POLITECHNIKA WARSZAWSKA
WYDZIAŁ ELEKTRYCZNY
INSTYTUT STEROWANIA I ELEKTRONIKI PRZEMYSŁOWEJ

PRACA DYPLOMOWA MAGISTERSKA
na kierunku INFORMATYKA

Imię NAZWISKO:

Michał Kapica
Nr imm. 172855

Imię NAZWISKO:

Krzysztof Cedzidło
Nr imm. 172843

Rok. akad. 2003/2004
Warszawa
data wydania tematu: 24.02.2004

TEMAT PRACY DYPLOMOWEJ
Studium środowisk programistycznych mobilnych robotów LEGO Mindstorms.

Zakres pracy:
1. *Wprowadzenie*
2. *Rys historyczny*
3. *Czujniki*
4. *Języki, środowiska programistyczne*
5. *Przykładowe programy*
6. *Porównanie, wnioski*
7. *Plan zajęć dla studentów na zajęcia laboratoryjne*

Podpis i pieczątka

Kierownika Zakładu Dydaktycznego

Opiekun naukowy:
Dr inż. Witold Czajewski

Konsultant:

Termin wykonania: czerwiec 2004
Praca wykonana i zaliczona pozostaje własnością Instytutu i nie będzie zwrócona wykonawcy

1 WSTĘP 1

2 WPROWADZENIE 2

2.1 HISTORIA MIKROKOMPUTERÓW LEGO 2
2.2 SILNIKI 6
2.3 CZUJNIKI LEGO 9
2.3.1 Czujnik dotykowy 11
2.3.2 Czujnik światła 12
2.3.3 Czujnik obrotów 13
2.3.4 Czujnik temperatury 14
2.4 EKSPERYMENTALNY CZUJNIK OBROTÓW 14

3 BRICX COMMAND CENTER 22

4 NOT QUITE C 34

4.1 REGUŁY LEKSYKALNE 34
4.1.1 Komentarze 34
4.1.2 Białe znaki 35
4.1.3 Stałe liczbowe 35
4.1.4 Identyfikatory i słowa kluczowe 36
4.1.5 Operatory przypisania 36
4.1.6 Wyrażenie złożone 37
4.1.7 Struktura programu 37
4.1.8 Pętle i instrukcje 44
4.1.9 Inne wyrażenia 47
4.1.10 Operatory 48
4.1.11 Wyrażenia warunkowe 48
4.1.12 Preprocesor 49
4.2 RCX NQC API 51
4.2.1 Czujniki 51
4.2.2 Silniki 53
4.2.3 Głośnik 56
4.2.4 Wyświetlacz LCD 56
4.2.5 Komunikacja 58
4.2.6 Timery 60
4.2.7 Liczniki – tylko RCX2 60
4.2.8 Kontrola dostępu – tylko RCX2 61
4.2.9 Zdarzenia – tylko RCX2 62
4.2.10 Rejestr danych (Datalog) 65
4.2.11 Inne polecenia 66

5 BRICKOS 68

5.1 REGUŁY LEKSYKALNE 68
5.1.1 Komentarze 68
5.1.2 Białe znaki 68
5.1.3 Słowa kluczowe 69
5.1.4 Zmienne i stałe 69
5.1.5 Zmienne ze znakiem i bez znaku 70
5.1.6 Operatory arytmetyczne 71
5.1.7 Operatory relacji 71
5.1.8 Operatory logiczne 72
5.1.9 Struktura programu 72
5.1.10 Funkcje w języku BrickOS 73
5.1.11 Pętle i instrukcje 74
5.2 RCX BRICKOS API 77
5.2.1 Obsługa wyświetlacza LCD (1) (lcd.h) 78
5.2.2 Obsługa wyświetlacza LCD (2) (romlcd.h) 78
5.2.3 Obsługa wyświetlacza LCD (3) (conio.h) 78
5.2.4 Obsługa wyświetlacza LCD (4) (dlcd.h) 79
5.2.5 Funkcje systemowe (romsystem.h) 80
5.2.6 Obsługa silników (dmotor.h) 80

5.2.7 Obsługa czujników (dsensor.h) ... 81
5.2.8 Obsługa głośniczka systemowego (dsound.h) ... 83
5.2.9 Obsługa klawiatury (1) (dkey.h) ... 84
5.2.10 Obsługa klawiatury (2) (dbutton.h) ... 85
5.2.11 Generator liczb losowych (stdlib.h) ... 86
5.2.12 Czas systemowy (time.h) ... 86
5.2.13 Czujnik baterii (battery.h) ... 86
5.2.14 Obsługa wątków (unistd.h) ... 87
5.2.15 Semafory (semaphore.h) ... 88

6 PASCAL ... 91

6.1 REGUŁY LEKSYKALNE ... 91
6.1.1 Komentarze ... 91
6.1.2 Białe znaki ... 91
6.1.3 Instrukcja przypisania ... 92
6.1.4 Identyfikatory i słowa kluczowe ... 92
6.1.5 Struktura programu ... 93
6.1.6 Deklaracja stałych ... 94
6.1.7 Deklaracja zmiennych ... 94
6.1.8 Definicja nazw własnych ... 95
6.1.9 Operatory arytmetyczne ... 95
6.1.10 Operatory relacji ... 96
6.1.11 Operatory logiczne ... 96
6.1.12 Procedury ... 96
6.1.13 Funkcje ... 98
6.1.14 Pętle i instrukcje ... 99
6.2 RCX PASCAL API ... 103
6.2.1 Obsługa wyświetlacza LCD (1) (lcd.h) ... 103
6.2.2 Obsługa wyświetlacza LCD (2) (romlcd.h) ... 103
6.2.3 Obsługa wyświetlacza LCD (3) (conio.h) ... 103
6.2.4 Funkcje systemowe (romsystem.h) ... 104
6.2.5 Obsługa silników (dmotor.h) ... 105
6.2.6 Obsługa czujników (dsensor.h) ... 105
6.2.7 Obsługa głośniczka systemowego (dsound.h) ... 107
6.2.8 Obsługa klawiatury (1) (dkey.h) ... 108
6.2.9 Obsługa klawiatury (2) (dbutton.h) ... 109
6.2.10 Generator liczb losowych (stdlib.h) ... 110
6.2.11 Czas systemowy (time.h) ... 110
6.2.12 Czujnik baterii (battery.h) ... 111
6.2.13 Obsługa wątków (unistd.h) ... 111

7 MINDSCRIPT ... 113

7.1 REGUŁY LEKSYKALNE ... 113
7.1.1 Komentarze ... 113
7.1.2 Białe znaki ... 113
7.1.3 Stałe liczbowe ... 113
7.1.4 Zmienne ... 114
7.1.5 Operatory ... 114
7.1.6 Struktura programu ... 115
7.1.7 Pętle i instrukcje ... 117
7.1.8 Preprocesor ... 120
7.2 RCX MINDSCRIPT API ... 120
7.2.1 Czujniki ... 120
7.2.2 Silniki ... 122
7.2.3 Głośnik ... 124
7.2.4 Wyświetlacz LCD ... 124
7.2.5 Komunikacja ... 124
7.2.6 Timery ... 125
7.2.7 Licznik (tylko RCX2) ... 125
7.2.8 Zdarzenia ... 125

7.2.9 Rejestr danych (Datalog) 127
7.2.10 Inne polecenia 127

8 RCX CODE 129

8.1 ŚRODOWISKO 129
8.2 STRUKTURA MENU 131
8.3 BLOKI FUNKCYJNE RCX CODE 133

9 PRZYKŁADOWE PROGRAMY, PORÓWNANIE JĘZYKÓW 141

9.1 NQC 141
9.2 BRICKOS 143
9.3 PASCAL 145
9.4 MINDSCRIPT 147
9.5 RCX CODE 149
9.6 PORÓWNANIE ORAZ WNIOSKI 152

10 PROJEKTY NA ZAJĘCIA LABORATORYJNE 154

10.1 GRA „KÓŁO I KRZYŻYK” 154
10.2 ROBOT „DOSTAWCA” 156

SPIS ILUSTRACJI 159

BIBLIOGRAFIA 161

1 WSTĘP

Niniejszy dokument ma uzupełnić lukę, jaka panuje na rynku publikacji, poruszających zagadnienia programowania robotów LEGO Mindstorms. Ponieważ obecnie nie istnieją kompleksowe opracowania traktujące o programowaniu robotów LEGO, dokument ten powinien być bardzo pomocnym przy budowie robotów oraz ich programowaniu. W pracy umieszczono opisy najbardziej popularnych języków, ze wszystkimi dostępnymi funkcjami oraz przykładami ich zastosowania. Praca zawiera także porównanie możliwości opisanych języków w taki sposób, aby czytelnik mógł wybrać język odpowiedni dla swoich umiejętności oraz postawionego zadania.

Atutem zestawu LEGO Mindstorms jest możliwość jego rozbudowy o czujniki własnej konstrukcji. Dostępna wiedza na temat specyfikacji wejść pozwoliła autorom na skonstruowanie dwóch eksperymentalnych czujników obrotów.

Ponieważ w planach uczelni jest wprowadzenie zajęć laboratoryjnych opartych o zestawy LEGO Mindstorms, w pracy umieszczono propozycje projektów do wykonania w czasie zajęć.

2 WPROWADZENIE

2.1 Historia mikrokomputerów LEGO

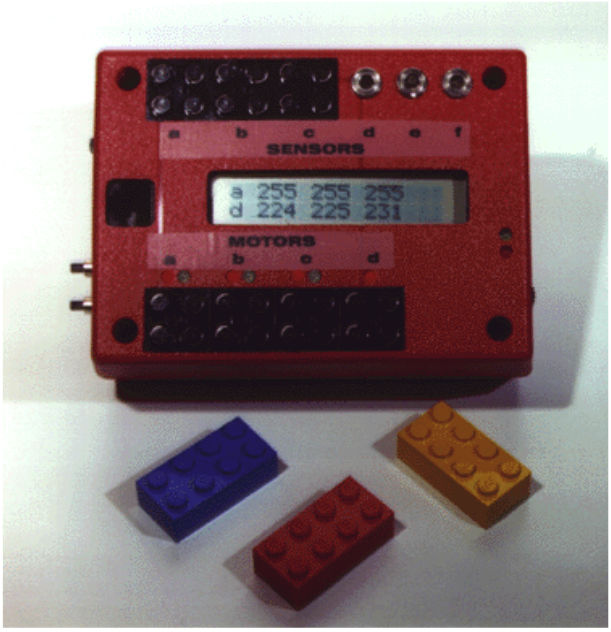
Idea budowy robotów z klocków LEGO narodziła się w Massachusetts Institute of Technology [15]. Pod koniec lat osiemdziesiątych, gdy w zestawach LEGO Technic można było znaleźć: silniki, światełka, przekładnie, koła zębate i elementy pneumatyczne, grupa pracowników w składzie: Fred Martin, Rand Sargent, Brian Silverman, Seymour Papert oraz Mitchel Resnick, postanowiła opracować nowy sterownik dla podzespołów LEGO. W ten sposób lista dostępnych elementów została powiększona o klocek – mikrokomputer, co umożliwiło budowę robotów oraz mechanizmów z klocków o złożonym sterowaniu.

W 1987 roku Rand Sargent opracował pierwszy model mikrokomputera – Programmable Brick 6502. Brian Silverman umieścił w sterowniku interpreter instrukcji Logo.

W latach 1993-1995 powstał Pocket Programmable Brick. Nazwę „pocket" (czyli kieszonkowy) otrzymał dzięki swoim małym wymiarom – bez problemu mieścił się w kieszeni. PPB posiadał osiem wejść oraz cztery wyjścia. Dodatkowo umieszczono w nim port podczerwieni oraz głośniczek. Konstrukcja wewnętrzna klocka sprowadzała się do trzech połączonych ze sobą płytek drukowanych, obsadzonych elementami obustronnie. Ze względu na małe wymiary całość zasilana była baterią 9V, co niestety było błędnym rozwiązaniem. Przy programach działających tylko z czujnikami taka bateria w zupełności wystarczała, lecz gdy do klocka podłączano silniki, bateria bardzo szybko się rozładowywała.
Rozwój Pocket Programmable Brick został dość szybko wstrzymany; trudność wykonania oraz drogie komponenty spowodowały, iż w roku 1995 projekt porzucono.

W roku 1994 powstał mikrokomputer Programmable Brick 120 (rys. 2.1). Jest to wersja „ekonomiczna" klocka Pocket PB. PB 120 zawierał mikroprocesor Motorola 68332/20MHz, 256 KB pamięci RAM; posiadał cztery wyjścia dla silników oraz sześć wejść dla czujników. Obecny był także jednokierunkowy port podczerwieni oraz głośniczek. Dodatkowo mogły zostać dołączone mikrofon oraz port nadawczy

podczerwieni. PB 120 był dwa razy większy od Pocket PB, ale dzięki temu rozwiązano problem z brakiem miejsca na baterie.



Rys. 2.1 Sterownik PB 120

PB 120 posiadał wbudowany interpreter jednej z wersji Logo – "Brick Logo". Język ten całkowicie wystarczał do obsługi silników, czujników dotyku, głośnika, wyświetlacza LCD. Fred Martin posiadając odpowiedni mikrokontroler (PB120), uruchomił na MIT kurs 6.270, w czasie którego studenci projektowali i budowali roboty z klocków LEGO. Kurs szybko zdobył taką sławę i popularność, że firma LEGO zdecydowała się wprowadzić na rynek zestawy Mindstorms.

Zestawy LEGO Mindstorms dedykowane są dla wszystkich lubiących konstruować roboty bądź inne technicznie zaawansowane konstrukcje z klocków. Mikrokomputer wchodzący w skład zestawu Mindstorms nosi nazwę RCX.

Rys. 2.2 Sterownik RCX wersja 1.0

„Programowalny klocek" firmy LEGO jest bardziej zaawansowany technicznie niż PB 120. Do tej pory firma LEGO wydała na rynek trzy różne wersje kontrolera RCX:

Sterownik RCX 1.0

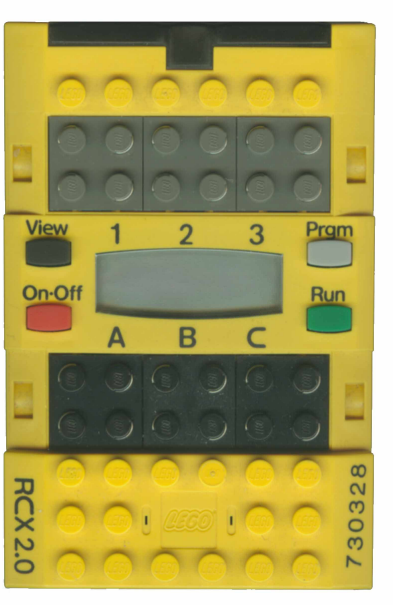
W 1996 roku firma LEGO wprowadziła do sprzedaży komputer RCX 1.0 (rys. 2.2). Konstrukcja opiera się na procesorze Hitachi H8/3292 [5]. Jest to procesor 16MHz pracujący pod napięciem 5V. Posiada 16KB wbudowanej pamięci Flash ROM oraz 32KB RAM. „Klocek" jest zasilany 6 bateriami typu AA (paluszkami). Aktualny stan pracy przedstawia wyświetlacz LCD, pokazujący aktywność każdego z trzech wejść i wyjść oraz dodatkowe informacje zdefiniowane w danym programie. Komputer ma także wbudowany głośniczek oraz dwukierunkowy port podczerwieni umożliwiający komunikowanie się RCX z komputerem lub innymi RCX.

Sterownik RCX 1.5

Jest to poprawiona wersja RCX 1.0. W obwody wstawiono kondensator 1000µF, aby podczas wymiany baterii nie kasować zawartości pamięci. Zlikwidowano możliwość zasilania mikrokomputera zewnętrznym zasilaczem. Usunięto również kilka błędów w oprogramowaniu.

Sterownik RCX 2.0

Ostatnia, najnowsza wersja mikrokomputera (rys. 2.3). Dokonano w nim kilku drobnych zmian w obwodach oraz poprawiono oprogramowanie.



Rys. 2.3 Sterownik RCX wersja 2.0

Mikrokontroler jest w stanie obsłużyć takie czujniki jak: dotyku, temperatury, światła. LEGO oferuje dodatkowy zestaw Vision Command, zawierający kolorową kamerę. Dzięki takiemu dodatkowi robot może widzieć co się przed nim znajduje i odpowiednio na to zareagować.

2.2 SILNIKI

W skład podstawowego zestawu Mindstorms wchodzą dwa silniki (nr 43362). Silniki są zasilane prądem stałym o napięciu 9V. Silniki te są urządzeniami dość nowymi – wprowadzono je do produkcji w roku 2002. Wcześniej używano silników nr 71427, które są silnikami cięższymi: 43362 ważą 28g zaś 71427 ważą 42g. Mniejsza waga silników 43362 jest ich dużą zaletą, np. odgrywa ona ogromną rolę przy budowie robotów kroczących.

CHARAKTERYSTYKA SILNIKA(43362)

Maksymalne zasilanie – 9V
Minimalny pobór prądu (bez obciążenia) – 10mA
Maksymalny pobór prądu (zablokowany) – 350mA
Maksymalna prędkość obrotowa silnika (bez obciążenia) – 350 obr/min
Prędkość obrotowa przy typowym obciążeniu silnika – 200 obr/min

ZABLOKOWANIE SILNIKA

Czasami dochodzi do zablokowania lub zatrzymania się silnika z różnych przyczyn. Bardzo ważnym jest unikanie tego typu sytuacji, gdyż można w ten sposób silnik przegrzać. Silnik 43362 jest wyposażony w zabezpieczenia, które powinny go uchronić przed spaleniem.

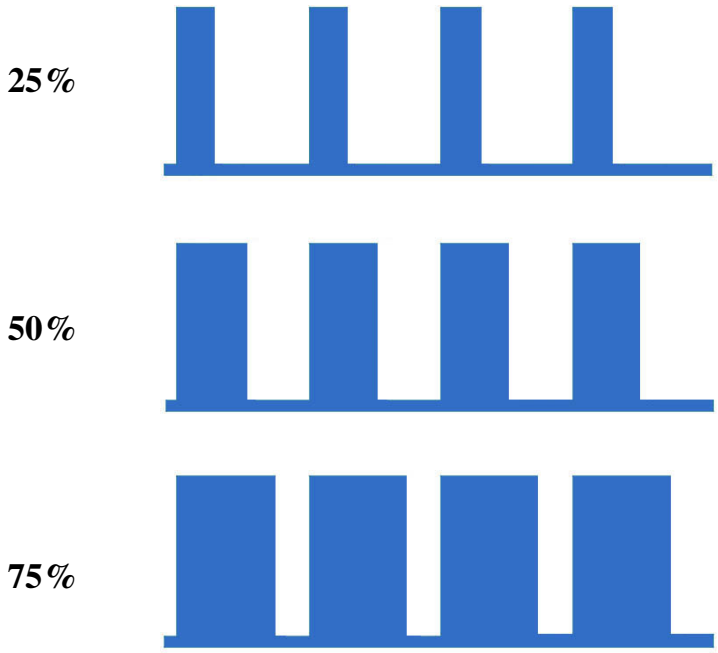
ZABEZPIECZENIA SILNIKA

- Termistor B1056 - rezystor zamontowany szeregowo z silnikiem, posiada bardzo małą rezystancję (około 1,7Ω) kiedy jest zimny, natomiast ze wzrostem temperatury wzrasta również jego rezystancja. Gdy silnik nadmiernie się grzeje, część napięcia jest odkładana na termistorze, co powoduje spadek obrotów silnika.
- Dioda BZW04-15B – połączona z silnikiem równolegle, zabezpiecza RCX przed dużymi skokami napięcia generowanymi przez motor. Nie pozwala także na dostarczenie do silnika napięcia wyższego niż 15V.

KONTROLA MOCY SILNIKA PRZEZ RCX

Moc doprowadzaną do silnika można kontrolować z poziomu programu. W języku NQC dzięki specjalnej instrukcji, można ustawić moc silnika w zakresie od 0 do 7 (w języku BrickOS jest do dyspozycji aż 256 możliwości).
RCX steruje silnikami poprzez modulację PWM (modulacja szerokości impulsu fali prostokątnej). Modulacja tego typu polega na tym, że silnik jest bardzo szybko naprzemiennie włączany i wyłączany (rys. 2.4). Moc wytworzona w trakcie jednego cyklu pracy zależy od tego jak długo silnik był włączony [6].

Gdy do silnika bez obciążenia podane zostanie zasilanie o mniejszym bądź większym stopniu wypełnienia, użytkownik nie zauważy istotnej zmiany w liczbie obrotów na sekundę. Dopiero podczas obciążenia można stwierdzić, że liczba obrotów ulega zmianie [8].



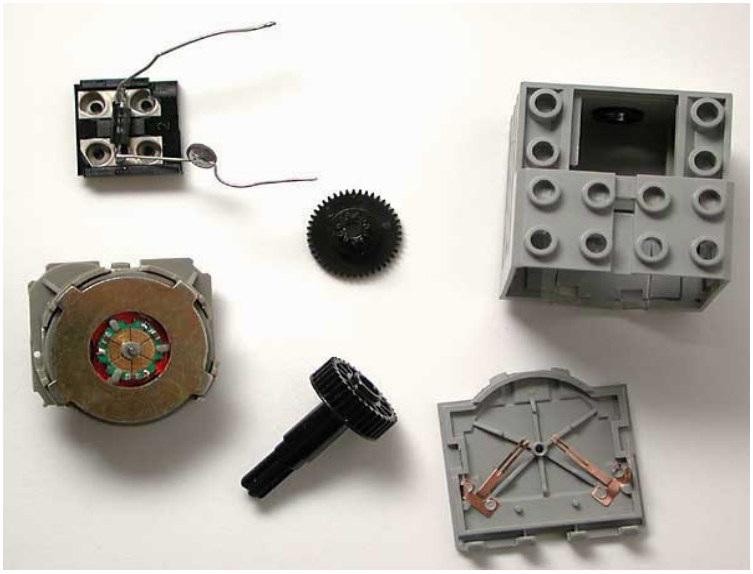
Rys. 2.4 Poziomy wypełnienia PWM

W rzeczywistości można odnieść wrażenie, że kontrolowana jest liczba obrotów silnika. Jednakże jest to tylko złudzenie – kontrolowana jest jego moc. Należy pamiętać, iż silnik bez obciążenia lub z minimalnym obciążeniem przy zmianie mocy nie zmieni

liczby obrotów. Dopiero przy normalnym obciążeniu można zaobserwować jak zmiana mocy wpływa na liczbę obrotów silnika [16].

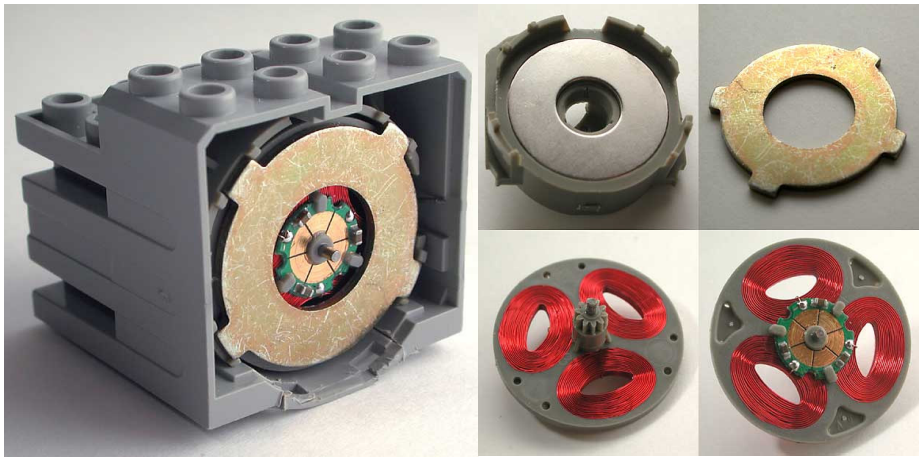
BUDOWA

Rys. 2.5 przedstawia wszystkie części składowe silnika.



Rys. 2.5 Części składowe silnika nr 43362

Na zdjęciu widoczne są: obudowa, tylnia ścianka wraz z umieszczonymi na stałe szczotkami, zestaw kół zębatych stanowiących przekładnię 1/14, cewki wraz z magnesami oraz port połączeniowy wraz z zabezpieczeniem termicznym oraz diodą.



Rys. 2.6 Cewki silnika nr 43362

Fragment rys. 2.6 z lewej strony pokazuje budowę silnika po zdjęciu tylnej ścianki. Prawa część rys. 2.6 ilustruje jak wygląda zespół napędowy silnika (cewki oraz magnesy).

Rys. 2.7 Silnik nr 43362 bez obudowy

Rys. 2.7 przedstawia całość wnętrza silnika po zdjęciu obudowy. Widoczne są: przekładnia, port połączeniowy, termistor, dioda, obudowa magnesów i cewek.

2.3 CZUJNIKI LEGO

RCX posiada trzy niezależne wejścia dla różnego rodzaju czujników [7]. Standardowo w zestawie LEGO Mindstorms można znaleźć dwa czujniki dotyku oraz jeden czujnik światła. Dodatkowo można dokupić czujniki temperatury oraz obrotów. Wiele osób stara się opracować swoje własne czujniki np. dźwięku, ciśnienia itp.

RCX odczytuje wyniki pomiarów z czujników, które mogą pracować w dwóch trybach: aktywnym (czujnik światła, obrotów) oraz pasywnym (dotyku, temperatury). Napięcie pojawiające się na wejściu jest przekształcane do wartości RAW odpowiednio: 0V = 0 oraz 5V = 1023. W zależności od potrzeb, z poziomu programu, będzie można odczytać tę wartość w różny sposób.

Michael Gasperi [7] opracował tabelę wartości RAW dla czujników światła, temperatury oraz dotyku.

Napięcie [V]	**Wartość RAW**	**Oporność czujnika [Ω]**	**Czujnik światła**	**Czujnik temperatury**	**Czujnik dotyku**
0.0	0	0	-	-	1
1.1	225	2816	-	70.0	1
1.6	322	4587	100	57.9	1
2.2	450	7840	82	41.9	1
2.8	565	12309	65	27.5	0
3.8	785	32845	34	0.0	0
4.6	945	119620	11	-20.0	0
5.0	1023	∞	0	-	0

- Jeśli wartość RAW odczytana z czujnika dotyku jest mniejsza lub równa 450 RCX zwraca 1, jeśli jest większa lub równa 565 RCX zwraca 0.
- RCX wskazuje odczytaną temperaturę w °C wg zależności:

  $$\text{temperatura} = \frac{(785 - RAW)}{8}$$

  Zakres czujnika wynosi od -20°C do +70°C.
- Przy pomiarze światła RCX zwraca wartość od 1 do 100 wyliczaną wg wzoru: światło=146-RAW/7

Dla czujników pasywnych takich jak czujnik dotyku czy temperatury, RCX odkłada na wejściowym rezystorze (10kΩ) napięcie 5V. Czujnik pasywny, podłączony równolegle do tego rezystora, musi stanowić pewną rezystancję, aby RCX mógł dokonać pomiaru.

Pomiar dla czujników aktywnych wygląda nieco inaczej. Komunikacja czujnika z RCX odbywa się dwufazowo:

1. okres 2,9ms – RCX zasila czujnik napięciem 8V.
2. okres 0,1ms – RCX dokonuje pomiaru analogowej wartości zwracanej przez czujnik (identycznie jak w przypadku czujnika pasywnego). W tej fazie napięcie zasilające jest odcięte, a czujnik pobiera energię z wewnętrznego kondensatora.

2.3.1 Czujnik dotykowy

Jednym z najprostszych czujników pasywnych w zestawie LEGO Mindstorms jest czujnik dotyku (rys. 2.8).

Rys. 2.8 Czujnik dotykowy

Jego konstrukcja opiera się na styczniku, połączonym szeregowo z rezystorem. W stanie rozwarcia rezystancja stycznika jest nieskończona, natomiast podczas zwarcia wynosi 500Ω. Czujnik ten doskonale nadaje się do wykrywania przeszkód.

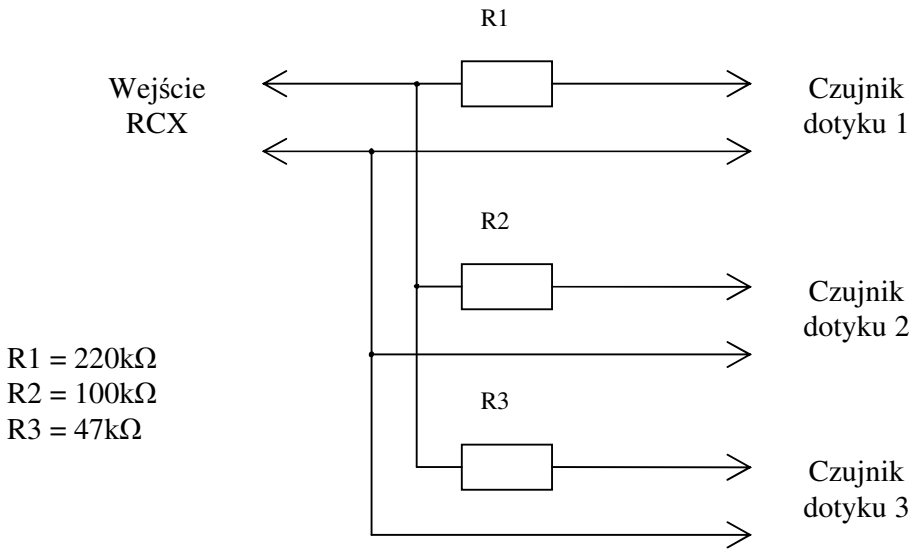
Niektóre projekty mogą wymagać podłączenia do RCX np. trzech czujników dotyku oraz czujnika światła. Nie można bezpośrednio do RCX podłączyć czterech czujników. Można łączyć czujniki ze sobą równolegle, jednakże uniemożliwia to identyfikację aktualnie naciśniętego czujnika.

Paul Hass opracował multiplekser [12], który rozwiązuje powyższy problem. Tego typu multiplekser (rys. 2.9), badając wartości zwracane RAW, pozwala stwierdzić, który z czujników został naciśnięty, można również rozróżnić czy dwa lub trzy czujniki zostały naciśnięte w tym samym czasie.

Tabela wartości RAW:

Wejście			**Wartość RAW**
3	**2**	**1**	
0	0	0	1023 – 1001
0	0	1	1000 - 955
0	1	0	954 - 912
0	1	1	911 - 869
1	0	0	868 - 830
1	0	1	829 - 798
1	1	0	797 - 768
1	1	1	767 - 0

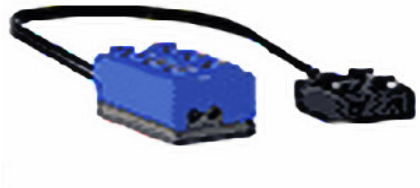
Schemat multipleksera:



Rys. 2.9 Multiplekser

2.3.2 Czujnik światła

Aktywny czujnik światła (rys.2.10) wchodzi w skład podstawowego kompletu LEGO Mindstorms. Dokonuje on pomiaru natężenia światła od 0,6 Lux do 760 Lux.



Rys. 2.10 Czujnik światła

Budowa czujnika światła jest dość skomplikowana[1]. Z przodu czujnika widoczne są dwa elementy elektroniczne: dioda oraz fototranzystor. Fototranzystor rejestruje

[1] dokładny schemat czujnika można znaleźć na stronie WWW: http://www.plazaearth.com/usr/gasperi/light.htm.

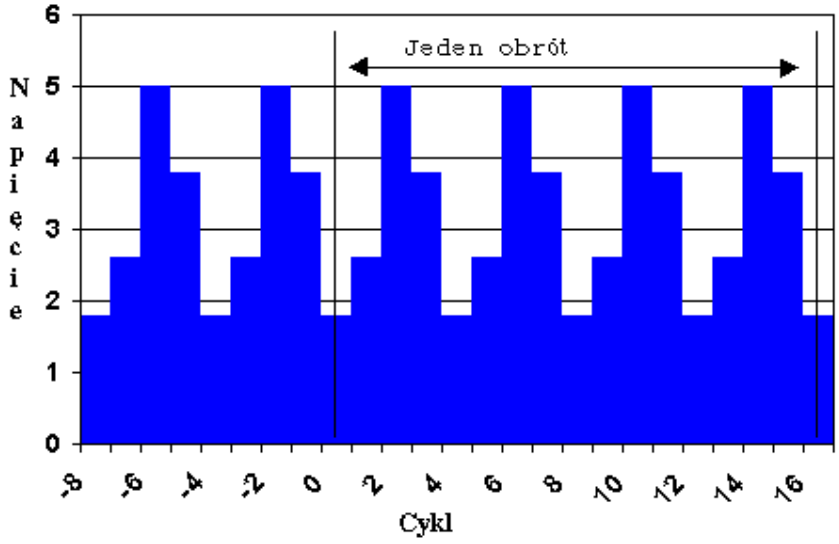
natężenie docierającego doń światła, natomiast dioda wykorzystywana jest do oświetlania otoczenia czujnika.

Czujnik światła może być wykorzystany np. do konstrukcji pojazdów jeżdżących wzdłuż linii (pojazdy, których trasa jazdy jest wyznaczona np. czarną linią na podłożu) czy też wykrywających źródło promieniowania IR.

2.3.3 Czujnik obrotów

Czujnik obrotów firmy LEGO (rys. 2.11) jest aktywnym czujnikiem, umożliwiającym komputerowi RCX pomiar obrotów z dość dużą rozdzielczością: 16 cykli na jeden obrót. (1 cykl = 22,5°).

Rys. 2.11 Czujnik obrotów



Rys. 2.12 Schemat podawanych napięć podczas jednego obrotu

Czujnik może zwrócić do RCX napięcie o wartości: 1,8V, 2,6V, 3,8V oraz 5V (rys. 2.12). Kolejność podawania tych napięć informuje RCX czy czujnik obraca się zgodnie z kierunkiem wskazówek zegara czy też nie. Na ¼ obrotu, czyli 90°, składają się cztery cykle; kolejność podawania napięć może wyglądać następująco (w zależności od kierunku rotacji): 1,8V -> 2,6V -> 5V -> 3,8V

2.3.4 Czujnik temperatury

Czujnik temperatury (rys. 2.13) jest czujnikiem biernym. Czujnik dokonuje pomiaru temperatury w zakresie od -20 do + 50°C.

Rys. 2.13 Czujnik temperatury

Element ten nie wchodzi w skład podstawowego zestawu Mindstorms. Można się pokusić o samodzielne wykonanie czujnika, gdyż jego konstrukcja jest bardzo prosta: jest to układ dwuelementowy składający się z termistora oraz szeregowo połączonego z nim rezystora 2,2 kΩ.

2.4 EKSPERYMENTALNY CZUJNIK OBROTÓW

W ramach pracy magisterskiej, skonstruowano dwa proste czujniki obrotów. Każdorazowo, gdy czujnik wykona jeden obrót, do RCX zwracany jest odpowiedni sygnał. Czujnik może być podłączony do jakiegokolwiek elementu obracającego się. W tym przypadku czujnik, podłączony do silnika, zliczał długość drogi, jaką pokonał robot.

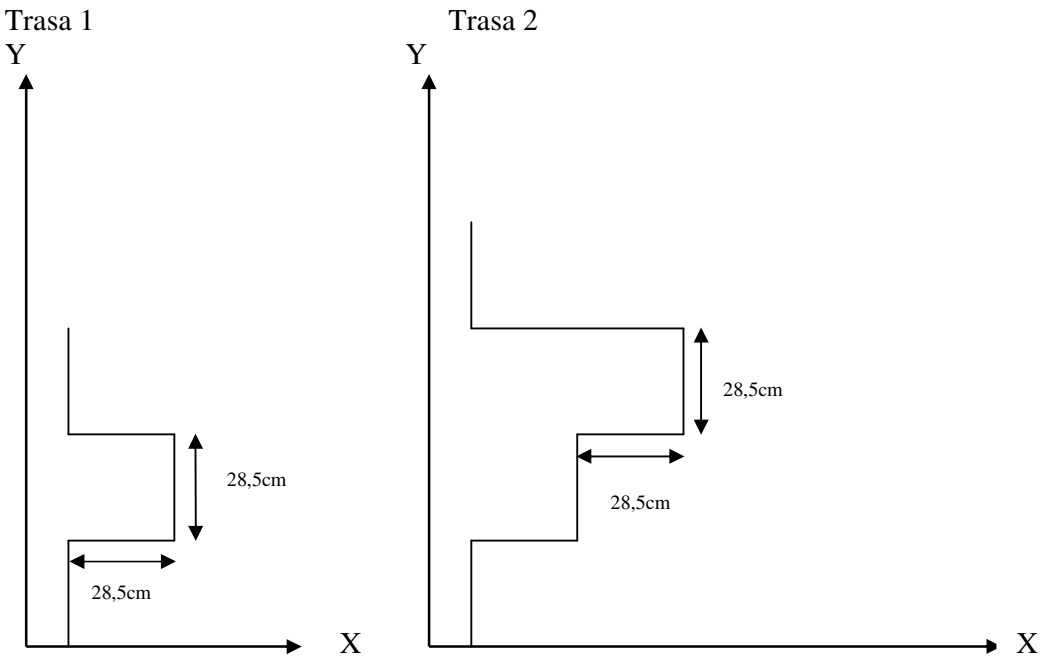
BUDOWA:

Konstrukcja czujnika oparta jest na czujniku dotyku. Wiedząc, że czujnik dotyku w czasie zwarcia stanowi opór 0,6kΩ, można było skonstruować własny czujnik (rys. 2.17 i 2.18, str. 24). Stycznik jest umieszczony tuż nad kołem zębatym. Za każdym razem, gdy koło wykona obrót, stycznik zostaje zamknięty. Do stycznika szeregowo podłączone są dwa potencjometry wyskalowane na 0,6kΩ. Całość jest podłączana do jednego z wejść RCX. Dla RCX zamknięty stycznik to nic innego jak wciśnięty standardowy czujnik dotyku. Z poziomu programu można z odpowiednią dokładnością stwierdzić jak długą trasę pokonał robot.

DOŚWIADCZENIE:

Robot porusza się po zaprogramowanej trasie (rys. 2.14) korzystając z czujnika ruchu.

Schemat trasy:



Rys. 2.14 Schematy tras

Idea jazdy po trasie z czujnikiem ruchu:

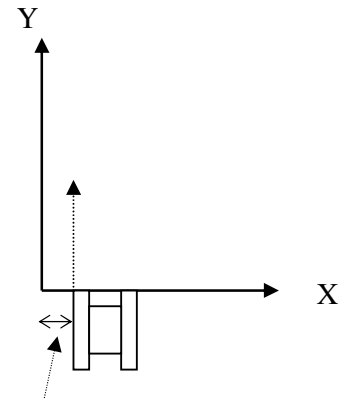
Robot oblicza pokonaną trasę korzystając z czujnika ruchu. Czujnik jest tak skonstruowany, że co 2,8cm przekazuje robotowi sygnał iż taka właśnie odległość została pokonana. Zakręty są wykonywane z wykorzystaniem czasu (timerów) - czujnik

ruchu nie bierze udziału przy wykonywaniu skrętu. Dokonano serii pomiarów czasu (czas, przez który jeden z silników ma być włączony), dla których robot wykona zakręt 90° w miejscu.
Robot zostaje zaprogramowany w taki sposób, że kolejno podaje mu się liczbę zliczeń czujnika ruchu (ile ma przejechać w linii prostej) oraz zakręty.

Przebieg doświadczenia:

Doświadczenie polegało na pokonaniu przez robota dwóch różnych tras.

Pozycja startowa dla wszystkich prób:



Rys. 2.15 Pozycja startowa robota

1) **trasa 1**

Program napisany w języku BrickOS <robots.c>:

```
#include <conio.h>
#include <unistd.h>
#include <dsensor.h>
#include <dmotor.h>

int i,j,k,l,m=0;
int odl1=20;
int odleglosc2=30;

void zakret_prawo(){                          //zakret w prawo o
90stopni
                      motor_a_speed(255);     //max predkosc silnika
                      motor_c_speed(255);
                      motor_a_dir(fwd);
                      motor_c_dir(rev);
```

```
                        msleep(1800);}          //niech sie obraca
                                                przez 180ms
void zakret_lewo(){                             //zakret w lewo o
90stopni

                        motor_a_speed(255);     //max predkosc silnika
                        motor_c_speed(255);
                        motor_a_dir(rev);
                        motor_c_dir(fwd);
                        msleep(1920);           //niech sie obraca
                                                przez 192ms
 }
void stopmotors(){                              //motory stop
    motor_a_speed(0);
    motor_c_speed(0);
}

void touch_sensor()
{
  lcd_refresh();
  motor_a_speed(255);
  motor_c_speed(255);
       while(1)
    {
    motor_a_dir(fwd);
    motor_c_dir(fwd);

    if(TOUCH_1!=0)                              //sprawdzamy czy czujka
                                                obrotow nacisnieta
        {  i=i+1;   j=j+1  ; k=k+1; l=l+1; m=m+1;
    delay(27);
     lcd_int(i);                                //wyswietla liczbe zliczonych
                                                obrotow
           if  (i==10) {
           zakret_prawo();
             }
           if(j==20){
           zakret_lewo();
             }
           if(k==30){                           //sekcja odpowiedzialna za
                                                ksztalt toru jazdy
           zakret_lewo();
             }
           if(l==40){
           zakret_prawo();
             }
           if(m==50){
           stopmotors();
             }
     }
    }
```

Wyniki:

Próba	Punkt startowy		Oczekiwany punkt końcowy		Rzeczywisty punkt końcowy		Błąd	
	X[cm]	Y[cm]	X[cm]	Y[cm]	X[cm]	Y[cm]	X[cm]	Y[cm]
1	5	0	5	85,5	1	84,5	4	1
2	5	0	5	85,5	8	84,5	3	1
3	5	0	5	85,5	2,5	86	2,5	0,5
4	5	0	5	85,5	4,5	89	0,5	3,5
5	5	0	5	85,5	6	87,5	1	2

2) trasa 2

Program napisany w języku BrickOS <robots2.c>:

```
#include <conio.h>
#include <unistd.h>
#include <dsensor.h>
#include <dmotor.h>

int i,j,k,l,m,n,o=0;
int odl1=20;
int odleglosc2=30;

void zakret_prawo(){                              //zakret w prawo o
90stopni
            motor_a_speed(255);                   //max predkosc silnika
            motor_c_speed(255);
            motor_a_dir(fwd);
            motor_c_dir(rev);
            msleep(1800);}          //niech sie obraca przez 180ms

void zakret_lewo(){                               //zakret w lewo o
90stopni
            motor_a_speed(255);
            motor_c_speed(255);                   //max predkosc silnika
            motor_a_dir(rev);
            motor_c_dir(fwd);
            msleep(1920);                         //niech sie obraca
                                                  przez 192ms
 }

void stopmotors(){                                //motory stop
    motor_a_speed(0);
    motor_c_speed(0);
}

void touch_sensor(){
  lcd_refresh();
  motor_a_speed(255);
  motor_c_speed(255);
       while(1)
    {
    motor_a_dir(fwd);
    motor_c_dir(fwd);
```

```
    if(TOUCH_1!=0)                                  //sprawdzamy czy czujka
                                                    obrotow nacisnieta
        {  i=i+1; j=j+1; k=k+1; l=l+1; m=m+1; n=n+1; o=o+1;
    delay(27);
    lcd_int(i);                                      //wyswietla liczbe
                                                     zliczonych obrotow
           if  (i==10) {
           zakret_prawo();
             }
           if(j==20){
           zakret_lewo();
             }
           if(k==30){
           zakret_prawo();
             }                                       //sekcja odpowiedzialna
                                                     za ksztalt toru jazdy
           if(l==40){
           zakret_lewo();
             }
           if(l==50){
           zakret_lewo();
             }
           if(l==70){
           zakret_prawo();
             }
           if(m==80){
           stopmotors();
             }
     }
    }
}
int main(){
      touch_sensor();

return 0;
}
```

Wyniki:

Próba	Punkt startowy		Oczekiwany punkt końcowy		Rzeczywisty punkt końcowy		Błąd	
	X[cm]	Y[cm]	X[cm]	Y[cm]	X[cm]	Y[cm]	X[cm]	Y[cm]
1	5	0	5	114	9	116,5	4	2,5
2	5	0	5	114	14	126	9	12
3	5	0	5	114	6	127,5	1	13,5
4	5	0	5	114	12	127	7	13
5	5	0	5	114	2	118	3	4

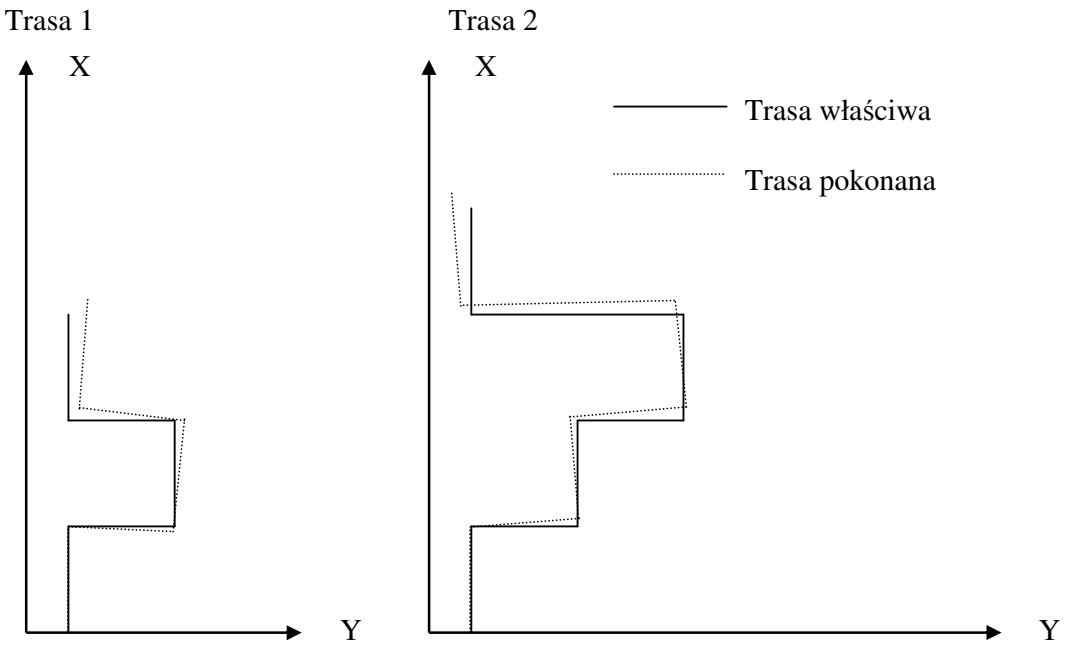
Wnioski:

Główną przyczyną powstawania błędów jest budowa podwozia robota. Testowany robot poruszał się na gąsienicach, które bardzo dobrze sprawdzają się przy jeździe na wprost, natomiast na zakrętach stawiają duży opór. Dodatkowym czynnikiem zwiększającym błędy na zakrętach (średni błąd przy zakrętach 90° wyniósł +/- 10°),

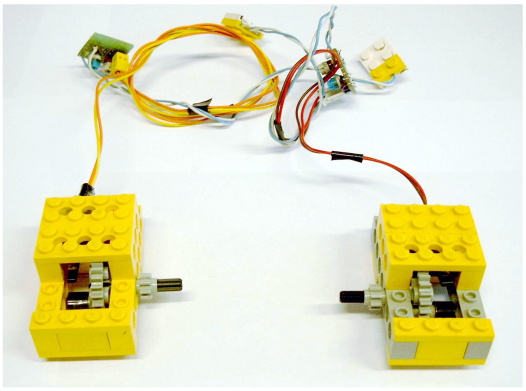
była wykładzina o dość chropowatej fakturze. Oba te czynniki powodowały przypadkowe i niemożliwe do skompensowania błędy. W porównaniu z trasą pierwszą, błędy dla trasy drugiej są większe ze względu na dłuższą trasę. Ponadto błąd rośnie wraz z zużyciem akumulatorków zasilających RCX (ma to bardzo duże znaczenie przy wykonywaniu zakrętów).

Skonstruowany czujnik nie ma zbyt dużej rozdzielczości. Jeden takt czujnika sygnalizuje pokonanie przez robota odległości 2,8cm. Z tego wniosek, że dana odległość (przy jeździe na wprost) może zostać pokonana z błędem +/- 5,6cm.

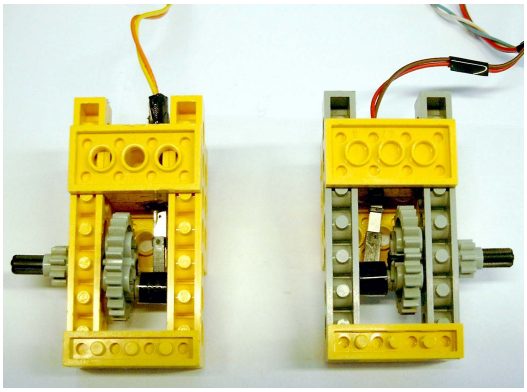
Przykładowe trajektorie trasy pokonanej przez robota:



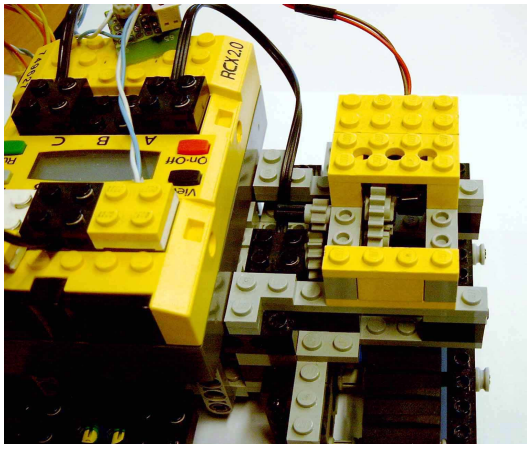
Rys. 2.16 Przykładowe trasy, po których poruszał się robot



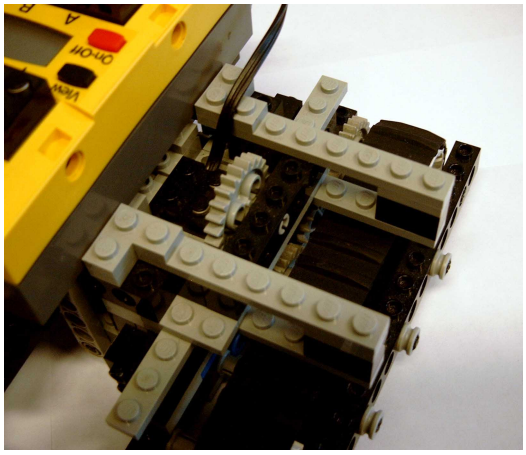
Rys. 2.17 Czujniki ruchu – widok z góry.



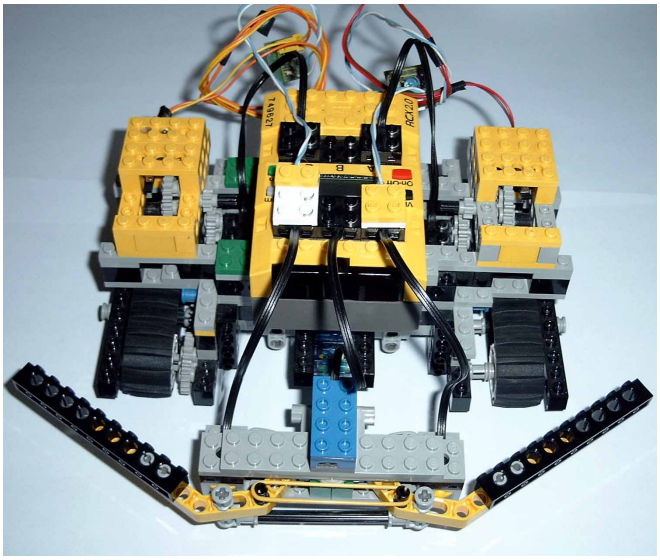
Rys. 2.18 Czujniki ruchu – widok od spodu.



Rys. 2.19 Czujnik zamontowany na robocie.



Rys. 2.20 Miejsce do montażu czujnika.



Rys. 2.21 Robot z zamontowanymi czujnikami gotowy do jazdy.

3 BRICX COMMAND CENTER

1. Wstęp

Bricx Command Center (BricxCC) [2] jest zintegrowanym środowiskiem programistycznym służącym do programowania robotów firmy LEGO: RCX (wszystkie wersje), Scout, Cybermaster oraz Spybot. Program pracuje w środowisku Windows (95, 98, ME, NT, W2K, XP). Początkowo obsługiwał tylko język NQC, ale w najnowszych wersjach (aktualnie najnowsza to 3.3.7.7), zapewnia wsparcie także dla innych języków między innymi: Mindscript, BrickOS, Pascal. Autorem programu jest Mark Overmars, a najnowsze wersje od 3.3, są rozwijane przez Johna Hansena.

2. Instalacja

W przypadku używania tylko języka NQC należy ściągnąć ze strony autora http://sourceforge.net/projects/bricxcc/ najnowszą wersję Bricx Command Center (3.3.7.7), która posiada już pełną dystrybucję języka NQC w wersji 3.0a1.

W pracy, oprócz języka NQC, zostało użytych także kilka innych języków i wystąpiła konieczność ściągnięcia dodatkowych plików.

1. The BricxCCSetupCygwin installer - jest to środowisko Cygwin z kompilatorem GNU GPC oraz wymaganymi bibliotekami
   http://bricxcc.sourceforge.net/BricxCCSetupCygwin.exe
2. The BricxCCbrickOSleJOS installer - środowisko BrickOS w wersji 0.2.6.10 oraz biblioteki umożliwiające użycie języka Pascal dla kompilatora BrickOS
   http://bricxcc.sourceforge.net/BricxCCbrickOSleJOS.exe
3. Bricx Command Center (3.3.7.7) http://sourceforge.net/projects/bricxcc/
4. Lego Mindstorms SDK 2.0 – potrzebny, aby programować w języku MindScript
   http://mindstorms.lego.com/sdk2/LEGOMindStormsSDK.zip

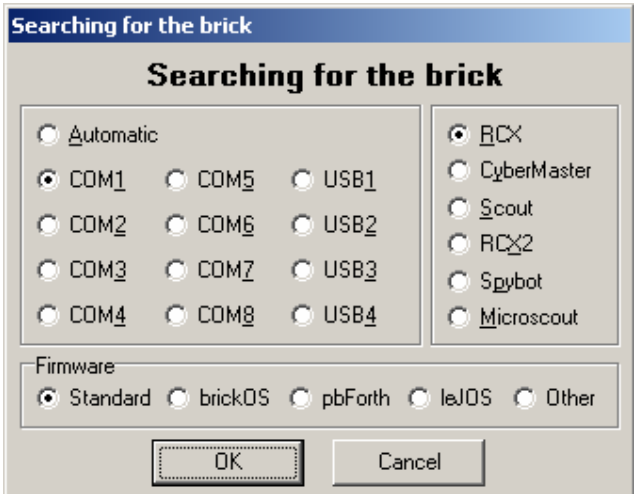
Po ściągnięciu powyższych plików należy je zainstalować w podanej powyżej kolejności. Zaleca się **nie modyfikować** ścieżek instalacyjnych, ponieważ będzie to wymagało ręcznej zmiany ścieżek w plikach konfiguracyjnych.

3. Uruchamianie

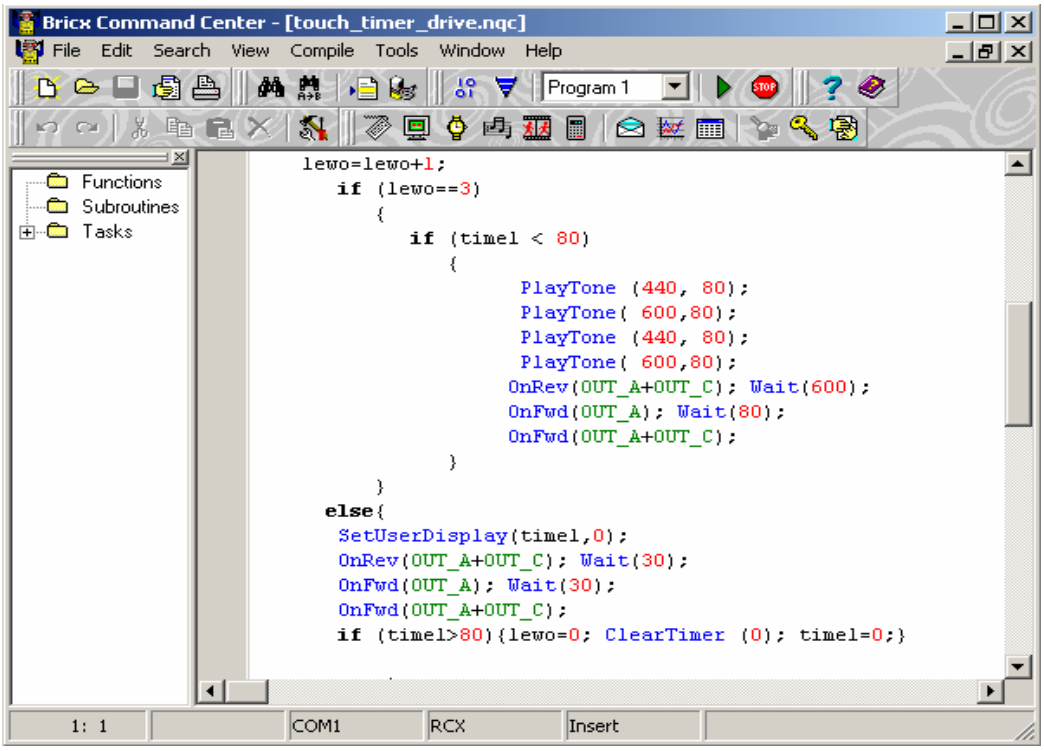
Po starcie programu pojawi się poniższe okienko. Służy ono do wyboru portu, przez który komputer komunikuje się z wieżą, typu robota i rodzaju firmware-u (specjalne

oprogramowanie znajdujące się w pamięci RCX, charakterystyczne dla danego języka). W przypadku portu najlepiej wybrać opcję *Automatic*, program sam znajdzie wieżę. Jako typ robota można wybrać RCX lub RCX2, ale dla firmware-ów innych niż standardowe jedynie opcja RCX jest aktywna. Typ firmware-u zależy od tego, w jakim języku użytkownik ma zamiar programować.

1. Standardowy – NQC, Mindscript
2. BrickOS – BrickOS, Pascal



Po wybraniu ustawień początkowych otworzy się właściwe okno programu:



Okno programu można podzielić na kilka części:

1. Menu główne – dostęp do wszystkich funkcji i narzędzi edytora
2. Paski narzędzi – przyciski do najczęściej używanych opcji np.: zapisz, otwórz, znajdź, kompiluj, uruchom
3. Code Explorer – jest to okno znajdujące się po lewej stronie okna edytora. Umożliwia ono dostęp do wszystkich funkcji, podprogramów i zadań w programie. Podwójne kliknięcie nazwy danej funkcji, podprogramu czy zadania ustawia kursor na jej początku
4. Okno edytora – służy do edycji otwartego programu. Styl pracy podobny jest do pracy ze standardowym edytorem tekstu.
5. Pasek stanu – pokazuje informacje na temat połączenia z robotem, numer aktualnej linii, w której znajduj się kursor
6. Okno błędów – pojawia się poniżej okna edytora, po kompilacji w przypadku, gdy w programie znajdują się błędy. Po kliknięciu w opis błędu kursor zostanie automatycznie przeniesiony do linii, w której znajduje się błąd.

4. Otwieranie i zamykanie plików

Do wszystkich typów plików, obsługiwanych przez program, można uzyskać dostęp z menu **File** lub za pomocą ikon w pasku narzędzi. W menu **File** znajdują się wszystkie podstawowe funkcje:

- **New -** otwórz pusty plik
- **Open** – otwórz plik
- **Save, Save as, Save all** – zapisz plik, zapisz jako, zapisz wszystkie
- **Close, Close all** – zamknij, zamknij wszystkie
- **Insert file** – wstawia wybrany plik na obecną pozycję kursora

W menu **File** znajdują się też opcje drukowania i ścieżki do ostatnio otwieranych plików.

5. Kompilacja, zgrywanie programu i uruchamianie na robocie

Aby skompilować program należy z menu **Compile** wybrać pozycję pierwszą **Compile** lub nacisnąć F5. Jeśli kompilator wykryje błędy otworzy się okno błędów. Po kliknięciu w opis błędu kursor zostanie automatycznie przeniesiony do linii, w której znajduje się błąd. Okno błędów można zamknąć wybierając opcję **Hide Errors** w menu **View**. Aby

uzyskać pełny opis błędu należy wybrać opcję **Show Code/Error Listing** z menu **View**.

Można także skompilować program i od razu załadować go do pamięci robota. Na początek należy w pasku narzędzi lub w menu **Compile** wybrać numer programu **Program Number**, do którego ma być zgrany program, a następnie wybrać opcję **Download** F6**.** Spowoduje to skompilowanie programu i w przypadku braku błędów przesłanie do robota.

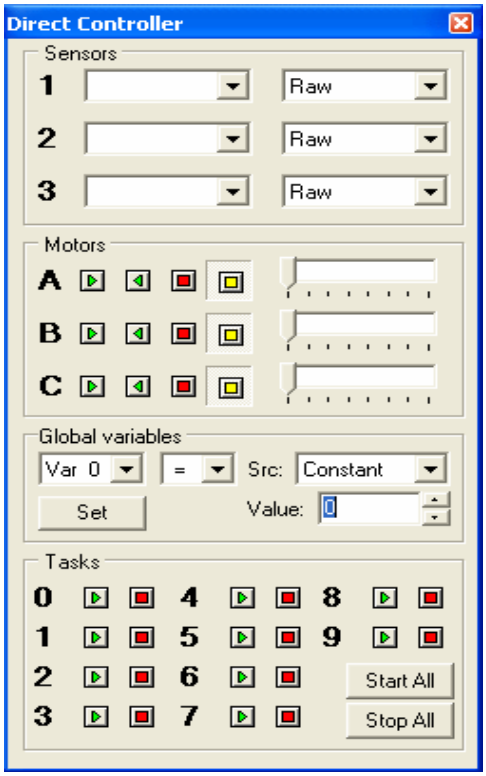
Aby natychmiast uruchomić program po zgraniu należy wybrać opcję **Download and run**.

Opcje **Run** i **Stop** służą do uruchamiania i zatrzymywanie programów już wgranych do robota.

6. Narzędzia

BricxCC posiada kilkanaście narzędzi przydatnych w czasie pracy z robotem RCX. Dostęp do poszczególnych narzędzi zależy od języka, w jakim aktualnie robot jest programowany.

6.1 Direct Controller



Narzędzie to umożliwia bezpośrednie sterowanie robotem.

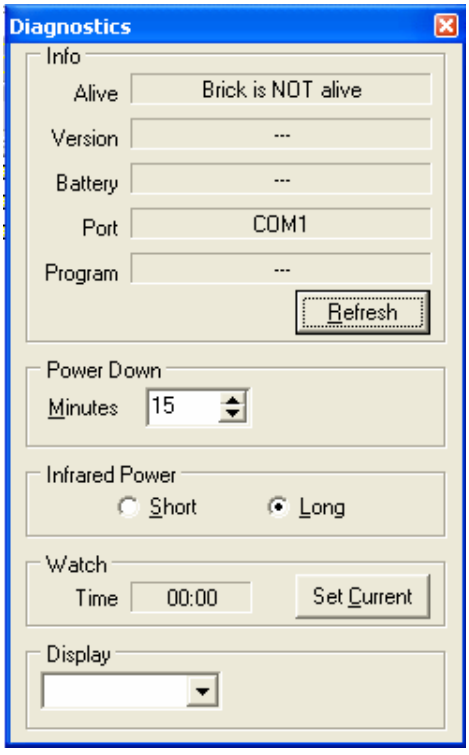
Sensors - tutaj ustawia się typ oraz tryb podłączonych czujników.

Motors - ustawienia silników. Można ustawić kierunek do przodu, do tyłu, wyłączyć je lub też ustawić na wolnym biegu. W tej sekcji ustawia się także prędkość silników.

Global variables – ustawienia zmiennych dla programu. Kolejno wybiera się numer zmiennej, operatora przypisania, typ zmiennej lub jej wartość. Zmiany zatwierdza się przyciskiem **Set**.

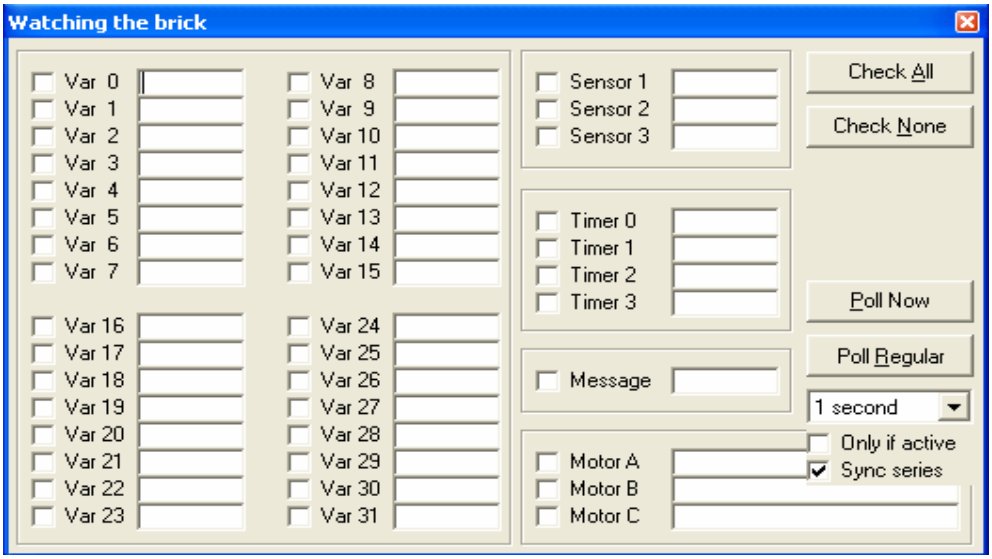
Tasks – w tej sekcji można rozpocząć lub zatrzymać zadania (dla RCX 1.0), które są aktualnie zapisane w robocie. Opcja ta jest przydatna podczas testowania programów.

6.2 Diagnostics



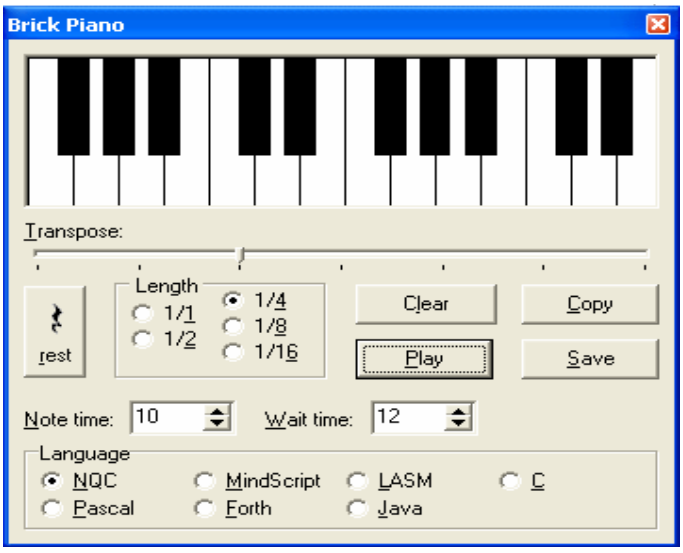
Wyświetla informacje o klocku: czy jest włączony, podaje wersję firmware-u, stan baterii, port, za pomocą którego komunikuje się wieża z komputerem, wybrany program. Przy pomocy tego narzędzia można również ustawić czas, po którym RCX wyłączy się samoczynnie, moc nadajnika IR, zegar czasu rzeczywistego, a także co ma być pokazywane na wyświetlaczu.

6.3 Watching the brick



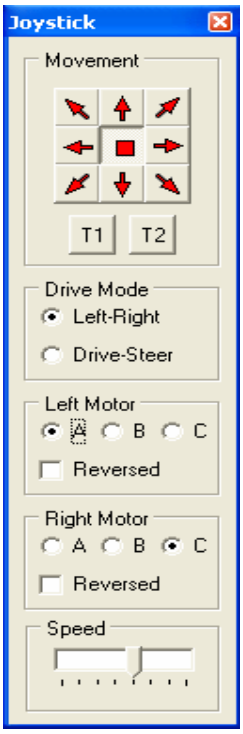
Narzędzie to wyświetla stan zmiennych, czujników, silników, timerów i bufora wiadomości. W zależności od tego, co ma być obserwowane, można zaznaczyć interesujące wartości bądź wybrać wszystkie przyciskiem **Check All**. Aby wyczyścić zaznaczenia wystarczy użyć przycisku **Check None**. Aby pobrać informacje wystarczy użyć przycisku **Poll Now**. Narzędzie pozwala także na ciągłe sprawdzanie stanu zmiennych, służy do tego przycisk **Poll Regular**. Wartości będą odczytywane okresowo, co 100ms – 10 s. Należy pamiętać, że odczyt wartości zajmuje czas i może spowolnić działanie komputera i robota.

6.4 Brick Piano



RCX umożliwia odgrywanie melodii. Narzędzie to ma na celu ułatwienie ich tworzenia. Za pomocą myszki można zagrać melodię jak na pianinie. Można też ustawić długość tonu. Melodię można odegrać, zapisać do pliku lub skopiować do okna edytora. W dolnej części okna można wybrać, w jakim języku ma być zapisana melodia.

6.5 Joystick

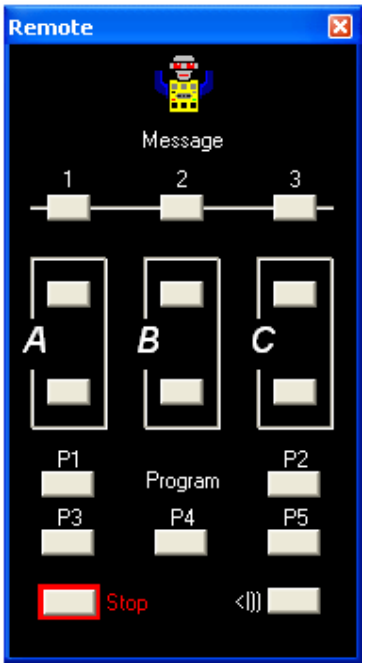


Jak nazwa wskazuje, narzędzie to służy do sterowania robotem w sposób podobny do joysticka. Joystick może działać w dwóch trybach pracy:

1. Left - Right – oba silniki napędzają po jednej gąsienicy
2. Drive – Steer – jeden silnik służy jako napęd, a drugi do sterowania

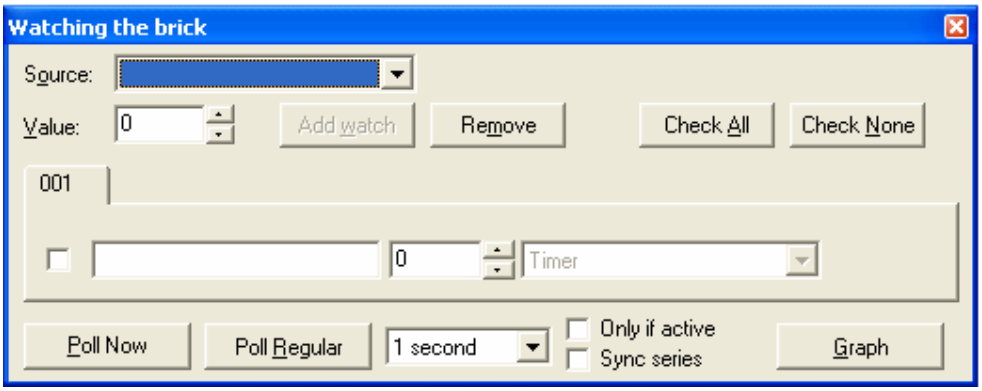
Po wybraniu trybu można ustawić poszczególne silniki i ich prędkość. Po ustawieniu silników można sterować robotem za pomocą strzałek na górze okna, klawiatury numerycznej (należy pamiętać o włączonym NumLock-u) lub za pomocą zwykłego joysticka. Przyciski **T1** i **T2** służą do uruchamiania zadań zapisanych w robocie. Do tego samego można wykorzystać przyciski na joysticku.

6.6 Remote



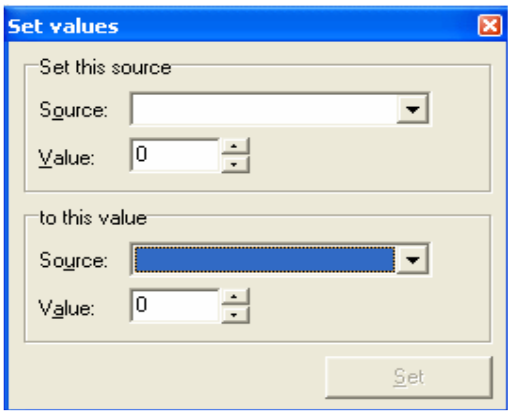
Jest to emulator klocka sterującego (zdalnego pilota), wysyłający polecenia sterujące do robota.

6.7 Configure watch



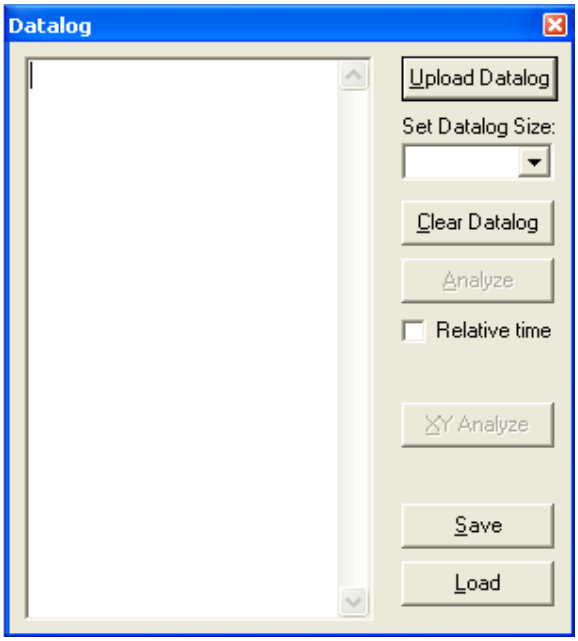
Narzędzie pozwala na sprawdzanie wartości zmiennych, timerów, liczników, podobne funkcjonalnie do **Watching the brick.**

6.8 Set values



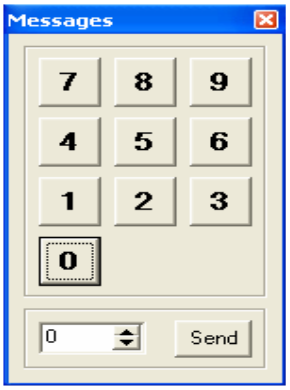
Pozwala na ustawienie wartości zmiennych, liczników, timerów

6.9 Datalog



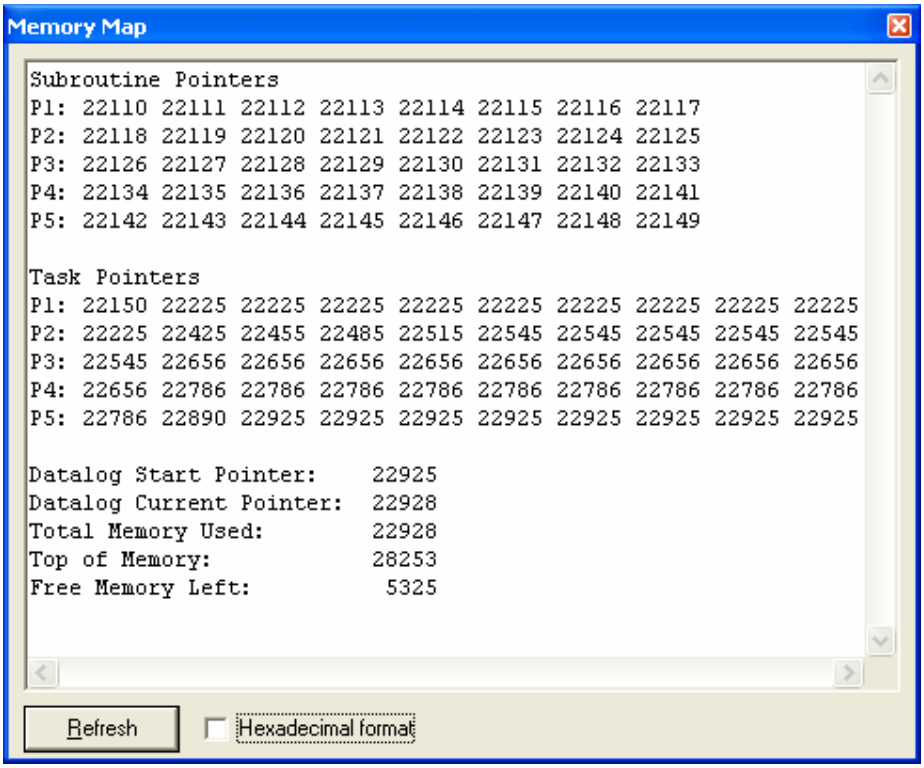
Narzędzie do pobierania danych z datalogu do komputera. Przycisk **Upload Datalog** pobiera dane z robota. Można też ustalić rozmiar datalogu, ale należy pamiętać, że każdy wpis zajmuje 3 bajty. **Clear Datalog** czyści rejestr przez ustawienie jego rozmiaru na 0. Po pobraniu danych można je wyświetlić na grafie. **Analyze** traktuje wszystkie dane jako jednakowego typu i zaznacza ich wartości na osi Y, a na osi X zaznacza kolejne wpisy. **XY Analyze** traktuje dane jako parę o współrzędnych (XY). Dane można zapisać też w pliku tekstowym.

6.10 Send Messages



Pozwalana na wysłanie wiadomości do robota, wiadomość może być liczbą z zakresu 0-255.

6.11 Memory Map



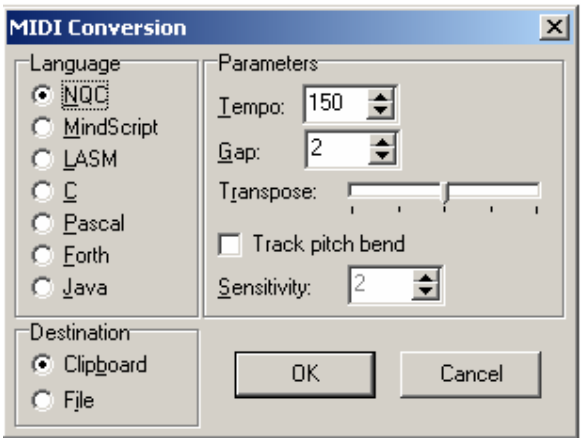
Podaje informacje pobrane z pamięci robota:

- Adresy początkowe 8 podprogramów w każdym programie
- Adresy początkowe 10 zadań w każdym programie
- Adres początkowy datalogu
- Adres ostatniego elementu w datalogu

- Ilość zajętej pamięci
- Ilość wolnej pamięci

6.12 Clear Memory – kasuje pamięć robota. Usuwa wszystkie zadania, podprogramy i datalog.

6.13 MDI Conversion



Konwertuje pliki muzyczne midi do zapisu w wybranym języku.

6.14 Find Brick

Otwiera okno jak przy starcie programu, służące do ustawienia typu robota, firmware-u i portu komunikacyjnego.

6.15 Turn Brick off

Wyłącza robota.

6.16 Close Communication

Zamyka połączenie z robotem.

6.17 Download Firmware

Umożliwia zgranie do robota firmware-u, zarówno standardowego jak i innego. Po wybraniu tego narzędzia otwiera się okno wyboru pliku (standardowy to plik z rozszerzeniem .lgo, a alternatywne z .srec). Po wybraniu pliku pojawi się pasek postępu, pokazujący szacunkowy postęp ładowania. W obecnej wersji programu 3.3.7.7

występuje błąd podczas ładowania powodujący uszkodzenie oprogramowania i ładowanie firmware-u należy wykonywać z poziomu środowiska Cygwin.

7. Download firmware-u

W poniższym punkcie przedstawiono jak załadować firmware standardowy i BrickOS do robota.

- Standardowy (do obsługi języków RCX Code, Mindscript, NQC)
    1. Wyjąć baterie na kilka sekund najlepiej, gdy kostka jest włączona
    2. Uruchomić Robotics Inventory System
    3. Wybrać opcję *Settings* – program powinien automatycznie rozpocząć instalację, jeśli nie, należy wybrać opcję ładowania firmware-u
- Firmware BrickOS (obsługuje języki BrickOS, Pascal)
    1. Uruchomić Cygwin
    2. W linii komend wpisać polecenie *export RCXTTY=USB* – ustawia komunikację za pomocą portu USB
    3. Skopiować pod Windowsem plik **firmdl3.exe** z katalogu *C:\cygwin\brickos\util* do katalogu *C:\cygwin\brickos\boot*
    4. W Cygwinie przejść do katalogu *C:\cygwin\brickos\boot*
    5. Wpisać komendę *./firmdl3.exe brickOS.srec*

8. Uwagi

1. Domyślnym katalogiem podczas pracy w BrickOS oraz Pascal jest *C:\cygwin\brickos\demo*. Aby móc kompilować swoje pliki powinny one być umieszczone w tym katalogu oraz trzeba dopisać w pliku Makefile nazwę bieżącego pliku z rozszerzeniem .lx

4 Not Quite C

NQC (Not Quite C) jest językiem programowania dla produktów serii LEGO Mindstorms [14]. NQC może być wykorzystany do zaprogramowania takich mikrokomputerów LEGO jak: RCX, RCX2, Scout, Cybermaster, Spybot.
Wszystkie te produkty posiadają w sobie interpreter kodu maszynowego, który jest wykorzystywany przy uruchamianiu programów. Kompilator języka NQC tłumaczy kod źródłowy na kod maszynowy, który następnie jest przesyłany i uruchamiany na mikrokomputerze.
Język NQC jest bardzo zbliżony swoją budową do języka C, jednakże nie jest to język uniwersalny – jest on sztywno dopasowany do programowania robotów LEGO. Jego ograniczenia wynikają z możliwości używanego firmware-u.

4.1 Reguły leksykalne

W tej części zawarte zostały reguły leksykalne, struktury programów, wyrażenia oraz operacje preprocesora.

4.1.1 Komentarze

W języku NQC występują dwa rodzaje komentarzy. Pierwszy typ został „zapożyczony" z języka C. Rozpoczyna się /* a kończy */. Komentarz ten pozwala na zaznaczenie jednej lub wielu linii jednocześnie, niedopuszczalnym jest zagnieżdżanie komentarzy.

Przykład:

```
/* komentarz jednoliniowy*/

/*komentarz
        składający się z
                kilku linii…*/
```

Niewłaściwe użycie komentarzy:

```
/*komentarz…
        /* pamiętajmy o zakończeniu komentarza
        przed rozpoczęciem następnego*/
                ta linia nie jest traktowana jako komentarz*/
```

Drugi typ komentarzy to komentarze jednoliniowe. Zaczynają się od sekwencji znaków //, a kończą się znakiem nowej linii.

Przykład:

```
//to jest komentarz jednoliniowy
```

Komentarze są ignorowane przez kompilator, ich jedynym zadaniem jest ułatwienie tworzenia dokumentacji programu.

4.1.2 Białe znaki

Białymi znakami określa się spacje, wcięcia, znaki nowej linii. Używane są po to, aby kod programu był przejrzysty i czytelny. Dopóki białe znaki nie wpływają na strukturę wyrażenia mogą być one dodawane lub usuwane bezkarnie.

Przykład:

```
x=1;    x=y+1;
x  =   1   ;
          x =y+   1;
```

Niektóre operatory wymagają użycia dwóch symboli. W tym przypadku nie wolno używać białych znaków pomiędzy takimi znakami.

Przykład:

```
x  >> 1;          //wyrażenie poprawne
x >  > 1   ;      //BŁĄD
```

4.1.3 Stałe liczbowe

Stałe liczbowe mogą być przypisywane zarówno w postaci dziesiętnej jak i heksadecymalnej. Stałe w zapisie szesnastkowym wymagają symboli 0x lub 0X przed właściwą wartością.

Przykład:

```
x = 10 ;
x = 0x10;
x = 0X10;
```

4.1.4 Identyfikatory i słowa kluczowe

Identyfikatory są używane do nazw zmiennych, zadań oraz funkcji. Identyfikatory muszą rozpoczynać się małą literą, wielką literą lub podkreśleniem (_). Pozostałe znaki w identyfikatorze mogą być dowolnym ciągiem liter lub cyfr.

Język NQC posiada też zastrzeżone identyfikatory. Nazwano je słowami kluczowymi i nie mogą być one użyte jako nazwy zmiennych, zadań bądź funkcji.

Oto kompletna lista zastrzeżonych słów kluczowych:

__event_src	break	for	start
__nolist	case	goto	stop
__res	catch	if	sub
__sensor	const	inline	switch
__taskid	continue	int	task
__type	default	monitor	true
abs	do	repeat	void
acquire	else	return	while
asm	false	sign	

4.1.5 Operatory przypisania

Przypisanie zmiennej wyrażenia ma postać:

zmienna operator_przypisania wyrażenie;

W języku NQC występuje kilkanaście operatorów przypisania. Poniższa tabela zawiera opis każdego z nich:

Operator	Opis
=	Przypisanie zmiennej wyrażenia
+=	Dodanie wyrażenia do zmiennej
-=	Odjęcie wyrażenia od zmiennej
*=	Pomnożenie zmiennej przez wyrażenie
/=	Podzielenie zmiennej przez wyrażenie
%=	Przypisanie zmiennej reszty z dzielenia jej przez wyrażenie
&=	Wykonuje operację AND na bitach zmiennej i wartości wyrażenia
|=	Wykonuje operację OR na bitach zmiennej i wartości wyrażenia
^=	Wykonuje operację XOR na bitach zmiennej i wartości wyrażenia
||=	Ustawia wartość bezwzględną wyrażenia
+-=	Ustawia znak (-1,+1,0) wyrażenia
>>=	Przesunięcie w prawo o stałą wartość
<<=	Przesunięcie w lewo o stałą wartość

Przykład:

```
a = 3;          // przypisanie a liczby 3
b = 5;          // przypisanie b liczby 3
a += b;         // a równa się 8, b wynosi 5
a *=b;          // a równa się 15, b wynosi 5
```

4.1.6 Wyrażenie złożone

Najprostszą strukturą sterującą jest wyrażenie złożone. Składa się ono z listy wyrażeń ograniczonych klamrami{ }:

Przykład:

```
{
        a=2;
        b=3;
}
```

Klamry mają znaczenie w przypadku innych pętli czy instrukcji warunkowych, ponieważ wiele z nich oczekuje na pojedynczą instrukcję, a zastosowanie wyrażenia złożonego pozwala ominąć to ograniczenie.

4.1.7 Struktura programu

Program napisany w NQC składa się z bloków kodu oraz zmiennych. Bloki kodu dzielą się na trzy niezależne typy: zadania, funkcje, podprogramy. Każdy z bloków posiada swoje możliwości i ograniczenia.

4.1.7.1 Zadania (tasks)

Zadania są definiowane słowem kluczowym *task*.

Przykład:

```
task nazwa()
{
        //tu należy wstawić kod
}
```

Nazwą zadania może być każdy dopuszczalny identyfikator. Każdy program musi zawierać w sobie przynajmniej jedno zadanie o nazwie *main*, które jest uruchamiane każdorazowo przy starcie programu. Maksymalna liczba zadań zależy od urządzenia docelowego:

Model	RCX	CyberMaster	Scout
Ilość zadań	10	4	6

Zadania mogą być rozpoczynane i zatrzymywane za pomocą wyrażeń *start* i *stop*. Istnieje także polecenie *StopAllTasks*, które zatrzymuje wszystkie wykonywane w danej chwili zadania.

4.1.7.2 Funkcje

Bardzo często ciągi wyrażeń grupuje się w pojedynczą funkcję, która może być wywoływana w razie potrzeby.

Przykład:

```
void nazwa(lista_argumentów)
{
        //kod funkcji
}
```

Język NQC dopuszcza funkcje z argumentami, ale nie pozwala na zwracanie wartości. Dlatego też, wszystkie funkcje w NQC rozpoczynają się słowem kluczowym *void*. Lista argumentów funkcji może być pusta albo zawierać jedną lub więcej definicji argumentów. Argument jest definiowany poprzez jego typ oraz nazwę własną. Kolejne argumenty są oddzielone od siebie przecinkami. Wszystkie wartości są 16 bitowymi liczbami całkowitymi.

NQC zezwala na użycie czterech typów argumentów w zależności od schematów i ograniczeń:

Typ	**Znaczenie**	**Ograniczenie**
int	przekazywane przez wartość	brak
const int	przekazywane przez wartość	tylko stałe
int&	przekazywane przez referencję	tylko zmienne
const int &	przekazywane przez referencję	funkcja nie może modyfikować argumentu
int*	przekazywane przez wskaźnik	tylko wskaźnik
const int*	przekazywane przez wskaźnik	funkcja nie może modyfikować argumentu wskaźnika

Argumenty typu *int są* przekazywane przez wartość, co zwykle oznacza, że kompilator musi utworzyć tymczasową zmienną do przechowywania argumentu. Ponieważ funkcja pracuje na kopii argumentu, z jakim została wywołana, to jakakolwiek zmiana tego argumentu w funkcji nie wpływa na ten argument poza nią. W poniższym przykładzie funkcja *zmień* próbuje zmienić wartość swojego argumentu na 3. Taka operacja jest dozwolona, ale ponieważ funkcja operuje tylko na kopii argumentu to argument oryginalny *a* pozostanie niezmieniony.

Przykład:

```
void zmień(int x)
{
        x=3;
}
task main()
{
        int a =1;       // a równa się 1
        zmień(a);       // po uruchomieniu funkcji a nadal równa się 1
}
```

Argumenty typu *const int* także przekazywane są przez wartość, ale ograniczeniem jest to, że mogą to być tylko wartości stałe (liczby).

Przykład:

```
void argument(const int x)
{
        PlaySound(x);
        x=3;                    //błąd nie można modyfikować argumentu funkcji
}
task main()
{
        argument(2);            // dobrze
        argument(3*4);          // dobrze
        argument(x);            // błąd argument funkcji nie jest stałą

}
```

Trzeci typ argumentu *int &* przekazywany jest przez referencję. Powoduje to, że funkcja może zmienić wartość argumentu, z którym jest wywoływana i zmiana ta wpływa na wartość argumentu poza funkcją. Powoduje to jednocześnie, że argument ten może być tylko zmienną.

Przykład:

```
void zmień(int &x){
        x=3;
}
task main()
{
        int a =1;       // a równa się 1
        zmień(a);       // po uruchomieniu funkcji a równa się 3
        zmień(2);       //błąd argument jest stałą a nie zmienną
}
```

Typ *const int &* jest dość niezwykły. Ze względu na to, że jest przekazywany przez referencję jednocześnie nie może być modyfikowany w funkcji. Własności te powodują, że kompilator pozwala przekazać do funkcji wszystko (nie tylko zmienne). W NQC jest to najbardziej skutecznym sposobem na przekazywanie argumentów.

Przykład:

```
void sprawdz(int x){
        if (x==x)       // zawsze jest spełnione
        PlaySound(SOUND_CLICK);
}

void graj(const int &x)
{
        if (x==x)       // może nie być spełnione gdyż wartość może ulec zmianie
        PlaySound(SOUND_CLICK);
}

task main()
{
sprwawdz (SENSOR_1);      // zagra
graj(2);                  // zagra
graj(SENSOR_1);           // może nie zagrać
}
```

Dwa ostatnie typy *int** i *const int** są przekazywane przez wskaźnik.

Przykład:

```
void wskaznik(int * p)
{
*p = 4;
}
task main()
{
int x = 2;
int* y = @x;            // y zawiera adres x
wskaznik(y);            // x = 4
}
```

Kompilator sprawdza czy funkcja została wywołana z odpowiednią liczbą argumentów i czy ich typy są właściwe.

Przykład:

```
void sprawdz(int a, cont int b)
{
        //kod funkcji
}

task main()
{
int x;              //inicjalizacja zmiennej
sprawdz (1,2) // dobrze
sprawdz (x,2) // dobrze
sprawdz (1,x) // źle, drugi argument musi być stałą
sprawdz (2)   // źle, zła liczba argumentów
}
```

W NQC funkcje są zawsze traktowane jako funkcje typu inline**.** Oznacza to, że każde wywołanie funkcji powoduje wstawienie jej kodu do programu i zwiększenie jego wielkości.

4.1.7.3 Podprogramy (Subroutines)

W odróżnieniu od funkcji podprogramy pozwalają na współdzielenie pojedynczej kopii podprogramu i wielokrotne jego wywoływanie bez umieszczania kolejnej jego kopii w kodzie programu. Jednak podprogramy mają pewne ograniczenia z tym związane. Podstawowym jest brak możliwości przekazywania do podprogramu argumentów, a także niemożliwość wywoływania podprogramu przez inny podprogram. Ostatnim ograniczeniem jest maksymalna liczba podprogramów, która w przypadku robota RCX wynosi 8. Dodatkowo w przypadku wersji RCX 1.0, gdy podprogram jest wywoływany przez kilka zadań, nie może posiadać lokalnych zmiennych ani wykonywać obliczeń wymagających użycia zmiennych tymczasowych.

Przykład:

```
sub nazwa()
{
        //kod podprogramu
}
```

4.1.7.4 Zmienne

W NQC występuje tylko jeden typ zmiennych. Są to 16-bitowe liczby całkowite ze znakiem. Zmienne deklaruje się za pomocą słowa *int,* po którym występuje lista nazw zmiennych oddzielonych przecinkiem, a zakończona średnikiem. Podczas deklaracji można od razu przypisać zmiennym wartość za pomocą znaku równości.

Przykład:

```
int a;          // deklaracja pojedynczej zmiennej
int a,b,c;      // deklaracja kilku zmiennych
int a,b,c=1;    // deklaracja kilku zmiennych oraz zmiennej c przypisujemy wartość 1
```

Zmienne globalne są deklarowane na początku programu, poza wszelkimi zadaniami, funkcjami czy podprogramami. Po zadeklarowaniu mogą być używane przez każde zadanie, funkcję czy podprogram w programie.

Natomiast zmienne lokalne deklaruje się wewnątrz zadania, funkcji, podprogramu, w którym ma być wykorzystana. Jej zasięg ogranicza się tylko do bloku, w którym została zadeklarowana. Grupy instrukcji ograniczone klamrami *{}* traktuje się w przypadku zmiennych lokalnych jako blok i zasięg zmiennych zadeklarowanych wewnątrz nich ogranicza się tylko do wnętrza tego bloku.

Przykład:

```
int a;          // zmienna globalna

task main()
{
        int b;          // zmienna lokalna dla zadania main
        a=b;            // operacja prawidłowa
        {               // grupa operacji
                int c;  // zmienna lokalna dla grupy
                a=c;    // dobrze
        }
        a=c;            // źle    zmienna c już nie istnieje
}

task zmienna()
{
a=1;            // dobrze
b=2;            // źle b nie jest zmienną globalną
}
```

W NQC występują też zmienne tymczasowe. Są to zmienne, których się nie deklaruje (kompilator NQC sam je tworzy) i służą one do przechowywania wartości w czasie obliczeń czy wartości argumentów przekazywanych do funkcji. Zmienne także wlicza się do liczby wszystkich zmiennych, jakie może przechowywać dany robot.
Każdy z robotów ma pewną ograniczoną przestrzeń pamięci na przechowywanie zmiennych. Kompilator NQC rozpoznaje dwa rodzaje zmiennych: lokalne i globalne, dla każdego z tych typów są inne ograniczenia, co do maksymalnej ich liczby.

Poniższa tabela przedstawia ile zmiennych danego typu na raz mogą przechowywać roboty RCX:

Wersja RCX	Globalne	Lokalne
RCX 1.0	32	0
RCX 2.0	32	16

4.1.7.5 Tablice

W wersji robota RCX 2 wprowadzono obsługę tablic. Tablice deklarowane są w taki sam sposób jak zmienne tylko, że po nazwie zmiennej w nawiasach *[]* określa się rozmiar tablicy.

Przykład:

```
int tablica[3];          // tablica trzy elementowa
```

W tablicach elementy rozpoznawane są po indeksie określającym jego miejsce w tablicy. Pierwszy element ma indeks 0, drugi 1 itd.

Przykład:

```
tablica[0]= 12;          // przypisanie pierwszemu elementowi liczby 12
tablica[2]=tablica[0]  // kopiowanie elementu pierwszego do trzeciego
```

Tablice mają pewne ograniczenia w obecnej wersji NQC:

- Tablica nie może być argumentem funkcji, ale jej element tak
- Ani wobec tablicy ani jej elementów nie można używać operatorów inkrementacji (++) oraz dekrementacji (--)
- Dla elementów tablicy można używać tylko prostych przypisań (=). Złożone, takie jak (+=) są zabronione
- Zabronione jest przypisywanie wartości w czasie deklarowana tablicy.

4.1.8 Pętle i instrukcje

4.1.8.1 Instrukcja warunkowa IF

Instrukcja *if* sprawdza czy warunek ma wartość logiczną *true,* czyli jest spełniony. Jeśli tak to wykonuje pojedyncze wyrażenie lub całe wyrażenie złożone. Dodatkowo można do instrukcji *if* dodać alternatywę, która będzie wykonywana, jeśli warunek nie został spełniony. Wyrażenie alternatywne występuje po słowie kluczowym *else*.

Przykład:

```
if (warunek) wyrażenie;
if (warunek) wyrażenie else wyrazenie_alternatywne;
if (a==1) { b=2 };
```

4.1.8.2 Pętla WHILE

Pętla *while* pozwala na wykonywanie wyrażenia dopóki warunek pętli jest spełniony.

Przykład:

```
while (warunek) wyrażenie;
```

Najczęściej stosuje się pętle *while* z wyrażeniem złożonym postaci:

Przykład:

```
while (warunek)
        {
        instrukcje
        }
```

Działanie pętli *while* polega na tym, że na początku sprawdzany jest warunek i jeśli jest on spełniony to wykonywane są instrukcje. Po każdym wykonaniu pętli warunek sprawdzany jest ponownie i pętla działa dopóki warunek jest spełniony.

4.1.8.3 Pętla DO..WHILE

Pętla *do..while* jest modyfikacją pętli *while*. Różnicą między pętlą *while* i *do..while* jest to, że w pętli do *while* instrukcja jest wykonywana na początku, a dopiero później jest sprawdzany warunek pętli. Prowadzi to do tego, że w przypadku pętli *do..while* instrukcja zostanie zawsze wykonana choć raz, nawet gdy warunek nie jest spełniony, co w pętli *while* nigdy nie nastąpi.

Przykład:

```
do
{
instrukcje
}
 while (warunek);
```

4.1.8.4 Pętla FOR

Pętla *for* działa według schematu:

- inicjuj zmienną
- sprawdź warunek
- jeśli warunek jest spełniony to wykonaj akcję i instrukcje pętli.

Za każdym razem wykonywania się pętli *for* powtarzane są dwa ostatnie kroki, aż do momentu, gdy warunek nie zwróci wartości logicznej *false*.

Przykład:

```
for (zmienna;warunek;akcja)
{
        instrukcje;
}
```

4.1.8.5 Pętla REPEAT

Pętla *repeat* wykonuje się tyle razy, ile jest zdefiniowane w wyrażeniu:

Przykład:

```
Repeat(wyrażenie)
{
        instrukcje;
}
```

Wyrażenie określa ile razy pętla ma być wykonana. W odróżnieniu od pętli *while* czy *do..while*, wyrażenie to sprawdzane jest tylko jeden raz, na początku wykonywania pętli.

4.1.8.6 Instrukcja SWITCH

Instrukcja *switch* pozwala na wykonanie jednego z kilku zestawów instrukcji w zależności od wartości wyrażenia.

Przykład:

```
switch(wyrażenie){
        case 0 : instrukcja; break;
        case 1 : instrukcja; break;
        case 2 :
        ….
        default : instrukcja; break;
}
```

Każda z instrukcji jest poprzedzona przez słowo *case* i zostanie wykonana w przypadku, jeśli wartość wyrażenia z *switc*h będzie równa wartości etykiety przy *case*. Etykieta ta musi być wartością stałą i unikalną w obrębie pętli *switch*. Instrukcje w *case* będą wykonywane, aż do napotkania słowa kluczowego *break* lub do końca pętli *switch* przy braku słowa *break*. Etykieta *default* będzie wykonana w przypadku, gdy żadna etykieta nie pasuje do wartości wyrażenia, ale nie jest konieczne jej stosowanie.

4.1.8.7 Instrukcja GOTO

Instrukcja goto wymusza skok do innego miejsca w programie.

Przykład:

```
petla:
x++;
goto petla;
```

Instrukcja *goto* nie jest zbyt często używana ze względu na to, że instrukcje *while* czy *if* spełniają podobną rolę, a czynią program łatwiejszym do czytania. Używając instrukcji *goto* należy pamiętać, aby etykieta, do której się kieruje nie znajdowała się wewnątrz wyrażenia *monitor* czy *acquire*, ponieważ instrukcje te przy wejściu i wyjściu z nich wykonują pewne operacje i mogą powodować dziwne zachowanie programu.

4.1.8.8 Pętla UNTIL

Język NQC posiada też pętlę *until*, która swoim działaniem przypomina pętlę *while*.

Przykład:

```
until(wyrażenie)
        {
        instrukcje
        }
```

Oznacza to, że pętla będzie wykonywana dopóki wyrażenie zwróci wartość logiczną *true*. Można ją wykorzystać na przykład do oczekiwania na sygnał z czujnika.

Przykład:

```
until(SENSOR_1 == 1);       // oczekuje aż czujnik zostanie wciśnięty
```

4.1.9 Inne wyrażenia

Aby wywołać funkcję lub podprogram należy podać jego nazwę i w nawiasie listę argumentów oddzielonych przecinkami.

Przykład:

```
nazwa_funkcji(lista_argumentów);
```

Aby uruchomić lub zatrzymać zadanie należy użyć następujących poleceń:

```
start nazwa_zadania;
stop nazwa_zadania;
```

W pętlach takich jak *while, do, until* używa się słowa *break*, które służą do zatrzymania instrukcji. Natomiast *continue* służy do rozpoczęcia następnej iteracji pętli.

Aby zakończyć wykonywanie funkcji przed osiągnięciem jej końca można użyć słowa kluczowego:

```
return;
```

4.1.10 Operatory

Poniższa tabela zawiera operatory uszeregowane od najwyższego do najniższego priorytetu:

Operator	Opis	Powiązania	Ograniczenia	Przykład
abs() sign()	Wartość bezwzględna Znak operandu	- -		abs(x) sign(x)
++ --	Inkrementacja Dekrementacja	Lewe	Tylko dla zmiennej Tylko dla zmiennej	x++ lub ++x x-- lub --x
- ~ !	Minus Negacja bitowa Negacja logiczna	Prawe	Tylko dla stałej	-x ~123 !x
* / %	Mnożenie Dzielenie Modulo	Lewe		x*y y/x 345%2
+ -	Dodawanie Odejmowanie	Lewe		x+y x-y
<< >>	Przesunięcie w lewo Przesunięcie w prawo	Lewe	Tylko dla stałej	
<,>,<=,>=	Operatory zależności	Lewe		
== !=	Równe Różne	Lewe		x==1
&	Bitowy AND	Lewe		
^	Bitowy XOR	Lewe	Tylko dla stałej	
|	Bitowy OR	Lewe		
&&	Logiczny AND	Lewe	Tylko dla stałej	
||	Logiczny OR	Lewe	Tylko dla stałej	

4.1.11 Wyrażenia warunkowe

Wyrażenia warunkowe (warunki) występują w instrukcjach sterujących wykonaniem programu. W większości przypadków warunek jest porównaniem wartości dwóch wyrażeń i zwykle zwraca wartość logiczną *true* lub *false*. Warunek może być zanegowany operatorem negacji lub dwa warunki mogą być połączone operatorami AND lub OR.

Warunek	Znaczenie
true	Zawsze prawdziwe
false	Zawsze fałszywe
wyrażenie	Prawdziwe, jeśli wyrażenie jest różne od 0
wyr1 == wyr2	Prawdziwe, jeśli wyrażenia są równe
wyr1 != wyr2	Prawdziwe, jeśli wyrażenia są różne
wyr1 < wyr2	Prawdziwe, jeśli *wyr1* jest mniejsze od *wyr2*
wyr1 <= wyr2	Prawdziwe, jeśli *wyr1* jest mniejsze lub równe od *wyr2*
wyr1 > wyr2	Prawdziwe, jeśli *wyr1* jest większe od *wyr2*
wyr1 >= wyr2	Prawdziwe, jeśli *wyr1* jest większe lub równe od *wyr2*
! warunek	Negacja logiczna wyrażenia
war1 && war2	Iloczyn logiczny warunku *war1* i *war2* (prawdziwy, gdy oba warunki są prawdziwe)
war1 || war2	Suma logiczna warunku *war1* i *war2* (prawdziwa, gdy conajmniej jeden warunek jest prawdziwy)

4.1.12 Preprocesor

W języku NQC zaimplementowano następujące polecenia*: #include, #define, #ifdef, #ifndef, #if, #elif, #else, #endif, #undef*. Ich zastosowanie jest bardzo podobne jak w języku C i występują tylko niewielkie różnice, które opisane zostaną w poniższym rozdziale.

4.1.12.1 #include

W NQC nazwa pliku musi być umieszczona w cudzysłowach, tak jak pokazuje to przykład:

```
#include „nazwa.nqh”
```

4.1.12.2 #define

Polecenie *#define* służy do definicji instrukcji (makr). Redefinicja makra w NQC jest traktowana jako błąd (error), a nie jako ostrzeżenie (warning) jak jest w C. Makro zwykle kończy się z końcem linii, ale można wymusić przejście do nowej linii za pomocą ukośnika „\”, co pozwala na pisanie makr wieloliniowych.

Przykład:

```
#define makro(x) do { bar(x); \
                      baz(x); } while (false)
```

Polecenie *#undef* służy do skasowania definicji makra.

4.1.12.3 Kompilacja warunkowa

Kompilacja warunkowa działa podobnie jak w C. W NQC można użyć następujących poleceń:

```
#if warunek
#ifdef symbol
#ifndef symbol
#else
#elif symbol
#endif
```

4.1.12.4 Inicjalizacja programu

Na początku każdego programu kompilator wstawia specjalną funkcję *_init*, która ustawia wszystkie trzy wyjścia na pełną moc i kierunek do przodu (są one nadal wyłączone). Funkcję tę można wyłączyć za pomocą polecenia:

#pragma noinit

Można też standardową inicjalizację zastąpić inną funkcją za pomocą polecenia:

#pragma init funkcja

4.1.12.5 Rezerwacja pamięci

NQC automatycznie przydziela zmiennym pamięć. Jednak czasami może wystąpić konieczność zabronienia kompilatorowi wykorzystania pewnego miejsca w pamięci. Wykonuje się to za pomocą polecenia:

#pragma reserve start
#pragma reserve start end

Polecenia te wymuszają na kompilatorze pominięcie określonych miejsc w pamięci. *Start* i *end* muszą być liczbami wskazującymi na poprawne miejsca w pamięci. Jeśli

zdefiniowany jest tylko start, wtedy zostaje zarezerwowana jedna pozycja. Jeśli obie są określone to zarezerwowany zostaje obszar od startu do *end* włącznie. Najczęściej rezerwuje się pozycje 0, 1, 2, które wykorzystują liczniki w RCX2.

Przykład:

```
#pragma reserve 0 1   // rezerwacja pamięci na dwa liczniki
```

4.2 RCX NQC API

NQC API (interfejs programowania aplikacji) definiuje stałe, funkcje, wartości i makra dające dostęp do różnych funkcji robota RCX. Niektóre z nich dostępne są tylko w określonej wersji robota. W przypadku, gdy nie będzie podana wersja robota RCX znaczy to, że dana funkcja czy stała, obsługiwana jest przez wszystkie odmiany RCX.

4.2.1 Czujniki

Robot RCX posiada trzy wejścia, do których można podłączyć czujniki i są one numerowane wewnętrznie 0, 1, 2 natomiast na zewnątrz są one numerowane 1, 2, 3. Dla ułatwienia zostały zdefiniowane nazwy poszczególnych wejść:

```
SENSOR_1, SENSOR_2, SENSOR_3
```

Nazwy te mogą być używane jako argumenty funkcji lub gdy program ma odczytać wartość czujnika.

Przykład:

```
x = SENSOR_1;       // odczytuje wartość czujnika pierwszego i podstawia pod x
```

Typy i tryby czujników

RCX pozwala na używanie szerokiej gamy czujników, także własnej konstrukcji. W programie należy zdefiniować jaki typ czujnika jest podłączony dodanego wejścia za pomocą polecenia *SetSensorType*. Standardowo firma LEGO zdefiniowała cztery typy czujników. Dodatkowo wprowadzono też definicję ogólnego czujnika pasywnego.

Typ czujnika	Znaczenie
SENSOR_TYPE_NONE	Ogólny czujnik pasywny
SENSOR_TYPE_TOUCH	Czujnik dotyku
SENSOR_TYPE_TEMPERATURE	Czujnik temperatury
SENSOR_TYPE_LIGHT	Czujnik światła
SENSOR_TYPE_ROTATION	Czujnik obrotów

Można też określić tryb pracy czujnika. Definiuje to, w jaki sposób ma być przetwarzana wartość pobrana z czujnika. Niektóre tryby pracy zdefiniowane są tylko dla określonych czujników np.: *SENSOR_MODE_ROTATION* używa się tylko dla czujnika obrotów. Tryb czujnika ustawia się poleceniem:

SetSensorMode

Tryb pracy czujnika	Znaczenie
SENSOR_MODE_RAW	Wartości od 0 do 1023
SENSOR_MODE_BOOL	Wartości binarna 0 i 1
SENSOR_MODE_EDGE	Zlicza zmiany stanu
SENSOR_MODE_PULSE	Zlicza okresy
SENSOR_MODE_PERCENT	Wartość od 0 do 100
SENSOR_MODE_FAHRENHEIT	Temperatura w °F
SENSOR_MODE_CELSIUS	Temperatura w °C
SENSOR_MODE_ROTATION	Obrót (16 impulsów na obrót)

W NQC można też za pomocą polecenia *SetSensor* ustawić jednocześnie typ i tryb pracy czujnika. Poniższa tabela zawiera standardowe ustawienia poszczególnych czujników.

Konfiguracja czujnika	Typ	Tryb
SENSOR_TOUCH	SENSOR_TYPE_TOUCH	SENSOR_MODE_BOOL
SENSOR_LIGHT	SENSOR_TYPE_LIGHT	SENSOR_MODE_PERCENT
SENSOR_ROTATION	SENSOR_TYPE_ROTATION	SENSOR_MODE_ROTATION
SENSOR_CELSIUS	SENSOR_TYPE_TEMPERATURE	SENSOR_MODE_CELSIUS
SENSOR_FAHRENHEIT	SENSOR_TYPE_TEMPERATURE	SENSOR_MODE_FAHRENHEIT
SENSOR_PULSE	SENSOR_TYPE_TOUCH	SENSOR_MODE_PULSE
SENSOR_EDGE	SENSOR_TYPE_TOUCH	SENSOR_MODE_EDGE

W RCX można dokonać przekształcenia sygnału na wartości binarne 0,1 nie tylko dla czujnika dotyku. Za wartość „niską” traktuje się sygnał o wartości mniejszej niż 460 i przypisuje się mu wartość binarną 1, a za wartość „wysoką” sygnał większy od 562 i

ma on wartość logiczną 0. Granice zakresów można modyfikować w zakresie 0 do 31. Jeśli wartość czujnika wykroczy poza granicę danego stanu w określonym czasie (3ms) to wtedy wartość binarna ulegnie zmianie.

SetSensor(sensor, konfiguracja) – określa typ i tryb danego czujnika
SetSensor(SENSOR_1, SENSOR_TOUCH);

SetSensorType(sensor, typ) – określa typ czujnika
SetSensorType(SENSOR_1, SENSOR_TYPE_TOUCH);

SetSensorMode(sensor, tryb) – określa tryb pracy czujnika
SetSensorMode(SENSOR_1, SENSOR_MODE_RAW);
SetSensorMode(SENSOR_1, SENSOR_MODE_RAW + 10); // modyfikacja zakresów konwersji binarnej o 10

ClearSensor(sensor) – zeruje wartość czujnika, dotyczy czujnika obrotu i czujnika typu pulse
ClearSensor(SENSOR_1);

SensorValue(n) – zwraca wartość odczytaną z czujnika o numerze n (0,1,2).
x=SensorValue(0); //zwraca wartość czujnika pierwszego

SensorType(n) – zwraca typ czujnika o numerze n (0,1,2).
x=SensorType(0);

SensorMode(n) – zwraca tryb czujnika o numerze n (0,1,2).
x=SensorMode(0);

SensorValueBool(n) – zwraca wartość binarną czujnika o numerze n (0,1,2).
x=SensorValueBool (0);

SensorValueRaw(n) – zwraca wartość liczbową czujnika o numerze n (0,1,2). Dla RCX w zakresie 0-1023.
x=SensorValueRaw (0);

4.2.2 Silniki

Dla silników zdefiniowano trzy nazwy określające poszczególne wyjścia robota: OUT_A, OUT_B, OUT_C. Jednocześnie można sterować kilkoma wyjściami np.: OUT_A+OUT_B oznacza jednoczesne sterowanie wyjściami A i B. Każde z wyjść określają trzy cechy: tryb, kierunek i moc.

Tryb	Znaczenie
OUT_OFF	Wyłącza wyjście (silnik się nie kręci)
OUT_ON	Włącza wyjście (silnik będzie zasilany)
OUT_FLOAT	Silnik na wolnym biegu

Kierunek i moc można ustawić w każdej chwili, ale będą one aktywne tylko gdy silnik jest włączony.

Kierunek	Znaczenie
OUT_FWD	Ustawia kierunek do przodu
OUT_REV	Ustawia kierunek do tyłu
OUT_TOGGLE	Zmienia kierunek na przeciwny do obecnego

Moc silnika można regulować w zakresie od 0 (najmniejszy) do 7 (największy). Dodatkowo zdefiniowane są trzy stałe określające pewne poziomy mocy: *OUT_LOW, OUT_HALF* oraz *OUT_FULL.*

Należy pamiętać, że standardowo podczas startu programu wszystkie trzy wyjścia są ustawione na pełną moc i kierunek do przodu.

SetOutput(wyjście,tryb) – ustawia tryb działania wyjścia
SetOutput(OUT_A+OUT_C,/OUT_ON); // włącza silniki A i C

SetDirection(wyjście,kierunek) – ustawia kierunek silnika
SetDirection(OUT_A,OUT_TOGGLE); // zmienia kierunek silnika A

SetPower(wyjście,moc) – ustawia moc na wyjściu
SetPower(OUT_A,OUT_HALF); //połowa mocy silnika A

OutputStatus(n) – zwraca aktualne ustawienia silnika o numerze n, przy czym n to 0, 1, 2
OutputStatus(1); // zwróci status silnika B

On(wyjścia) – włącza wyjścia
On(OUT_A); // włącza wyjście A

Off(wyjścia) – wyłącza wyjścia
Off(OUT_A+OUT_B); // włącza wyjście A i B

Float(wyjścia) – odłącza wyjścia (wolny bieg)
Float(OUT_A);

Fwd(wyjścia) – ustawia kierunek wyjścia do przodu

Rev(wyjścia) - ustawia kierunek wyjścia do tyłu

Toggle(wyjścia) – zmienia kierunek wyjść

OnFwd(wyjścia) – włącza wyjścia i ustawia ich kierunek do przodu

OnRev(wyjścia) - włącza wyjścia i ustawia ich kierunek do tyłu

OnFor(wyjścia,czas) - włącza wyjścia na podany czas. Czas podawany jest w setnych sekundy (1s =100). Czas może być wyrażeniem.

OnFor(OUT_A,x);

Sterowanie globalne

Instrukcje te są dostępne od wersji **RCX2**.

SetGlobalOutput(wyjścia,tryb) – W przypadku ustawienia trybu *OUT_OFF* wyjścia zostaną wyłączone i wszystkie instrukcje *SetOutput()* czy *On()* będą ignorowanie. Ponowne włączenie wyjść nastąpi poprzez użycie tej instrukcji z trybem *OUT_ON* . Należy pamiętać, że nie spowoduje to uruchomienia silnika, a jedynie pozwoli na późniejsze zastosowanie funkcji *SetOutput()*. Tryb *OUT_FLOAT* przełączy wyjścia na wolny bieg przed ich odłączeniem.

```
SetGlobalOutput(OUT_A, OUT_OFF);      // Wyłącza wyjście A
SetGlobalOutput(OUT_A, OUT_ON);       // Włącza wyjście A
```

SetGlobalDirection(wyjścia,kierunek) – odwraca lub przywraca kierunek wyjść. Normalny kierunek jest w przypadku *OUT_FWD*. Gdy zastosuje się *ON_REV* wtedy zostanie odwrócony aktualny kierunek wyjścia. *OUT_TOGGLE* przełącza między normalnym kierunkiem przeciwnym.

```
SetGlobalDirection(OUT_A, OUT_REV);   // kierunek przeciwny
SetGlobalDirection(OUT_A, OUT_FWD);   // kierunek normalny
```

SetMaxPower(wyjścia,moc) – ustawia maksymalną moc dla wyjść. Moc może być zmienną, ale musi zawierać się między *OUT_LOW* i *OUT_FULL.*

```
SetMaxPower(OUT_A, OUT_HALF);
```

GlobalOutputStatus(n) – zwraca aktualne ustawienia globalne wyjść o numerze n, przy czym n to 0, 1, 2.

```
x = GlobalOutputStatus(2);   // status globalny wyjścia C
```

4.2.3 Głośnik

PlaySound(dźwięk) – odgrywa jeden z sześciu standardowych dźwięków RCX. Dźwięk jest jednym z zdefiniowanych dźwięków:

SOUND_CLICK
SOUND_DOUBLE_BEEP
SOUND_DOWN
SOUND_UP
SOUND_LOW_BEEP
SOUND_FAST_UP.

PlaySound(SOUND_CLICK);

PlayTone(częstotliwość,czas) – odgrywa dźwięk o podanej częstotliwości, przez określony czas. Częstotliwość podawana jest w Hz i w przypadku RCX musi być stałą, a dla RCX2 może być zmienną. Czas musi być stałą i podawany jest w setnych częściach sekundy.

PlayTone(440, 50); // odgrywa dźwięk o częstotliwości 440 Hz przez pół sekundy

MuteSound() – wyłącza odgrywanie wszystkich dźwięków. Tylko **RCX2**

UnmuteSound() – włącza odgrywanie wszystkich dźwięków. Tylko **RCX2**

ClearSound() – czyści bufor dźwięków. Tylko **RCX2**

4.2.4 Wyświetlacz LCD

W wersji RCX występuje siedem trybów pracy wyświetlacza. Standardowo ustawiony jest *DISPLAY_WATCH*

Tryb	Wyświetla
DISPLAY_WATCH	Zegar systemowy
DISPLAY_SENSOR_1	Wartość czujnika 1
DISPLAY_SENSOR_2	Wartość czujnika 2
DISPLAY_SENSOR_3	Wartość czujnika 3
DISPLAY_OUT_A	Wartość wyjścia A
DISPLAY_OUT_B	Wartość wyjścia B
DISPLAY_OUT_C	Wartość wyjścia C

Dodatkowo, w nowej wersji RCX2 wprowadzono tryb *DISPLAY_USER*. W tym trybie odczytuje on ciągle wartość źródła i wyświetla jego stan na wyświetlaczu. W tym trybie można używać kropki na każdej pozycji, co daje złudzenie pracowania z wartościami niecałkowitymi, mimo że operuje się na wartościach całkowitych.

Przykład:

```
SetUserDisplay(1234, 2);      // wyświetli „12.34”
```

Przykład:

```
task main()
{
        ClearTimer(0);
        SetUserDisplay(Timer(0), 0);
        until(false);
}
```

Ze względu na ciągłe odświeżanie wyświetlacza, występują pewne ograniczenia co do wyświetlanej wartości. W przypadku wyświetlania, zmienna musi być dostępna globalnie, co najłatwiej zapewnić deklarując ją jako zmienną globalną. Dodatkowo w przypadku, gdy zmienna jest używana do obliczeń, na wyświetlaczu mogą pojawić się wartości pośrednie obliczeń.

SelectDisplay(tryb) – ustawia tryb wyświetlacza

SetUserDisplay(wartość,precyzja) – ustawia wyświetlacz na ciągłe monitorowanie źródła i ustawienie precyzji. W przypadku ustawienia precyzji na 0 nie pojawi się kropka. (tylko RCX2)

```
SetUserDisplay(Timer(0), 0);          // wyświetla timer 0
```

4.2.5 Komunikacja

4.2.5.1 Wiadomości

Język NQC pozwala na wysyłanie i odbieranie wiadomości przez port IR. Wiadomość może mieć wartość z zakresu 0-255, przy czym wiadomość o wartości 0 jest pomijana. Ostatnio przesłana wiadomość jest zapamiętywana i dostępna jest za pomocą *Message()*. W przypadku braku wiadomości, *Message()* zwróci 0. Ze względu na sposób działania portu IR niemożliwe jest jednoczesne wysyłanie i odbieranie wiadomości.

ClearMessage() – kasuje bufor wiadomości.

Przykład:

```
ClearMessage();          // kasuje bufor
Until(Message()>0);      // czeka na następną wiadomość
```

SendMessage(wiadomość) – wysyła wiadomość, która może być wyrażeniem, ale RCX może wysłać wiadomość tylko z zakresu 0 – 255, więc tylko 8 najniższych bitów jest brane pod uwagę

Przykład:

```
SendMessage(3);      // wysyła 3
SendMessage(259);    // wysyła  także 3
```

SetTxPower(moc) – ustawia moc nadajnika IR. Moc może być jedną ze zdefiniowanych stałych: *TX_POWER_LO, TX_POWER_HI.*

4.2.5.2 Transmisja szeregowa – tylko RCX2

W najnowszej wersji robota RCX2 wprowadzono możliwość transmisji szeregowej przez port IR. Przed rozpoczęciem transmisji należy zdefiniować ustawienia pakietów. Następnie przesyłane dane umieszczane są w buforze i wysyłane za pomocą funkcji *SendSerial()*.

Ustawienia transmisji:

Opcje	Ustawienie
SERIAL_COMM_DEFAULT	Ustawienia standardowe
SERIAL_COMM_4800	4800 bodów
SERIAL_COMM_76KHZ	76 kHz

Standardowe ustawienie to 2400 bodów i 38 kHz.

Ustawienia paczki danych:

Opcje	Ustawienie
SERIAL_PACKET_DEFAULT	Bez formatu paczki, same dane
SERIAL_PACKET_PREAMBLE	Paczka z nagłówkiem
SERIAL_PACKET_CHECKSUM	Dodaje sumę kontrolną do paczki
SERIAL_PACKET_RCX	Standardowy format RCX (nagłówek, suma kontrolna i dopełnienie)

Bufor potrafi przechowywać do 16 bajtów.

Poniższy program przesyła dwa bajty (0x12, 0x34):

Przykład:

```
SetSerialComm(SERIAL_COMM_DEFAULT);
SetSerialPacket(SERIAL_PACKET_DEFAULT);
SetSerialData(0, 0x12);
SetSerialData(1, 0x34);
SendSerial(0, 2);
```

SetSerialComm(ustawienie) – ustawienie sposobu transmisji przez IR. W przypadku, gdy trzeba ustawić tryb na 4800 bodów, stosuje się następujący zapis:

```
SetSerialComm(SERIAL_COMM_4800);
```

SetSerialPacket(ustawienie) – typ transmitowanej paczki danych.

SetSerialPacket(SERIAL_PACKET_DEFAULT);

SetSerialData(n, wartość) – zapisuje bajt do bufora transmisji; n to indeks w buforze (0-15), a wartość może być wyrażeniem

SetSerialData(3, x);

SerialData(n) – zwraca wartość bajtu z bufora transmisji (nie bajtu odebranego); n to numer indeksu: 0-15

SerialData(6); //zwraca bajt numer 6

SendSerial(początek,licznik) – służy do zbudowania paczki danych i wysłania jej przez port IR. Start określa pierwszy bajt, a licznik liczbę bajtów do wysłania.

SendSerial(0,3); // wysyła pierwsze trzy bajty

4.2.6 Timery

RCX zapewnia dostęp do 4 niezależnych timerów o rozdzielczości 100 ms (10 zliczeń na sekundę).Timery potrafią zliczyć od 0 do 32767 zliczeń, co odpowiada około 55 minutom. Timery posiadają identyfikatory o numerze od 0 do 3.

ClearTimer(n) – zeruje wartość timera o numerze n

Timer(n) – zwraca aktualną wartość timera o numerze n (rozdzielczość 100 ms)

x=Timer(1);

SetTimer(n,wartość) – ustawia wartość timera o numerze n

SetTimer(0,x);

FastTimer(n) - zwraca aktualną wartość timera o numerze n (rozdzielczość 10 ms). (tylko RCX2)

4.2.7 Liczniki – tylko RCX2

Liczniki to zmienne, które można inkrementować, dekrementować i zerować. RCX2 zapewnia trzy liczniki (0,1 i 2). Liczniki są przechowywane globalnie w pamięci w lokacjach 0-2 dlatego należy zarezerwować te miejsca za pomocą polecenia *#pragma reserv n*, gdzie n to numer licznika 0-2.

ClearCounter(n) – zeruje licznik o numerze n (0,1,2)

IncCounter(n) – zwiększa licznik n o 1

DecCounter(n) – zmniejsza licznik n o 1

Counter(n) – odczytuje wartość licznika o numerze n

4.2.8 Kontrola dostępu – tylko RCX2

W wersji RCX2 wprowadzono kontrolę dostępu. Umożliwia ona zadaniu przejęcie na „własność" dostępu do jednego lub kilku zasobów. W NQC dokonuje się tego za pomocą komendy:

acquire (zasób) – przejęcie kontroli nad zasobem, gdzie zasób to stała określająca zasób

acquire (zasób) instrukcje
acquire (zasób) instrukcje catch instrukcje

Nazwa zasobu	Zasób
ACQUIRE_OUT_A	Wyjścia
ACQUIRE_OUT_B	
ACQUIRE_OUT_C	
ACQUIRE_SOUND	Głośnik
ACQUIRE_USER_1	Zasoby zdefiniowane przez użytkownika
ACQUIRE_USER_2	
ACQUIRE_USER_3	
ACQUIRE_USER_4	

Za pomocą powyższej instrukcji, zadanie występuje o przejęcie kontroli nad danym zasobem. W przypadku, gdy inne zadanie o wyższym priorytecie już jest właścicielem zadania to żądanie zostanie odrzucone i wykonane zostaną instrukcje występujące po słowie *catch*, które jest opcją nieobowiązkową. W przypadku, gdy zasób jest dostępny zostaną wykonane instrukcje. Jeżeli podczas wykonywania instrukcji inne zadanie o wyższym priorytecie zgłosi żądanie dostępu, wykonywanie instrukcji oraz zadanie straci dostęp do zasobu. W momencie utraty dostępu automatycznie zostaną wykonane instrukcje występujące po słowie *catch* (jeśli oczywiście istnieje). Po prawidłowym wykonaniu instrukcji, zasób zostaje zwrócony do systemu i zadania o mniejszym priorytecie dostają do niego dostęp, lecz przerwane zadanie nie zostanie wznowione.

Przykład:

```
acquire(ACQUIRE_OUT_A)
{
Wait(1000);
}
catch
{
PlaySound(SOUND_UP);
}
```

Powyższy kod przejmuje dostęp do wyjścia A na 10 sekund, a w przypadku, gdy nie będzie to możliwe odegra dźwięk.

SetPriority(p) – ustawia priorytet p zadania. W RCX2 priorytet jest w zakresie 0-255, im mniejsza liczba tym większy priorytet.

4.2.9 Zdarzenia – tylko RCX2

NQC zapewnia 16 w pełni konfigurowalnych zdarzeń, które mogą zostać mapowane do kilkunastu źródeł. Zdarzenia rozpoznawane są za pomocą numerów zakresu 0-15. Źródłem zdarzenia mogą być czujniki, timery, liczniki czy wiadomości bufora. Zdarzenia konfiguruje się za pomocą *SetEvent(zdarzenie,źródła,typ),*gdzie *zdarzenie* to numer z zakresu 0-15, *źródło* określa źródło zdarzenia a typ jest jednym z poniższej tabeli.

Typ	Warunki	Źródło
EVENT_TYPE_PRESSED	Wartość staje się on	Tylko czujniki
EVENT_TYPE_RELEASED	Wartość staje się off	Tylko czujniki
EVENT_TYPE_PULSE	Wartość przechodzi z off do on	Tylko czujniki
EVENT_TYPE_EDGE	Wartość przechodzi z on do off, lub na odwrót	Tylko czujniki
EVENT_TYPE_FASTCHANGE	Wartość zmienia się szybko	Tylko czujniki
EVENT_TYPE_LOW	Wartość staje się low	Każde
EVENT_TYPE_NORMAL	Wartość staje się norma	Każde
EVENT_TYPE_HIGH	Wartość staje się high	Każde
EVENT_TYPE_CLICK	Wartość przechodzi z low przez high do low	Każde
EVENT_TYPE_DOUBLECLICK	Dwa kliknięcia przez określony czas	Każde
EVENT_TYPE_MESSAGE	Nowa wiadomość odebrana	Tylko Message()

Pierwsze cztery odnoszą się do czujników zwracających wartości binarne 0,1 i najczęściej stosuje się w przypadku czujników dotyku. Dla typów *EVENT_TYPE_PULSE* oraz *EVENT_TYPE_EDGE* czujnik musi pracować odpowiednio w trybie *SENSOR_MODE_PULSE* lub *SENSOR_MODE_EDGE*.
Kolejne trzy *EVENT_TYPE_LOW*, *EVENT_TYPE_NORMAL*, *EVENT_TYPE_HIGH* konwertują wartości źródła do trzech zakresów i wywołują się w przypadku przejścia z jednego do drugiego. Poniższy przykład pokazuje konfigurację zdarzenia, które zostanie wywołane, gdy czujnik światła wykryje jaśniejsze światło.

Przykład:

```
SetEvent(3, SENSOR_2, EVENT_TYPE_HIGH);
SetLowerLimit(3, 50);
SetUpperLimit(3, 80);
```

EVENT_TYPE_CLICK wykorzystuje się, gdy wartość źródła przechodzi ze stanu Low do High i powraca do Low w czasie krótszym od kliknięcia ustawionym w zdarzeniu. Do podwójnego takiego przejścia wykorzystuje się zdarzenie typu *EVENT_TYPE_DOUBLECLICK*.

Ostatni typ wykorzystuje się tylko, gdy nadejdzie nowa wiadomość niezależnie od jej treści.

W NQC zaimplementowany jest też mechanizm monitorowania zdarzeń – *monitor,* podobny do mechanizmu kontroli dostępu *acquire.*

monitor (zdarzenia) – monitorowanie zdarzeń

monitor (zdarzenia) instrukcje
monitor (zdarzenia) instrukcje handler_list

Gdzie *handler_list* to jedna lub więcej instrukcji typu:

catch (catch_events) instrukcje

W monitorze numer określający zdarzenie należy przekonwertować do maski zdarzenia za pomocą makra *EVENT_MASK()*. W przypadku monitorowania kilku zdarzeń stosuje się operator bitowy OR.
Monitor będzie wykonywał instrukcje jednocześnie monitorując zadane zdarzenia. W momencie wywołania jednego ze zdarzeń przeskoczy do *handler* (uchwytu)

obsługującego to zdarzenie. Jeżeli taki nie istnieje dla danego zdarzenia, instrukcje nadal będą wykonywane.

Poniższy przykład czeka 10 sekund jednocześnie monitorując zdarzenie 2,3,4. Wywołanie któregoś z nich powoduje wydanie sygnału dźwiękowego.

Przykład:

```
monitor( EVENT_MASK(2) | EVENT_MASK(3) | EVENT_MASK(4) )
{
Wait(1000);
}
catch ( EVENT_MASK(4) )
{
PlaySound(SOUND_DOWN);          // zdarzenie 4 zostało wywołane
}
catch
{
PlaySound(SOUND_UP);     // zdarzenie 2 lub 3 zostało wywołane
}
```

ActiveEvents(zadanie) – zwraca zestaw zdarzeń wywołanych w danym zadaniu

x= ActiveEvents(0);

CurrentEvents() - zwraca zestaw zdarzeń wywołanych w aktywnym zadaniu

x= CurrentEvents(0);

Events(zdarzenie) – ręcznie wywołuje zdarzenie

Event(EVENT_MASK(3));

SetEvent(zdarzenie,źródło,typ) – ustawienie źródła i typu zdarzenia o numerze 0-15

SetEvent(3, SENSOR_2, EVENT_TYPE_HIGH);

ClearEvent(zdarzenie) - kasuje ustawienia danego zdarzenia, co zapobiega jego wywołaniu

ClearAllEvents() – kasuje ustawienia wszystkich zdarzeń

EventState(zdarzenie) – zwraca stan zdarzenia: 0–niski, 1-normalny, 2-wysoki, 3-nieokreślony, 4-rozpoczęcie kalibracji, 5-w czasie kalibracji

CalibrateEvent(zdarzenie,lower,upper,hyst) – kalibruje zdarzenie pobierając aktualne wartości z czujnika i ustawiając poziomy dolny i górny. Kalibracja zależy od typu czujnika i jest opisana w LEGO SDK. Kalibracja nie jest natychmiastowa i zwykle trwa około 50ms.

```
CalibrateEvent2,50,50,20);
until(EventState(2) !=5);       // czeka na koniec kalibracji
```

SetUpperLimit(zdarzenie,limit) – ustawia górny limit dla zdarzenia

```
SetUpperLimit(1,x);
```

UpperLimit(zdarzenie) – zwraca górny limit dla zdarzenia

```
x=UpperLimit(2);
```

SetLowerLimit(zdarzenie,limit) – ustawia dolny limit dla zdarzenia

```
SetUpperLimit(1,x);
```

LowerLimit(zdarzenie) – zwraca dolny limit dla zdarzenia

```
x=LowerLimit(2);
```

SetClickTime(zdarzenie,wartość) – ustawia wartość czasu kliknięcia dla zdarzenia (czas w setnych częściach sekundy)

```
SetClickTime(2,10);
```

ClickTime(zdarzenie) – zwraca czas kliknięcia

```
x=ClickTime(2);
```

SetClickCounter(zdarzenie,wartość) – ustawia licznik kliknięć

```
SetClickCounter(2,x);
```

ClickCounter(zdarzenie) – zwraca licznik kliknięć dla danego zdarzenia

```
x=ClickCounter(2);
```

4.2.10 Rejestr danych (Datalog)

RCX posiada rejestr danych (datalog), w którym można zapisywać wartości odczytane z czujników, timerów, zmiennych czy zegara systemowego. W przypadku, gdy ma być używany datalog, należy go zainicjować poleceniem *CreateDatalog(rozmiar)*, gdzie rozmiar określa ile wartości może być dodanych do datalogu. W przypadku danych

takich jak stałe czy wartości losowe, datalog potrafi je zapisać, ale przed zapisem zostaną one zapisane do zmiennych i jako zmienne zapamiętane w datalogu. RCX nie potrafi bezpośrednio odczytać datalogu. Dane z niego należy przesłać do komputera, gdzie można je następnie obrabiać. Po pobraniu danych do komputera wyświetlona zostanie wartość jak i źródło, z którego została pobrana (timer, czujnik). Pobranie danych z datalogu można wykonać na różne sposoby, np.: w wersji NQC uruchamianej z linii komend, służą do tego polecenia;

nqc -datalog

nqc –datalog_full

CreateDatalog(rozmiar) – tworzy datalog o zadanym rozmiarze, gdzie rozmiar musi być stałą. W przypadku rozmiaru równego 0, czyści obecny datalog bez tworzenia nowego

AddToDatalog(wartość) – dodaje wartość do datalogu. W przypadku zapełnienia datalogu pomija tą instrukcję

Przykład:

```
AddToDatalog(Timer(0));   // dodaje wartość timera 0
AddToDatalog(x);          // dodaje wartość zmiennej x
AddToDatalog(7);          // dodaje 7, ale będzie ona traktowana jako zmienna
```

UploadDatalog(start,liczba) – inicjuje pobieranie datalogu do komputera. Rzadko używana, ponieważ zwykle komputer inicjalizuje pobieranie danych (narzędzie BXCC). Start – początek, od którego mają być pobierane wartości, liczba określa ile ma być pobranych danych.

4.2.11 Inne polecenia

Wait(czas) – usypia zadanie na określony czas (w setnych częściach sekundy). Czas może być stałą lub wyrażeniem.

```
Wait(100);                // czeka 1 sekundę
Wait(Random(100));        // czeka losową wartość czasu od 0 do 1 sekundy
```

StopAllTask() – zatrzymuje wszystkie działające zadania, a więc w konsekwencji cały program

Random(n) – zwraca losową wartość z przedziału 0-n

SetRandomSeed(n) – inicjacja, ustala zarodek dla generatora liczb losowych

SetSleepTime(minuty) – określa czas w minutach do uśpienia robota. W przypadku podania wartości 0 wyłącza funkcję usypiania.

SleepNow() – usypia natychmiast robota

Program() – zwraca numer aktualnie wybranego programu.

x=Program();

SelectProgram(n) - wybiera program o numerze n i wykonuje go. Programy numerowane są 0-4 a nie 1-5 jak na wyświetlaczu. (tylko RCX2)

BatteryLevel() – zwraca stan baterii w miliwoltach. (tylko RCX2)

x= BatteryLevel();

FirmwareVersion() – podaje wersję oprogramowania jako liczbę całkowitą. Np: 3.2.6 jako 326. (tylko RCX2)

x= FirmwareVersion();

Watch() – zwraca wartość zegara systemowego w minutach

x=Watch();

SetWatch(godzina,minuty) – Ustawia zegar systemowy na podany czas. Godzina zawiera się w 0-23, a minuty 0-59

SetWatch(5,55); //ustawia godzinę na 5:55

5 BrickOS

5.1 Reguły leksykalne

5.1.1 Komentarze

W języku BrickOS istnieją dwa rodzaje komentarzy. Pierwszy zaczyna się symbolem //. Znak ten informuje kompilator, że ma pominąć wszystko od tego miejsca aż do znaku końca linii. Drugi rodzaj komentarza rozpoczyna się od /*. Jest to informacja dla kompilatora, że ma pominąć wszystko aż do napotkania znaku */. Należy pamiętać, aby każdy znak /* miał swoje zamknięcie.

Przykład:

```
// to jest komentarz
        to już nie jest komentarz
```

Przykład:

```
/*rozpoczynamy komentarz
        dalej komentujemy
        */
to już nie jest komentarz
```

5.1.2 Białe znaki

Do białych znaków zalicza się spacje, tabulacje, znaki końca linii. Używanie białych znaków jest dozwolone, dopóki nie mają one niepożądanego wpływu na strukturę wyrażenia.

Przykład:

```
x:=1;
        x := 1          ;
y:=   x+   2;
```

W wyrażeniach, w których potrzebne jest użycie dwóch znaków, nie dopuszcza się stosowania białych znaków.

5.1.3 Słowa kluczowe

Słowa kluczowe są zarezerwowane dla kompilatora, nie można nazywać funkcji, zmiennych takimi samymi nazwami.

Lista zastrzeżonych słów kluczowych:

break	for	float	void
case	int	short	while
char	long	signed	else
const	return	struct	
continue	do	switch	
default	double	unsigned	

5.1.4 Zmienne i stałe

5.1.4.1 Zmienne

BrickOS oferuje podstawowe typy zmiennych takie jak:

Typ zmiennej	Zakres
Int	-32768 do 32767
Char	0-255 (256 znaków)
Double	2.2e-308 do 1.8e308
Float	1.2e-38 do 1.8e38

Zmienne tworzy się poprzez napisanie nazwy typu, a po niej listy nazw zmiennych oddzielonych przecinkami.

Przykład:

```
int  wiek, x, y;
char tablica[5];          //wektor 5-cio elementowy
int tablica2d[5][5];      //macierz o wymiarach 5x5
```

Do przypisywania wartości do zmiennych używa się operatora „=”. Wartość może zostać przypisana do zmiennej w trakcie jej tworzenia.

Przykład:

```
wiek=32;
bool status = false;
int tablica[2][2] = {1,2,3,4};
```

Wszystkie nazwy zmiennych przed użyciem muszą być zadeklarowane.

5.1.4.2 Stałe

Stałe, podobnie jak i zmienne przechowują dane, lecz nie zmieniają swojej wartości. W języku BrickOS stałe można zadeklarować przy użyciu słowa kluczowego *const*.

Przykład:

```
const int threshold = 30;
const char znak = 'j';
```

5.1.5 Zmienne ze znakiem i bez znaku

Każdą zmienną typu całkowitego można określić jako posiadającą znak lub nie. Do określania rodzaju zmiennej służą dwa słowa kluczowe: *signed* (liczby ze znakiem) oraz *unsigned* (liczby bez znaku). Zmiennym typu *unsigned* można przypisać tylko wartości nieujemne, natomiast zmiennym typu *signed* wartości dodatnie oraz ujemne. Zaletą stosowania zmiennych ze znakiem lub bez znaku jest „podwojenie" dodatniego zakresu danej zmiennej.

Przykładowo *signed int* reprezentuje: -32767 do 32767, natomiast *unsigned int*: 0 do 65535.

Makrodefinicje

Istnieje możliwość zlecenia preprocesorowi zastępowania pewnych ciągów znaków wskazanym ciągiem symboli. Są to tzw. makrodefinicje.

Przykład:

```
#define czas_jazdy 200      //w całym kodzie, podczas kompilacji, czas_jazdy zostanie
                            //zastąpiony wartością 200
```

5.1.6 Operatory arytmetyczne

Operator	Operacja
+	Dodawanie
-	Odejmowanie
*	Mnożenie
/	Dzielenie (real)
%	Dzielenie modulo

Przykład:

```
Zmienna1  = a / b;
Zmienna2  = 35.3 * -2   //źle, operatory nie mogą występować obok siebie
Zmienna3  = 34.2 * (-2)
Zmienna4  = 3.5 * (2 + 3)
```

Ponieważ najczęściej dodawaną (inkrementacja) i odejmowaną (dekrementacja) liczbą od zmiennej jest jedynka, zaimplementowano dla tych operacji specjalne operatory:

Przykład:

```
i++;      // dodaje jeden do zmiennej i
m--;      // odejmuje jeden od zmiennej m
```

5.1.7 Operatory relacji

Operatorów relacji używa się do określenia czy dwie wartości są równe lub różne od siebie.

Operator	Znaczenie
= =	Równy
>	Większy od
<	Mniejszy od
<=	Mniejszy bądź równy
>=	Większy bądź równy
!=	Różny

5.1.8 Operatory logiczne

Operatory logiczne zazwyczaj są stosowane, gdy istnieje potrzeba sprawdzenia więcej niż jednego warunku za jednym razem. Program jest w stanie sprawdzić czy dane wyrażenia są prawdziwe, ewentualnie czy inny warunek jest spełniony i w zależności od tego wykonać daną operację.

Operator	**Symbol**
AND	&&
OR	||
NOT	!

AND oblicza dwa wyrażenia i jeśli oba są prawdziwe zwraca wartość logiczną *true*.

Przykład:

```
if ((x==4) && (y>6))
```

OR oblicza dwa wyrażenia i jeśli przynajmniej jedno z nich jest prawdziwe zwraca wartość logiczną *true*.

Przykład:

```
if ((x==4) || (y==6))
```

NOT zwraca wartość logiczną *true*, gdy badane wyrażenie reprezentuje wartość logiczną *false*.

Przykład:

```
if (x!=5)            // jest prawdziwe tylko gdy x nie jest równe 5
```

5.1.9 Struktura programu

Struktura programów napisanych w BrickOS jest identyczna z programami napisanymi w C. Oto przykładowa struktura programu napisanego w BrickOS:

```
#include <conio.h>                  //sekcja wstawiania plików nagłówkowych
#include<rom/lcd.h>

#define speed_norm 200              //opcjonalna sekcja definicji
#define speed_low   110
```

```
void jedz(){                    //opcjonalna sekcja funkcji
…
}

void main(){                    //obowiązkowa sekcja mian()
…
}
```

5.1.10 Funkcje w języku BrickOS

Funkcje to podprogramy, które mogą modyfikować dane, zmienne, zwracać wartość. Program napisany w języku BrickOS musi zawierać przynajmniej jedną funkcję *main()*. Gdy program zostanie uruchomiony, funkcja *main()* zostanie automatycznie wywołana. Funkcja ta może uruchomić inne funkcje, które z kolei mogą uruchomić następne itd.

Każda funkcja posiada swój unikalny identyfikator. Wywołanie funkcji polega na wpisaniu jej nazwy w kodzie programu. Gdy program napotka taką nazwę funkcji, przechodzi do wykonywania kodu tej funkcji. Po zakończeniu wykonywania funkcji, program wraca do miejsca gdzie funkcja była wywołana – przechodzi do następnego polecenia.

Zalecane jest rozbijanie dużych skomplikowanych zadań na funkcje, takie postępowanie czyni program czytelnym oraz łatwiejszym do zrozumienia dla innego programisty.

Deklaracja funkcji wskazuje, jaką nazwę posiada dana funkcja, jaki typ wartości zwraca, jakie parametry wejściowe jest w stanie przyjąć.

Zwrócenia wartości przez funkcję można dokonać za pośrednictwem słowa kluczowego *return*.

```
        zwracany_typ nazwa_funkcji ( typ nazwa_parametru, typ nazwa_parametru,…){
                //instrukcje
                return zwracana_wartość;
        }
```

Przykład:

```
int odczytaj_czujnik();        //zwraca int, brak parametrów wejściowych
void jedz_przod();             //zwraca void, brak parametrów wejściowych

int przelicz_odleglosc(int odległość){ //zwraca int, przyjmuje int jako parametr
        int odl_new;
        odl_new=odległość*4;
        return odl_new;
}
```

Przykład: (funkcja typu wakeup_t)

```
wakeup_t touch_wakeup(wakeup_t ignore)
{
        return (TOUCH_1);
}
```

Funkcje typu *wakeup_t* są używane do obsługi zdarzeń. Funkcja ta musi przyjmować argument, ale nie musi go używać (*ignore*).

5.1.11 Pętle i instrukcje

5.1.11.1 Instrukcja warunkowa IF

Instrukcja *if* sprawdza czy jakiś warunek jest spełniony. Jeśli tak, pozwala przejść do specjalnej części kodu, jeśli nie wykonuje kod zawarty po słowie kluczowym *else*.

Przykład:

```
if (wyrażenie) {
        instrukcja1;
        instrukcja2;
        …       }
else {
        instrukcja;
}
```

Instrukcje *if* można zagnieżdżać.

Przykład:

```
if(wyrażenie1){
```

```
        if(wyrażenie2)
                instrukcja1;
        else{
                if(wyrażenie3)
                        instrukcja2;
                else
                        instrukcja3;
        }
}
else
instrukcja4;
```

5.1.11.2 Pętla WHILE

Zezwala na wykonywanie instrukcji tak długo, dopóki warunek jest prawdziwy.

Przykład1:

```
        while ( warunek){
                        instrukcje;
        }
```

Przykład2:

```
        while ( i<5){
                lcd_int(i);
                sleep(1);
                i++;
                }
```

W pętli *while* najpierw sprawdzany jest warunek i jeśli jest on prawdziwy to wykonywane są instrukcje w pętli. Jeśli warunek nie zostanie spełniony, zawartość pętli zostanie pominięta.

5.1.11.3 Pętla DO..WHILE

Pętla tego typu gwarantuje, iż jej zawartość zostanie wykonana, chociaż raz. Warunek jest sprawdzany po wykonaniu zawartości pętli.

Przykład1:

```
do{
        instrukcje;
}while (warunek);
```

Przykład2:

```
do{
        lcd_int(i);
        sleep(1);
        i++;
}while (i<5);
```

5.1.11.4 Pętla FOR

Pętla *for* działa według schematu:

- Inicjuj zmienną
- Sprawdź warunek
- Jeśli warunek jest spełniony to wykonaj akcję i instrukcje pętli.

Za każdym razem wykonywania się pętli *for*, powtarzane są dwa ostatnie kroki aż do momentu, gdy warunek nie zwróci wartości logicznej *false*.

Przykład1:

```
for (zmienna;warunek;akcja){
        instrukcje;
}
```

Przykład2:

```
for (i=0;i<10;i++){
        lcd_int(i);
        sleep(i);          // liczby będą się zmieniać coraz wolniej
}
```

5.1.11.5 Instrukcja SWITCH

Instrukcja ta oblicza wartość podanego wyrażenia, następnie porównuje go z wartościami podanymi przy *case*. Jeśli któraś z nich jest równa wartości wyrażenia to program wykonuje instrukcje podane po słowie kluczowym *case***,** aż do napotkania

słowa kluczowego *break*. Należy pamiętać o każdorazowym zamknięciu sekwencji *case* instrukcją *break*. Jeśli program nie znajdzie żadnej wartości, która równałaby się wyrażeniu, wówczas wykonane zostaną instrukcje po słowie kluczowym *default* (umieszczenie *default* nie jest konieczne).

- Miejsce etykiety *default* jest dowolne, ale może wystąpić tylko jeden raz.
- Może być tylko jedna etykieta z tą sama wartością
- Etykiety wyboru oraz wyrażenie muszą być wartościami całkowitymi

Jeśli po etykiecie *case* nie zostanie umieszczona instrukcja *break*, będą wykonywane instrukcje umieszczone po kolejnej etykiecie *case*.

Przykład1:

```
switch(i){
        case 0 : instrukcja1; break;
        case 1 : instrukcja2; break;
        ....
        default : instrukcja; break;
}
```

Przykład2:

```
cputs(“press”);
switch(getchar()){
        case KEY_PRGM : cputs(“prgm”); break;
        case KEY_VIEW : cputs(“view”); break;
}
```

5.2 RCX BrickOS API

Dostępne funkcje sterujące mikrokomputerem RCX 2.0 są zdefiniowane w plikach nagłówkowych w katalogu …/brickos/include. Aby mieć dostęp do danej funkcji lub instrukcji trzeba uprzednio zadeklarować w programie bibliotekę, w której się ona znajduje [3].

Deklaracji pliku dokonuje się na samym początku programu dyrektywą ***#include*<…>** podając w nawiasach jego nazwę z rozszerzeniem.

Przykład:

```
#include<dmotor.h>
#include<rom/lcd.h>
```

5.2.1 Obsługa wyświetlacza LCD (1) (lcd.h)

lcd_init(); - inicjacja wyświetlacza lcd

lcd_power_on(); - włączenie zasilania wyświetlacza lcd

lcd_power_off(); - wyłączenie zasilania wyświetlacza lcd

lcd_refresh(); - odświeżenie lcd

5.2.2 Obsługa wyświetlacza LCD (2) (romlcd.h)

lcd_int(*liczba*); - wyświetla na wyświetlaczu liczbę typu integer (pozycja 0 nie jest używana)

lcd_unsigned(*liczba*); - wyświetla liczbę bez znaku typu integer

lcd_clock(*t*); - wyświetla liczbę *t* w postaci zegarowej np.

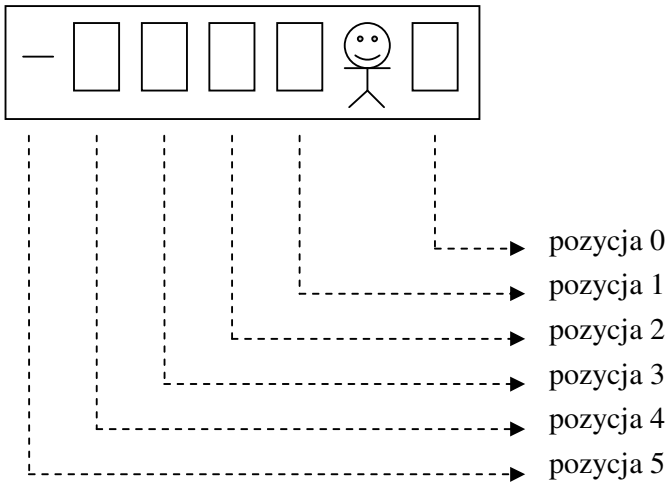
lcd_clock(1015); -> wyświetli 10.15

lcd_digit(*d*); - wyświetla cyfrę *d* po prawej stronie „ludzika”

lcd_clear(); - czyści zawartość wyświetlacza lcd

5.2.3 Obsługa wyświetlacza LCD (3) (conio.h)

delay(*t*); - [t – czas w milisekundach] - czekaj t milisekund



cputc_hex_0(*liczba*); - wypisuje liczbę na 0 pozycji lcd

cputc_hex_1(*liczba*); - wypisuje liczbę na 1 pozycji lcd

cputc_hex_2(*liczba*); - wypisuje liczbę na 2 pozycji lcd

cputc_hex_3(*liczba*); - wypisuje liczbę na 3 pozycji lcd

cputc_hex_4(*liczba*); - wypisuje liczbę na 4 pozycji lcd

cputc_hex_5(*liczba*); - wypisuje liczbę na 5 pozycji lcd

cputc_hex(*c, pos*); - wypisuje cyfrę *c* na danej pozycji *pos* wyświetlacza lcd

cputc(*c, pos*); - wypisuje znak ASCII na pozycji *pos* wyświetlacza lcd

cputs(*string*); - wyświetla napis na lcd (tylko pierwsze pięć liter) np. cputs('czesc');

cls(); - czyści ekran LCD

5.2.4 Obsługa wyświetlacza LCD (4) (dlcd.h)

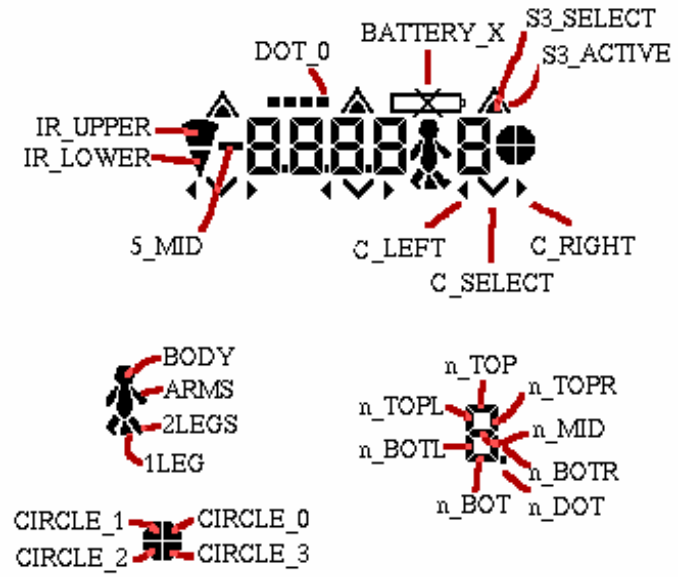
Funkcje zawarte w tej bibliotece pozwalają ukryć bądź pokazać wybrane segmenty wyświetlacza LCD.

Rys. 5.1 Wszystkie segmenty wyświetlacza

dlcd_hide (*mask*); - ukryj segment *mask* wyświetlacza

dlcd_show (*mask*); - pokaż segment *mask* wyświetlacza

Za zmienną mask można podstawić:



Rys. 5.2 Nazwy poszczególnych segmentów wyświetlacza LCD

Wszystkie nazwy elementów należy poprzedzić przedrostkiem *LCD_* , np.:

```
dlcd_show(LCD_CIRCLE_1);
dlcd_show(LCD_S2_SELECT);
dlcd_show(LCD_2_MID);
dlcd_show(LCD_ARMS);
```

5.2.5 Funkcje systemowe (romsystem.h)

power_off(); - uśpienie systemu, UWAGA!!! Może powodować zawieszanie się RCX

reset(); - kasuje firmware BrickOS z pamięci

5.2.6 Obsługa silników (dmotor.h)

Podobnie jak w języku Pascal, w BrickOS jest do dyspozycji 256 różnych prędkości silników. Prędkości silników definiuje liczba zawarta między 0 a 255. W pliku motor.h jest zdefiniowana prędkość maksymalna oraz minimalna:

MIN_SPEED – prędkość 0

MAX_SPEED – prędkość 255

Prędkość jest ustalana niezależnie dla każdego z trzech motorów:

motor_a_speed(*speed*);

motor_b_speed(*speed*);

motor_c_speed(*speed*);

gdzie *speed* przyjmuje wartość od 0 do 255. (można też wpisać np. MAX_SPEED)

Kierunek obrotów oraz tryb pracy silniczków jest definiowany przez polecenia:

motor_a_dir(*dir*);

motor_b_dir(*dir*);

motor_c_dir(*dir*);

gdzie *dir* określa tryb pracy danego silniczka:

- off – silnik wolny
- fwd – jedź do przodu
- rev – jedź do tyłu
- brake – „zaciągnięty hamulec” – UWAGA!!! Włączenie hamulca powoduje bardzo szybkie wyładowanie się baterii.

5.2.7 Obsługa czujników (dsensor.h)

Wejścia dla czujników, w RCX, pracują w dwóch stanach: czynnym oraz biernym. Domyślnie wszystkie wejścia są ustawione na pracę w trybie biernym. W trybie biernym można do wejścia podłączyć np. czujnik dotyku, zaś w trybie czynnym czujnik natężenia światła. Stan pracy wejść jest definiowany niezależnie.

ds_active(&*nr_portu_we*); -> ustawia port na aktywny

ds_passive(&*nr_portu_we*); -> ustawia port na pasywny

Za *nr_portu_we* należy wstawić nazwę portu, którego stan pracy użytkownik chce zmienić:

- SENSOR_1 - port numer 1
- SENSOR_2 - port numer 2
- SENSOR_3 - port numer 3

Czujnik dotyku:

Makro TOUCH zwraca prawdę lub fałsz w zależności czy czujnik dotyku został wciśnięty.

Przykładowe użycie makra:

```
if (TOUCH(SENSOR_1) = = true) {…
if (TOUCH_1){…
```

Można też sprawdzić stan czujnika nie używając do tego makra TOUCH:

```
if (SENSOR_1 < 0xf000) {… → sprawdza czy czujnik dotyku naciśnięty
if (SENSOR_1 > 0xf000) {… → sprawdza czy czujnik dotyku wolny
```

Czujnik natężenia światła:

Czujnik może pracować w dwóch trybach:

- Aktywnym – pomiar światła odbitego (zapalona dioda oświetlająca otoczenie)
- Pasywnym – pomiar światła (zgaszona dioda oświetlająca otoczenie)

Makro LIGHT przelicza uzyskane z pomiarów czujnikiem światła wyniki na liczbę całkowitą typu *integer* z przedziału 0-100 (0-ciemność, 100-max natężenie światła).

Przykładowe użycie makra:

```
poziom=LIGHT(SENSOR_2);
lcd_int(poziom);
```

5.2.8 Obsługa głośniczka systemowego (dsound.h)

dsound_set_duration(*duration*); - czas trwania dźwięku w milisekundach

dsound_stop(); - przerywa, zatrzymuje granie dźwięku

dsound_system(DSOUND_BEEP); - gra dźwięk systemowy (beep)

dsound_set_internote(*duration*); - określa długość trwania „ciszy" pomiędzy granymi dźwiękami

dsound_play(*const note_t *notes*); - gra sekwencje dźwięków, funkcja ta nie jest blokująca, system nie będzie czekał na koniec gry

Parametrem dla *dsound_play* jest macierz z zapisanymi w odpowiedniej kolejności tonami, np.:

```
const note_t gierka[] = {
  { PITCH_G4, QUARTER },
  { PITCH_G4, HALF },
  { PITCH_F4, QUARTER },
  { PITCH_F4, HALF },
  { PITCH_E4, QUARTER },
  { PITCH_PAUSE, HALF },
  { PITCH_END, 0 }          // należy pamiętać aby na końcu
}                           // macierzy zawsze było PITCH_END
```

Definicja:	**Znaczenie nutowe:**
WHOLE	Cała nuta
HALF	Półnuta
QUARTER	Ćwierćnuta
EIGHT	Ósemka

PITCH_PAUSE – pauza

PITCH_END – zaznaczenie końca macierzy t_note

Oto pełna lista dostępnych tonów:

PITCH_A0	PITCH_D2	PITCH_G3	PITCH_C5
PITCH_Am0	PITCH_Dm2	PITCH_Gm3	PITCH_Cm5
PITCH_H0	PITCH_E2	PITCH_A3	PITCH_D5
PITCH_C1	PITCH_F2	PITCH_Am3	PITCH_Dm5
PITCH_Cm1	PITCH_Fm2	PITCH_H3	PITCH_E5
PITCH_D1	PITCH_G2	PITCH_C4	PITCH_F5
PITCH_Dm1	PITCH_Gm2	PITCH_Cm4	PITCH_Fm5
PITCH_E1	PITCH_A2	PITCH_D4	PITCH_G5
PITCH_F1	PITCH_Am2	PITCH_Dm4	PITCH_Gm5
PITCH_Fm1	PITCH_H2	PITCH_E4	PITCH_A5
PITCH_G1	PITCH_C3	PITCH_F4	PITCH_Am5
PITCH_Gm1	PITCH_Cm3	PITCH_Fm4	PITCH_H5
PITCH_A1	PITCH_D3	PITCH_G4	PITCH_C6
PITCH_Am1	PITCH_Dm3	PITCH_Gm4	PITCH_Cm6
PITCH_H1	PITCH_E3	PITCH_A4	PITCH_D6
PITCH_C2	PITCH_F3	PITCH_Am4	PITCH_Dm6
PITCH_Cm2	PITCH_Fm3	PITCH_H4	PITCH_E6
PITCH_D2	PITCH_C7	PITCH_G7	PITCH_D8
PITCH_Dm2	PITCH_Cm7	PITCH_Gm7	PITCH_Dm8
PITCH_E2	PITCH_D7	PITCH_A7	PITCH_E8
PITCH_F2	PITCH_Dm7	PITCH_Am7	PITCH_F8
PITCH_Fm2	PITCH_E7	PITCH_H7	PITCH_Fm8
PITCH_G2	PITCH_F7	PITCH_C8	PITCH_G8
PITCH_Gm2	PITCH_Fm7	PITCH_Cm8	PITCH_Gm8
PITCH_A			

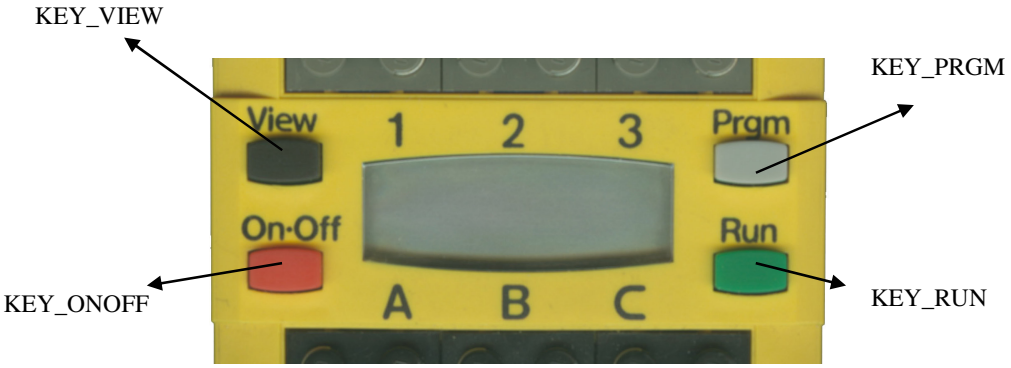
5.2.9 Obsługa klawiatury (1) (dkey.h)

Robot jest wyposażony w prostą klawiaturę, na którą składają się cztery przyciski (rys. 5.3). Biblioteka dkey.h oferuje możliwość sprawdzenia który, z przycisków został naciśnięty w trakcie uruchomionego programu.

Getchar(); - program zatrzymuje się i czeka aż zostanie naciśnięty jakiś guzik, następnie zwraca jego wartość. Funkcja może zwrócić wartości: KEY_ONOFF, KEY_RUN, KEY_VIEW, KEY_PRGM.

Przykład:

```
if (getchar() == KEY_PRGM){ …
```



Rys. 5.3 Kody przycisków (dkey.h)

5.2.10 Obsługa klawiatury (2) (dbutton.h)

Dzięki funkcjom z biblioteki dbutton.h można sprawdzić, który z przycisków w danej chwili jest wciśnięty lub wyciśnięty. W przeciwieństwie do funkcji *getchar;* funkcje te nie zatrzymują programu.

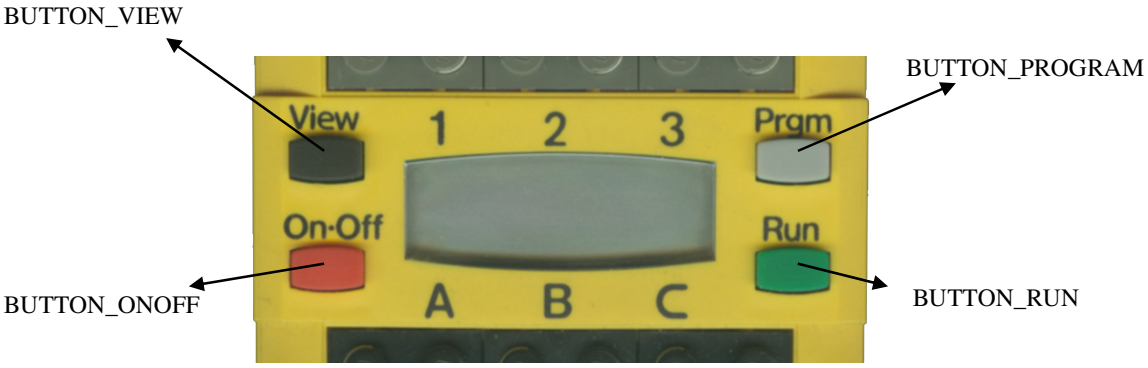
Przykład:

```
if (PRESSED(dbutton(), BUTTON_PROGRAM) ==true){…

if (RELEASED(dbutton(), BUTTON_PROGRAM) == true){…
```

W miejsce BUTTON_PROGRAM można wpisać w zależności od potrzeb:

- BUTTON_ONOFF
- BUTTON_RUN
- BUTTON_VIEW



Rys. 5.4 Kody przycisków (dbutton.h)

Można wymusić zatrzymanie się programu/procesu, aby czekał na zmianę stanu guzika. Do tego typu zdarzenia należy użyć wątków:

```
wakeup_t button_press_wakeup(wakeup_t data) {
   return PRESSED(dbutton(),data); }
```

wywołanie:

```
wait_event(&button_press_wakeup,BUTTON_VIEW);
```

W rezultacie program będzie czekał do momentu, aż guzik VIEW zostanie wciśnięty.

5.2.11 Generator liczb losowych (stdlib.h)

Generator liczb losowych wymaga inicjacji, aby za każdym razem losowana przez niego wartość była inna:

srandom(*liczba*); - zarodek generatora liczb losowych, w miejsce *liczba* najlepiej podać czas systemowy np. *get_system_up_time* (generator będzie uzależniony od czasu systemowego)

Przykładowe użycie funkcji losującej:

```
v := random();
```

5.2.12 Czas systemowy (time.h)

get_system_up_time(); - zwraca liczbę milisekund, które upłynęły od momentu włączenia mikrokomputera.

Get_system_up_time można podzielić przez *TICKS_PER_SEC* wtedy otrzymaną wartością będzie czas w sekundach.

5.2.13 Czujnik baterii (battery.h)

get_battery_mv(); - zwraca bieżące napięcie baterii w mV. Jeśli wynik jest większy od 6700 to poziom naładowania baterii jest dobry. Jeśli wynik jest

mniejszy od 6300 to oznacza to, iż baterie są już słabe i nadają się do wymiany.

battery_refresh(); - odświeża stan czujnika baterii.

5.2.14 Obsługa wątków (unistd.h)

Nowy wątek jest tworzony za pomocą słowa kluczowego *execi.* Funkcja zwraca wartość *taskID* lub -1 w przypadku niepowodzenia.

execi (*funkcja* , argc , argv , priorytet , rozmiar_stosu);

opis:

- Funkcja - wskaźnik do funkcji, która ma być wykonana oraz jej argumenty; jeśli argumenty nie są przekazywane to odpowiednio za *argc* i *argv* trzeba wstawić 0 oraz NIL.
- Priorytet - priorytet nowo utworzonego wątku, zawierający się pomiędzy PRIO_LOWEST i PRIO_HIGHEST czyli między 1 a 20.
- Rozmiar stosu - rozmiar stosu przeznaczonego dla danego wątku, zwykle przyjmuje się DEFAULT_STACK_SIZE = 512B

Metody kończenia pracy wątków:

kill(tid_t taskID); - zakończenie wykonywania wątku o identyfikatorze *taskID*

killall(priority_t priority); - zakończenie wątków o priorytecie mniejszym bądź równym *priority*

Przykład:

```
tid_t tid1;
[…]
int zadanie1 (){
        int i;
        do {
                i=SENSOR_2;
                lcd_int(i);
                msleep(200);
        }until i<2000;
}
[…]
```

```
void main(){
        tid1=execi(zadanie1,0,NULL,2,DEFAUT_STACK_SIZE);
        sleep(5);
        kill(tid1);
}
```

wait_event(&*funkcja, data*);- zawiesza wykonywanie wątku, zadania dopóki funkcja nie zwróci wartości innej od 0.

- &funkcja - wskazuje na funkcję, która zwraca niezerową wartość, jeśli zaszło odpowiednie zdarzenie.
- Data - określa dodatkową wartość, która jest przekazywana do funkcji.

Przykład:

```
wakeup_t sensor_pressed(wakeup_t data) {
   return (TOUCH_1 || TOUCH_2);
 }
```

wywołanie:

```
        wait_event(&sensorpressed,0);
```

sleep(*sec*); - zawiesza wykonywanie zadania na *sec* sekund.

msleep(*msec*); - zawiesza wykonywanie zadania na *msec* milisekund.

5.2.15 Semafory (semaphore.h)

Semafory są przydatnym narzędziem np. do synchronizacji wątków. Semafor blokuje dostęp, gdy jego wartość jest równa 0, natomiast udostępnia zasoby, gdy jego wartość jest większa od zera. Semafory w BrickOS nie są semaforami binarnymi, mogą przyjmować wartości 0,1,2,…

Za każdym razem, gdy proces korzysta z zasobów, wartość semafora powinna być dekrementowana i od razu inkrementowana po zwolnieniu zasobu przez proces. Jeśli dostęp do zasobu ma być wyłączny (np.: tylko jeden proces w danej chwili może z niego korzystać) wartość początkowa semafora powinna być równa 1.

Nazwę nowego semafora należy zadeklarować np.

sem_t MojSemafor;

int sem_init(sem_t *semafor, int pshared, int wartość) – inicjacja nowego semafora

- sem_t *semafor – nazwa semafora
- int pshared – wartość pomijana – należy wstawić 0
- int wartość – wartość semafora

int sem_wait(sem_t *semafor) - zawiesza wątek aż do momentu, gdy wskazany semafor stanie się niezerowy. W tym momencie wartość semafora jest zmniejszana o jeden, natomiast wątek oczekujący na semafor wznawia działanie.

int sem_trywait(sem_t *semafor) - jest odmianą *sem_wait* bez blokowania wątku. Jeżeli semafor jest niezerowy, wówczas funkcja ta zmniejsza jego wartość, a następnie zwraca 0. W przeciwnym przypadku zwraca do wątku EAGAIN.

int sem_post(sem_t *semafor) – natychmiastowe zwolnienie semafora oraz zwiększenie jego wartości o 1.

int sem_getvalue(sem_t *semafor, int *sval) - przypisanie zmiennej wskazywanej przez *sval* wartości semafora.

Int sem_destroy(sem_t *sem) – gdy semafor zostanie wykorzystany, należy go usunąć. Funkcja ta niszczy semafor oraz zwalnia zajmowane przez niego zasoby.

Przykład:

```
/* Demonstracja użycia wątków oraz semaforów.                                    */
/* Wszystkie trzy wątki są uruchamiane na raz.                                   */
/* Semafory służą do określenia kolejności działania/synchronizacji wątków:      */
/* - semafor1 zwalnia wątek go1 oraz go2 – silniki zaczynają pracować w tym samym momencie*/
/* - po upływie dwóch sekund semafor2 zwalnia wątek stopmotor – silniki stop     */
/* Bez mechanizmu semaforów silniki nie rozpoczęłyby pracy równocześnie          */
/* (drugi wystartowałby z minimalnym opóźnieniem)                                */

tid_t pid1, pid2, pid3;  //identyfikatory procesów
sem_t semafor1,semafor2; // deklaracja nazw semaforów

void go1(){
    sem_wait(&semafor1); //czekaj
    motor_c_dir(fwd);
    sem_post(&semafor1); //zwalniamy semafor, zwiększamy jego wartość o jeden
}

void go2(){
    sem_wait(&semafor1); //czekaj
    motor_a_dir(fwd);
    sem_post(&semafor1); //zwalniamy semafor, zwiększamy jego wartość o jeden
}

void stopmotor()
{
    sem_wait(&semafor2); //czekaj
    motor_a_dir(off);
    motor_c_dir(off);
    sem_post(&semafor2);// zwalniamy semafor, zwiększamy jego wartość o jeden
}

int main(){
    pid1=execi(&go1, 0, NULL, 2, DEFAULT_STACK_SIZE);   //uruchamiamy kolejne procesy
    pid2=execi(&go2, 0, NULL, 2, DEFAULT_STACK_SIZE);
    pid3=execi(&stopmotor, 0, NULL, 2, DEFAULT_STACK_SIZE);
    sleep(2);
    sem_init(&semafor1,0,2); //inicjujemy semafor1 - przesylamy znak startu dla procesu 1 i 2
    sleep(2);
    kill(pid1); //zabijamy proces 1
    kill(pid2); //zabijamy proces 2
    sem_init(&semafor2,0,1);        //silniki nadal się kreca, inicjujemy semafor2 - przesylamy sygnał
                                    startu dla //procesu 3
    msleep(1);
    sem_destroy(&semafor1);         //niszczymy semafory - nie beda juz potrzebne
    sem_destroy(&semafor2);
    kill(pid3); //zabijamy proces 3
    return 0;
}
```

6 PASCAL

Język programowania Pascal został zaadaptowany do programowania klocka RCX przez Johna Bindera. Obsługa języka Pascal pojawiła się w środowisku programistycznym Brick Command Center od wersji 3.3.7.7 wzwyż.

6.1 Reguły leksykalne

6.1.1 Komentarze

W języku Pascal dostępne są dwa rodzaje komentarzy. Pierwszy rodzaj komentarza jest taki sam jak w C, NQC. Rozpoczyna się sekwencją znaków „//", a kończy znakiem końca nowej linii. Pozwala on na umieszczenie komentarza tylko w jednej linii.

Przykład:

```
// to jest komentarz
        to już nie jest komentarz
```

Drugi rodzaj pozwala na umieszczenie komentarza w wielu liniach. Rozpoczyna się symbolem { a kończy }.

Przykład:

```
{rozpoczynamy komentarz
        dalej komentujemy
        }
to już nie jest komentarz
```

Komentarze są ignorowane przez kompilator, służą tylko do umieszczania własnych spostrzeżeń, opisów bądź „odznaczania" chwilowo niepotrzebnych bloków programu.

6.1.2 Białe znaki

Są to odstępy, wcięcia itp. Używanie białych znaków jest dozwolone dopóki nie mają one niepożądanego wpływu na strukturę wyrażenia.

Przykład:

```
x:=1;
        x := 1          ;
y:=    x+   2;
```

W wyrażeniach, w których potrzebne jest użycie dwóch znaków, nie dopuszcza się stosowania białych znaków.

Przykład:

```
if zmienna <    > 10 then      //błąd – powinno być <>
```

6.1.3 Instrukcja przypisania

Stałe liczbowe można przypisywać zmiennym poprzez zapis typu:

```
x :=10;  // poprawny zapis
z = 2;    // zapis niepoprawny, brak dwukropka
```

6.1.4 Identyfikatory i słowa kluczowe

Identyfikatory są używane do nazw zmiennych, zadań oraz funkcji. Język Pascal posiada zastrzeżone nazwy identyfikatorów. Nazwano je słowami kluczowymi i nie mogą być one użyte jako nazwy zmiennych bądź funkcji.

Lista zastrzeżonych słów kluczowych:

and	else	mod	record
array	end	not	repeat
begin	for	of	then
const	function	or	to
div	if	procedure	type
do	in	program	until
downto	nil	while	var
writeln	write		

Identyfikatory:

abs	false
boolean	integer
char	real
cos	round
exp	sin
true	

6.1.5 Struktura programu

Poniższy przykład przedstawia strukturę programu napisanego w języku Pascal:

Przykład:

```
program nazwa_programu;

uses
        {tu należy zadeklarować używane moduły}
const
        { tu należy zadeklarować stałe }
var
        { tu należy zadeklarować zmienne }

 {tu można umieścić: procedury, funkcje }

begin
        { tu należy umieścić kod programu głównego}
end.
```

Program napisany w języku Pascal składa się z bloków kodu. W skład bloków kodu wchodzą: instrukcje, procedury, funkcje.

Słowo kluczowe *end*, kończące program musi być zakończone symbolem „.”

6.1.6 Deklaracja stałych

Deklarację stałych umieszcza się w języku Pascal przed blokiem kodu programu. Do wywołania sekcji z deklaracją stałych należy użyć słowa kluczowego *const.*

Przykład:

```
const
    name = 'Robot';              //stała typu string
    letter = 'a';                //stała typu charakter
    rok = 1997;                  //stała typu integer
    pi = 3.1415926535897932;     //stała typu real
    wartosc = TRUE;              //stała typu boolean
```

Wartości stałych w programie nie ulegają zmianie!!!

6.1.7 Deklaracja zmiennych

Deklarację zmiennych umieszcza się w języku Pascal przed blokiem kodu programu. Do wywołania sekcji z deklaracją zmiennych trzeba użyć słowa kluczowego *var.*

Przykład:

```
var
        wiek, wzrost : integer;
        wynik : real;
        znak : char;
        marker : Boolean;
        macierz1d : array[1..10] of integer;        //wektor o wymiarach 1x10
                                                    //składający się z liczb typu
                                                    integer
        macierz2d : array[1..10,1..10] of integer;  //macierz o wymiarach 10x10
                                                    //składająca się z liczb typu
                                                    integer
```

Zakresy zmiennych:

integer	od -32768 do 32767.
real	od $3.4x10^{-38}$ do $3.4x10^{38}$
char	‘A’ … ‘Z’, ‘a’…’z’ oraz znaki np.‘+’
boolean	TRUE lub FALSE

6.1.8 Definicja nazw własnych

W języku Pascal istnieje możliwość definiowania nazw własnych dla długich, rozwiniętych poleceń. Na przykład polecenie *motor_a_dir(mdFwd)* można zastąpić przyjaznym skrótem: *a_motor_przod*. Definicje powinny być umieszczanie na początku programu.

Przykład:

{$define a_motor_przod motor_a_dir(mdFwd)}

Jak wcześniej napisano, w języku Pascal nawiasy{} określają komentarz. W przypadku tego typu definicji nawiasy te nie pełnią roli znaków początku i końca komentarza.

6.1.9 Operatory arytmetyczne

Operator	Operacja	Typ składników	Typ wyniku
+	Dodawanie	real lub integer	real lub integer
-	Odejmowanie	real lub integer	real lub integer
*****	Mnożenie	real lub integer	real lub integer
/	Dzielenie (real)	real lub integer	real
div	Dzielenie (integer)	integer	integer

Przykład:

```
Zmienna1 : = a div b;
Zmienna2 : = 35.3 * -2   //źle, operatory nie mogą występować obok siebie
Zmienna3 : = 34.2 * (-2)
Zmienna4 : = 3.5 * (2 + 3)
```

Uwaga: Każdej zmiennej może być przypisana wartość, która musi być zgodna z zadeklarowanym wcześniej typem zmiennej. Nie można przypisać zmiennej typu *integer* wartości odpowiadającej zmiennej typu *real*.

6.1.10 Operatory relacji

Operatory relacji są stosowane, gdy użytkownik chce porównać dwie lub więcej wartości.

Operator	Znaczenie
<	mniejszy od
>	większy od
=	równy
<=	mniejszy bądź równy
>=	większy bądź równy
<>	różny

Przykład:

```
if wartosc1 <> wartosc2 then
        begin
        …
        end;
```

6.1.11 Operatory logiczne

Operatory logiczne służą do wykonywania operacji logicznych na wartościach typu logicznego, oraz wartościach całkowitych.

Do podstawowych operatorów logicznych języka można zaliczyć:

Operator	Znaczenie
AND	I (koniunkcja)
OR	Lub (alternatywa)
NOT	Nie (negacja)

6.1.12 Procedury

Procedura to wyodrębniona część programu, która posiada swoją nazwę i realizuje określone zadania. Procedury stosuje się do wykonywania czynności wielokrotnie powtarzanych w programie lub takich, które mogą być wykorzystane w innych programach. Stosowanie procedur przyczynia się do bardziej efektywnego wykorzystania pamięci, większej przejrzystości programu i pozwala podzielić duże problemy na mniejsze, rozwiązywane przez różne procedury.

Procedury, można nazwać „podprogramami w programie.”

Część opisowa procedury może zawierać takie same elementy jak część opisowa programu z wyjątkiem deklaracji modułów. Wszystkie zmienne zadeklarowane w części opisowej procedury mają zasięg lokalny - są dostępne tylko w definiowanej procedurze.

6.1.12.1 Procedury bez parametrów

Przykład:

```
procedure nazwa;
  const
    {stałe}
  var
    {zmienne}
  begin
    {kod programu}
  end;
```

Słowo kluczowe *end*, kończące procedurę, musi się kończyć średnikiem a nie kropką. Aby wywołać procedurę w programie wystarczy w bloku kodu wywołać jej nazwę.

6.1.12.2 Procedury z parametrem(ami)

Przykład:

```
procedure nazwa(a : integer);
begin
        motor_a_dir(mdFwd);     // uruchamia motor a
        motor_c_dir(mdFwd);     // uruchamia motor c
        sleep(a);               // niech silniki działają a sekund
        motor_a_dir(mdOff);     // wyłącza motor a
        motor_c_dir(mdOff);     // wyłącza motor c

end;
```

Aby wywołać procedurę z parametrem w programie, należy wpisać jej nazwę wraz z parametrem w bloku kodu.

Przykład:

```
nazwa(20);
```

Wynikiem jej działania będzie uruchomienie silników a i c na czas a podany w sekundach.

6.1.13 Funkcje

Podobnie jak procedury, funkcje są podprogramami posiadającymi swoją nazwę, listę argumentów. Funkcje, w przeciwieństwie do procedur, zawsze zwracają pewną wartość.

Przykład:

```
function dzialanie(a,b : integer) : integer;
begin
        dzialanie :=(a+b)div10;
end;
```

Wywołuje się je w kodzie programu w wyrażeniach np.

```
a:= dzialanie(5,55)+4;
```

Uwaga!!!

Gdy program będzie zawierał w sobie procesy wywoływane poleceniem *execi*, wywoływane funkcje muszą być typu *code_start_t*.

Przykład:

```
[…]
function zadanie1 :code_start_t;
begin
   instrukcje;
end;
[…]
execi(@zadanie1,0,NIL,2,DEFAULT_STACK_SIZE);
```

6.1.14 Pętle i instrukcje

6.1.14.1 Instrukcja warunkowa IF

Instrukcja warunkowa uzależnia wykonywanie innej lub innych instrukcji od spełnienia określonego warunku. Ogólna jej postać jest następująca:

if *wyrażenie* then *instrukcja*

lub

if *wyrażenie* then *instrukcja*
else *instrukcja*

Wyrażenie powinno zwracać wartość logiczną *true* lub *false*. Instrukcja występująca po słowie kluczowym *then* lub *else* może być dowolną instrukcją. Jeśli wartością wyspecyfikowanego wyrażenia jest *true*, to zostanie wykonana instrukcja podana po słowie kluczowym *then*. W przeciwnym przypadku wykonana będzie następna instrukcja po instrukcji *if* (gdy brak *else*) lub instrukcja podana po słowie kluczowym *else*.

Przykład1:

```
if  warunek then
        instrukcja;          //instrukcja zostanie wykonana jeśli
                             //warunek jest prawdą
```

Przykład2:

```
if  warunek then
        instrukcja1
else
        instrukcja2;         //instrukcja2 zostanie wykonana jeśli
                             //warunek nie jest prawdą
```

6.1.14.2 Instrukcja sterująca CASE

Dyrektywy tej należy użyć w przypadku, gdy wykonanie dwóch różnych części programu jest uzależnione od stanu pewnej zmiennej.

Przykład1:

case *wyrazenie* of
    sekwencja instrukcji wyboru
else
    instrukcja
end;

Instrukcja *case* oblicza wartość *wyrażenia*, następnie wyszukiwana jest instrukcja z *sekwencji instrukcji wyboru*. Pierwsza *sekwencja instrukcji wyboru,* poprzedzona stałą obliczoną na podstawie wyrażenia, jest wykonywana. Po wykonaniu instrukcji sterowanie jest przekazywane do instrukcji występującej po słowie *end*. W przypadku, gdy żadna stała wyboru nie odpowiada wartości obliczonej, wykonywana jest instrukcja występująca po *else*.

Przykład2:

```
case b of
    1,7 : instrukcja1;
    2037,5: instrukcja2;
else
    instrukcja3;
end;
```

6.1.14.3 Instrukcja iteracyjna FOR

Dzięki tej pętli można dokładnie określić ile razy zostanie powtórzony podany ciąg instrukcji.

Przykład1:

for *wyrażenie1* to *wyrażenie2*
do
    instrukcja

Instrukcja zostanie wykonana tyle razy ile wynosi różnica między *wyrażenie1* a *wyrażenie2.*

Przykład2:

```
for i:=1 to 50
do
        begin
                lcd_int(i);
                delay(50);
        end;
```

Zasada działania pętli jest taka, że po każdej iteracji zmienna i zwiększana jest o 1. Można również zmniejszyć zmienną *i* o 1 dzięki słowu kluczowemu *downto*.

Przykład3:

```
for i:=1 downto 50
do
        begin
                lcd_int(i);
                delay(50);
        end;
```

6.1.14.4 Instrukcja powtarzania REPEAT..UNTIL

Wykonując pewną liczbę iteracji (powtórzeń) należy skorzystać z instrukcji *repeat.*

Warunek kontynuowania powtórzenia jest sprawdzany na końcu.

Przykład1:

```
repeat
        instrukcja1;
        instrukcja2;
        …
        instrukcja;
until warunek;
```

Warunek zwraca wartość logiczną. Sekwencje instrukcji będą powtarzana dopóki *warunek* nie zostanie spełniony (zwróci wartość logiczną *true*).

Przykład2:

```
repeat
        i:=i+1;
        motor_a_dir(mdFwd);
        motor_c_dir(mdFwd);
        delay(20);
until i=10;
```

6.1.14.5 Instrukcja powtarzania WHILE..DO

Instrukcji *while..do* należy użyć, gdy zachodzi potrzeba wykonania instrukcji (bloków instrukcji) dopóki *warunek* jest spełniony. Cechą charakterystyczną tej instrukcji jest sprawdzenie warunku jeszcze przed wykonywaniem poleceń zawartych w bloku *begin…end*. Jeśli warunek przerywający przy rozpoczęciu tej pętli ma wartość logiczną *false*, to instrukcje zawarte w pętli nie zostaną wykonane ani razu.

Przykład1:

```
while warunek
do
        begin
                instrukcja 1;
                instrukcja 2;
                ...
                instrukcja n;
        end;
```

Przykład2:

```
while i = 10
do
        begin
                motor_a_dir(mdFwd);
                motor_c_dir(mdFwd);
                delay(20);
        end;
```

6.2 RCX Pascal API

Wszystkie funkcje sterujące mikrokomputerem RCX 2.0 są zawarte w plikach nagłówkowych w katalogu …/brickos/include. Zastosowanie danej funkcji wymaga uprzednio zadeklarowania odpowiedniego pliku nagłówkowego, w którym znajduje się dana funkcja.

Deklarację pliku należy umieścić w sekcji *uses*, podając jego nazwę bez rozszerzenia, na początku programu.

Przykład:

```
uses
        conio, dmotor, romlcd;
```

6.2.1 Obsługa wyświetlacza LCD (1) (lcd.h)

lcd_init; - inicjacja wyświetlacza lcd

lcd_power_on; - włączenie zasilania wyświetlacza lcd

lcd_power_off; - wyłączenie zasilania wyświetlacza lcd

lcd_refresh; - odświeżenie lcd

6.2.2 Obsługa wyświetlacza LCD (2) (romlcd.h)

lcd_int(*liczba*); - wyświetla liczbę typu integer (pozycja 0 nie jest używana)

lcd_unsigned(*liczba*); - wyświetla nieoznaczoną liczbę typu integer

lcd_clock(*t*); - wyświetla liczbę *t* w postaci zegarowej np.

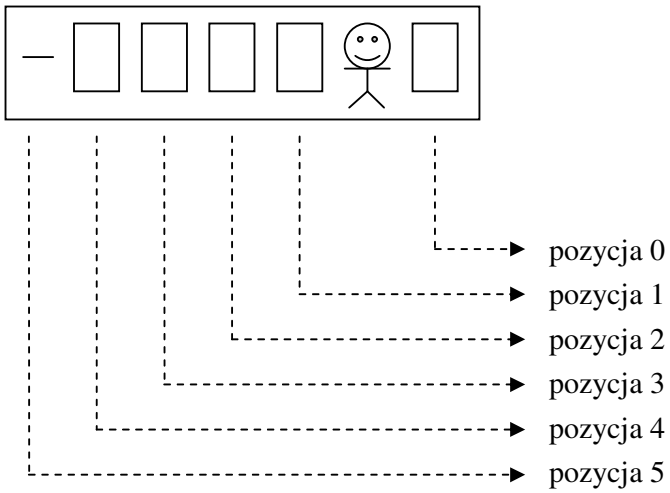
lcd_clock(1015); -> wyświetli 10.15

lcd_digit(*d*); - wyświetla jedną cyfrę po prawej stronie „ludzika”

lcd_clear; - czyści zawartość wyświetlacza LCD

6.2.3 Obsługa wyświetlacza LCD (3) (conio.h)

delay(*t*); - [t – czas w milisekundach] - czekaj t milisekund



cputc_hex_0(*liczba*);	- wypisuje liczbę na 0 pozycji lcd
cputc_hex_1(*liczba*);	- wypisuje liczbę na 1 pozycji lcd
cputc_hex_2(*liczba*);	- wypisuje liczbę na 2 pozycji lcd
cputc_hex_3(*liczba*);	- wypisuje liczbę na 3 pozycji lcd
cputc_hex_4(*liczba*);	- wypisuje liczbę na 4 pozycji lcd
cputc_hex_5(*liczba*);	- wypisuje liczbę na 5 pozycji lcd

cputc_hex(*c, pos*);	- wypisuje cyfrę *c* na danej pozycji *pos* wyświetlacza lcd
cputc(*c, pos*);	- wypisuje znak ASCII na pozycji *pos* wyświetlacza lcd
cputs(*string*);	- wyświetla napis na lcd (tylko pierwsze pięć liter) np. cputs('czesc');
cls;	- czyści ekran lcd

6.2.4 Funkcje systemowe (romsystem.h)

power_off;	- system przechodzi do stanu uśpienia (software-standby) - przycisk on/off reaktywuje RCX
reset;	- kasuje firmware BrickOS z pamięci

6.2.5 Obsługa silników (dmotor.h)

W języku Pascal prędkość motorów określa liczba zawarta między 0 a 255.

MIN_SPEED – prędkość 0

MAX_SPEED – prędkość 255

Prędkość jest definiowana niezależnie dla każdego z trzech motorów:

motor_a_speed(*speed*);

motor_b_speed(*speed*);

motor_c_speed(*speed*);

gdzie *speed* przyjmuje wartość od 0 do 255.

Kierunek oraz zachowanie się silników jest definiowane przez polecenia:

motor_a_dir(*MotorDirection dir*);

motor_b_dir(*MotorDirection dir*);

motor_c_dir(*MotorDirection dir*);

gdzie *MotorDirection dir* określa czynność jaką ma wykonać silnik:

- mdOff – silnik wolny
- mdFwd – jedź do przodu
- mdRev – jedź do tyłu
- mdBrake – „zaciągnięty” hamulec – UWAGA!!! Włączenie hamulca powoduje bardzo szybkie wyładowanie się baterii.

6.2.6 Obsługa czujników (dsensor.h)

Wejścia dla czujników w RCX pracują w dwóch stanach: czynnym oraz biernym. Domyślnie wszystkie wejścia są ustawione na pracę w trybie biernym. W trybie

biernym można do wejścia podłączyć np. czujnik dotyku, zaś w trybie czynnym czujnik natężenia światła. Stan pracy wejść jest definiowany niezależnie.

ds_active(@*nr_portu_we*); -> zmiana trybu pracy portu na aktywny
ds_passive(@*nr_portu_we*); -> zmiana trybu pracy portu na pasywny

Za *nr_portu_we* należy wstawić nazwę portu, którego stan pracy ma być zmieniony:

- SENSOR_1 - port numer 1
- SENSOR_2 - port numer 2
- SENSOR_3 - port numer 3

Czujnik dotyku:

Makro TOUCH zwraca prawdę lub fałsz w zależności czy czujnik dotyku został wciśnięty.

Przykładowe użycie makra:

```
if TOUCH(SENSOR_1) = true then…
```

Można też sprawdzić stan czujnika nie używając do tego makra TOUCH:

```
if SENSOR_1 < $F000 then… → sprawdza czy czujnik dotyku naciśnięty
if SENSOR_1 > $F000 then… → sprawdza czy czujnik dotyku wolny
```

Czujnik natężenia światła:

Czujnik może pracować w dwóch trybach:

- Aktywnym – pomiar światła odbitego (zapalona dioda oświetlająca otoczenie)
- Pasywnym – pomiar światła (zgaszona dioda oświetlająca otoczenie)

Makro LIGHT przelicza uzyskane z pomiarów czujnikiem światła wyniki na liczbę całkowitą typu integer z przedziału 0-100. (0-ciemność, 100-max natężenie światła)

Przykładowe użycie makra:

```
poziom:=LIGHT(SENSOR_2);
lcd_int(poziom);
```

6.2.7 Obsługa głośniczka systemowego (dsound.h)

dsound_set_duration(*unsigned_duration*); - czas trwania dźwięku w milisekundach

dsound_stop; - przerywa, zatrzymuje granie dźwięku

dsound_system(DSOUND_BEEP); - gra dźwięk systemowy (beep)

dsound_set_internote(*unsigned_duration*); - określa długość trwania „ciszy" pomiędzy granymi dźwiękami

dsound_play(*const note_t *notes*); - gra sekwencje dźwięków, funkcja ta nie jest blokująca (podczas gry, RCX będzie wykonywał następne zadania)

Parametrem dla dsound_play jest macierz z zapisanymi w odpowiedniej kolejności tonami, np.:

```
const
        macierz : array[0..3] of t_note =(
                (PITCH_G4,QUARTER),
                (PITCH_F4,HALF),
                (PITCH_END,0)   // należy pamiętać aby na końcu
                );              // macierzy zawsze było PITCH_END
```

Definicja:	**Znaczenie nutowe:**
WHOLE	Cała nuta
HALF	Półnuta
QUARTER	Ćwierćnuta
EIGHT	Ósemka

PITCH_PAUSE - pauza

PITCH_END - zaznaczenie końca macierzy t_note

Oto pełna lista dostępnych tonów:

PITCH_A0	PITCH_D2	PITCH_G3	PITCH_C5
PITCH_Am0	PITCH_Dm2	PITCH_Gm3	PITCH_Cm5
PITCH_H0	PITCH_E2	PITCH_A3	PITCH_D5
PITCH_C1	PITCH_F2	PITCH_Am3	PITCH_Dm5
PITCH_Cm1	PITCH_Fm2	PITCH_H3	PITCH_E5
PITCH_D1	PITCH_G2	PITCH_C4	PITCH_F5
PITCH_Dm1	PITCH_Gm2	PITCH_Cm4	PITCH_Fm5
PITCH_E1	PITCH_A2	PITCH_D4	PITCH_G5
PITCH_F1	PITCH_Am2	PITCH_Dm4	PITCH_Gm5
PITCH_Fm1	PITCH_H2	PITCH_E4	PITCH_A5
PITCH_G1	PITCH_C3	PITCH_F4	PITCH_Am5
PITCH_Gm1	PITCH_Cm3	PITCH_Fm4	PITCH_H5
PITCH_A1	PITCH_D3	PITCH_G4	PITCH_C6
PITCH_Am1	PITCH_Dm3	PITCH_Gm4	PITCH_Cm6
PITCH_H1	PITCH_E3	PITCH_A4	PITCH_D6
PITCH_C2	PITCH_F3	PITCH_Am4	PITCH_Dm6
PITCH_Cm2	PITCH_Fm3	PITCH_H4	PITCH_E6
PITCH_D2	PITCH_C7	PITCH_G7	PITCH_D8
PITCH_Dm2	PITCH_Cm7	PITCH_Gm7	PITCH_Dm8
PITCH_E2	PITCH_D7	PITCH_A7	PITCH_E8
PITCH_F2	PITCH_Dm7	PITCH_Am7	PITCH_F8
PITCH_Fm2	PITCH_E7	PITCH_H7	PITCH_Fm8
PITCH_G2	PITCH_F7	PITCH_C8	PITCH_G8
PITCH_Gm2	PITCH_Fm7	PITCH_Cm8	PITCH_Gm8
PITCH_A			

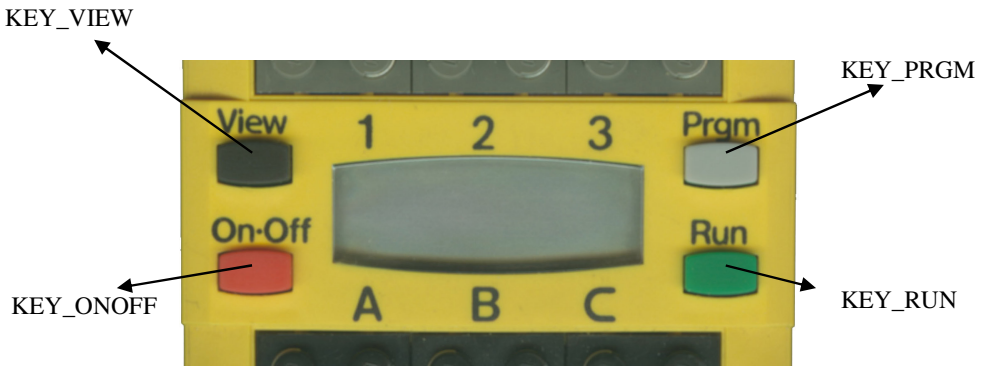
6.2.8 Obsługa klawiatury (1) (dkey.h)

Robot jest wyposażony w prostą klawiaturę, na którą składają się cztery przyciski (rys. 6.1). Biblioteka dkey.h oferuje możliwość sprawdzenia, który z przycisków został naciśnięty w trakcie uruchomionego programu.

getchar; - program zatrzymuje się i czeka aż zostanie naciśnięty jakiś guzik, następnie zwraca jego wartość. Funkcja może zwrócić wartości: KEY_ONOFF, KEY_RUN, KEY_VIEW, KEY_PRGM.

Przykład:

if getchar = KEY_PRGM then …



Rys. 6.1 Kody przycisków (dkey.h)

6.2.9 Obsługa klawiatury (2) (dbutton.h)

Dzięki funkcjom z biblioteki dbutton.h można sprawdzić, który z przycisków w danej chwili jest wciśnięty lub wyciśnięty. W przeciwieństwie do funkcji *getchar;* funkcje te nie zatrzymują programu.

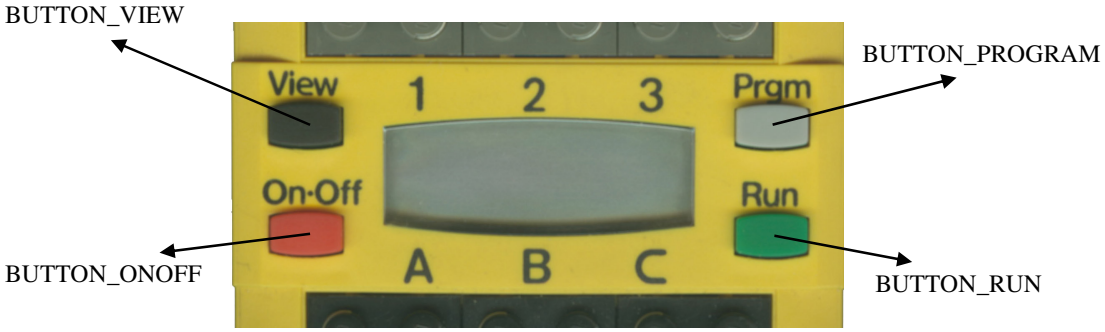
Przykład:

```
if PRESSED(dbutton, BUTTON_PROGRAM) = true then…

if RELEASED(dbutton, BUTTON_PROGRAM) = true then…
```

W miejsce BUTTON_PROGRAM można wpisać w zależności od potrzeb:

- BUTTON_ONOFF
- BUTTON_RUN
- BUTTON_VIEW



Rys. 6.2 Kody przycisków (dbutton.h)

Można wymusić zatrzymanie się programu, aby czekał na zmianę stanu guzika. Dokonać tego można za pomocą wątków:

```
function button_press_wakeup(data : wakeup_t) : wakeup_t;
begin
        if PRESSED(dbutton,data)=true then
                result:=1;
        else
                result:=0;
end;
```

wywołanie:

```
wait_event(@button_press_wakeup, BUTTON_PROGRAM);
```

W rezultacie program będzie czekał do momentu aż guzik PRGM zostanie wciśnięty.

6.2.10 Generator liczb losowych (stdlib.h)

Generator liczb losowych wymaga inicjacji, aby za każdym razem losowana przez niego wartość była inna:

srandom(*liczba*); - zarodek generatora liczb losowych, w miejsce *liczba* najlepiej podać czas systemowy np. *get_system_up_time* (generator będzie uzależniony od czasu systemowego)

Przykładowe użycie funkcji losującej:

```
v := random;
```

6.2.11 Czas systemowy (time.h)

get_system_up_time; - zwraca liczbę milisekund, które upłynęły od momentu włączenia mikrokomputera.

Get_system_up_time można podzielić przez *TICKS_PER_SEC* – wynikiem tego działania będzie czas w sekundach.

6.2.12 Czujnik baterii (battery.h)

get_battery_mv; - zwraca bieżące napięcie baterii w mV. Jeśli wynik jest większy od 6700 to stan baterii jest OK. Jeśli wynik jest mniejszy od 6300 to oznacza to że baterie są już słabe i nadają się do wymiany.

battery_refresh; - odświeża stan czujnika baterii.

6.2.13 Obsługa wątków (unistd.h)

Nowy wątek jest tworzony za pomocą słowa kluczowego *execi.* Funkcja zwraca wartość *taskID* lub -1 w przypadku niepowodzenia.

execi (@*funkcja* , argc , argv , priorytet , rozmiar_stosu);

opis:

- @funkcja - wskaźnik do funkcji, która ma być wykonana oraz jej argumenty; jeśli argumenty nie są przekazywane to odpowiednio za argc i argv trzeba wstawić 0 oraz NIL.
- Priorytet - priorytet nowoutworzonego wątku, zawierający się pomiędzy PRIO_LOWEST i PRIO_HIGHEST czyli między 1 a 20.
- Rozmiar stosu - rozmiar stosu przeznaczonego dla danego wątku, zwykle przyjmuje się DEFAULT_STACK_SIZE = 512B

Metody zakańczania wątków:

kill(tid_t taskID); - zakończenie wykonywania wątku o identyfikatorze taskID

killall(priority_t priority); - zakończenie wątków o priorytecie mniejszym bądź równym *priority*

Przykład:

```
var
        tid1 : tid_t;
[…]
function zadanie : code_start_t;
begin
        motor_a_dir(mdFwd);
```

```
        sleep(2);
        motor_a_dir(mdOff);
end;
[…]
begin
        pid:=execi(@zadanie,0,NIL,2,DEFAUT_STACK_SIZE);
        sleep(5);
        kill(tid1);
end.
```

wait_event(@*funkcja, data*);- zawiesza wykonywanie wątku, zadania dopóki funkcja nie zwróci wartości innej od 0.

- @funkcja - wskazuje na funkcję, która zwraca niezerową wartość, jeśli zaszło odpowiednie zdarzenie.
- Data - określa dodatkową wartość, która jest przekazywana do funkcji.

Przykład:

```
function button_press_wakeup(data : wakeup_t) : wakeup_t;
begin
        if PRESSED(dbutton,data)=true then
                result:=1;
        else
                result:=0;
end;
```

wywołanie:

```
        wait_event(@button_press_wakeup, BUTTON_PROGRAM);
```

sleep(*sec*); - zawieś wykonywanie zadania na *sec* sekund.

msleep(*msec*); - zawieś wykonywanie zadania na *msec* milisekund.

7 MINDSCRIPT

7.1 Reguły leksykalne

7.1.1 Komentarze

W Mindscript występują dwa rodzaje komentarzy: jednoliniowe oraz komentarz wieloliniowy. Pierwszy typ zaczyna się od sekwencji znaków //,a kończą się znakiem nowej linii.

Przykład:

```
//to jest komentarz jednoliniowy
```

Drugi rozpoczyna się /*, a kończy */. Komentarz ten pozwala na zaznaczenie jednej lub wielu linii na raz.

Przykład:

```
/* komentarz jednoliniowy*/

/*komentarz
        składający się z
                kilku linii…*/
```

7.1.2 Białe znaki

Białymi znakami określa się spacje, wcięcia, znaki nowej linii. Używane są po to, aby kod programu był przejrzysty i czytelny. Dopóki białe znaki nie wpływają na strukturę wyrażenia mogą być one dodawane lub usuwane bezkarnie.

7.1.3 Stałe liczbowe

Mindscript pozwala na używanie stałych liczbowych. Stałe mogą być deklarowane jako globalne, w takim przypadku można używać ich w całym programie lub jako lokale, dostęp do nich możliwy jest tylko w danym zadaniu. Stałe mogą być tylko liczbami całkowitymi.

Przykład:

```
const nazwa=wartość          // deklarowanie stałej
```

Stale globalne deklaruje się na początku programu przed wszystkimi zadaniami.

7.1.4 Zmienne

W Mindscript rozróżnia się dwa typy zmiennych: globalne i lokalne. Są one liczbami całkowitymi.

Zmienne globalne muszą być zadeklarowane na początku programu przed zadaniem *main*. Zmienne globalne mogą być używane przez wszystkie zadania.

Przykład:

```
var nazwa                    // deklaracja zmiennej globalnej
var nazwa=wartość            //deklaracje i przypisanie jej wartości
```

Zmienne lokalne deklarowane są na początku zadań, w których są używane i dostęp do nich jest możliwy tylko wewnątrz danego zadania. W odróżnieniu od innych języków np. NQC, deklaracja zmiennej lokalnej jest różna od globalnej.

Przykład:

```
local nazwa                  // deklaracja zmiennej lokalnej
local nazwa=wartość          //deklaracje i przypisanie jej wartości
```

Aby wyzerować wartość zmiennej należy użyć komendy:

```
clear nazwa
```

7.1.5 Operatory

Mindscript obsługuje wszystkie podstawowe operatory arytmetyczne. Brakuje jednak dość przydatnych operatorów inkrementacji i dekrementacji.

Operator arytmetyczny	Opis
+	Dodawanie
-	Odejmowanie
*	Mnożenie
/	Dzielenie

Operator przypisania	Opis
=	Przypisanie zmienne wyrażenia
+=	Dodanie wyrażenia do zmiennej
-=	Odjęcie wyrażenia od zmiennej
*=	Pomnożenie zmiennej przez wyrażenie
/=	Podzielenie zmiennej przez wyrażenie

Powyższe operatory są wydajne, ponieważ nie potrzebują zmiennych tymczasowych.

Operator relacji	Opis
=	Równy
>	Większy
<	Mniejszy
>=	Większy lub równy
<=	Mniejszy lub równy
<>	Różny

7.1.6 Struktura programu

Poniższy przykład pokazuje strukturę programu napisanego w języku Mindscript.

Przykład:

```
program nazwa {
        // dołączanie plików nagłówkowych
        // deklaracje zmiennych globalnych, czujników, zdarzeń
main {
instrukcje
}
watcher {
}
task{   // zadania
}
}
```

7.1.6.1 Zadania (task)

Zadania są definiowane słowem kluczowym *task*.

Przykład:

```
task nazwa{
    //tu należy wstawić kod
}
```

Każdy program musi zawierać w sobie przynajmniej jedno zadanie o nazwie *main*, które jest uruchamiane każdorazowo przy starcie programu za pomocą przycisku *Run* lub za pomocą komendy *start main*. Pozostałe zadania uruchamiane i zatrzymywane są za pomocą komendy *start i stop*. Zadania pozwalają na deklarowanie w sobie zmiennych lokalnych.

7.1.6.2 Podprogramy (Subroutines) Tylko RCX2

Dla wersji RCX2 wprowadzono obsługę podprogramów. Dla RCX2 maksymalna liczba podprogramów wynosi 8. Są one numerowane od 0 do 7.

Przykład:

```
sub nazwa (parametry){
    //tu należy wstawić kod
}

nazwa                 // wywołanie podprogramu bez parametrów
nazwa(parametry)      // wywołanie podprogramu z parametrami
```

7.1.6.3 Makra

Makra są definiowane i wywoływane za pomocą nazwy.

Przykład:

```
macro nazwa (parametry) { instrukcje }
nazwa                 // wywołanie makra bez parametrów
nazwa(parametry)      // wywołanie makra z parametrami
```

Makra mogą być uruchamiane zarówno z parametrami jak i bez. Makra pozwalają na wywoływanie makra przez inne makro, a także możliwe jest równoczesne wywołanie makra przez kilka zadań. Makra są wstawiane do kodu programu w miejscu ich wywołania.

7.1.7 Pętle i instrukcje

7.1.7.1 Instrukcja warunkowa IF

Instrukcja *if* sprawdza czy warunek ma wartość logiczną *true,* czyli jest spełniony. Jeśli tak to wykonuje instrukcje. Dodatkowo można do instrukcji *if* dodać alternatywę, która będzie wykonywana, jeśli warunek nie został spełniony. Wyrażenie alternatywne występuje po słowie kluczowym *else*.

Przykład:

```
if warunek {instrukcja}
if warunek {instrukcja} else {wyrazenie_alternatywne}
```

W języku Mindscript, oprócz standardowych operatorów relacji można zastosować dwa operatory pozwalające na sprawdzenie czy wartość zawiera się nie w podanym przedziale.

Przykład:

```
if wartość is przedział {instrukcja}// sprawdza czy wartość znajduje się w zakresie
if wartość is not przedział {instrukcja}// sprawdza czy wartość nie znajduje się w zakresie
```

Przedział definiuje się podając wartość początkową i końcową przedzielone dwiema kropkami.

Przykład:

```
if x is 40..50 { display x}// jeśli wartość x jest w przedziale (40.50) to wyświetli
                           // x na LCD
```

7.1.7.2 Pętla WHILE

Pętla *while* pozwala na wykonywanie instrukcji dopóki warunek pętli jest spełniony.

Przykład:

```
while warunek {instrukcja}
```

Podobnie jak w przypadku instrukcji *if,* w warunku można testować czy wartość zawiera się lub nie w podanym przedziale.

Przykład:

```
while wartość is przedział {instrukcja}// sprawdza czy wartość znajduje się w zakresie
while wartość is not przedział {instrukcja}// sprawdza czy wartość nie znajduje się w zakresie
```

7.1.7.3 Pętla REPEAT

Pętla *repeat* pozwalana na wykonanie pętli określoną liczbę razy.

Przykład:

```
repeat liczba {instrukcja}
```

Minscript pozwala też na wywołanie pętli *repeat* losową liczbę razy.

Przykład:

```
repeat random 5 {instrukcja}          // wywoła losową ilość razy z przedziału 0-5
repeat random  2 to 8 {instrukcja}   // wywoła losową ilość razy z przedziału 2-8
```

Można też wywoływać pętle do czasu napotkania zdarzenia

Przykład:

```
repeat {instrukcja} until nazwa_zdarzenia
```

7.1.7.4 Pętla FOREVER

Pętla ta będzie wykonywana do czasu zakończenia zadania bądź jego restartu.

Przykład:

```
forever {instrukcja}
```

7.1.7.5 Instrukcja SELECT

Instrukcja *select* pozwala na wykonanie jednej z kilku instrukcji w zależności od wartości wyrażenia. Jest bardzo podobna do instrukcji *switch* występującej w językach takich jak C, czy NQC.

Przykład:

```
select  wyrażenie {
        when wartość1 {instrukcja}
        when wartość2 {instrukcja}
        when wartość3 {instrukcja}
        else {instrukcja}
```

Każda z instrukcji poprzedzona jest słowem *when* i zostanie wykonana w przypadku, gdy wartość wyrażenia z select będzie równa z wartością przy *when.* Instrukcja po *else* zostanie wykonana, w przypadku, gdy wartość wyrażenie nie będzie pasowała do żadnego z powyższych wartości. Jej stosowanie jest opcjonalne.

7.1.7.6 Instrukcja TRY

Instrukcję *try* wykorzystuje się, gdy dwa lub więcej zadań lub podprogramów próbują uzyskać dostęp do tego samego zadania. Jednoczesny dostęp mógłby spowodować nieprzewidywalne zachowanie robota lub nawet jego zawieszenie. Instrukcję stosuje się wewnątrz zadania lub podprogramu. W momencie jej wykonania program sprawdza czy dany zasób nie jest używany przez inne zadanie. W przypadku, gdy jest wolny, wykonywane są instrukcje, jeśli instrukcje nie zostaną wykonane to zostanie podjęta akcja zdefiniowana przed wyrażeniem *on fail*.

Przykład:

```
sub podprogram{
        try {instrukcje} akcja on fail
        }
```

Gdy nie zostanie wykonana instrukcja *try* mogą zostać podjęte następujące akcje:

retry – próbuje wykonać instrukcję *try* raz jeszcze

abort - wykonuje dalszy program

restart – rozpoczyna od nowa zadanie

stop - przechodzi na koniec zadania

Instrukcja *try* respektuje priorytet danego zadania i najpierw wykonana zostanie instrukcja *try* dla zadania o wyższym priorytecie.

7.1.8 Preprocesor

#include

Pozwala na dołączenie do programu dodatkowych plików, najczęściej plików nagłówkowych. Nazwy plików umieszcza się w ostrych nawiasach i gdy plik nie znajduje się w katalogu domyślnym należy podać pełną ścieżkę do niego.

Przykład:

```
#include <RCX2.h>
```

7.2 RCX Mindscript API

7.2.1 Czujniki

Aby możliwe było użycie czujnika trzeba go najpierw zadeklarować[4]:

sensor nazwa on port

gdzie *nazwa* to nadana przez użytkownika nazwa czujnika a *port* to numer portu do którego podłączony jest czujnik dla RCX to 1,2,3

Ustawienie typu czujnika:

nazwa is typ_sensora

Dostępne typy czujnkiów:

Typ	Opis
light	Czujnik światła
rotation	Czujnik obrotów
switch	Czujnik dotyku
temperature	Czujnik temperatury
unknown	Czujnik nieznanego typu

Ustawienia trybu pracy czujnika:

nazwa as tryb_pracy

Dostępne tryby pracy czujników:

Tryb	Opis
angle	Kąt obrotu
boolean	Wartość binarną 0 lub 1
celsius	Temperatura w °C od –20 do 50
fahrenheit	Temperatura w °F od –4 do 122
percent	Wartość procentowa od 0 do 100
periodic	Zlicza okresy 0-1-0 lub 1-0-1
raw	Wartości od 0 do 1023
transition	Zlicza przejścia między stanami 0 a 1

Dla trybu *celsius* i *fahrenheit* wartość przechowywana jest to wartość faktyczna pomnożona przez 10. Należy pamiętać, że zapis:

```
temperatura=100      // oznacza 10 stopni
```

Można także zdefiniować jednocześnie typ i tryb czujnika:

```
s is temperature as celsius
```

Dla czujnika typu *switch* można wyróżnić dwa stany: opened – otwarty i closed – zamknięty.

clear nazwa_sensora – zeruje wartość danego czujnika

get nazwa_sensora – pobiera wartość czujnika

Po zadeklarowaniu czujnika i ustawieniu jego typu i trybu pracy, wartość jego można przypisać zmiennej:

```
zmienna=sensor1          //przypisanie zmiennej wartości czujnika1
zmienna=sensor1.raw      //przypisanie zmiennej wartości czujnika1 przy
                         jednoczesnej konwersji do raw
```

7.2.2 Silniki

Na początku programu należy zadeklarować nazwy, jakie będą używane dla poszczególnych wyjść. Za pomocą tych nazw w programie będzie można do nich odwoływać.

output nazwa on port // port to numer danego portu 1,2,3

on nazwa – włącza wyjście o podanej nazwie. Można włączać kilka wyjść naraz. W takim przypadku, nazwy wyjść zapisuje się w nawiasach kwadratowych

on [lewy prawy]
on lewy for czas // włącza wyjście lewy na określony czas w milisekundach
on lewy for random czas // włącza wyjście na losowy czas od 0 do czas
on lewy for random czas1 to czas2 //włącza wyjście na losowy czas od czas1 do czas2

off nazwa – wyłącza dane wyjście. Podobnie jak i dla on można sterować kilkoma wyjściami naraz.

off [lewy prawy]

float nazwa - włącza wolny bieg dla danego wyjścia. Podobnie jak i dla on można sterować kilkoma wyjściami naraz.

float [lewy prawy]

forward nazwa / fd nazwa – ustawia kierunek wyjścia do przodu

backward nazwa / bk nazwa – ustawia kierunek wyjścia do tyłu

direction nazwa nazwa / dir nazwa nazwa - ustawia kierunek do przodu/ tyłu wyjść.

dir [lewy środkowy] prawy // wyjścia lewy i środkowy do przodu a prawy do // tyłu
dir prawy lewy // wyjście prawe do przodu a lewe do tyłu

reverse nazwa – zmienia kierunek wyjść

power nazwa moc – ustawia moc wyjść. Moc jest liczbą z zakresu 1-8. Można też ustawić moc jako liczbę losową

```
power lewy 3
power prawy random 1 to 5
```

Istnieją też komendy do ustawień globalnych dla wyjść. Pozwalają one na początku programu zmienić ustawienia początkowe wyjść.

global on nazwa – włącza wyjście o podanej nazwie. Nie można ustawić czasu działania wyjścia

global off nazwa – wyłącza dane wyjście

global float nazwa - włącza wolny bieg dla danego wyjścia

global forward nazwa / global fd nazwa – ustawia kierunek wyjścia do przodu

global backward nazwa / global bk nazwa – ustawia kierunek wyjścia do tyłu
global direction nazwa nazwa / global dir nazwa nazwa - ustawia kierunek do przodu/ tyłu wyjść.

```
dir [lewy środkowy]  prawy   // wyjścia lewy i środkowy do przodu a prawy do
                             // tyłu
dir prawy lewy               // wyjście prawe do przodu a lewe do tyłu
```

global reverse nazwa – zmienia kierunek wyjść

global power nazwa moc – ustawia maksymalną moc wyjść. Moc jest liczbą z zakresu 1-8. Można też ustawić moc jako liczbę losową

```
global power lewy 3
global power prawy random 1 to 5
```

7.2.3 Głośnik

sound n – odgrywa dźwięk, gdzie n jest z zakresu 1-6

sound on – włącza głośnik

sound off – wyłącza głośnik

clear sound – czyści bufor głośnika

tone częstotliwość for czas – odgrywa dźwięk o danej częstotliwości (Hz) przez określony czas

W pliku nagłówkowym RCX2Sounds.h zdefiniowane zostały makra definiujące kilkanaście podstawowych dźwięków, które można użyć w programie. Makra mają nazwy od *sound_1* do *sound_28*.

7.2.4 Wyświetlacz LCD

display wartość – wyświetla wartość na wyświetlaczu. Wartość może być stałą, zmienną, wartością czujnika, licznika, timera. Zmienna lokalna nie może być wyświetlana.

Przykład:

```
display  x
display 314:2  // wyświetli 314 z dwoma miejscami po przecinku czyli 3,14.
                  // Liczba miejsc po przecinku może wynosić 1,2 lub 3.
```

7.2.5 Komunikacja

Język Mindscript pozwala na wysyłanie wiadomości za pomocą portu IR. Wiadomością może być liczba całkowita z zakresu 1-255. Wiadomości przechowywane są w specjalnym rejestrze, dlatego powinno się wykasować całą jego zawartość na początku programu.

clear message – kasuje wszystkie odebrane wiadomości

get message – pobiera ostatnią wiadomość

send wiadomość – wysyła wiadomość

7.2.6 Timery

RCX udostępnia 4 niezależne timery o rozdzielczości 100 ms (10 zliczeń na sekundę). Na początku programu należy zadeklarować nazwy timerów.

timer nazwa – deklaracja użycia timera
clear nazwa – zeruje dany timer

7.2.7 Licznik (tylko RCX2)

counter nazwa – deklaracja globalnego licznika
clear nazwa – zeruje licznik

Liczniki różnią się tym od zmiennych, że można dokonywać tylko ich inkrementacji, dekrementacji oraz zerowania.

7.2.8 Zdarzenia

Źródłem zdarzeń mogą być czujniki, timery, liczniki lub wiadomości. Mogą być wykorzystywane przez watcher, monitor, pętlę repeat. Na początku należy zdefiniować zdarzenie. Służy do tego komenda:
event nazwa_zdarzenia when warunek – warunek ma zwykle postać
nazwa_źródła.stan

Poniżej znajduje się lista możliwych deklaracji zdarzeń:

event nazwa when z.pressed - czujnik jest wciśnięty
event nazwa when z.released - czujnik jest zwolniony
event nazwa when z.high - wartość jest powyżej górnego limitu
event nazwa when z.normal - wartość jest pomiędzy dolnym i górnym limitem
event nazwa when z.low - wartość jest poniżej dolnego limitu
event nazwa when z.click - pojedyncze kliknięcie
event nazwa when z.doubleclick - podwójne kliknięcie
event nazwa when wiadomość - zdarzenie wywoływane określoną wiadomością
event nazwa when warunek - spełniony jest podany warunek

Zdarzenia oparte na czujnikach, są automatycznie kalibrowane podczas startu programu, ale mogą być ponownie skalibrowane w każdym miejscu programu za pomocą poniższej komendy:

calibrate nazwa_zdarzenia

Dla niektórych zdarzeń można zdefiniować własne własności takie jak low, high, time, state. Wykonuje się to za pomocą:

nazwa_zdarzenia.własność = wartość

nazwa_zdarzenia.low - dolny limit
nazwa_zdarzenia.high - górny limit
nazwa_zdarzenia.time - czas dla pojedynczego i podwójnego kliknięcia
nazwa_zdarzenia.state - stan zdarzenia. Są 4 stany: 0 = niski, 1 = normalny, 2= wysoki, 3 = nieokreślony

Wartości tych własności można też odczytać, podstawiając je do zmienne.

x=zdarzenie.low

Do monitorowania zdarzeń służy polecenie:

monitor zdarzenie { instrukcje} akcja on ewent.

Możliwe są następujące akcje, w przypadku nastąpienia zdarzenia, zanim zostaną wykonane instrukcje zawarte w monitorze:

retry – próbuje wykonać instrukcje raz jeszcze
abort - wykonuje dalszy program po monitorze
restart – rozpoczyna od nowa zadanie
stop - przechodzi na koniec zadania

Gdy zdarzenie zostało już zdefiniowane, należy zdefiniować zdarzenie zwane *watcher*, którego zadaniem jest monitorowanie zdarzeń. W momencie nastąpienia zdarzenia wykonywany jest kod zawarty w tym zadaniu, a po zakończeniu nastąpi powrót do wykonywania głównego programu.

watcher nazwa monitor zdarzenie { instrukcje }

Uruchamianie jest identyczne z uruchamianiem zadań:

start nazwa

Gdy nastąpi zdarzenie, a nie zostanie zakończone wykonywanie kodu watcher-a, zdarzenie to jest ignorowane chyba, że watcher ma postać:

watcher nazwa monitor zdarzenie { instrukcje } restart on event

w takim przypadku wykonywanie instrukcji zostanie przerwane i rozpoczęte od nowa.
fire zdarzenie – wywołuje zdarzenie
trigger zdarzenie – sprawdza warunki zdarzenia i je wywołuje jeśli są spełnione

7.2.9 Rejestr danych (Datalog)

RCX posiada rejestr danych (datalog), w którym można zapisywać wartości pobrane z zmiennych, czujników czy timerów. Na początek należy zarezerwować pamięć potrzebną do przechowywania pobranych wartości. Należy pamiętać, że każda pobrana wartość zajmuje 3 bajty.
clear data n – rezerwuje pamięć na przechowanie n elementów
log nazwa – zapisuje do datalogu wartość zmiennej, timera, czujnika, licznika o podanej nazwie
get data początek,n – pobiera zapisane n elementów od elementu początkowego

7.2.10 Inne polecenia

abs (wyrażenie) – zwraca wartość bezwzględną wyrażenia

brick alive? – zwraca 1 jeśli robot włączony, gdy wyłączony zwraca 0

brick battery? – zwraca stan baterii w milivoltach

brick version? – zwraca wersję ROM i firmware

brick transmitter power n – ustawia moc nadajnika robota n=0 lub 1

brick tx power n - ustawia moc nadajnika robota n=0 lub 1 (skrócona wersja)
boot firmware – odblokowuje firmware

boot rom – przechodzi do trybu boot przed dowloadem firmwaru

clear tasks – kasuje wszystkie zadania

clear task nazwa – kasuje zadanie

clear subs – kasuje wszystkie podprogramy

clear sub nazwa – kasuje podprogram

clear sleep – resetuje odliczanie czasu do wyłączenia robota

get map – pobiera mapę pamięci (tylko **RCX2**)

priority priorytet – ustawia priorytet dla zadań, watcher-ów. Priorytet jest liczbą całkowitą z zakresu 1-256, gdzie 1 to priorytet najwyższy, a 256 najniższy

randomize – ustala zarodek dla generatora liczb losowych

random liczba/ random liczba1 to liczba2 – generuje liczbę losową z zakresu 0 – liczba lub liczba1 – liczba2

sleep - wyłącza robota

sleep after czas - wyłącza robota po określonym czasie (w minutach).

slot numer – wybiera numer programu (1-5) (tylko **RCX2**)

wait czas – zatrzymuje zadanie na określony czas.

watch godzina:minuty – ustawia zegar robota (tylko **RCX2**)

8 RCX Code

Standardowym, dostarczonym wraz z zestawem LEGO Mindstorms, środowiskiem do programowania w języku RCX (Robotics Command System) Code jest Robotics Invention System 2.0. RCX Code jest językiem w pełni funkcjonalnym, a zarazem bardzo prostym w użyciu – wszystkie polecenia są zobrazowane w postaci różnokolorowych klocków, które należy ułożyć w odpowiedniej kolejności. Jego prosta struktura nie przeszkadza w pisaniu skomplikowanych, rozbudowanych programów.

8.1 Środowisko

Rys. 8.1 Ekran powitalny RIS 2.0

Okno powitalne programu Robotics Invention System 2.0 (rys. 8.1) zawiera szereg różnych opcji:

- **TOUR** – wyświetla krótki film o robotach LEGO
- **MISSIONS** – misje treningowe

  - *Training Missions* – Nauka podstawowych zasad programowania w języku RCX Code

  - *Challenges* – Proste zadania, które uczą jak programować, jak przesłać gotowy program do robota

- ***Pro Challenges*** – Skomplikowane zadania, które wymagają dość dużej inwencji twórczej podczas programowania.

- **PROGRAM** – menu programowania robotów

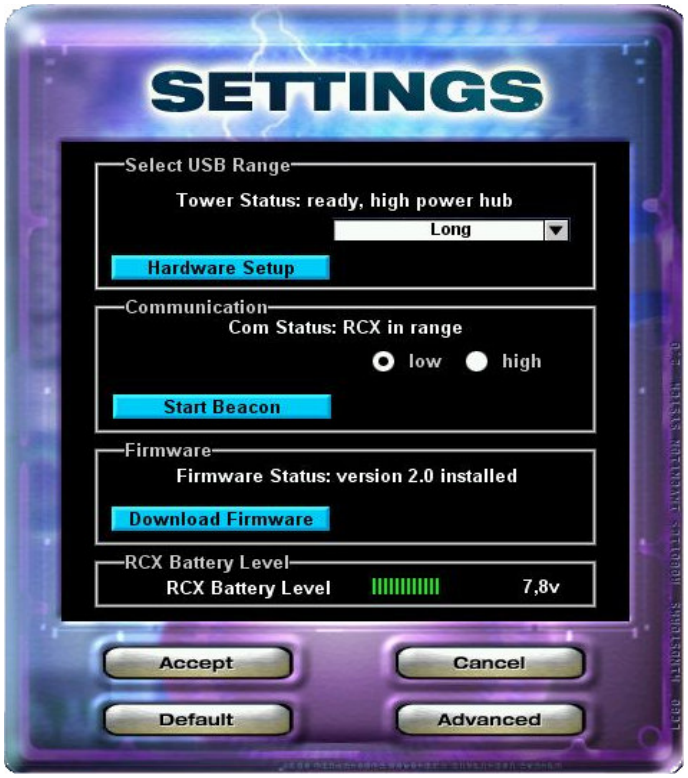
  - ***Pick a Robot*** – programowanie jednego z dostępnych, standardowych robotów

  - ***Freestyle*** – programowanie dowolnego, skonstruowanego robota

  - ***Vault*** – edycja wcześniej napisanych programów

- **LIBRARY** – biblioteka, w której znajdują się przykładowe programy oraz użyteczne wskazówki, jak budować roboty LEGO.

- **SETTINGS** – **podstawowy panel kontrolny** (rys. 8.2), w którym jest możliwość ustawienia zasięgu nadajnika/odbiornika USB, załadowania standardowego firmware-u. Można także odczytać poziom naładowania baterii.



Rys. 8.2 Podstawowe opcje konfiguracyjne.

Zaawansowany panel kontrolny (rys. 8.3), można w nim ustawić czas, po którym robot sam się wyłączy, systemowy timer oraz zablokować sloty od 1 do 5. Można ponadto wykasować lub przywrócić załadowane standardowo programy. Gdy użytkownik posiada dodatkowe czujniki takie jak czujnik ruchu lub temperatury, należy zaznaczyć ich obecność w odpowiednim polu.

Rys. 8.3 Zaawansowane opcje konfiguracyjne.

8.2 Struktura menu

Menu, w którym znajdują się polecenia, jest podzielone na sekcje. W każdej sekcji jest zbiór poleceń. Pierwsza sekcja - Big Blocks – zawiera w sobie zestawy predefiniowanych bloków dla standardowych robotów takich jak: Roverbot, Acrobot, Inventorbot itp. Każdy z dużych bloków zawiera w sobie zestaw poleceń, które w odpowiedni sposób wykonują zdefiniowaną „akcję". Poniżej grupy Big Blocks, znajdują się pozostałe grupy z podstawowymi blokami funkcyjnymi.

Budowa struktury menu została przedstawiona poniżej:

BIG BLOCKS
SMALL BLOCKS
- ***Power***
  - On
  - On for
  - Off
  - Set power
  - Set direction
  - Reverse direction
- ***Sound***
  - Beep
  - Tone
  - Mute sounds
  - Un-mute sounds
- ***Comm***
  - Send IR message
  - Clear IR message
  - Display value
  - Display clock
- ***Variable***
  - Set
  - Add
  - Subtract
  - Multiply
  - Divide
  - Make positive
  - Make negative
- ***Reset***
  - Reset light
  - Reset timer
- ***Advanced***
  - Set priority
  - Global reverse
  - Global set direction
  - Connect power
  - Disconnect power
  - End program

MY BLOCKS
- Create my New block

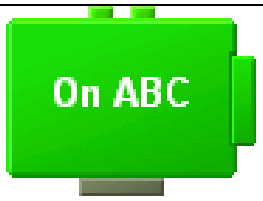
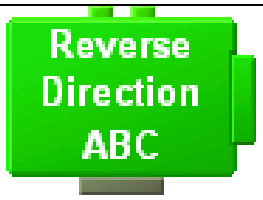
WAIT
- Wait for
- Wait until

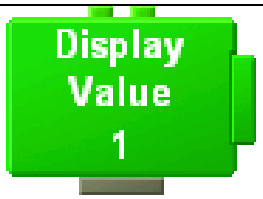
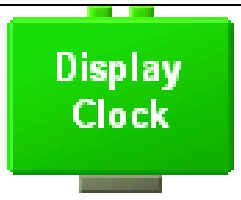
REPEAT
- Repeat for
- Repeat forever
- Repeat while
- Repeat until

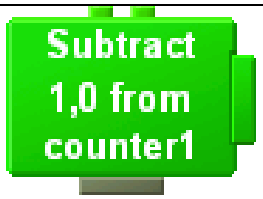
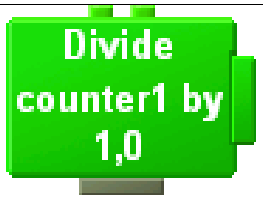
YES OR NO
- Yes or no

SENSORS
- Touch sensor
- Light sensor
- Timer sensor
- IR sensor
- Variable sensor

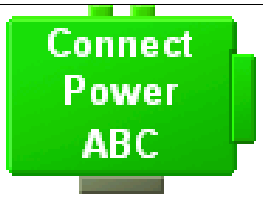
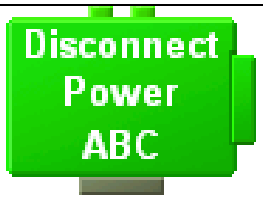
8.3 Bloki funkcyjne RCX Code

	**Power On** – komenda służąca do włączania silników lub innych urządzeń podłączonych do portów wyjściowych. Urządzenia włączone tym blokiem pozostaną aktywne, dopóki nie zostaną wyłączone specjalnym rozkazem. W przeciwnym wypadku nawet, gdy program się zakończy urządzenia dalej będą pracować.
	**Power On For** – blok włączający porty wyjściowe A,B lub C na określony czas. Po upływie wskazanego czasu blok automatycznie wyłącza porty. Domyślnie blok włącza wszystkie trzy porty na 1 sekundę.
	**Power Off** – komenda wyłączająca urządzenia na wyjściach A,B lub C.
	**Set Power** – blok zmieniający prędkość obrotową silników. Do wyboru jest 8 możliwości: 1-najwolniejszy, 8-najszybszy. Prędkość jest ustawiana domyślnie na 8 za każdym razem, gdy program jest na nowo uruchamiany.
	**Set Direction** – komenda określająca kierunek obrotów silnika na wyjściu A,B lub C. Bieżący kierunek jest pokazany przez strzałkę. W zależności od potrzeb, można zmieniać kierunek obracania się silników niezależnie.
	**Reverse Direction –** kolejny blok określający kierunek obrotów podłączonych silniczków. Każdorazowe użycie tego bloku powoduje natychmiastową zmianę kierunku obracania się silników.

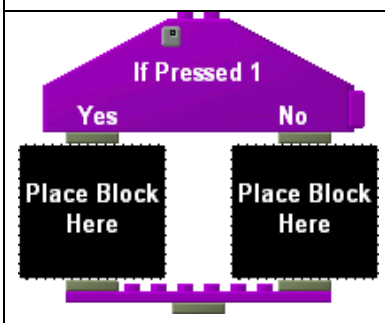
	**Beep** – komenda, dzięki której RCX zagra jeden z sześciu dostępnych, standardowych tonów.
	**Tone –** blok, który umożliwia sprecyzowanie wydawanego dźwięku – można wybrać częstotliwość od 31 do 20000hz oraz czas trwania dźwięku.
	**Mute Sounds** – blok, który nakazuje RCX ignorowanie wszelkich innych bloków typu *tone* lub *beep*, dopóki nie dostanie instrukcji *un–mute sounds*.
	**Un-Mute Sounds** – instrukcja włączająca dźwięki po zastosowaniu instrukcji *mute sounds*.
	**Send IR Message** – blok wysyłający podczerwienią wiadomość do innego RCX. Wiadomością może być liczba z zakresu <1,255>.
	**Clear IR Message** – blok, który kasuje wcześniej otrzymaną wiadomość. Blok ten musi być użyty, jeśli użytkownik chce, aby RCX odczytał nową wiadomość.
	**Display Value –** komenda wyświetlająca na wyświetlaczu: - wartość wiadomości IR - zmienną - timer - wartość czujnika - liczbę z zakresu -9999 do 9999 lub -999,9 do 999,9
	**Display Clock** – blok wyświetlający na wyświetlaczu LCD wartość wbudowanego licznika – jest to czas, jaki upłynął od momentu włączenia programu.

	**Set Variable** – komenda ustawiająca wartość danej zmiennej. Kiedy po raz pierwszy zmienna jest tworzona jej wartość jest ustawiana na zero. *Set Variable* umożliwia zmianę jej wartości w zakresie -3276,8 do 3276,7. Blok ten umożliwia również przypisanie do zmiennej timera, wartości losowej, wartości czujnika lub wartości wiadomości otrzymanej podczerwienią.
	**Add Variable** – blok, który umożliwia dodanie do zmiennej: - dowolnej liczby z zakresu -3276,8 do 3276,7 - liczby z wiadomości otrzymanej podczerwienią - losowej wartości wygenerowanej przez generator - wartości danego czujnika - wartości innej zmiennej - inną zmienną
	**Subtract Variable** – blok odejmuje pewną wartość od danej zmiennej. Od zmiennej można odjąć: - liczbę z zakresu do -3276,8 do 3276,7 - liczbę wygenerowaną przez generator liczb losowych - wartość licznika - wartość czujnika - wartość wiadomości otrzymanej podczerwienią - inną zmienną
	**Multiply Variable** – komenda mnożąca daną zmienną przez żądaną wartość. Zmienna może być mnożona przez: - liczbę z zakresu do -3276,8 do 3276,7 - liczbę wygenerowaną przez generator liczb losowych - wartość licznika - wartość czujnika - wartość wiadomości otrzymanej podczerwienią - inną zmienną
	**Divide Variable** – blok dzielący zmienną przez żądaną wartość. Zmienna może być dzielona przez: - liczbę z zakresu do -3276,8 do 3276,7 z wyjątkiem 0 - liczbę wygenerowaną przez generator liczb losowych - wartość licznika - wartość czujnika - wartość wiadomości otrzymanej podczerwienią - inną zmienną

	**Make Positive** – blok zmienia daną wartość reprezentowaną przez dowolną zmienną na dodatnią. Jeśli wartość zmiennej jest ujemna, a do obliczeń potrzebna jest jej wartość dodatnia, stosować należy komendę *make positive*.
	**Make Negative** – blok zmienia daną wartość reprezentowaną przez dowolną zmienną na ujemną. Jeśli wartość zmiennej jest dodatnia, a do obliczeń potrzebna jest jej wartość ujemna, stosować należy komendę *make negative*.
	**Reset Light Sensor** – blok resetujący domyślne ustawienia czujnika światła; dostosowuje parametry czujnika tak aby poprawnie funkcjonował w nowym pomieszczeniu. Komenda stosowana tylko, gdy czujnik światła pracuje w trybie automatycznym, czyli sam wykrywa jasność lub ciemność.
	**Reset Timer** – blok ustawiający wartość timera na 0. RCX posiada trzy timery, których wartość można niezależnie resetować.
	**Set Priority** – blok ustawiający priorytet czujnika zdarzeń. Przykładowo, gdy w programie występuje obsługa dwóch lub więcej czujników i gdy dwa z nich zostaną naciśnięte w tym samym momencie, może zaistnieć sytuacja, że będą chciały się odwołać do tego samego zasobu (np. dźwięk). Aby uniknąć takiej sytuacji należy stosować komendę *set priority* – czujnik o wyższym priorytecie dostanie zezwolenie na wykorzystanie zasobu, natomiast czujnik o niższym priorytecie zostanie zignorowany. Najwyższy priorytetem jest proces o numerze 1 natomiast najniższy jest proces o numerze 8. (UWAGA, nie wolno nadawać procesom numeru 8, RCX może się zawiesić)

	**Global Reverse** – blok, który zmienia polaryzację napięcia na wyjściach A, B lub C. Silniki podłączone do tych wyjść zmienią kierunek obrotów. Powtórne użycie komendy *global reverse*, przywróci stan wyjść do stanu początkowego. UWAGA!!! Dostępne są dwie komendy zmieniające kierunek obrotów silników: reverse direction oraz *global reverse*. *Reverse direction* zmienia kierunek jednakże komendy takie jak *forward* lub *reverse* będą działały prawidłowo. Komenda *global reverse* powoduje zmianę kierunku oraz powoduje zmianę funkcjonowania komend *forward* oraz *reverse* na przeciwne.
	**Global Set Direction** – komenda ustawia polaryzację napięcia na wyjściach A,B lub C. Jeśli zostanie ustawiony kierunek obracania się silników zgodnie z kierunkiem ruchu wskazówek zegara, komendy *forward* oraz *backward* będą działały poprawnie. Gdy kierunek obracania się silników zostanie zmieniony na przeciwny do ruchu wskazówek zegara, powyższe komendy będą działały na odwrót. Blok ten jest użyteczny, gdy stosuje się bloki typu *global reverse*, a użytkownik chce przywrócić wybranym silnikom ich pierwotny kierunek.
	**Connect Power** – komenda przywracająca zasilanie do portów A,B lub C, odłączonych komendą *disconnect power*.
	**Disconnect Power** – odłącza napięcie na portach A,B lub C. Przydatna komenda przy testowaniu kodu programu; robot nie będzie się poruszał.

	**End Program** – blok sygnalizujący koniec programu. Jeśli w programie są użyte czujniki(monitory) zdarzeń, a na końcu programu nie zostanie umieszczony blok *end program*, po zakończeniu programu czujniki dalej będą pracować.
	**My Block** – blok *my block* służy do przechowywania w sobie sekwencji poleceń, rozkazów. Stosowany go, gdy pewien fragment kodu jest niezmiennie wykorzystywany w różnych częściach programu. Stosowanie tego bloku zapobiega powstawaniu nadmiernej ilości kodu w programie.
	**Wait for** – komenda zatrzymująca wykonywanie kolejnych części programu na określony czas. Czas czekania musi zawierać się w granicach od 0,01 do 327,67 sekund. Liczba ta może być ustawiana ręcznie, może być liczbą wygenerowaną losowo lub może być wartością danej zmiennej.
	**Wait Until** – blok czekający na zaistnienie pewnego zdarzenia. Gdy monitorowane zdarzenie zaistnieje, zostają wykonywane kolejne części kodu. Monitorowane mogą być: - czujnik dotyku - czujnik światła - port IR - zmienna - timer
	**Repeat For** – blok powtarzający sekwencję kodu zadaną ilość razy. Gdy kod, zawierający się w bloku *repeat*, zostanie wykonany oczekiwaną przez nas ilość razy, program przejdzie do wykonywania następnych instrukcji. Liczba powtórzeń musi się zawierać w przedziale od 2 do 32767, ich liczba może zostać ustawiona ręcznie, poprzez zmienną lub wartość wygenerowaną losowo.

	**Repeat Forever** – blok powtarzający pewną sekwencję kodu nieskończoną liczbę razy, aż do zakończenia programu.
	**Repeat While** – blok powtarzający pewną sekwencję kodu, dopóki ustalony warunek jest prawdą. Blok sprawdza czy warunek jest prawdą każdorazowo po wykonaniu zawierającego w sobie kodu. Jeśli warunek jest prawdą kod zostanie wykonany ponownie, jeśli jest fałszem program przejdzie do wykonywania kolejnych instrukcji. Blok repeat za warunek do działania może przyjąć: - stan czujnika dotyku - stan czujnika światła - wartość wiadomości otrzymanej z portu IR - wartość zmiennej
	**Repeat Until** – blok powtarzający pewną sekwencję kodu, dopóki ustalony warunek jest prawdą. Jeśli określony warunek został spełniony, pętla repeat jest natychmiast przerywana, program przechodzi do wykonywania kolejnych instrukcji. Warunek do działania pętli może być ustalony na podstawie: - stanu czujnika dotyku - stanu czujnika światła - wartości wiadomości otrzymanej z portu IR - wartośći zmiennej
	**Yes or No** – blok sprawdzający dane zdarzenie. Jeśli warunki zostaną spełnione, zostanie uruchomiony zestaw komend, jeśli warunki nie zostaną spełnione, zostanie uruchomiony inny zestaw komend. Zdarzeniem do sprawdzenia np. może być: - wartość czujnika dotyku, światła, zmiennej - wartość wiadomości z portu IR

	**Touch Sensor** – Monitor czujnika dotyku. Określa stan czujnika, po którym sekwencja kodu ma zostać wykonana (czujnik naciśnięty, wolny, „klik”) **Light Sensor** – Monitor czujnika światła. Określa stan czujnika, po którym sekwencja kodu ma zostać wykonana (ciemność, jasność, błysk). **Timer Sensor** – Monitor wartości timera. Gdy timer osiągnie żądaną wartość, zostaje wykonana określona sekwencja kodu. Za każdym razem, gdy program jest włączany, timer zaczyna liczyć czas od zera, co sekundę. **IR Message Sensor** – Monitor wiadomości IR. Odpowiednia sekwencja kodu zostanie wykonana, gdy wartość wiadomości otrzymanej z portu IR jest zgodna z ustalonym warunkiem. **Variable Sensor** – Monitor wartości zmiennej. Gdy zmienna osiągnie żądaną wartość zostanie wykonana określona sekwencja kodu.

9 Przykładowe programy, porównanie języków

Aby sprawdzić funkcjonalność języków programowania, dla każdego języka zostały napisane dwa programy.

Pierwszy program – **wyjazd z tunelu** – ma za zadanie wyprowadzić robota ze ślepo zakończonego tunelu. Program ten zapobiega „zakleszczaniu się" pojazdu we wszelkiego rodzaju narożach oraz ciasnych miejscach. Gdy czujnik robota zostanie wciśnięty, zostaje uruchomiony timer. Jeśli w przeciągu ośmiu sekund czujnik dotyku zostanie wciśnięty trzy razy (pojazd nieprzerwanie będzie napotykał tę samą przeszkodę), robot wycofa się oraz wykona duży zakręt. Jeśli w przeciągu ośmiu sekund czujnik nie zostanie wciśnięty trzy razy, robot będzie próbował objechać przeszkodę.

Może on stanowić samodzielny program, a także może być wykorzystany jako część większego projektu. Przy konstrukcji programu zostały wykorzystane takie mechanizmy jak: timery, czujniki, wyświetlacz, głośniczek.

Drugi program – **poszukiwacz latarni** – jak sama nazwa wskazuje, ma odnaleźć w pomieszczeniu specjalny nadajnik IR. Robot będzie się obracał dopóki czujnik światła nie natrafi na promień IR. Jeśli natrafi na promień, zacznie się poruszać w kierunku „latarni".

9.1 NQC

Program wyjazd z tunelu

```
task main (){
int var =0;
int time1=0  ;    //czas dla czujnika1
int lewo =0;
int prawo=0;
int time2=0;      //czas dla czujnika2
  SetSensor (SENSOR_1, SENSOR_TOUCH);  //do portu 1 i 3 podlaczone czujniki dotyku
  SetSensor (SENSOR_3, SENSOR_TOUCH);
  OnFwd(OUT_A+OUT_C);
    ClearTimer (0); ClearTimer (1);
    while (true){
      if (SENSOR_1 == 1){          //jeśli czujnik1 zostal nacisniety
      time1=Timer(0);              //zacznij liczyc czas dla czujnika lewego
      lewo=lewo+1;                 //zliczaj uderzenia
        if (lewo==3){              //jesli uderzenia = 3 oraz czas <8sek
          if (time1 < 80){         //zagraj melodie oraz wykonaj duzy zwrot
            PlayTone (440, 80);
            PlayTone( 600,80);
            PlayTone (440, 80); PlayTone( 600,80);
            OnRev(OUT_A+OUT_C); Wait(600);
```

```
            OnFwd(OUT_A); Wait(80);
            OnFwd(OUT_A+OUT_C);
          }
      }
      else{                                  //jesli czas > 8sek
      SetUserDisplay(time1,0);               //wykonaj maly zwrot
      OnRev(OUT_A+OUT_C); Wait(30);
      OnFwd(OUT_A); Wait(30);
      OnFwd(OUT_A+OUT_C);
      if (time1>80){lewo=0; ClearTimer (0); time1=0;}
      }
    }
    if (SENSOR_3 == 1){          //jeśli czujnik3 zostal nacisniety
      time2=Timer(1);            //zacznij liczyc czas dal czujnika prawego
      prawo=prawo+1;             //zliczaj uderzenia
      if (prawo==3){             //jesli uderzenia = 3 oraz czas <8sek
        if (time2 < 80){         //zagraj melodie oraz wykonaj duzy zwrot
          PlayTone (440, 80); PlayTone( 600,80);
          PlayTone (440, 80); PlayTone( 600,80);
          OnRev(OUT_A+OUT_C); Wait(600);
          OnFwd(OUT_C); Wait(80);
          OnFwd(OUT_A+OUT_C);
        }
      }
      else{                                  //jesli czas > 8sek
        SetUserDisplay(time2,0);             //wykonaj maly zwrot
        OnRev(OUT_A+OUT_C); Wait(30);
        OnFwd(OUT_C); Wait(30);
        OnFwd(OUT_A+OUT_C);
        if (time2>80){prawo=0; ClearTimer (1); time2=0;}
    }
  }
  }
}
```

Program poszukiwacz latarni

```
#define POZIOMSWIATLA 65

task main(){
  SetSensor(SENSOR_2,SENSOR_LIGHT);    //czujnik swiatla podlaczony do portu 2
  SetPower(OUT_A, 4);                  //ustawiamu moc silnikow na 4
  SetPower(OUT_C, 4);
  OnFwd(OUT_A);                        //niech robot sie obraca w poszukiwaniu
  OnRev(OUT_C);                        //silnego zrodla swiatla
  while (true){
    if (SENSOR_2 > POZIOMSWIATLA){     //jesli czujnik wykryje natezenie swiatla
      SetPower(OUT_A, 7);              //wieksze od POZIOMSWIATLA, robot
      SetPower(OUT_C, 7);              //przestanie sie obracac i pojedzie w kierunku
      OnFwd(OUT_A);                    //jego zrodla
      OnFwd(OUT_C); Wait(110);
    }
    if (SENSOR_2 < POZIOMSWIATLA){     //jesli natezenie swiatla nie bedzie wieksze
      SetPower(OUT_A, 4);              //od POZIOMSWIATLA, robot w dalszym ciagu
      SetPower(OUT_C, 4);              //będzie się obracal
      OnFwd(OUT_A);
      OnRev(OUT_C);
    }
  }
}
```

9.2 BRICKOS

Program wyjazd z tunelu

```
#include <time.h>
#include <sys/tm.h>
#include <conio.h>
#include <unistd.h>
#include <dsensor.h>
#include <dmotor.h>
#include <rom/system.h>

void go(){
  int lewo=0;
  int prawo=0;
  int t21,t11,czl1,czl4,czp1,czp4;
  time_t t1,czas1;
  time_t t2,czas2;
  lcd_clear();
  motor_a_speed(210);
  motor_c_speed(210);
  motor_a_dir(fwd);
  motor_c_dir(fwd);
  t1 = get_system_up_time()/TICKS_PER_SEC;          //pobieramy czas systemowy
  t11=(int)t1;                                      //czas t1do integer-a
  while(1){
    if(TOUCH_1!=0){                                 //gdy czujnik1 zostanie nacisniety
      t2 = get_system_up_time()/TICKS_PER_SEC;      //znow pobierz czas systemowy
      t21=(int)t2;                                  //czas t2 do integer-a
      lewo=lewo+1;                                  //zlicz uderzenie
      if (lewo==1){                     //gdy jest to pierwsze uderzenie
        czl1=(int)t2;                   //czas t2 do integera
      }
      if (lewo==4){                     //gdy jest to trzecie uderzenie
        czl4=(int)t2;                   //czas t3 do integer-a
        if (czl4-czl1<8){               //obliczamy ile sekund minelo od uderzenia 1 do 3
          motor_a_dir(rev);             //jesli <8 wykonaj duzy zwrot
          motor_c_dir(rev);
          msleep(1600);
          motor_a_dir(fwd);
          motor_c_dir(rev);
          msleep(1800);
          motor_a_dir(fwd);
          motor_c_dir(fwd);
          lewo=0;                       //zerujemy licznik uderzen
        }
      }
      else{                             //jesli minelo 8 sekund wykonaj maly zwrot
        motor_a_dir(rev);
        motor_c_dir(rev);
        msleep(300);
        motor_a_dir(fwd);
        motor_c_dir(rev);
        msleep(300);
        motor_a_dir(fwd);
        motor_c_dir(fwd);
        if (czl4-czl1>=8){lewo=0;}      //jesli czas od zderzenia 1 do 3 jest wiekszy od 8 sek
      }                                 //zerujemy lewy licznik uderzen
    }
```

```
    if(TOUCH_3!=0){                                  //gdy czujnik3 zostanie nacisniety
      t2 = get_system_up_time()/TICKS_PER_SEC;       //pobierz czas systemowy
      t21=(int)t2;
      prawo=prawo+1;                                 //zliczaj uderzenia
      if (prawo==1){                                 //pobierz czas przy pierwszym
uderzeniu
        czp1=(int)t2;
      }
      if (prawo==4){
        czp4=(int)t2;                  //pobierz czas przy trzecim uderzeniu
        if (czp4-czp1<8){              //obliczamy ile sekund minelo od uderzenia 1 do 3
          motor_a_dir(rev);            //jesli <8 wykonaj durzy zwrot
          motor_c_dir(rev);
          msleep(1600);
          motor_a_dir(rev);
          motor_c_dir(fwd);
          msleep(1800);
          motor_a_dir(fwd);
          motor_c_dir(fwd);
          prawo=0;                     //zerujemy licznik uderzen
        }
      }
      else{                            //jesli minelo 8 sekund wykonaj maly zwrot
        motor_a_dir(rev);
        motor_c_dir(rev);
        msleep(300);
        motor_a_dir(rev);
        motor_c_dir(fwd);
        msleep(300);
        motor_a_dir(fwd);
        motor_c_dir(fwd);
        if (czp4-czp1>=8){prawo=0;}    //jesli czas od zderzenia 1 do 3 jest wiekszy od 8 sek
      }                                //zerujemy prawy licznik uderzen
    }
  }
}
int main(){
  go();
  return 0;
}
```

Program poszukiwacz latarni

```
#include <sys/tm.h>
#include <dmotor.h>
#include <dsensor.h>
#include <unistd.h>
#include <sys/lcd.h>

#define LIGHT_INPUT          SENSOR_2
#define LIGHT_SENSOR           LIGHT_2

int main(){
  motor_a_speed(250);
  motor_c_speed(250);
  ds_active(&LIGHT_INPUT);             //ustawiamy port 2 na aktywny
  while(1){
    if(LIGHT_SENSOR > 65){             //jesli poziom wykrytego swiatla jest wiekszy od 65
      motor_a_dir(fwd);                //podazaj w jego kierunku
      motor_c_dir(fwd);
```

```
      sleep(1);
    }
    if(LIGHT_SENSOR < 65){          //jesli poziom swiuatla jest mniejszy od 65
      motor_a_dir(rev);             //obracaj sie w poszukiwaniu mocnego zrodla
      motor_c_dir(fwd);
    }
  } }
```

9.3 PASCAL

Program wyjazd z tunelu

```
uses
  types, tm, conio, unistd, dsensor, dmotor, lcd, time,romlcd;
var
  temp:integer;
  tm1  :time_t;
  tm2  :time_t;
  czl1,czl4,czp1,czp4:integer;
  tm11:integer;
  lewo,prawo : integer;

begin
  lewo:=0;
  prawo:=0;
  repeat;
  motor_a_dir(mdFwd);
  motor_c_dir(mdFwd);
  tm1:=get_system_up_time div ticks_per_sec; //pobieramy czas systemowy
  if  (SENSOR_1 < $F000) then begin         //jesli czujnik lewy nacisniety
    tm2:=get_system_up_time div ticks_per_sec;
    lewo:=lewo+1;                           //zliczaj uderzenia
    if lewo = 1 then begin
      czl1:=tm2;                            //pobierz czas przy pierwszym uderzeniu
    end;
    if lewo = 4 then begin                  //pobierz czas przy czwartym uderzeniu
      czl4:=tm2;
      if (czl4-czl1)<8 then begin           //jesli od uderzenia pierwszego do czwartego nie minelo
        motor_a_dir(mdRev);                 //8sekund wykonaj duzy zwrot
        motor_c_dir(mdRev);
        delay(1000);
        motor_a_dir(mdFwd); motor_c_dir(mdRev);
        delay(500);
        lewo:=0;
      end;
    end
    else begin                              //jesli od uderzenia pierwszego do czwartego minelo 8
sekund
      motor_a_dir(mdRev);                   //wykonaj maly zwrot
      motor_c_dir(mdRev);
      delay(100);
      motor_a_dir(mdFwd); motor_c_dir(mdRev);
      delay(100);
      if (czl4-czl1)>=8 then begin
        lewo:=0;                            //po 8 sekundach zeruj licznik uderzen
      end ;
    end;
  end;
```

```
  if  (SENSOR_3 < $F000) then begin          //jesli czujnik lewy nacisniety
    tm2:=get_system_up_time div ticks_per_sec;
    prawo:=prawo+1;                          //zliczaj uderzenia
    if prawo = 1 then begin                  //pobierz czas przy pierwszym uderzeniu
      czp1:=tm2;
    end;
    if prawo = 4 then begin                  //pobierz czas przy czwartym uderzeniu
      czp4:=tm2;
      if (czp4-czp1)<8 then begin            //jesli od uderzenia pierwszego do czwartego nie minelo
        motor_a_dir(mdRev);                  //8sekund wykonaj duzy zwrot
        motor_c_dir(mdRev);
        delay(1000);
        motor_c_dir(mdFwd); motor_a_dir(mdRev);
        delay(500);
        lewo:=0;
      end;
    end
    else begin                               //jesli od uderzenia pierwszego do czwartego minelo 8
sekund
      motor_a_dir(mdRev);                    //wykonaj maly zwrot
      motor_c_dir(mdRev);
      delay(100);
      motor_c_dir(mdFwd); motor_a_dir(mdRev);
      delay(100);
      if (czp4-czp1)>=8 then begin           //po 8 sekundach zeruj licznik uderzen
        prawo:=0;
      end ;
    end;
 end;
 until temp >100;
end.
```

Program poszukiwacz latarni

```
uses
  types, conio, tm, unistd, dsensor, dmotor, romlcd;

{$DEFINE LIGHTSENS SENSOR_2}

procedure znalazl;
begin
  if LIGHT(LIGHTSENS)>80 then begin          //jesli prog swiatla przekroczony
    motor_a_dir(mdFwd);                      //poruszaj sie w kierunku zrodla
    motor_c_dir(mdFwd);
    delay(100);
   end;
end;

procedure nieznalazl;
begin
  if LIGHT(LIGHTSENS)<79 then begin          //jesli prog swiatla nie przekroczony
    motor_a_dir(mdFwd);                      //obracaj sie w poszukiwaniu latarni
    motor_c_dir(mdRev);
  end;
end;

begin
  ds_active(@LIGHTSENS);                     //ustawiamy wejscie nr 2 na aktywne
  repeat;
```

```
  nieznalazl;
  znalazl;
  until LIGHT(LIGHTSENS)= 200;
end.
```

9.4 MINDSCRIPT

Program wyjazd z tunelu

```
program tunel{
  timer t1
  timer t2
  var liczlewo =0
  var liczprawo = 0
  output prawy on 1                                  //motor 1 jest z prawej strony
  output lewy on 3                                   //motor 2 jest z lewej strony
  sensor czujkalewa on 1                             //czujkalewa po lewej
  sensor czujkaprawa on 3                            //czujkaprawa po prawej
 event nacisnietolewa when czujkalewa.pressed        //zdarzenie nacisnietolewa gdy czujnik
                                                     //lewy wcisniety
 event nacisnietoprawa when czujkaprawa.pressed      //zdarzenie nacisnietoprawa gdy czujnik
                                                     //prawy wcisniety

  main{
    fd [lewy prawy] on [lewy prawy]                  //jedz do przodu
    clear t1
    clear t2
    start hit                                        //start obserwatorow
    start hit1
  }

  watcher hit monitor nacisnietolewa{
    liczlewo  = liczlewo+1                           //zliczaj uderzenia
    if liczlewo =4 {                                 //jesli uderzy 4 razy i czas < 8sek
      if t1<80{                                      //wykonaj duzy zwrot
        sound 3
         bk [lewy prawy]
         on [lewy prawy] for 500
         dir prawy lewy //obrot w prawo
         on [lewy prawy] for 80
         fd[lewy prawy]
         on[lewy prawy]
         liczlewo =0
         clear t1
      }
      } else {                                       //jeśli czas >8sek wykonaj maly zwrot
        bk [lewy prawy]
         on [lewy prawy] for 30
         dir prawy lewy //obrot w prawo
         on [lewy prawy] for 30
         fd[lewy prawy]
         on[lewy prawy]
         clear sound
         if  t1>80  {
           clear t1                                  //zeruj timer oraz licznik uderzen
           liczlewo=0
         }
      }
```

```
    }
  watcher hit1 monitor nacisnietoprawa{
    liczprawo = liczprawo+1                  //zliczaj uderzenia
    display t2:0
    if liczprawo =4 {                        //jesli uderzy 4 razy i czas < 8sek
      if t2<80{
        sound 3
         bk [lewy prawy]                     //wykonaj duzy zwrot
         on [lewy prawy] for 500
         dir lewy prawy //obrot w lewo
         on [lewy prawy] for 80
         fd[lewy prawy]
         on[lewy prawy]
         liczprawo =0
         clear t2
      }
    } else {
      bk [lewy prawy]                        //jeśli czas >8sek wykonaj maly zwrot
      on [lewy prawy] for 30
      dir lewy prawy //obrot w lewo
      on [lewy prawy] for 30
      fd[lewy prawy]
      on[lewy prawy]
      clear sound
      if  t2>80  {
        clear t2                             //zeruj timer oraz licznik uderzen
         liczprawo=0
      }
    }
  }
}
```

Program poszukiwacz latarni

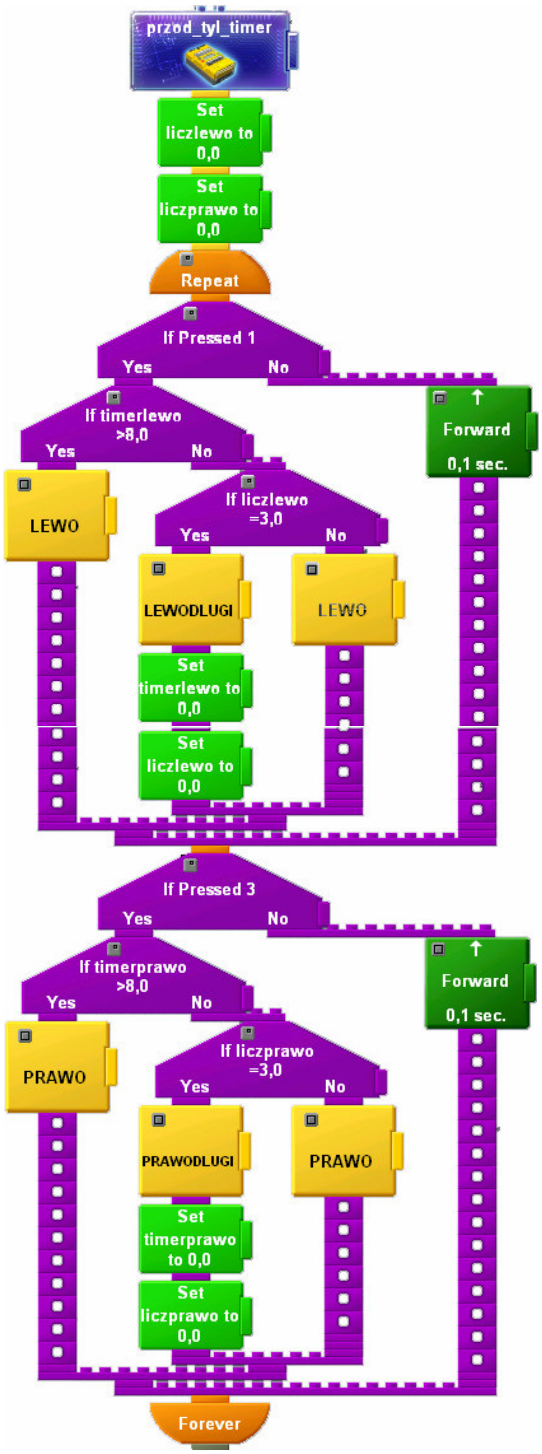
```
program szukaj latarni{
  output prawy on 1                          //motor 1 jest z prawej strony
  output lewy on 3                           //motor 2 jest z lewej strony
  sensor swiatelko on 2                      //czujnik swiatla na wejsciu 2
  swiatelko is light as percent              //czujnik zwraca wartosc w procentach
  event znalazl when swiatelko is 75..100    //jesli zwroci wiecej niz 75 wtedy znalazl
  event nieznalazl when swiatelko is 0..74   //jesli zwroci mniej niz 75 wtedy nie znalazl

  main{
    forever{
      start hit
      monitor nieznalazl {                   //jesli nie znalazl
        dir lewy prawy                       //niech się obraca
        on[lewy prawy]
      } abort on event                       //obracaj sie dopoki znajdziesz
    }
  }

  watcher hit monitor znalazl{
    dir [prawy ] []                          //jeśli znajdzie niech jedze do przodu
    fd [lewy prawy]
    on [lewy prawy]
  }
}
```

9.5 RCX Code

Program wyjazd z tunelu



Rys. 9.1 Program wyjazd z tunelu z rozwiniętymi blokami funkcyjnymi

Program poszukiwacz latarni

Rys. 9.2 Program poszukiwacz latarni
z rozwiniętymi blokami funkcyjnymi

Oba programy, napisane dla każdego języka, spełniają swoje zadanie w 100%. Sposobów napisania programów, spełniających wyżej wymienione zadania, jest wiele. Ze względu na konstrukcję składni języków, programy dla BrickOS oraz Pascal są budową do siebie zbliżone. W przypadku pozostałych języków zapis programu jest zupełnie inny, aczkolwiek wszystkie postawione zadania zostały wykonane.

9.6 Porównanie oraz wnioski

FUNKCJA	BrickOS	Pascal	NQC	Mindscript	RCX Code
Obsługa silników	✓	✓	✓	✓	✓
Obsługa czujników	✓	✓	✓	✓	✓
Obsługa LCD	✓	✓	✓ uproszczona	✓ uproszczona	✓ uproszczona
Obsługa brzęczyka	✓	✓	✓	✓	✓
Obsługa klawiatury	✓	✓	✗	✗	✗
Wielowątkowość	✓	✓	✗	✗	✗
Semafory	✓	✗	✗	✗	✗
Obsługa zdarzeń	✓	✓	✓	✓	✗
Generator liczb losowych	✓	✓	✓	✓	✓
Timery	✗	✗	✓	✓	✓
Obsługa czasu systemowego	✓	✓	✓	✓	✗
Obsługa czujnika baterii	✓	✓	✓	✓	✗

Najbardziej funkcjonalnym z dostępnych języków jest BrickOS. Jak wynika z powyższej tabeli spełnia on niemal wszystkie kryteria. W budowie przypomina język C, więc osoby znające ten język nie powinny mieć kłopotu z szybkim opanowaniem jego składni. BrickOS jest językiem polecanym do stosowania na zajęciach laboratoryjnych, jak i do prac badawczych ze względu na największą elastyczność i możliwości jakie oferuje programiście.

Pascal, jeśli chodzi o funkcjonalność, jest językiem podobnym do BrickOS. Ponieważ został on zaadaptowany do programowania RCX dość niedawno, a tym samym jest

jeszcze niedopracowany, można się więc spodziewać niewielkich kłopotów (zawieszanie się RCX) z mechanizmem wielowątkowości oraz obsługą zdarzeń.

Język Not Quite C jest językiem dobrym dla początkujących. Posiada prostą składnię (podobną do C) jednak nie oferuje aż tak zaawansowanych funkcji jak BrickOS lub Pascal. Do jego pracy nie jest wymagana wymiana firmware-u w komputerze RCX.

Mindscript jest dość skomplikowanym językiem. Mimo, iż jest oferowany jako dodatek przez firmę LEGO, dobre opanowanie zasad programowania w tym języku wymaga wielu ćwiczeń.

RCX Code standardowo dołączony do zestawu Mindstorms jest językiem najprostszym w obsłudze. Specjalny interfejs użytkownika oraz graficzna postać komend w znaczny sposób ułatwiają pracę. Mimo, iż język ten z pozoru może wydawać się „banalny", można w nim konstruować programy o dość złożonej budowie.

10 Projekty na zajęcia laboratoryjne

10.1 Gra „kółko i krzyżyk”

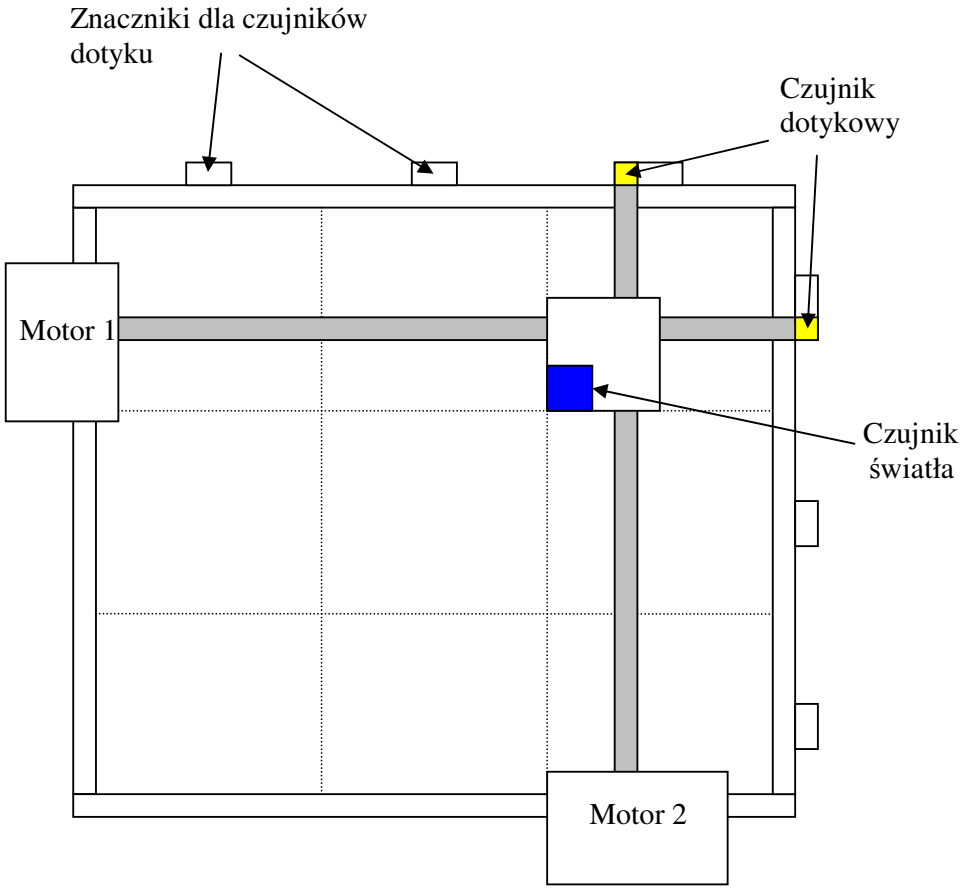
Cel projektu:

Zbudować robota do gry „kółko i krzyżyk”.

Fazy projektu:

1. Zaprojektować oraz zbudować odpowiedni model robota
2. Napisać program do obsługi robota
   a) mechanizm poruszania się po planszy
   b) algorytm do gry „kółko i krzyżyk”

Model robota:



Rys. 10.1 Przykładowy model robota do gry „kółko i krzyżyk”

Robot powinien swoim wyglądem przypominać ploter, tzn.: powinien posiadać dwie prowadnice (rys. 10.1). Jedna powinna poruszać się po osi X a druga po osi Y. Każda z prowadnic powinna być napędzana jednym silnikiem. Na przecięciu się prowadnic powinien znajdować się czujnik światła. Obszar, nad którym poruszają się prowadnice, powinien być podzielony na 9 części (pól). Aby RCX wiedział, nad którym polem znajduje się punkt przecięcia prowadnic, należy użyć czujników dotyku.

Przebieg gry:

Zamiast stawiania symboli O oraz X, należy używać znaczników np.: kwadracików o kolorach białym (puste pole), czarnym (robot), zielonym (gracz). Robot skanuje po kolei wszystkie pola - po kolei zatrzymuje się nad każdym z nich oraz mierzy natężenie światła (rozpoznaje czy pole należy do niego, przeciwnika czy jest puste). Po zapamiętaniu stanu wszystkich komórek, naprowadza czujnik światła nad pole, na którym chce umieścić swój znacznik oraz wydaje sygnał dźwiękowy (człowiek umieszcza znacznik robota na wskazanym przez niego polu). Gdy znacznik robota zostanie wstawiony, czas na ruch gracza. Należy wybrać odpowiednie pole, wstawić swój znacznik oraz nacisnąć przycisk np.: View dając tym samym do zrozumienia robotowi, iż gracz postawił swój symbol. Robot znów skanuje planszę i wskazuje miejsce dla nowego znacznika.

Przykładowy algorytm:

Gracz rozpoczyna grę	**Robot rozpoczyna grę**
1. Gracz wstawia swój znacznik, naciska przycisk View. 2. Robot skanuje planszę. 3. Według odpowiedniego algorytmu wskazuje miejsce na swój znacznik oraz wydaje sygnał dźwiękowy. 4. Czynności od 1 do 3 powtarzane aż do wypełnienia wszystkich pól lub stwierdzenia wygranej. 5. Po wypełnieniu wszystkich wolnych pól, robot powinien zeskanować planszę w celu stwierdzenia remisu lub wygranej jednej ze stron.	1. Robot wskazuje miejsce na swój znacznik oraz wydaje sygnał dźwiękowy. 2. Gracz wstawia swój znacznik, naciska przycisk View. 3. Robot skanuje planszę. 4. Czynności od 1 do 3 powtarzane aż do wypełnienia wszystkich pól lub stwierdzenia wygranej. 5. Po wypełnieniu wszystkich wolnych pól, robot powinien zeskanować planszę w celu stwierdzenia remisu lub wygranej jednej ze stron.

Możliwe udoskonalenia:

- Robot może skanować pola ruchem płynnym, tzn. nie musi się zatrzymywać nad każdym segmentem
- Można dobudować do modelu kontener z klockami w trzech kolorach, aby robot sam pobierał i wstawiał klocki na odpowiednie pola (wymaga trzeciego silnika)
- Zamiast czujników dotyku można zastosować czujniki obrotów.
- Do podnoszenia i upuszczania znaczników można użyć elementów pneumatycznych.
- Robot może zapamiętywać poprzednie ruchy i skanować tylko pola, które przed ruchem gracza były wolne

10.2 Robot „dostawca”

Cel projektu:

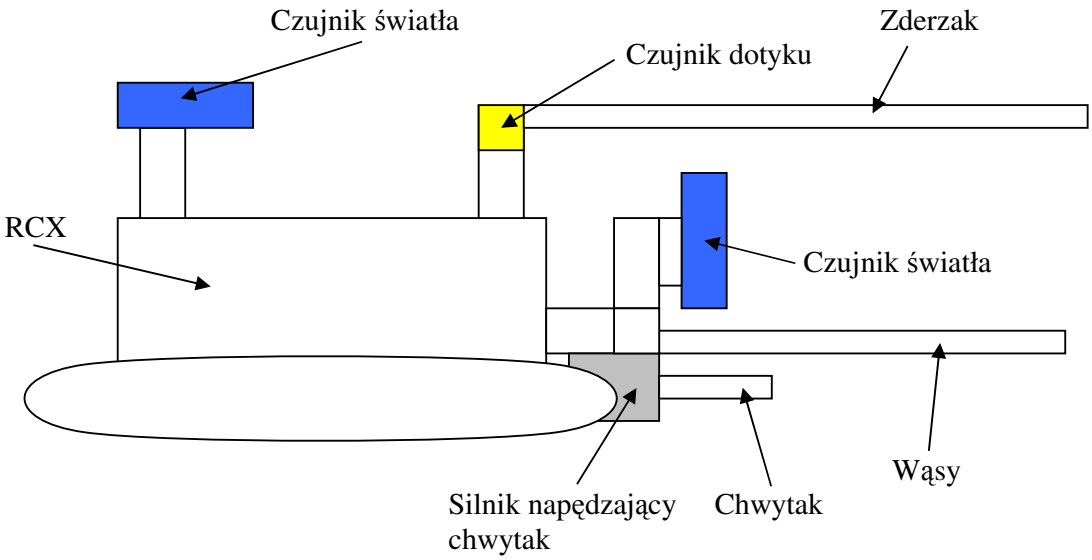
Zbudować robota, który zbierze porozrzucane klocki i dostarczy je do wyznaczonego miejsca.

Fazy projektu:

1. Zaprojektować oraz zbudować odpowiedni model robota
2. Napisać program do obsługi robota

Model robota:

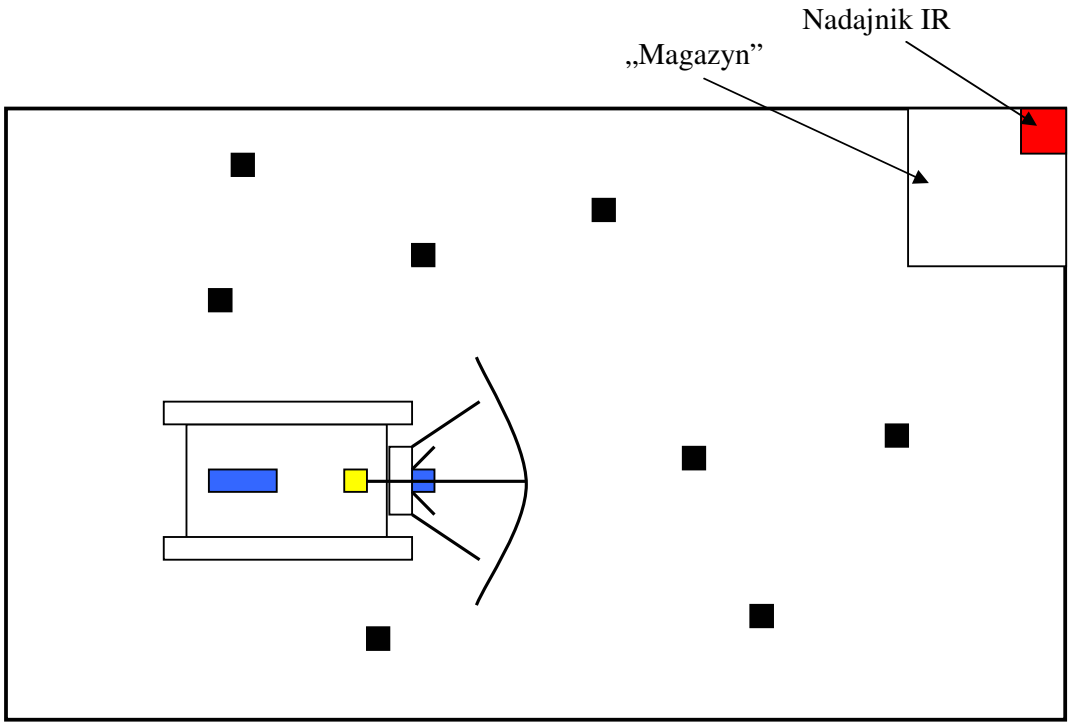
Robot powinien być pojazdem osadzonym na podwoziu gąsienicowym lub kołowym. Z przodu robota powinien znajdować się mechanizm złożony z czujnika światła, czujnika dotyku oraz z „wąsów” zagarniających klocki (rys. 10.2). „Wąsy” powinny zagarnąć klocek pod czujnik światła, który będzie wykrywać jego obecność w zasięgu ramienia chwytającego. Nad „wąsami” powinien znajdować się mechanizm wykrywający ściany ograniczające pole gry. Drugi czujnik światła powinien znajdować się na robocie. Jego zadaniem jest wykrywanie latarni IR, określającej miejsce docelowe dla transportowanych klocków.



Rys. 10.2 Robot dostawca – widok z boku

Przebieg gry:

Celem gry jest dostarczenie w ciągu ustalonego czasu jak największej ilości jednokolorowych klocków do wyznaczonego obszaru (magazynu). Miejsce magazynu wyznacza latarnia IR. Robot wyrusza z magazynu i szuka klocków. Jeśli robot zderzy się ze ścianą, robot wycofa się i obróci o kąt ustalony w programie. Gdy „wąsy" zagarną klocek pod czujnik światła, należy zamknąć chwytak. W tym momencie należy uruchomić drugi czujnik światła znajdujący się na robocie, który ma odnaleźć magazyn. Gdy robot dojedzie do magazynu, zderzenie ze ścianą poinformuje o dotarciu na miejsce przeznaczenia, chwytak zwolni klocek, robot wycofa się, obróci i rozpocznie dalsze poszukiwania.



Rys. 10.3 Model projektu „Robot dostawca”

Możliwe udoskonalenia:

- Rozróżnianie kolorów – zbieranie tylko klocków o jednym kolorze
- Jednoczesne cofanie i zawracanie robota
- Zastosowanie elementów pneumatycznych do chwytania klocka.

Spis ilustracji

Rys. 2.1 Sterownik PB 120

Rys. 2.2 Sterownik RCX wersja 1.0

Rys. 2.3 Sterownik RCX wersja 2.0

Rys. 2.4 Poziomy wypełnienia PWM

Rys. 2.5 Części składowe silnika nr 43362

Rys. 2.6 Cewki silnika nr 43362

Rys. 2.7 Silnik nr 43362 bez obudowy

Rys. 2.8 Czujnik dotykowy

Rys. 2.9 Multiplekser

Rys. 2.10 Czujnik światła

Rys. 2.11 Czujnik obrotów

Rys. 2.12 Schemat podawanych napięć podczas jednego obrotu

Rys. 2.13 Czujnik temperatury

Rys. 2.14 Schematy tras

Rys. 2.15 Pozycja startowa robota

Rys. 2.16 Przykładowe trasy, po których poruszał się robot

Rys. 2.17 Czujniki ruchu – widok z góry.

Rys. 2.18 Czujniki ruchu – widok od spodu

Rys. 2.19 Czujnik zamontowany na robocie.

Rys. 2.20 Miejsce do montażu czujnika.

Rys. 2.21 Robot z zamontowanymi czujnikami gotowy do jazdy.

Rys. 2.13 Czujniki ruchu – widok z góry

Rys. 2.14 Czujniki ruchu – widok od spodu

Rys. 2.16 Miejsce do montażu czujnika

Rys. 2.15 Czujnik zamontowany na robocie.

Rys. 2.17 Robot z zamontowanymi czujnikami gotowy do jazdy.

Rys. 5.1 Wszystkie segmenty wyświetlacza

Rys. 5.2 Nazwy poszczególnych segmentów wyświetlacza LCD

Rys. 5.3 Kody przycisków (dkey.h)

Rys. 5.4 Kody przycisków (dbutton.h)

Rys. 6.1 Kody przycisków (dkey.h)

Rys. 6.2 Kody przycisków (dbutton.h)

Rys. 8.1 Ekran powitalny RIS 2.0

Rys. 8.2 Podstawowe opcje konfiguracyjne.

Rys. 8.3 Zaawansowane opcje konfiguracyjne.

Rys. 9.1 Program wyjazd z tunelu z rozwiniętymi blokami funkcyjnymi

Rys. 9.2 Program poszukiwacz latarni z rozwiniętymi blokami funkcyjnymi

Rys. 10.1 Przykładowy model robota do gry „kółko i krzyżyk”

Rys. 10.2 Robot dostawca – widok z boku

Rys. 10.3 Model projektu „Robot dostawca”

Bibliografia

[1] http://home.elka.pw.edu.pl/~mbrudka/ERO/index.html

[2] http://bricxcc.sourceforge.net – strona domowa Bricx Command Center

[3] http://legos.sourceforge.net/docs/CommandRef.htm - spis komend dla LegOS ver. 0.2.4

[4] http://users.ncable.net.au/~blane/smartParts/index.htm - strona o języku mindscript oraz NQC

[5] http://www.crynwr.com/lego-robotics/ - ogólna strona o budowie oraz o programowaniu

[6] http://www.philohome.com/motors/motorcomp.htm - porównanie silników

[7] http://www.plazaearth.com/usr/gasperi/lego.htm - RCX sensor input page

[8] http://www.philohome.com/motors.htm - motor 43362 internals

[9] http://www.lysator.liu.se/c/bwk-tutor.html#working-c – kurs programowania w C

[10] http://www.strath.ac.uk/IT/Docs/Ccourse/ - kurs programowania w C

[11] http://web.mit.edu/taoyue/www/tutorials/pascal/contents.html - kurs programowania w PASCAL

[12] http://www.plazaearth.com/usr/gasperi/3rmux.htm - multiplekser dla czujników dotyku

[13] http://mindstorms.lego.com/sdk2/LEGOMindStormsSDK.zip - dokumentacja pakietu LEGO Mindstorms SDK

[14] http://bricxcc.sourceforge.net/nqc/index.html - NQC FAQ

[15] http://lcs.www.media.mit.edu/groups/el/projects/programmable-brick/more.html - opis PB120

[16] Ferrari M., Ferrari G., Hempel R.: Building robots with Lego Mindstorms, Syngress 2001