    turn_around(75);
    Wait(1000);
    turn_around(75);
    Off(OUT_AC);
}
```

V tomto programe máme definovaný podprogram, ktorý zaistí otočenie robota okolo jeho stredu. Úloha **main** volá tento podprogram tri krát. Všimnite si, že podprogram voláme napísaním jeho mena a poslaním číselného argumentu, ktorý je napísaný v zátvorke. Ak podprogram neprijíma argumenty, stačí za meno podprogramu pridať prázdne zátvorky. Volanie podprogramu teda vyzerá ako mnoho príkazov, ktoré sme si ukázali.

Hlavnou výhodou podprogramov je, že sú v pamäti NXT uložené len raz a tak pamäť šetria. Keď sú však podprogramy krátke, môže byť lepšie použiť namiesto toho funkciu **inline**. Takéto funkcie nie sú uchovávané samostatne, ale sú kopírované na miesto, kde majú byť použité. Takto je použité viac pamäte, ale neexistuje obmedzenie počtu inline funkcií. Takéto podprogramy môžu byť definované takto:

```
inline int Name( Args ) {
    //body;
    return x*y;
}
```

Definovanie a volanie inline funkcií je úplne rovnaké ako v prípade podprogramov. Takže predchádzajúci príklad pomocou inline funkcie by vyzeral takto:

```
inline void turn_around()
{
    OnRev(OUT_C, 75);
    Wait(900);
    OnFwd(OUT_AC, 75);
}

task main()
{
    OnFwd(OUT_AC, 75);
    Wait(1000);
    turn_around();
```

```
    Wait(2000);
    turn_around();
    Wait(1000);
    turn_around();
    Off(OUT_AC);
}
```

V predchádzajúcom príklade, je funkcia bez prametra. V nasledujúcom príklade pridáme funkcii parametre:

```
inline void turn_around(int pwr, int turntime)
{
    OnRev(OUT_C, pwr);
    Wait(turntime);
    OnFwd(OUT_AC, pwr);
}

task main()
{
    OnFwd(OUT_AC, 75);
    Wait(1000);
    turn_around(75, 2000);
    Wait(2000);
    turn_around(75, 500);
    Wait(1000);
    turn_around(75, 3000);
    Off(OUT_AC);
}
```

Všimnite si, že v zátvorke za menom inline funkcie sme zadali argument(y) funkcie. V tomto prípade definujeme, že argumenty sú celé čísla (sú aj iné možnosti) a ich meno je **pwr** a **turntime**. Keď je definovaných viacero argumentov, je treba ich oddeliť čiarkami. Zapamätajte si, že v NXC je **sub** to isté ako **void**. Samozrejme, funkcie môžu mať aj iný typ návratovej hodnoty ako **void**, môžu vracať celé číslo alebo reťazcovú hodnotu: detaily sú v príručke NXC.

3. Definícia makier

Existuje ešte jeden spôsob ako pomenovať malé časti kódu. V NXC môžete definovať makro (nemýliť si s makrami v BricxCC). Už sme si ukázali ako sa dajú definovať konštanty – pomocou **#define** a zadaním mena konštanty. Ale rovnaký spôsob možno využiť na na definovanie časti kódu. Tu je znova rovnaký program, ale tentokrát je na otáčanie použité makro:

```
#define turn_around \
```

```
    OnRev(OUT_B, 75); Wait(3400); OnFwd(OUT_AB, 75);

task main()
{
    OnFwd(OUT_AB, 75);
    Wait(1000);
    turn_around;
    Wait(2000);
    turn_around;
    Wait(1000);
    turn_around;
    Off(OUT_AB);
}
```

Za príkazom **#define** je slovo **turn_around**, ktoré je vlastne skratkou pre text za ním. Teraz vždy keď v kóde napíšete **turn_around**, je toto slovo nahradené zadaným textom. Dajte pozor na to, že zadaný text musí byť na jednom riadku (existujú síce spôsoby na rozdelenie príkazu **#define** na viacero riadkov, ale nie sú odporúčané) .

Príkazy #define sú trochu viac výkonné. Môžu tiež mať argumenty. Napríklad môžeme vložiť ako argument príkazu čas otočenia. Nasleduje príklad, v ktorom definujeme štyri makrá – jedno pre posúvanie vpred, jedno pre posúvanie vzad, jedno pre otočenie vľavo a jedno pre otočenie vpravo. Každé z nich má dva argumenty – rýchlosť a čas:

```
#define turn_right(s,t) \
    OnFwd(OUT_A, s);OnRev(OUT_B, s);Wait(t);
#define turn_left(s,t)  \
    OnRev(OUT_A, s);OnFwd(OUT_B, s);Wait(t);
#define forwards(s,t)   OnFwd(OUT_AB, s);Wait(t);
#define backwards(s,t)  OnRev(OUT_AB, s);Wait(t);

task main()
{
    backwards(50,10000);
    forwards(50,10000);
    turn_left(75,750);
    forwards(75,1000);
    backwards(75,2000);
    forwards(75,1000);
    turn_right(75,750);
    forwards(30,2000);
    Off(OUT_AB);
}
```

Definovať makrá je veľmi užitočné, pretože robia kód kompaktnejší a čitateľnejší. Rovnako takto umožňujú oveľa jednoduchšie meniť kód, napríklad keď zmeníte pripojenie motorov.

4. Zhrnutie

V tejto kapitole sme hovorili o úlohách, podprogramoch, inline funkciách a makrách. každé má iné použitie. Úlohy bežia v rovnakom čase a starajú sa o rôzne veci, ktoré majú byť vykonané naraz. Podprogramy sú užitočné pre väčšiu časť kódu, ktorý má byť vykonaný na viacerých miestach jednej úlohy. Inline funkcie sú zase užitočné keď sa majú časti kódu použiť na mnohých rôznach miestach v rôznych úlohách, avšak zaberajú viac miesta v pamäti. A nakoniec makrá, sú to veľmi malé časti kódu, ktoré majú byť použité na rôznych miestach. Makrá môžu mať parametre, a tak sú oveľa užitočnejšie.

VII. Tvorba hudby

NXT má zabudovaný reproduktor, ktorý môže prehrávať tóny alebo dokonca zvukové súbory. Táto vlastnosť je dôležitá najmä keď chcete urobiť aby NXT oznámil, že sa niečo udialo. Ale môže to tiež byť zábavné mať robota, ktorý hrá hudbu alebo rozpráva, kým chodí dookola.

1. Prehrávanie zvukových súborov

BricxCC má zabudovaný konvertor súborov .wav na súbory .rso, ktorý je dostupný prostredníctvom menu Tools → Sound conversion. Po konverzii možno ukladať zvukové súbory .rso vo flash pamäti NXT pomocou iného nástroja, prieskumníka NXT pamäte (Tools → NXT explorer) a prehrávať ich pomocou príkazu:

```
PlayFileEx(filename, volume, loop?)
```

Jeho argumenty sú meno zvukového súboru, hlasitosť (číslo od 0 do 4) a opakovanie. Opakovanie môže mať hodnotu **1** (TRUE) pre opakovanie prehrávania alebo **0** (FALSE) ak má byť súbor prehraný iba raz.

```
#define TIME 200
#define MAXVOL 7
#define MINVOL 1
#define MIDVOL 3
#define pause_4th Wait(TIME)
#define pause_8th Wait(TIME/2)
#define note_4th \
    PlayFileEx("! Click.rso",MIDVOL,FALSE); pause_4th
#define note_8th \
    PlayFileEx("! Click.rso",MAXVOL,FALSE); pause_8th

task main()
{
    PlayFileEx("! Startup.rso",MINVOL,FALSE);
    Wait(2000);
    note_4th;
    note_8th;
    note_8th;
    note_4th;
    note_4th;
    pause_4th;
    note_4th;
    note_4th;
    Wait(100);
```

```
}
```

Tento program naprv prehrá zvuk, ktorý počujete pri zapnutí NXT, potom používa štandardný zvuk pre kliknutie pre zahranie “Shave and a haircut”, ktorý robí Roger Rabbit! V tomto prípade sú veľmi užitočné makrá, ktoré výrazne zjednodušujú zápis úlohy **main**.

2. Prehrávanie hudby

Pre zahranie tónu možno použiť príkaz:

```
PlayToneEx(frequency, duration, volume, loop?)
```

Príkaz má štyri argumenty. Prvý je frekvencia v Hertzoch, druhý je trvanie v milisekundách (obdobne ako v príkaze wait) a zvyšné sú hlasitosť a opkavanie (rovnaké ako pri zvukových súboroch). Možno použiť aj

```
PlayTone(frequency, duration)
```

V tomto prípade je hlasitosť nastavená z NXT menu a opakovanie je vypnuté.

Prehľad užitočných frekvencií:

Sound	3	4	5	6	7	8	9
B	247	494	988	1976	3951	7902	
A#	233	466	932	1865	3729	7458	
A	220	440	880	1760	3520	7040	14080
G#		415	831	1661	3322	6644	13288
G		392	784	1568	3136	6272	12544
F#		370	740	1480	2960	5920	11840
F		349	698	1397	2794	5588	11176
E		330	659	1319	2637	5274	10548
D#		311	622	1245	2489	4978	9956
D		294	587	1175	2349	4699	9398
C#		277	554	1109	2217	4435	8870
C		262	523	1047	2093	4186	8372

Ako v prípade **PlayFileEx**, NXT nečaká na dokončenie noty. Takže, ak používate viacero tónov v rade za sebou, bude potrebné medzi nich pridať (trochu dlhšie) príkazy **wait**. Takto vyzerá príklad:

```
#define VOL 3

task main()
{
    PlayToneEx(262,400,VOL,FALSE);              Wait(500);
    PlayToneEx(294,400,VOL,FALSE);              Wait(500);
    PlayToneEx(330,400,VOL,FALSE);              Wait(500);
```

```
    PlayToneEx(294,400,VOL,FALSE);              Wait(500);
    PlayToneEx(262,1600,VOL,FALSE);             Wait(2000);
}
```

Hudbu môžete veľmi jednoducho vytvoriť pomocou Brick Piano, ktorý je súčasťou BricxCC.

Ak chcete aby NXT hralo počas pohybu hudbu, je lepšie na prehrávanie použiť samostatnú úlohu. Tu je príklad celkom zbytočného programu, v ktorom NXT posúva robota dopredu a dozadu a pri tom stále hrá hudbu:

```
task music()
{
    while (true)
    {
        PlayTone(262,400);        Wait(500);
        PlayTone(294,400);        Wait(500);
        PlayTone(330,400);        Wait(500);
        PlayTone(294,400);        Wait(500);
    }
}

task movement()
{
    while(true)
    {
        OnFwd(OUT_AC, 75); Wait(3000);
        OnRev(OUT_AC, 75); Wait(3000);
    }
}

task main()
{
    Precedes(music, movement);
}
```

3. Zhrnutie

V tejto kapitole ste sa naučili ako nechať NXT prehrávať zvuky a hudbu. Tiež ste sa naučili ako na prehrávanie hudby použiť samostatnú úlohu.

VIII. Viac o motoroch

Existuje niekoľko ďalších príkazov pre riadenie motora, pomocou ktorých možno riadiť motory oveľa presnejšie. V tejto kapitole si popíšeme: **ResetTachoCount**, **Coast** (Float), **OnFwdReg**, **OnRevReg**, **OnFwdSync**, **OnRevSync**, **RotateMotor**, **RotateMotorEx** a základné koncepty PID.

1. Plynulé zastavenie

Pri použití príkazu **Off()** sú servomotory okamžite zastavené, ich osi zabrzdené a motory uchovávajú pozíciu. Motory však možno zastaviť aj oveľa elegantnejšie, bez použitia bŕzd. Na tento účel slúžia príkazy **Float()** alebo **Coast()**, ktoré jdnoducho vypnú napájanie motora. Nasledujúci príklad najprv zastaví motory pomocou bŕzd a potom bez bŕzd. Sledujte rozdiel v zastavení, ktorý však nemusí byť s týmto typom robta veľmi viditeľný, ale môže byť oveľa viditeľnejší pri inom type.

```
task main()
{
    OnFwd(OUT_AC, 75);
    Wait(500);
    Off(OUT_AC);
    Wait(1000);
    OnFwd(OUT_AC, 75);
    Wait(500);
    Float(OUT_AC);
}
```

2. Pokročilé príkazy

Príkazy **OnFwd()** a **OnRev()** sú najjednoduchšie programy na riadenie motorov. Ale NXT servomotory majú zabudovaný kodér, pomocou ktorého možno oveľa presnejšie riadiť pozíciu natočenia a rýchlosť; NXT firmware implementuje **PID** (*Proportional Integrative Derivative*) radič s uzatvoreným cyklom na riadenie pozície osí motora a rýchlosti pomocou kodéra ako spätnej väzby.

Ak chcete aby sa robot pohyboval presne rovno, môžete použiť vlastnosť synchronizácie, ktorá zaistí, že sa zvolená skupina motorov bude otáčať spoločne – teda budú jeden na druhý čakať, ak bol jeden spomalený, prípadne zablokovaný. Podobným spôsobom možno skupinu motorov spustiť spolu synchronizovane, ale s percentuálnym údajom pre zatočenie vľavo, vpravo, či otáčanie sa na mieste. Ale stále synchronizovane! Pre získanie plnej moci nad motormi je teda využiť ďalšie príkazy:

OnFwdReg(‘porty',‘rýchlosť',‘režim') ovláda motory, zadané pomocou portov zadanou rýchlosťou, ale aplikuje regulačný režim, ktorý môže byť jednou z hodnôt **OUT_REGMODE_IDLE**, **OUT_REGMODE_SPEED** alebo **OUT_REGMODE_SYNC**. Ak je zvolený režim **IDLE**, nie je aplikovaná regulácia PID. Ak je zvolený režim **SPEED**, NXT reguluje jeden motor tak, aby dosahoval konštantnú rýchlosť, hoci sa záťaž motora mení. A nakoniec, ak je zvolený režim **SYNC**, je zadaná skupina motorov pri otáčaní synchronizovaná.

OnRevReg() vykonáva presne tú istú funkciu, ale pre spätný chod motorov.

```
task main()
{
    OnFwdReg(OUT_AC,50,OUT_REGMODE_IDLE);
    Wait(2000);
    Off(OUT_AC);
    PlayTone(4000,50);
    Wait(1000);
    ResetTachoCount(OUT_AC);
    OnFwdReg(OUT_AC,50,OUT_REGMODE_SPEED);
    Wait(2000);
    Off(OUT_AC);
    PlayTone(4000,50);
    Wait(1000);
    OnFwdReg(OUT_AC,50,OUT_REGMODE_SYNC);
    Wait(2000);
    Off(OUT_AC);
}
```

Tento program ukazuje rôzne spôsoby regulácie, ak sa pokúsite rukami zastaviť kolesá: prvý režim (IDLE) – pri zastavení kolesa si nič nevšimnete; pri druhom režime (SPEED) sa pri pokuse zastaviť koleso bude robot snažiť zvýšiť výkon motora tak, aby prekonal vaše držanie s cieľom udržať konštantnú rýchlosť otáčania. Nakoniec, pri treťom režime (SYNC), spôsobí zastavenie jedného kolesa i zastavenie druhého, ktoré bude „čakať“ na to blokované.

OnFwdSync(‘porty',‘rýchlosť',‘otočenie') je vlastne rovnaký príkaz ako **OnFwdReg()** v režime SYNC, ale tentokrát možno zadať parameter, ktorý umožňuje synchronizované otáčanie v percentách (od -100 do 100).

OnRevSync() je rovnaký príkaz ako predchádzajúci, len pre spätný chod motorov. Nasledujúci program ukazuje tieto príkazy – skúste si zmeniť hodnotu otočného pomeru a sledujte zmeny správania:

```
task main()
{
```

```
    PlayTone(5000,30);
    OnFwdSync(OUT_AC,50,0);
    Wait(1000);
    PlayTone(5000,30);
    OnFwdSync(OUT_AC,50,20);
    Wait(1000);
    PlayTone(5000,30);
    OnFwdSync(OUT_AC,50,-40);
    Wait(1000);
    PlayTone(5000,30);
    OnRevSync(OUT_AC,50,90);
    Wait(1000);
    Off(OUT_AC);
}
```

Nakoniec, motory môžu byť otáčané o zadaný počet stupňov (nezabudnite, že celé otočenie je 360°). V oboch nasledujúcich príkazoch možno zmeniť smer otáčania bu zmenou znamienka rýchlosti, alebo zmenou znamienka uhla otočenia. Takže ak uhol i rýchlosť majú rovnaké znamienko, motor bude spustený vpred, ak budú ich znamienka rôzne, moor bude spustený dozadu.

RotateMotor(‘porty',‘rýchlosť',‘stupne') otáča os motora zadaného pomocou portu o zadaný uhol zadanou rýchlosťou (v rozsahu 0 až 100).

```
task main()
{
    RotateMotor(OUT_AC, 50,360);
    RotateMotor(OUT_C, 50,-360);
}
```

RotateMotorEx(‘porty', ‘rýchlosť', ‘stupne', ‘otočenie', ‘sync', 'stop') je rozšírením predchádzajúceho príkazu, pomocou ktorého môžeme synchronizovať dva motory (napr. OUT_AC) zadaním percenta otáčania (od -100 do 100) a logickou hodnotou ‘sync' (ktorá môže byť true alebo false). Možno tiež určiť spôsob zastavenia motorov,a to pomocou zabrzdenia alebo bez neho, pomocou logického parametra 'stop'.

```
task main()
{
    RotateMotorEx(OUT_AC, 50, 360, 0, true, true);
    RotateMotorEx(OUT_AC, 50, 360, 40, true, true);
    RotateMotorEx(OUT_AC, 50, 360, -40, true, true);
    RotateMotorEx(OUT_AC, 50, 360, 100, true, true);
}
```

3. Riadenie PID

NXT firmware implementuje digitálny radič PID (*Proportional Integrative Derivative*) na presné riadenie pozície a rýchlosti servomotorov. Tento typ radiča je jedným z najjednoduchších ale dostatočne efektívnych radičov s uzatvorenou spätnou väzbou, známych z automatizácie a je často používaný.

Toto riadenie pracuje zhruba takto (popíšem reguláciu pozície pre radič diskrétneho času):

Váš program udáva radiču bod **R(t)**, ktorý má dosiahnuť; radič poháňa motor príkazom **U(t)**, meria jeho pozíciu **Y(t)** pomocou zabudovaného kodéra a vypočítava chybu **E(t) = R(t) – Y(t)**: kvôli tomu sa volá "radič s uzatvoreným cyklom", pretože výstupná pozícia Y(t) je kvôli výpočtu chyby prenesená späť na vstup radiča. radič transformuje chybu E(t) do príkazu U(t) takže:

U(t) = P(t) + I(t) + D(t), kde

P(t) = KP · E(t),

I(t) = KI·(I(t–1) + E(t)),

D(t) = KD·(E(t) – E(t –1))

Pre nováčikov to môže byť trochu zložité, ale skúsim to vysvetliť najlepšie ako viem. Príkaz je súčtom troch prvkov, proporcionálnej časti **P(t)**, integračnej časti **I(t)** a derivačnej časti **D(t)**.

- **P(t)** zasiťuje rýchlosť radiča v čase, ale nezaručuje nulovú chybu pri rovnáhe;
- **I(t)** poskytuje radiču „pamäť“, v zmysle, že zhromažďuje akumulovaných chýb a ich kompenzácií, so zaistením nulovej chyby pri rovnováhe;
- **D(t)** poskytuje „budúcu predpoveď” pre radič (ako matematickú deriváciu), čím zrýchľuje odozvu.

Viem, že to môže byť mätúce, ale berte do úvahy, že o tomto boli napísané celé akedemické knihy! Avšak stále to môžete vyskúšať v reále, pomocou svojej NXT kocky! Jednoduchý program, ktorý ukladá veci do pamäte vyzerá takto:

```
#define P 50
#define I 50
#define D 50

task main(){
    RotateMotorPID(OUT_A, 100, 180, P, I, D);
    Wait(3000);
}
```

Príkaz **RotateMotorPID(port, rýchlosť, uhol, Pgain, Igain, Dgain)** dovoľuje presúvať motor pomocou nastavenia rôznych vstupov PID. Skúste nastavenie nasledujúcich hodnôt:

- **(50,0,0)**: motor sa neotočí presne o 180°, pretože ostáva nevykompenzovaná chyba
- **(0,x,x)**: bez proporčnej časti, chyba je príliš veľká
- **(40,40,0)**: tu je veľký presah, to znamená, že sa os motora otočí za nastavený bod a potom sa vráti späť
- **(40,40,90)**: dobrá presnosť a čas (čas na dosiahnutie cieľového bodu)
- **(40,40,200)**: os osciluje, pretože derivačná časť je príliš vysoká

Vyskúšajte aj iné hodnoty, aby ste si vyskúšali ako tieto vstupy ovplyvňujú výkon motora.

4. Zhrnutie

V tejto kapitole ste sa naučili o pokročilých príkazoch riadenia motorov: **Float(), Coast()**, ktoré plynulo zastavujú motory; **OnXxxReg()** a **OnXxxSync()**, ktoré umožňujú spätnoväzobné riadenie rýchlosti motorov a ich synchronizáciu; **RotateMotor()** a **RotateMotorEx()**, ktoré sú používané na otočenie osí motorov o presný počet stupňov. Naučili ste sa aj niečo o riadení PID, ktorý nebol vysvetlený detailne, ale mohol byť pre vás novinkou, a tak pre získanie podrobnejších informácií použite web.

IX. Viac o senzoroch

V kapitole V sme si ukázali základné aspekty použitia senzorov. Ale so senzormi je možné robiť oveľa viac. V tejto kapitole sa pozrieme na rozdiel medzi režimom senzora a typom senzora, uvidíte tiež ako použiť staré kompatibilné RCX senzory, pripojené k NXT pomocou konverzných káblov.

1. Režim a typ senzora

Príkaz **SetSensor()**, ktorý sme si už spomenuli, robí dve veci: nastavuje typ senzora a režim, v ktorom senzor pracuje. Pomocou samostatného nastavenia režimu a typu senzora, možno riadiť správanie senzorov oveľa precíznejšie, čo môže byť pre niektoré aplikácie potrebné.

Typ senzora možno nastaviť pomocou príkazu **SetSensorType()**. Existuje veľa rôznych typov, ale my si spomenieme len základné:

- **SENSOR_TYPE_TOUCH** – dotykový senzor,
- **SENSOR_TYPE_LIGHT_ACTIVE** – svetelný senzor (so zapnutou LED),
- **SENSOR_TYPE_SOUND_DB** – zvukový senzor, a
- **SENSOR_TYPE_LOWSPEED_9V** – ultrazvukový senzor.

Nastavenie typu senzora je dôležité aj na to, aby NXT rozlíšila, či senzor potrebuje napájanie alebo nie (napr. zapnutie LED vo svetelnom senzore), alebo udanie, že senzor je digitálny a má byť čítaný pomocou sériového protokolu I2C. Starý RCX senzor možno v NXT použiť pomocou:

- **SENSOR_TYPE_TEMPERATURE** – pre senzor teploty,
- **SENSOR_TYPE_LIGHT** – pre starý svetelný senzor,
- **SENSOR_TYPE_ROTATION** – pre RCX rotačný senzor (o ňom viac neskôr).

režim senzora nastavíte pomocou príkazu **SetSensorMode()**. K dispozícii je osem rôznych režimov. Najdôležitejší je režim **SENSOR_MODE_RAW**. V tomto režime je hodnota získaná pri čítaní stavu senzora číslo v rozsahu 0 až 1023. Je to pôvodná hodnota poskytovaná senzorom. Čo táto hodnota znamená, to závisí na aktuálnom senzore. Napríklad pre dotykový senzor, keď senzor nie je stlačený je táto hodnota blízko 1023. Keď je senzor naplno stlačený, je hodnota blízko 50. Keď je stlačený čiastočne, je hodnota niekde medzi 50 a 1000. Takže ak nastavíte dotykový senzor do režimu **raw**, môžete získať aj stav, keď je stlačený čiastočne. Keď je to svetelný senzor, hodnota osciluje v rozmedzí od 300 (veľmi svetlé) do 800 (veľmi tmavé). Takto možno získať oveľa presnejšiu hodnotu, ako pri použití príkazu **SetSensor()**. Ďalšie podrobnosti sú v Príručke programátora NXC.

Druhý režim senzorov je **SENSOR_MODE_BOOL**. V tomto režime je hodnota 0 alebo 1. Keď je raw hodnota nižšia ako 562 je hodnota 0, inak je 1. **SENSOR_MODE_BOOL** je

predvolený režim pre dotykový senzor, ale môže byť použitý aj pre iné typy. Režimy **SENSOR_MODE_CELSIUS** a **SENSOR_MODE_FAHRENHEIT** sú užitočné len pre senzory teploty a poskytujú hodnotu teploty v príslušných jednotkách.

SENSOR_MODE_PERCENT mení raw hodnotu na hodnotu medzi 0 a 100. **SENSOR_MODE_PERCENT** je predvolená hodnota pre svetelný senzor. **SENSOR_MODE_ROTATION** je zase použiteľný len pre rotačný senzor (viz ďalej).

Existujú ešte dva zaujímavé režimy: **SENSOR_MODE_EDGE** a **SENSOR_MODE_PULSE**. Tieto počítajú prechody, teda zmeny z nízkej na vysokú raw hodnotu alebo naopak. Napríklad stlačenie dotykového senzora spôsobí prechod z vysokej do nízkej raw hodnoty. Keď ho uvoľníte dostanete prechod do opačného smeru. Keď nastavíte senzor do režimu **SENSOR_MODE_PULSE**, sú počítané len zmeny z nízkej na vysokú hodnotu, takže každé stlačenie a uvoľnenie dotykového senzora je počítané ako jedna. Keď nastavíte senzor do režimu **SENSOR_MODE_EDGE**, sú počítané oba prechody, takže každé stlačenie a uvoľnenie dotykového senzora je počítané ako dva. Takto možno počítať ako často bol dotykový senzor stlačený alebo to môžete využiť v spojení so svetelným senzorom na počítanie ako často je (silné) svetlo zapnuté a vypnuté. Samozrejme, keď počítate hrany alebo pulzy, musíte mať možnosť nastaviť počítadlo na 0. Pre tento účel slúži príkaz **ClearSensor()**, ktorý vynuluje počítadlo zadaného senzora.

Pozrime sa na príklad. Nasledujúci program používa dotykový senzor na ovládanie robota. Pripojte senzor pomocou dlhého vodiča k vstupu 1. Ak rýchlo stlačíte senzor dva krát za sebou, robot sa pohybuje dopredu. Ak ho stlačíte iba raz, zastaví sa.

```
task main()
{
    SetSensorType(IN_1, SENSOR_TYPE_TOUCH);
    SetSensorMode(IN_1, SENSOR_MODE_PULSE);
    while(true)
    {
        ClearSensor(IN_1);
        until (SENSOR_1 > 0);
        Wait(500);
        if (SENSOR_1 == 1) {Off(OUT_AC);}
        if (SENSOR_1 == 2) {OnFwd(OUT_AC, 75);}
    }
}
```

Nezabudnite, že najprv je potrebné nastaviť typ senzora a až potom jeho režim. Je to veľmi dôležité, pretože nastavenie typu senzora ovplyvní aj jeho režim.

2. Rotačný senzor

Rotačný senzor je veľmi užitočným typom senzora: je to optický kodér, skoro taký istý ako ten zabudovaný v NXT servomotoroch. Rotačný senzor má otvor, cez ktorý môžete prestrčiť osku, ktorej relatívne otočenie merané. Jedno celé otočenie osky sa skladá zo 16 krokov (alebo –16, ak otáčate opačným smerom), tzn. rozlíšenie 22.5°, veľmi hrubé v porovnaní s 1°rozlíšením servomotora. Tento starý typ rotačného senzora však môže byť užitočný na monitorovanie osky bez potreby použiť motor; tiež zvážte, že použitie motora ako kodéra vyžaduje veľký krútiaci moment (silu) na jeho otočenie, oproti tomu starý rotačný senzor možno otáčať veľmi ľahko. Ak potrebujete presnejšie rozlíšenie ako 16 krokov na otáčku, stále môžete použiť ozubené kolieska na mechanické zvýšenie to počtu tikov na otáčku. Nasledujúci príklad je zo starého tutoriálu pre RCX.

Štandardným použitím je mať dva rotačné senzory pripojené k dvom kolesám robota, ktorého ovládate pomocou dvoch motorov. Pre priamy pohyb robota sú vyžadované rovnaké otáčky oboch kolies. Nanešťastie, motory sa väčšinou netočia rovnako rýchlo. Pomocou rotačných senzorov možno zistiť, ktoré koleso sa otáča rýchlejšie a potom môžete dočasne zastaviť motor (najlepšie pomocou **Float()**), kým oba senzory neukazujú rovnakú hodnotu. Nasledujúci program robí presne toto. Program jednoducho riadi robota v priamom smere. Aby ste ho mohli použiť, zmeňte pripojenie svojho robota pripojením dvoch rotačných senzorov k dvom kolesám a senzory pripojte k vstupom 1 a 3:

```
task main()
{
    SetSensor(IN_1, SENSOR_ROTATION); ClearSensor(IN_1);
    SetSensor(IN_3, SENSOR_ROTATION); ClearSensor(IN_3);
    while (true)
    {
        if (SENSOR_1 < SENSOR_3)
            {OnFwd(OUT_A, 75); Float(OUT_C);}
        else if (SENSOR_1 > SENSOR_3)
            {OnFwd(OUT_C, 75); Float(OUT_A);}
        else
            {OnFwd(OUT_AC, 75);}
    }
}
```

Program najprv nastavuje oba senzory na rotačné senzory a nuluje ich počítadlá. Potom vstupuje do nekonečného cyklu, v ktorom kontrolujeme či sú hodnoty oboch senzorov rovnaké. Ak sú, robot jednoducho pokračuje v pohybe dopredu. Ak je jedna hodnota väčšia, je príslušný motor zastavený dokiaľ oba senzory zase neukazujú rovnakú hodnotu.

Je to naozaj veľmi jednoduchý program, ktorý si môžete rozšíriť tak, aby sa robot pohyboval na presné vzdialenosti alebo nechať ho robiť veľmi presné zatáčky.

3. Viac senzorov na jednom vstupe

Na začiatku tejto časti je potrebná jedna oprava! Kvôli novej štruktúre vylepšených NXT senzorov a 6-žilovým káblom, už nie je také jednoduché ako predtým (s RCX) pripojiť viacero senzorov na rovnaký port. Podľa môjho názoru je len jedno osvedčené (a jednoducho realizovateľné) riešenie, a to urobiť si analógový multiplexer dotykového senzora a použiť ho v kombinácii s konverzným káblom. Alternatívou je komplexný digitálny multiplexer, ktorý dokáže riadiť I2C komunikáciu s NXT, ale toto určite nie je riešenie pre začiatočníkov.

NXT má na pripojenie senzorov štyri vstupy. Keď sa chystáte vytvoriť komplikovanejšieho robota (a kúpili ste si dodatočné senzory) môže vám to byť málo. Našťastie, pomocou malých trikov, možno pripojiť dva (alebo dokonca viac) senzorov k jednému vstupu.

Najjednoduchšie je pripojiť na jeden vstup dva dotykové senzory. Ak je jeden z nich (alebo oba) stlačený je hodnota 1, inak je 0. Nedokážete síce tieto dva senzory rozlišovať, ale niekedy to nie je potrebné. Napríklad, keď umiestnite jeden dotykový senzor vpredu a druhý vzadu, môžete rozlišovať stlačený senzor na základe smeru pohybu robota. Ale môžete tiež nastaviť režim senzora na raw (viz vyššie). Teraz môžete získať oveľa viac informácií. Ak máte šťastie, tak hodnota stlačeného senzora sa na senzoroch líši. Ak je to váš prípad, môžete na základe tejto hodnoty dva senzory rozlišovať. A keď sú stlačené oba, dostanete oveľa nižšiu dolnú hodnotu (okolo 30), takže možno zisťovať i tento stav.

Tiež môžete na jeden vstup spojiť svetelný a dotykový senzor (len RCX senzory). Nastavte typ senzora na light (inak nebude pracovať svetelný senzor). Nastavte režim na raw. Takto keď bude stlačený dotykový senzor, získate raw hodnotu nižšiu ako 100. Ak nie je stlačený, získate hodnotu svetelného senzora, ktorá nikdy nie je nižšia ako 100.

Nasledujúc program využíva práve túto myšlienku. Robot musí byť vybavený svetelným senzorom smerujúcim dolu a predným nárazníkom, pripojeným k dotykovému senzoru. Oba senzory sú pripojené k vstupu 1. Robot sa bude náhodne pohybovať v svetlej oblasti. Keď svetelný senzor zistí tmavú čiaru (raw hodnota > 750) trochu sa vráti. Keď dotykový senzor na niečo narazí, (raw hodnota < 100) urobí to isté:

```
mutex moveMutex;
int ttt,tt2;

task moverandom()
{
    while (true)
    {
        ttt = Random(500) + 40;
        tt2 = Random();
```

```
        Acquire(moveMutex);
        if (tt2 > 0)
            { OnRev(OUT_A, 75); OnFwd(OUT_C, 75); Wait(ttt); }
        else
            { OnRev(OUT_C, 75); OnFwd(OUT_A, 75); Wait(ttt); }
        ttt = Random(1500) + 50;
        OnFwd(OUT_AC, 75); Wait(ttt);
        Release(moveMutex);
    }
}

task submain()
{
    SetSensorType(IN_1, SENSOR_TYPE_LIGHT);
    SetSensorMode(IN_1, SENSOR_MODE_RAW);
    while (true)
    {
        if ((SENSOR_1 < 100) || (SENSOR_1 > 750))
        {
            Acquire(moveMutex);
            OnRev(OUT_AC, 75); Wait(300);
            Release(moveMutex);
        }
    }
}

task main()
{
    Precedes(moverandom, submain);
}
```

Myslím, že tento program je jasný. Skladá sa z dvoch úloh. Úloha **moverandom** sa stará o náhodný pohyb robota. Úloha **submain** najprv spúšťa úlohu **moverandom**, nastavuje senzor a potom čaká na nejakú udalosť. Ak sa hodnota senzora stane príliš nízka (stlačenie) alebo príliš vysoká (čierna čiara) zastaví náhodný pohyb, trochu vráti robota a znova spustí náhodný pohyb.

4. Zhrnutie

V tejto kapitole sme si ukázali niekoľko ďalších možností práce so senzormi. Ukázali sme si ako samostatne nastaviť typ a režim senzora, a ako to možno využiť na získanie dodatočných informácií. Naučili sme sa o použití starého rotačného senzora a popísali sme

si ako pripojiť viacero senzorov na jedn vstup NXT. všetky tieto triky môžu byť veľmi užitočné pri konštrukcii komplikovanejších robotov.

X. Paralelné úlohy

Ako sme už spomínali, úlohy sú v NXC vykonávané simultánne, alebo ako sa zvykne hovoriť paralelne. Táto vlastnosť je veľmi dôležitá, pretože dovoľuje v jednej úlohe čakať na senzory, zatiaľ čo iná úloha presúva robota a ďalšia prehráva nejakú hudbu. Ale pralelné úlohy môžu spôsobiť aj problémy, pretože jedna úloha môže narušovať inú .

1. Zlý program

Pouvažujte nad nasledujúcim programom. V ňom jedna úloha riadi robota dookola v štvorcoch (ako sme už často robili) a druhá úloha kontroluje stav dotykového senzora. Keď je senzor stlačený, posunie robota späť a urobí 90° otočenie:

```
task check_sensors()
{
    while (true)
    {
        if (SENSOR_1 == 1)
        {
            OnRev(OUT_AC, 75);
            Wait(500);
            OnFwd(OUT_A, 75);
            Wait(850);
            OnFwd(OUT_C, 75);
        }
    }
}

task submain()
{
    while (true)
    {
        OnFwd(OUT_AC, 75); Wait(1000);
        OnRev(OUT_C, 75); Wait(500);
    }
}

task main()
{
    SetSensor(IN_1,SENSOR_TOUCH);
    Precedes(check_sensors, submain);
}
```

Na prvý pohľad to isto vyzerá ako perfektný funkčný program. Ale keď ho spustíte, s najväčšou pravdepodobnosťou zistíte neočakávané správanie. Skúste nasledovné: nechajte robota počas pohybu do niečoho naraziť. Začne sa vracať späť, ale okamžite sa začne znova pohybovať vpred, naraziac do prekážky. Dôvod takéhoto správania je, že úlohy sa rušia. nastáva toto: robot zatáča vpravo, teda, prvá úloha je v druhom príkaze čakania. Teraz robot stlačí senzor a začne sa pohybovať dozadu, ale v krátkom okamihu je prvá úloha na konci čakania a nasmeruje robota znovu vpred, do prekážky. Druhá úloha teraz čaká, a tak nezaregistruje kolíziu. Toto naozaj nie je správanie, ktoré sme chceli. Problém je, že kým druhá úloha čaká na dokončenie pohybu, neberieme do úvahy, že prvá úloha je stále spustená, a tak jej príkazy rušia príkazy druhej úlohy.

2. Kritické sekcie a mutexy

Jedným zo spôsobov riešenia tohto problému je zaistenie, že v jednom okamžiku môže robota ovládať len jedna úloha. Tohto riešenia sme sa dotkli v Kapitole VI. Zopakujme si znova program:

```
mutex moveMutex;

task move_square()
{
    while (true)
    {
        Acquire(moveMutex);
        OnFwd(OUT_AC, 75); Wait(1000);
        OnRev(OUT_C, 75); Wait(850);
        Release(moveMutex);
    }
}

task check_sensors()
{
    while (true)
    {
        if (SENSOR_1 == 1)
        {
            Acquire(moveMutex);
            OnRev(OUT_AC, 75); Wait(500);
            OnFwd(OUT_A, 75); Wait(850);
            Release(moveMutex);
        }
    }
```

```
}

task main()
{
    SetSensor(IN_1,SENSOR_TOUCH);
    Precedes(check_sensors, move_square);
}
```

Problém je, že obe úlohy, **check_sensors** aj **move_square**, môžu riadiť motory len ak ich nepoužíva druhá úloha: čo sme dosiahli pomocou príkazu **Acquire**, ktorý čaká pred použitím motorov na uvoľnenie premennej vzájomného vylúčenia (*mutual exclusion*) **moveMutex**. Opakom príkazu **Acquire** je príkaz **Release**, ktorý uvoľňuje mutexovú premennú, takže druhá úloha môže použiť kritický zdroj, v našom prípade motory. Kód medzi príkazmi **acquire – release** je nazývaný kritická sekcia: kritická v zmysle použitia zdieľaných zdrojov. Takto nemôže jedna úloha rušiť druhú.

3. Použitie semaforov

Existuje i ručná alternatíva mutexových premenných, ktorá je explicitnou implementáciou príkazov **Acquire** a **Release**.

Štandardným spôsobom riešenia tohto problému je použitie premennej, ktorá udáva, ktorá úloha riadi motory. Druhá úlohy nemôže ovládať motory pokiaľ prvá úloha neindikuje, pomocou premennej, že sú pripravené. Takáto premenná je zvyčajne nazývaná semafor. Nazvime taký semafor **sem** (podobne ako mutex). Povedzme, že hodnota 0 indikuje, že žiadna úloha neriadi motory (zdroj je voľný). Teraz vždy keď úloha chce urobiť niečo s motormi, vykoná tieto príkazy:

```
until (sem == 0);
sem = 1;        //Acquire(sem);
// Vykonať niečo s motormi
// kritická sekcia
sem = 0;       //Release(sem);
```

Takže najprv počkáme kým nikto nepoužíva motory. Potom si vyhradíme právo na ich riadenie pomocou nastavenia **sem** na 1 a môžeme ovládať motory. Po dokončení nastavíme **sem** späť na 0. Takže si ukážme predchádzajúci program, ale tentokrát s použitím semaforov. Keď je stlačený dotykový senzor, je semafor nastavený a je vykonaná operácia cúvania. Počas tejto procedúry musí úloha **move_square** čakať. V okamžiku dokončenia cúvania je semafor nastavený na 0 a úloha **move_square** môže pokračovať:

```
int sem;

task move_square()
```

```
{
    while (true)
    {
        until (sem == 0); sem = 1;
        OnFwd(OUT_AC, 75);
        sem = 0;
        Wait(1000);
        until (sem == 0); sem = 1;
        OnRev(OUT_C, 75);
        sem = 0;
        Wait(850);
    }
}

task submain()
{
    SetSensor(IN_1, SENSOR_TOUCH);
    while (true)
    {
        if (SENSOR_1 == 1)
        {
            until (sem == 0); sem = 1;
            OnRev(OUT_AC, 75); Wait(500);
            OnFwd(OUT_A, 75); Wait(850);
            sem = 0;
        }
    }
}

task main()
{
    sem = 0;
    Precedes(move_square, submain);
}
```

Možno sa budete hádať, že nie je potrebné nastavovať v úlohe **move_square** semafor na 1 a späť na 0. Je to však užitočné, najmä preto, že príkaz **OnFwd()** sú vlastne dva príkazy (viz kapitola VIII). A nechcete predsa aby bola táto postupnosť prerušená inou úlohou.

Semafory sú veľmi užitočné a keď píšete komplikované programy s paralelnými úlohami, sú skoro vždy nutné (hoci stále existuje malá šanca, že zlyhajú – skúste si predstaviť prečo) .

4. Zhrnutie

V tejto kapitole sme študovali neiktoré problémy, ktoré môžu nastať pri používaní rôznych úloh. Vždy buďte opatrní ohľadom neželaných efektov. Väčšina nežiadúcich efektov je spôsobená práve nimi. Ukázali sme si dva rôzne spôsoby riešenia týchto problémov. Prvé riešenie zastavuje a znova spúšťa úlohy s cieľom zaistiť aby v jednom okamihu bežala len jedna kritická úloha. Druhé riešenie používa na riadenie vykonávania úloh semafory, ktoré garantujú, že vždy je vykonávaná len jedna kritická sekcia.

XI. Komunikácia medzi robotmi

Ak vlastníte viac ako jednu NXT, je táto kapitola pre vás (hoci aj ostatní môžu komunikovať prostredníctvom PC, hoci majú len jednu NXT). Roboty môžu komunikovať s ostatnými prostredníctvom technológie Bluetooth, a tak môžete mať viacero spolupracujúcich robotov (alebo bojujúcich) alebo môžete vybudovať komplexnejšieho robota s použitím dvoch NXT – a teda použiť až šesť motorov a osem senzorov.

Pri starých RCX, bola komunikácia jednoduchá – posielali infračervené správy, ktoré prijímali všetky roboty v dosahu. V prípade NXT to trochu inak. Pre jednotlivé kocky (alebo NXT a PC) musíte najprv vytvoriť spojenie pomocou Bluetooth menu v kocke a až potom možno posielať správy takto pripojeným zariadeniam.

NXT, ktorá začína spojenie je nazývaná **Master** a môžu sa k nej pripojiť až 3 podriadené zariadenia, zvané **Slave**, na linkách 1,2,3; Všetky zariadenia **Slave** majú vždy **Master** pripojený na linke 0. Správy možno posielať do 10 dostupných schránok.

1. Komunikácia Master – Slave

Ukážeme si dva programy – jeden pre zariadenie **Master** a druhý pre zariadenie **Slave**. Tieto základné programy vás naučia ako môže byť rýchly nepretržitý prúd textových reťazcov spravovaný dvomi NXT v bezdrôtovej sieti.

Program master najprv kontroluje, či je slave správne pripojený na linke 1 (konštanta **BT_CONN**) pomocou funkcie **BluetoothStatus(conn)**, potom pripraví a pošle správy s predponou M a zvyšuje číslo pomocou **SendRemoteString(conn, queue, string)**, kým prijme správy od slave pomocou **ReceiveRemoteString(queue, clear, string)** a zobrazí dáta.

```
//MASTER
#define BT_CONN 1
#define INBOX 1
#define OUTBOX 5

sub BTCheck(int conn){
    if (!BluetoothStatus(conn)==NO_ERR){
        TextOut(5,LCD_LINE2,"Error");
        Wait(1000);
        Stop(true);
    }
}

task main(){
    string in, out, iStr;
```

```
    int i = 0;
    BTCheck(BT_CONN); //check slave connection
    while(true){
        iStr = NumToStr(i);
        out = StrCat("M",iStr);
        TextOut(10,LCD_LINE1,"Master Test");
        TextOut(0,LCD_LINE2,"IN:");
        TextOut(0,LCD_LINE4,"OUT:");
        ReceiveRemoteString(INBOX, true, in);
        SendRemoteString(BT_CONN,OUTBOX,out);
        TextOut(10,LCD_LINE3,in);
        TextOut(10,LCD_LINE5,out);
        Wait(100);
        i++;
    }
}
```

Program slave je veľmi podobný, len používa **SendResponseString(queue, string)** namiesto **SendRemoteString**, pretože slave môže posielať správy len svojmu master, ktorého vidí na linke 0.

```
//SLAVE
#define BT_CONN 1
#define INBOX 5
#define OUTBOX 1

sub BTCheck(int conn){
    if (!BluetoothStatus(conn)==NO_ERR){
        TextOut(5,LCD_LINE2,"Error");
        Wait(1000);
        Stop(true);
    }
}

task main(){
    string in, out, iStr;
    int i = 0;
    BTCheck(0); //check master connection
    while(true){
        iStr = NumToStr(i);
        out = StrCat("S",iStr);
        TextOut(10,LCD_LINE1,"Slave Test");
        TextOut(0,LCD_LINE2,"IN:");
        TextOut(0,LCD_LINE4,"OUT:");
```

```
        ReceiveRemoteString(INBOX, true, in);
        SendResponseString(OUTBOX,out);
        TextOut(10,LCD_LINE3,in);
        TextOut(10,LCD_LINE5,out);
        Wait(100);
        i++;
    }
}
```

Všimnite si, že ak prerušíme jeden z programov, druhý bude pokračovať v posielaní správ a zvyšovaním počtu, bez toho aby vedel, že jeho správy sa strácajú, pretože na druhej strane ich nikto neprijíma. Aby sme predišli tomuto problému, budeme musieť vytvoriť lepší komunikačný protokol, ktorý bude potvrdzovať doručenie správy.

2. Posielanie čísel s potvrdením

Tu je iná dvojica programov. Tentokrát master posiela čísla pomocou **SendRemoteNumber(conn, queue, number)** a ukončí čakanie na potvrdenie od slave (cyklus until, v ktorom hľadáme **ReceiveRemoteString**). Master pokračuje v posielaní správ len, ak slave čaká na správy a posiela potvrdenia o ich prijatí. Slave jednoducho prijíma číslo pomocou **ReceiveRemoteNumber(queue, clear, number)** a posiela potvrdenie pomocou **SendResponseNumber**. Naše master-slave programy musia používať zhodný kód potvrdenia, v tomto prípade som zvolila šestnástkovú hodnotu **0xFF**.

Master posiela náhodné čísla a čaká na potvrdenie od slave; vždy keď prijme potvrdenie so správnym kódom, musí vymazať premennú **ack**, inak by master pokračoval v posielaní správ bez ich potvrdenia, pretože by si premenná ponechala hodnotu z potvrdenia predchádzajúcej správy.

Slave nepretržite kontroluje svoju schránku a ak nie je prázdna, zobrazí prečítanú hodnotu a pošle potvrdenie master. Na začiatku programu je poslané jedno potvrdenie bez čítania správ, a to kvôli odblokovaniu master. Bez tohto triku by sa mohlo stať, že ak je program master spustený ako prvý, môže zamrznúť, hoci neskôr slave spustíme. Takýmto spôsobom síce niekoľko prvých správ stratíme, ale môžeme spúšťať jednotlivé programy v rôznych okamihoch, bez obáv z ich zamrznutia.

```
//MASTER
#define BT_CONN 1
#define OUTBOX 5
#define INBOX 1
#define CLEARLINE(L)   \
    TextOut(0,L,"                      ");

sub BTCheck(int conn){
```

```
    if (!BluetoothStatus(conn)==NO_ERR){
        TextOut(5,LCD_LINE2,"Error");
        Wait(1000);
        Stop(true);
    }
}

task main(){
    int ack;
    int i;
    BTCheck(BT_CONN);
    TextOut(10,LCD_LINE1,"Master sending");
    while(true){
        i = Random(512);
        CLEARLINE(LCD_LINE3);
        NumOut(5,LCD_LINE3,i);
        ack = 0;
        SendRemoteNumber(BT_CONN,OUTBOX,i);
        until(ack==0xFF) {
            until(ReceiveRemoteNumber(INBOX,true,ack) == NO_ERR);
        }
        Wait(250);
    }
}
```

```
//SLAVE
#define BT_CONN 1
#define OUT_MBOX 1
#define IN_MBOX 5
sub BTCheck(int conn){
   if (!BluetoothStatus(conn)==NO_ERR){
      TextOut(5,LCD_LINE2,"Error");
      Wait(1000);
      Stop(true);
   }
}
task main(){
   int in;
   BTCheck(0);
   TextOut(5,LCD_LINE1,"Slave receiving");
   SendResponseNumber(OUT_MBOX,0xFF); //unblock master
   while(true){
```

```
     if (ReceiveRemoteNumber(IN_MBOX,true,in) !=
STAT_MSG_EMPTY_MAILBOX) {
       TextOut(0,LCD_LINE3,"                   ");
       NumOut(5,LCD_LINE3,in);
        SendResponseNumber(OUT_MBOX,0xFF);
     }
     Wait(10); //take breath (optional)
   }
}
```

3. Priame príkazy

Existuje aj iná zaujímavá vlastnosť Bluetooth komunikácie: master môže priamo ovládať svoje slave.

V nasledujúcom príklade, posiela master do slave priame príkazy na prehrávanie zvukov a spúšťanie motorov. Program pre slave nie je potrebný, pretože správy na slave prijíma a spracováva priamo firmware slave NXT!

```
//MASTER
#define BT_CONN 1
#define MOTOR(p,s) RemoteSetOutputState(BT_CONN, p, s, \
    OUT_MODE_MOTORON+OUT_MODE_BRAKE+OUT_MODE_REGULATED, \
    OUT_REGMODE_SPEED, 0, OUT_RUNSTATE_RUNNING, 0)

sub BTCheck(int conn){
    if (!BluetoothStatus(conn)==NO_ERR){
        TextOut(5,LCD_LINE2,"Error");
        Wait(1000);
        Stop(true);
    }
}

task main(){
    BTCheck(BT_CONN);
    RemotePlayTone(BT_CONN, 4000, 100);
    until(BluetoothStatus(BT_CONN)==NO_ERR);
    Wait(110);
    RemotePlaySoundFile(BT_CONN, "! Click.rso", false);
    until(BluetoothStatus(BT_CONN)==NO_ERR);
    //Wait(500);
    RemoteResetMotorPosition(BT_CONN,OUT_A,true);
    until(BluetoothStatus(BT_CONN)==NO_ERR);
    MOTOR(OUT_A,100);
```

```
    Wait(1000);
    MOTOR(OUT_A,0);
}
```

4. Zhrnutie

V tejto kapitole sme skúmali niektoré základné aspekty Bluetooth komunikácie medzi robotmi: pripojenie dvoch NXT, posielanie a prijímanie reťazcov, čísel a čakanie na potvrdenie. Tento posledný aspekt je veľmi dôležitý, keď je potrebný spoľahlivý komunikačný protokol.

Ako extra vlastnosť, sme sa naučili ako posielať do slave kocky priame príkazy.

XII. Viac príkazov

NXC má veľa ďalších príkazov. V tejto kapitole si popíšeme tri typy: použitie časovača, príkazy na ovládanie displeja a použitie súborového systému NXT.

1. Časovače

NXT má zabudovaný časovač, ktorý nepretržite beží. Tento časovač tiká (zvyšuje hodnotu svojho počítadla) každú tisícinu sekundy. Aktuálnu hodnotu časovača možno získať pomocou funkcie **CurrentTick()**. Nasleduje príklad použitia časovača – nasledujúci program náhodne pohybuje robotom počas 10 sekúnd.

```
task main()
{
    long t0, time;
    t0 = CurrentTick();
    do
    {
        time = CurrentTick()-t0;
        OnFwd(OUT_AC, 75);
        Wait(Random(1000));
        OnRev(OUT_C, 75);
        Wait(Random(1000));
    }
    while (time<10000);
    Off(OUT_AC);
}
```

Môžete si tento program porovnať s tým, ktorý je v Kapitole IV a ktorý robí presne rovnakú úlohu. Určite zbadáte, že program s použitím časovača je výrazne jednoduchší.

Časovače sú veľmi užitočné ako náhrada príkazu **Wait()**. Zaistiť čakanie pomocou časovača na určitú dobu možno prostým vynulovaním časovača a následným čakaním na dosiahnutie príslušnej hodnoty. Počas čakania môžete reagovať aj na iné udalosti (napr. zo senzorov). Nasledujúci jednoduchý program je práve príkladom takéhoto použitia – necháva robot bežať buď 10 sekúnd (ako predtým) alebo kým nie je stlačený dotykový senzor.

```
task main()
{
    long t3;
    SetSensor(IN_1,SENSOR_TOUCH);
    t3 = CurrentTick();
    OnFwd(OUT_AC, 75);
    until ((SENSOR_1 == 1) || ((CurrentTick()-t3) > 10000));
```

```
    Off(OUT_AC);
}
```

Pamätajte, že časovač pracuje v intervaloch 1/1000 sekundy, rovnako ako príkaz **Wait**.

2. Bodový maticový displej

NXT kocka disponuje čiernobielym bodovým maticovým displejom s rozlíšením 100x64 pixelov. NXC poskytuje mnoho API funkcií pre vykresľovanie textových reťazcov, čísel, bodov, čiar, pravouholníkov, kruhov a dokonca aj bitmapových obrázkov (súborov .ric). Nasledujúci príklad sa pokúša pokryť všetky tieto prípady. Pixel označený súradnicami (0,0) je v ľavom dolnom rohu.

```
#define   X_MAX     99
#define   Y_MAX     63
#define   X_MID     (X_MAX+1)/2
#define   Y_MID     (Y_MAX+1)/2

task main(){
    int i = 1234;
    TextOut(15,LCD_LINE1,"Display", true);
    NumOut(60,LCD_LINE1, i);
    PointOut(1,Y_MAX-1);
    PointOut(X_MAX-1,Y_MAX-1);
    PointOut(1,1);
    PointOut(X_MAX-1,1);
    Wait(200);
    RectOut(5,5,90,50);
    Wait(200);
    LineOut(5,5,95,55);
    Wait(200);
    LineOut(5,55,95,5);
    Wait(200);
    CircleOut(X_MID,Y_MID-2,20);
    Wait(800);
    ClearScreen();
    GraphicOut(30,10,"faceclosed.ric");
    Wait(500);
    ClearScreen();
    GraphicOut(30,10,"faceopen.ric");
    Wait(1000);
}
```

Myslím, že všetky tieto funkcie sú samo vysvetľujúce, ale i tak si popíšeme ich parametre podrobnejšie:

- **ClearScreen()** – vymaže obrazovku;
- **NumOut(x, y, number)** – vypíše číslo na zadané súradnice;
- **TextOut(x, y, string)** – pracuje rovnako ako predchádzajúca, ale s reťazcom;
- **GraphicOut(x, y, filename)** – zobrazí bitmapu zo súboru .ric;
- **CircleOut(x, y, radius)** – nakreslí kruh so zadaným polomerom a stredom v daných súradniciach;
- **LineOut(x1, y1, x2, y2)** – nakreslí čiaru z bodu (x1,x2) do bodu (x2,y2)
- **PointOut(x, y)** – nakreslí bodku an zadaných súradniciach;
- **RectOut(x, y, width, height)** – nakreslí pravouholník s ľavým dolným rohom v (x,y) a zadanými rozmermi;
- **ResetScreen()** – resetuje obrazovku.

3. Súborový systém

NXT môže zapisovať a čítať súbory, uložené v pamäti flash. Takto možno uložiť záznam (*datalog*) dát senzorov alebo čítať čísla počas vykonávania programu. Jediným obmedzením v počte súborov a ich veľkosti je veľkosť pamäte flash . NXT API funkcie umožňujú spravovať súbory (vytvorenie, premenovanie, mazanie, hľadanie), umožňujú čítať a zapisovať textové reťazce, čísla i samotné bajty.

V nasledujúcom príklade si ukážeme ako vytvoriť súbor, zapísať do neho reťazce a premenovať ho.

Najprv program vymaže súbory, s menami, ktoré budeme používať. Toto nie je dobré riešenie (mali by sme skontrolovať existenciu súboru a ak existuje ručne ho zmazať alebo automaticky zvoliť iné meno súboru pre pracovný súbor), ale to v našom jednoduchom príklade nie je problém. Náš súbor je vytvorený pomocou **CreateFile("Danny.txt", 512, fileHandle)**, zadávame meno súboru, veľkosť a manipulátor súboru, kde NXT firmware zapíše číslo pre interné použitie.

Potom program pripraví reťazec a zapíše ho do súboru spolu so znakom konca riadku (*carriage return*) pomocou **WriteLnString(fileHandle, string, bytesWritten)**, pričom všetky parametre musia byť premennými. Nakoniec je súbor zatvorený a premenovaný. Dôležité je, aby bol súbor pred začatím inej operácie zatvorený, takže ak vytvoríte súbor, môžete do neho zapisovať. Ak však z neho chcete čítať, musí byť najprv zatvorený a znova otvorený pomocou **OpenFileRead()**. Rovnako musí byť zatvorený pred zmazaním alebo premenovaním.

```
#define OK LDR_SUCCESS
task main(){
    byte fileHandle;
    short fileSize;
```

```
    short bytesWritten;
    string read;
    string write;
    DeleteFile("Danny.txt");
    DeleteFile("DannySays.txt");
    CreateFile("Danny.txt", 512, fileHandle);
    for(int i=2; i<=10; i++ ){
        write = "NXT is cool ";
        string tmp = NumToStr(i);
        write = StrCat(write, tmp," times!");
        WriteLnString(fileHandle, write, bytesWritten);
    }
    CloseFile(fileHandle);
    RenameFile("Danny.txt", "DannySays.txt");
}
```

Výsledok si môžete pozrieť tak, že prostredníctvom NXT menu kocky **BricxCC→Tools→NXT Explorer** nahráte súbor DannySays.txt do počítača a tam si ho pozriete. Takže sme pripravení na ďalší príklad! Vytvoríme tabuľku ASCII znakov:

```
task main(){
    byte handle;
    if (CreateFile("ASCII.txt", 2048, handle) == NO_ERR) {
        for (int i=0; i < 256; i++) {
            string s = NumToStr(i);
            int slen = StrLen(s);
            WriteBytes(handle, s, slen);
            WriteLn(handle, i);
        }
        CloseFile(handle);
    }
}
```

Veľmi jednoduché, program vytvára súbor a ak vytvorenie prebehne bez chýb, zapíše do neho čísla od 0 do 255 (konvertované na reťazce) pomocou **WriteBytes(handle, s, slen)**, čo je spôsob zapisovania reťazcov bez znaku konca riadku; potom zapíše čísla pomocou **WriteLn(handle, value)**, teda so znakmi konca riadku. Výsledok, ktorý môžete vidieť rovnako ako v predchádzajúcom príklade – otvorením **ASCII.txt** v textovom editore, je samo vysvetľujúci: čísla zapísané ako reťazce sú v čitateľnej podobe, zatiaľčo čísla zapísané ako šestnástková hodnota sú interpretované ako ASCII kód.

Ostáva ukázať dve dôležité funkcie, a to **ReadLnString** pre čítanie reťazcov a **ReadLn** pre čítanie čísel zo súboru.

Najprv príklad pre prvú funkciu: úloha **main** volá podprogram **CreateRandomFile**, ktorý vytvára súbor s náhodnými číslami (zapísanými ako reťazce). Túto časť môžete zakomentovať a použiť nejaký ručne vytvorený textový súbor.

Úloha **main** potom tento súbor otvorí na čítanie, číta ho po riadkoch až do konca súboru pomocou funkcie **ReadLnString** a prečítaný text zobrazuje na displeji.

V podprograme **CreateRandomFile** generuje preddefinovaný počet náhodných čísel, konvertujeme ich na reťazce a zapisujeme do súboru.

Funkcia **ReadLnString** vyžaduje zadanie manipulátora súboru a reťazcovej premennej. A po zavolaní vracia reťazec, ktorý obsahuje riadok textu alebo chybový kód, ktorý možno použiť na zistenie konca súboru.

```
#define FILE_LINES 10

sub CreateRandomFile(string fname, int lines){
    byte handle;
    string s;
    int bytesWritten;
    DeleteFile(fname);
    int fsize = lines*5;
    //create file with random data
    if(CreateFile(fname, fsize, handle) == NO_ERR) {
        int n;
        repeat(FILE_LINES) {
            int n = Random(0xFF);
            s = NumToStr(n);
            WriteLnString(handle,s,bytesWritten);
        }
        CloseFile(handle);
    }
}

task main(){
    byte handle;
    int fsize;
    string buf;
    bool eof = false;
    CreateRandomFile("rand.txt",FILE_LINES);
    if(OpenFileRead("rand.txt", fsize, handle) == NO_ERR) {
        TextOut(10,LCD_LINE2,"Filesize:");
        NumOut(65,LCD_LINE2,fsize);
        Wait(600);
        until (eof == true){   // read the text file till the end
```

```
        if(ReadLnString(handle,buf) != NO_ERR) eof = true;
            ClearScreen();
            TextOut(20,LCD_LINE3,buf);
            Wait(500);
        }
    }
    CloseFile(handle);
}
```

V poslednom príklade ukážem ako možno zo súboru čítať čísla. Zároveň v ňom chcem predviesť malú ukážku podmienenej kompilácie. Na začiatku kódu je definícia, ktorá nie je použitá pre žiadne makro ani ako alias – proste definujeme **INT**. A takto vyzerá príkaz preprocesora:

```
#ifdef INT
    ...Code...
#endif
```

ktorý jednoducho informuje kompilátor, že kód medzi týmito dvomi príkazmi má kompilovať len ak bol predtým definovaný identifikátor **INT**. Takže v príklade bude kompilovaná prvá verzia kódu úlohy **main**, ak je definovaný **INT**, alebo druhá verzia úlohy **main**, ak bol definovaný **LONG**.

Pomocou tejto metódy môžem v jednom príklade ukázať spôsob ako môžu byť čítané oba typy premenných, typ **int** (16 bit) aj typ **long** (32 bit), pomocou rovnakej funkcie **ReadLn(handle, val)**.

Rovnako ako predtým, funkcia potrebuje zadať manipulátor súboru a číselnú premennú, a vracia chybový kód. Funkcia prečíta zo súboru 2 bajty, ak je poslaná premenná deklarovaná ako **int**, alebo číta 4 bajty, ak je argument typu **long**. Rovnakým spôsobom možno zapisovať a čítať logické (*bool*) premenné.

```
#define INT     // INT or LONG

#ifdef INT
task main () {
    byte handle, time = 0;
    int n, fsize,len, i;
    int in;
    DeleteFile("int.txt");
    CreateFile("int.txt",4096,handle);
    for (int i = 1000; i<=10000; i+=1000){
        WriteLn(handle,i);
    }
    CloseFile(handle);
    OpenFileRead("int.txt",fsize,handle);
```

```
    until (ReadLn(handle,in)!=NO_ERR){
        ClearScreen();
        NumOut(30,LCD_LINE5,in);
        Wait(500);
    }
    CloseFile(handle);
}
#endif

#ifdef LONG
task main () {
    byte handle, time = 0;
    int n, fsize,len, i;
    long in;
    DeleteFile("long.txt");
    CreateFile("long.txt",4096,handle);
    for (long i = 100000; i<=1000000; i+=50000){
        WriteLn(handle,i);
    }
    CloseFile(handle);
    OpenFileRead("long.txt",fsize,handle);
    until (ReadLn(handle,in)!=NO_ERR){
        ClearScreen();
        NumOut(30,LCD_LINE5,in);
        Wait(500);
    }
    CloseFile(handle);
}
#endif
```

4. Zhrnutie

V tejto kapitole ste sa stretli s pokročilými funkciami,ktoré ponúka NXT: časovač s vysokým rozlíšením, bodový maticový displej a súborový systém. okrajovo ste tiež mali možnosť vidieť ukážku podmienenej kompilácie kódu.

XIII. Záverečné poznámky

Ak ste si s pomocou tohto tutoriálu našli svoj spôsob práce a vyskúšali si vlastné príklady, môžete sa považovať za experta v NXC. Ak ste to zatiaľ neurobili, je čas na vlastné experimenty. Pomocou tvorivého návrhu a programovania môžú vaše lego roboty robiť neuveriteľné veci.

Tento tutoriál nepopisuje všetky možnosti BricxCC. Odporúčam vám čítať Príručku NXC pri každej kapitole. Nezabúdajte, že NXC je v neustálom vývoji, budúce verzie môžu pridať ďalšie funkcie. Rovnako v tomto tutoriáli nie sú rozoberané programovacie koncepty a celkom sme sa vyhli problematike učiacich sa robotov a ďalších aspektov umelej inteligencie.

Lego roboty možno riadiť priamo z PC. Na toto si potrebujete napísať program v jazyku ako C++, Visual Basic, Java alebo Delphi. Môžete tiež vytvoriť program, ktorý spolupracuje s NXC programom, ktorý beží v NXT samotnom. Takáto kombinácia je veľmi výkonná a ak sa zaujímate o tento spôsob programovania svojho robota, najlepším začiatkom je stiahnutie Fantom SDK a Open Source dokumentov zo sekcie NXTreme webovej stránky Lego MindStorms:

http://mindstorms.lego.com/Overview/NXTreme.aspx

Tento web je perfektným zdrojom dodatočných informácií. Niektoré ďalšie štartovacie body sú na LUGNETe, neoficiálna LEGO® Users Group Network:

http://www.lugnet.com/robotics/nxt