Пособие по программированию в среде NXC.

Not eXactly C.

Учимся программировать LEGO роботов с помощью языка NXC.

2013

ДТДиМ. Конструктив ™.

13.09.2013

# Оглавление

[Первая программа 5](#_Toc366859953)

[Постройка робота 5](#_Toc366859954)

[Запуск Bricx Command Center 5](#_Toc366859955)

[Написание программы 6](#_Toc366859956)

[Запуск программы 7](#_Toc366859957)

[Ошибки в программе 8](#_Toc366859958)

[Изменяем скорость 9](#_Toc366859959)

[Подводим итоги 9](#_Toc366859960)

[Более интересная программа 10](#_Toc366859961)

[Учимся делать повороты 10](#_Toc366859962)

[Повторяем команды 10](#_Toc366859963)

[Добавляем комментарии 11](#_Toc366859964)

[Подводим итоги 12](#_Toc366859965)

[Использование переменных 12](#_Toc366859966)

[Движение по спирали 12](#_Toc366859967)

[Случайные числа 14](#_Toc366859968)

[Подводим итоги 15](#_Toc366859969)

[Управляющие структуры 15](#_Toc366859970)

[Оператор "if" 15](#_Toc366859971)

[Оператор "do" 16](#_Toc366859972)

[Подводим итоги 17](#_Toc366859973)

[Сенсоры 17](#_Toc366859974)

[Ждём информацию с сенсора 18](#_Toc366859975)

[Действия при срабатывании сенсора касания 18](#_Toc366859976)

[Сенсор освещенности 19](#_Toc366859977)

[Звуковой сенсор 20](#_Toc366859978)

[Ультразвуковой дальномер 21](#_Toc366859979)

[Подводим итоги 21](#_Toc366859980)

[Задачи и функции 22](#_Toc366859981)

[Задачи 22](#_Toc366859982)

[Функции 23](#_Toc366859983)

[Определяем макрос 25](#_Toc366859984)

[Подводим итоги 26](#_Toc366859985)

[Создаём музыку 26](#_Toc366859986)

[Проигрываем звуковые файлы 26](#_Toc366859987)

[Играем музыку 27](#_Toc366859988)

[Подводим итоги 28](#_Toc366859989)

[Еще раз о моторах 28](#_Toc366859990)

[Плавная остановка 29](#_Toc366859991)

[Дополнительные команды 29](#_Toc366859992)

[ПИД-управление 31](#_Toc366859993)

[Подводим итоги 32](#_Toc366859994)

[Дополнительная информация о сенсорах 32](#_Toc366859995)

[Режимы и типы сенсоров 33](#_Toc366859996)

[Сенсор вращения 34](#_Toc366859997)

[Подключение нескольких сенсоров к одному входу 35](#_Toc366859998)

[Подводим итоги 37](#_Toc366859999)

[Параллельные задачи 37](#_Toc366860000)

[Неправильная программа 37](#_Toc366860001)

[Критические секции и "мьютекс"-переменные 38](#_Toc366860002)

[Использование семафоров 39](#_Toc366860003)

[Подводим итоги 40](#_Toc366860004)

[Коммуникации между роботами 40](#_Toc366860005)

[Отправка сообщений мастер-подчиненный 41](#_Toc366860006)

[Отправка чисел с подтверждением 42](#_Toc366860007)

[Прямые команды 44](#_Toc366860008)

[Подводим итоги 44](#_Toc366860009)

[Дополнительные возможности 44](#_Toc366860010)

[Таймеры 45](#_Toc366860011)

[Дисплей 45](#_Toc366860012)

[Файловая система 46](#_Toc366860013)

[Подводим итоги 50](#_Toc366860014)

[Как определить цвет, когда датчик "врет"? 50](#_Toc366860015)

[NXC: кнопочная интерактивность 51](#_Toc366860016)

[NXC: работаем с датчиками вращения мотора 52](#_Toc366860017)

[NXC: работаем с цветовым сенсором. 54](#_Toc366860018)

[NXC: датчики, енкодеры, кнопки - взаимодействуем с внешним миром 55](#_Toc366860019)

[Пример 1. Датчик расстояния 55](#_Toc366860020)

[Пример 2. Измерение отраженного света. 55](#_Toc366860021)

[Пример 3. Измерение окружающего света 56](#_Toc366860022)

[Пример 4. Использование необработанных данных 56](#_Toc366860023)

[Пример 5. Датчик вращения двигателя (енкодер) 57](#_Toc366860024)

[Пример 6. Скорость опроса датчика расстояния 58](#_Toc366860025)

[Пример 7. Определение цвета 58](#_Toc366860026)

[Пример 8. Опрос датчика цвета в режиме датчика освещенности 59](#_Toc366860027)

[NXC: использование циклов 59](#_Toc366860028)

[Пример 2. while() в одну строчку 60](#_Toc366860029)

[Пример 3. until() 60](#_Toc366860030)

[Пример 4. Скорость исполнения цикла. 61](#_Toc366860031)

[NXC: повороты - практика программирования моторов 61](#_Toc366860032)

[Пример 1. Поворот одним двигателем 62](#_Toc366860033)

[Пример 2. Поворот двумя двигателями 62](#_Toc366860034)

[Пример 3. Распределение мощности при управлении двумя двигателями 63](#_Toc366860035)

[Пример 4. Альтернативный поворот двумя двигателями 64](#_Toc366860036)

[Сортировка данных на NXT 64](#_Toc366860037)

[Пузырьковая сортировка 65](#_Toc366860038)

[Сортировка вставками 66](#_Toc366860039)

[Сортировка Шелла 67](#_Toc366860040)

# 

# Первая программа

В этой главе я покажу, как написать очень простую программу. Мы запрограммируем робота на движение 4 секунды вперед, 4 - назад и остановку в конце. Не очень впечатляюще, но на этом примере вы сможете получить представления об простейших способах программирования. Кроме того, вы сможете понять, насколько это просто. Но, прежде чем писать программу, нам нужно собрать робота.

## Постройка робота

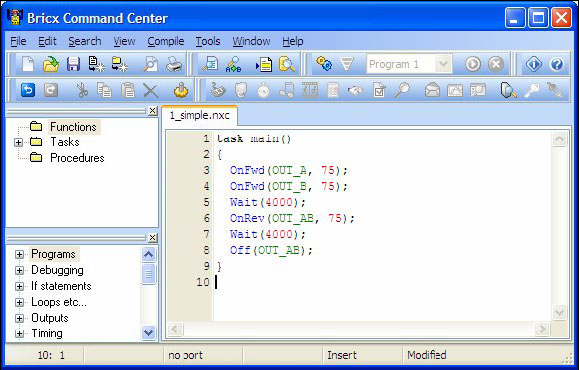
Робот, которого мы будем использовать в этом курсе - Tribot - первый колёсный робот, которого инструкция к NXT советует собрать после покупки набора. Единственное отличие будет заключаться в том, что вы должны подключить правый мотор в порт A, левый мотор в порт C, а мотор захвата в порт B.

[](http://roboforum.ru/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Lego_NXT_Tribot.jpg)

Убедитесь, что вы правильно установили драйверы Mindstorms NXT Fantom Drivers, поставляемые с набором.

## Запуск Bricx Command Center

Мы будем писать программы используя Bricx Command Center. Запустите его двойным щелчком по иконке BricxCC (я предполагаю, что вы уже установили BricxCC. Если нет, загрузите его с сайта (см. введение) и установите его в любой удобный вам каталог). Программа попросит предоставить ей возможность подключиться к роботу. Включите робота и нажмите "ОК". Программа (скорее всего) автоматически найдёт робота. Ниже приведено избражение интерфейса пользователя, появляющееся после успешного завершения синхронизации (закладки с текстовым окном быть не должно).

[](http://roboforum.ru/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:BricxCCScreenShot1.jpg)

Интерфейс выглядит как стандартный текстовый редактор с обычным меню, кнопками для открытия и сохранения файлов, печати, редактирования и т.п. Однако среди них есть несколько специальных меню для компилирования программ и загрузки их в робота, а также для получения отладочной информации; пока вы можете не обращать на них внимание. Сейчас мы хотим написать новую программу, так что нажимайте кнопку "New File" для создания нового окна с текстовым редактором.

## Написание программы

Теперь введите текст программы:

task main()

{

OnFwd(OUT\_A, 75);

OnFwd(OUT\_C, 75);

Wait(4000);

OnRev(OUT\_AC, 75);

Wait(4000);

Off(OUT\_AC);

}

На первый взгляд выглядит немного сложно, так что давайте разберемся в ней.

Программы NXC состоят из задач ("task"). Наша программа содержит всего одну задачу с именем "main". Любая программа на языке NXC должна содержать задачу с именем "main", которая и будет исполняться роботом (о задачах мы поговорим подробнее в главе 6). Задача состоит из набора команд - операторов. Операторы заключены в фигурные скобки для того, чтобы была ясна их принадлежность к конкретной задаче. Каждый оператор заканчивается точкой с запятой. Это позволяет понять, где кончается один оператор и начинается следующий. В целом задача обычно выглядит следующим образом:

task main()

{

statement1;

statement2;

…

}

Наша программа содержит 6 операторов. Давайте разберем их по очереди:

OnFwd(OUT\_A, 75);

Этот оператор говорит роботу включить выход "A" модуля NXT. Это означает, что мотор, подключенный нами ранее, будет двигать модель вперед. Число, следующее за указанием порта, устанавливает скорость мотора в 75% от максимальной.

OnFwd(OUT\_C, 75);

Тот же оператор, но теперь включаем мотор "C". После выполнения этих операторов оба мотора вращаются, и робот едет вперёд.

Wait(4000);

Теперь время немного подождать. Этот оператор говорит нам подождать 4 секунды. Аргумент, которым является число в скобках, указывается в 1/1000 секунды, так что мы можем с большой точностью указать программе необходимую задержку. Следующие 4 секунды программа будет ждать, а робот с включенными двигателями ехать вперед.

OnRev(OUT\_AC, 75);

Робот проехал вперед достаточно далеко, так что мы скажем ему ехать в обратном направлении, т.е. назад. Обратите внимание, что мы можем установить режим работы обоих моторов сразу используя аргумент OUT\_AC; таким же образом можно заменить первые два оператора задачи.

Wait(4000);

Снова ждём 4 секунды.

Off(OUT\_AC);

И наконец, выключаем оба мотора.

Вот и вся программа. Она включает на 4 секунды оба мотора на движение вперед, затем назад и в конце выключает их.

Вы, наверное, заметили при написании программы автоматическое выделение фрагментов кода разными цветами. Эти цвета и стили используемые текстовым редактором можно настроить, если они вам кажутся неудобными.

## Запуск программы

После написания программы нам потребуется её скомпилировать (превратить её в двоичные коды, которые робот может разобрать и выполнить) и переслать полученные двоичные коды в робота, используя USB-кабель или Bluetooth-донгл (сокращенно: загрузить программу в робота).

[Изображение:BricxCCScreenShot2.jpg](http://roboforum.ru/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:BricxCCScreenShot2.jpg)

На этой панели расположены кнопки, которые позволяют (слева направо) компилировать программу, загружать её в робота, запускать выполнение программы на роботе и останавливать его.

Нажмите вторую кнопку, и, если вы не сделали ошибок при написании программы, она корректно скомпилируется и загрузится в робота (если вы допустили ошибки, об этом будет выдано сообщение, см. более подробно ниже).

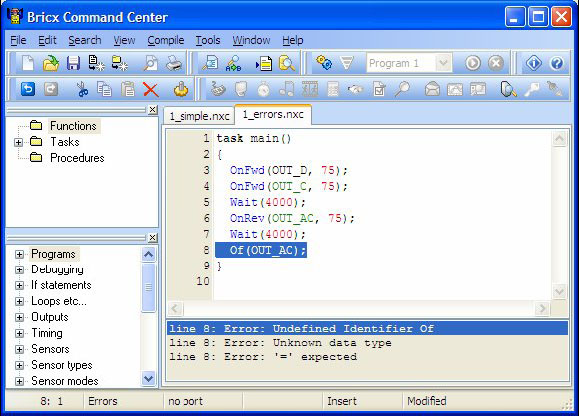
Теперь мы можем запустить нашу программу на выполнение. Чтобы сделать это зайдите в меню "My Files" на модуле, "Software files", и запустите ("run") программу с названием "1\_simple". Помните: файлы программ в файловой системе NXT имеют такие же имена, как и исходные NXC-файлы.

Кроме того, ваша программа может запускаться автоматически после загрузки. Используйте комбинацию клавиш CTRL+F5 или после загрузки программы на робота нажмите зелёную кнопку запуска программы.

Делает ли робот то, что мы запрограммировали? Если нет, проверьте соединения проводов.

## Ошибки в программе

Когда вы набираете программы, существует вероятность допустить ошибки. Компилятор сообщает обо всех обнаруженных им ошибках внизу окна, в котором вы писали программу, как это показано на рисунке:

[](http://roboforum.ru/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:BricxCCScreenShot3.jpg)

Он автоматически выделяет первую ошибку (неправильное написание наименования мотора). Когда у вас выдалось несколько ошибок, вы можете щёлкнуть на интересующем вас сообщении об ошибке и перейти к соответствующему месту программы. Обратите внимание, что часто ошибки в начале программы могут повлечь ошибки в других местах. Так что лучше исправлять только первые несколько ошибок, после чего скомпилировать программу еще раз. Так же заметим: подсветка синтаксиса помогает избежать ошибок при разработке. Например, в последней строке мы написали Of вместо Off. Так как это неизвестная команда, она не подсвечена.

Могут быть ошибки, которые не обнаруживаются компилятором. Если мы указали OUT\_B в управлении мотором, то будет вращаться не тот двигатель, который нам нужен. Если ваш робот ведёт себя не так, как вы запланировали, скорее всего вы что-то неправильно написали в управляющей им программе.

## Изменяем скорость

Как вы заметили, робот перемещается достаточно быстро. Чтобы изменить его скорость, вам потребуется изменить второй параметр внутри круглых скобок. Скорость вращения моторов задаётся числом между 0 и 100: 100 - это самая большая скорость, 0 - полная остановка (сервопривод NXT будет удерживать положение). Вот новая версия нашей программы, под управлением которой робот будет перемещаться медленно:

task main()

{

OnFwd(OUT\_AC, 30);

Wait(4000);

OnRev(OUT\_AC, 30);

Wait(4000);

Off(OUT\_AC);

}

## Подводим итоги

В этой главе вы написали свою первую программу на языке NXC, используя среду разработки BricxCC. Теперь вы умеете набирать программу, загружать её в робота и запускать её выполнение. BricxCC может делать многие другие вещи, чтобы получить о них больше информации, прочитайте документацию поставляемую с этим пакетом. Этот обучающий курс в первую очередь направлен на освоение языка NXC и затрагивает только те функции BricxCC, без которых мы не можем обойтись.

Также вы освоили некоторые важные аспекты языка NXC. Прежде всего вы узнали, что каждая программа имеет задачу с именем "main", которая будет выполняться роботом. Также вы изучили 3 простые команды для управления двигателями: OnFwd(), OnRev() и Off(). И наконец, вы узнали об операторе Wait().

# Более интересная программа

Наша первая программа была не сильно впечатляющая. Так что давайте попробуем сделать её более интересной. Мы будем делать это в несколько шагов, показывая некоторые важные возможности изучаемого языка программирования NXC.

## Учимся делать повороты

Вы можете заставить робота поворачиваться путём остановки или включении в обратном направлении одного из двух моторов. Вот пример того, как это можно сделать. Введите его, загрузите полученную программу в робота и запустите. Робот должен проехать немного вперед, после чего повернуться на 90 градусов вправо.

task main()

{

OnFwd(OUT\_AC, 75);

Wait(500);

OnRev(OUT\_C, 75);

Wait(360);

Off(OUT\_AC);

}

Вам может потребоваться попробовать немного другие числа, вместо 360 указанного во втором операторе Wait(), чтобы сделать 90 градусный поворот. Это зависит от типа поверхности по которой перемещается робот. Чтобы не отыскивать в больших программах такие числа, которые могут потребовать настройки существует возможность дать этому числу имя и определить его в начале программы. В языке NXC вы можете определять постоянные значения (константы) так, как показано в следующей программе:

#define MOVE\_TIME 500

#define TURN\_TIME 360

task main()

{

OnFwd(OUT\_AC, 75);

Wait(MOVE\_TIME);

OnRev(OUT\_C, 75);

Wait(TURN\_TIME);

Off(OUT\_AC);

}

Первые две строки определяют две константы. Теперь их можно использовать везде в программе вместо этих чисел. Использование констант хорошо по двум причинам: оно делает программы более читаемыми и более удобно менять значения констант, чем искать во всей программе где еще у вас время поворота было прописано как 360 тысячных секунды. Обратите внимание, что BricxCC обозначает операторы define отдельным цветом. Как мы увидим в главе 6, вы можете подобным образом определять не только константы.

## Повторяем команды

Теперь давайте попробуем написать программу, которая заставить робота проехать по квадрату. Проехать по квадрату означает: проехать вперед, повернуться на 90 градусов, снова проехать вперед, снова повернуть на 90 градусов и так далее. Мы можем просто написать 4 раза кусок указанного выше кода, но значительно проще это можно сделать с использованием оператора цикла (repeat).

#define MOVE\_TIME 500

#define TURN\_TIME 360

task main()

{

repeat(4)

{

OnFwd(OUT\_AC, 75);

Wait(MOVE\_TIME);

OnRev(OUT\_C, 75);

Wait(TURN\_TIME);

}

Off(OUT\_AC);

}

Число внутри скобок оператора цикла указывает сколько раз повторить операторы в следующих за оператором фигурных скобках. Обратите внимание, что в приведенной выше программе мы сместили операторы внутри цикла вправо. Это не является обязательным, но делает программу более легко читаемой.

В качестве окончательного примера давайте заставим робота 10 раз проехать по квадрату:

#define MOVE\_TIME 500

#define TURN\_TIME 360

task main()

{

repeat(10)

{

repeat(4)

{

OnFwd(OUT\_AC, 75);

Wait(MOVE\_TIME);

OnRev(OUT\_C, 75);

Wait(TURN\_TIME);

}

}

Off(OUT\_AC);

}

Теперь у нас один оператор цикла внутри другого. Это называется “вложенный” оператор цикла. Вы можете делать операторы цикла любого уровня вложенности. Внимательно посмотрите на фигурные скобки и отступы, использованные в программе. Задача начинается с первой фигурной скобки и заканчивается последней. Первый оператор цикла начинается содержит операторы внутри второй и пятой фигурных скобок. Вложенный в него оператор цикла содержит операторы между третьей и четвертой фигурной скобкой. Как вы видите фигурные скобки всегда идут парами и кусок программы между ними выделяется отступом слева.

## Добавляем комментарии

Чтобы сделать вашу программу еще более читаемой, неплохо бы добавить в неё комментарии. Когда мы пишем в строке // всё что после этих символов компилятор будет считать комментарием и проигнорирует. Длинный комментарий на несколько строк можно размещать между символами /\* и \*/. BricxCC выделяет комментарии отдельным цветом в текстовом редакторе. Полный код программы с комментариями выглядит так:

/\* 10 SQUARES

This program make the robot run 10 squares

\*/

#define MOVE\_TIME 500 // Time for a straight move

#define TURN\_TIME 360 // Time for turning 90 degrees

task main()

{

repeat(10) // Make 10 squares

{

repeat(4)

{

OnFwd(OUT\_AC, 75);

Wait(MOVE\_TIME);

OnRev(OUT\_C, 75);

Wait(TURN\_TIME);

}

}

Off(OUT\_AC); // Now turn the motors off

}

## Подводим итоги

В этой главе вы освоили использование операторов цикла и использование комментариев. Так же вы увидели зачем используются вложенные фигурные скобки и отступы. С имеющимися на этом этапе знаниями вы уже можете писать программы под управлением которых робот будет двигаться по практически каким угодно траекториям. Рекомендуется перед переходом к следующей главе попробовать самостоятельно запрограммировать робота для движения по разным интересным траекториям. Для этого можете попробовать варьировать программы из этой или предыдущей главы.

# Использование переменных

Переменные составляют важную часть любого языка программирования. Переменные это ячейки памяти, в которых мы можем хранить какие-либо значения. Мы можем использовать эти значения в различных местах программы и мы можем изменять эти значения. Давайте разберем использование переменных на следующем примере.

## Движение по спирали

Предположим, что мы хотим адаптировать приведенную выше программу таким образом, чтобы робот двигался по спирали. Это можно сделать путём увеличения времени которое робот движется прямо после каждого такого движения. То есть мы хотим после каждого такого движения увеличивать значение константы MOVE\_TIME. Но как же мы сделаем это? Ведь MOVE\_TIME это константа и поэтому не может быть изменена при выполнении программы. Поэтому нам потребуется заменить в нашей программе константу MOVE\_TIME на переменную move\_time. Как видно из программы, представленной ниже и реализующей движение по спирали, в языке NXC работа с переменными организована очень просто.

#define TURN\_TIME 360

int move\_time; // define a variable

task main()

{

move\_time = 200; // set the initial value

repeat(50)

{

OnFwd(OUT\_AC, 75);

Wait(move\_time); // use the variable for sleeping

OnRev(OUT\_C, 75);

Wait(TURN\_TIME);

move\_time += 200; // increase the variable

}

Off(OUT\_AC);

}

Наиболее важные строки программы снабжены комментариями. Сначала мы объявляем переменную указывая ключевое слово "int" и следом имя переменной, которое мы выбрали. (Обычно принято использовать нижний регистр для имён переменных и верхний регистр для констант, но это не принципиально для компилятора.) Имя переменной должно начинаться с буквы, но может содержать в себе цифры и символ подчеркивания. Другие символы в именах переменных запрещены. (То же самое касается названий констант, задач и т.д.) Ключевое слово "int" означает "integer", то есть целое число. Только целые числа могут храниться в такой переменной.

Во второй строке мы присваиваем значение 200 нашей только что объявленной переменной. С этого момента при использовании переменной её значение будет 200. Теперь последуем внутрь цикла, в котором мы используем переменную для определения того, сколько времени двигаться вперед, и в конце цикла увеличим значение переменной на 200. Таким образом первый раз робот будет ехать прямо 200/1000=0.2 секунды, второй раз 0.4 секунды, третий раз 0.6 секунды и так далее.

Кроме добавления числа к переменной мы можем умножать её на число используя \*=, вычитать из неё число используя -= и делить на число используя /=. (Обратите внимание, что для деления результат будет округлён до ближайшего целого.) Вы можете также добавлять одни переменные к другим и вообще записывать более сложные выражения. Следующий пример программы не будет никаким образом проявлять себя через исполнительные устройства робота, так как мы до сих пор не научились использовать дисплей NXT!

int aaa;

int bbb,ccc;

int values[];

task main()

{

aaa = 10;

bbb = 20 \* 5;

ccc = bbb;

ccc /= aaa;

ccc -= 5;

aaa = 10 \* (ccc + 3); // aaa is now equal to 80

ArrayInit(values, 0, 10); // allocate 10 elements = 0

values[0] = aaa;

values[1] = bbb;

values[2] = aaa\*bbb;

values[3] = ccc;

}

Заметим, что во второй строке мы объявили сразу несколько переменных за один раз. Мы можем таким образом объявить все переменные из первой и второй строки в одну строку. Переменная названная "values" является массивом, то есть, эта переменная может содержать больше чем просто число: элементы массива, к которым можно обращаться указывая число в квадратных скобках за переменной-массивом. В языке NXC массивы целых чисел объявляются так:

int name[];

Еще один интересный для нас оператор в приведённой программе - инициализация массива, который определяет, что в массиве values будет 10 элементов и инициализирует их все значением 0:

ArrayInit(values, 0, 10);

## Случайные числа

Во всех приведенных ранее программах мы строго определяли что должен делать. Но может быть интересно создать ситуацию, в которой мы не будем знать что именно будет делать робот. Пусть мы хотим внести некую непредсказуемость в его движения. Для таких ситуаций в языке NXC существует возможность получения случайных чисел. Следующая программа показывает как сделать, чтобы робот перемещался случайным образом. Она без конца движется вперед случайное количество времени и потом поворачивается на случайный угол.

int move\_time, turn\_time;

task main()

{

while(true)

{

move\_time = Random(600);

turn\_time = Random(400);

OnFwd(OUT\_AC, 75);

Wait(move\_time);

OnRev(OUT\_A, 75);

Wait(turn\_time);

}

}

Программа объявляет две переменные, а затем присваивает им случайные значения. Random(600) означает случайное число между 0 и 600 (при этом верхняя граница диапазона не включена во множество возможных ответов, т.е. вернётся число из диапазона 0..599). При каждом вызове функции Random получаемые числа будут отличаться. Обратите внимание, что мы могли избежать использования переменных и напрямую использовать полученное случайное число, написав оператор Wait(Random(600)).

Вы так же можете увидеть в этой программе новый тип цикла. Вместо использования цикла repeat мы использовали цикл while(true). Оператор цикла while повторяет включенные в него операторы до тех пор, пока условие заключенное в круглые скобки остаётся истинным. Специальное ключевое слово "true" всегда истинно, так что операторы этого цикла будут повторяться бесконечно (ну или по крайней мере до того момента, когда вы нажмёте тёмно-серую кнопку на вашем NXT). Больше информации об операторе while вы узнаете в главе 4.

## Подводим итоги

В этой главе вы научились использованию переменных и массивов. Вы можете объявлять переменные других типов данных, отличных от "int": это могут быть "short", "long", "byte", "bool" и "string".

Типы данных

Int – целочисленный тип данных диапазон значений: 0 … 65 535

Short, int - целочисленный тип данных диапазон значений: –32 768 … 32 767

Char – целочисленный тип данных, диапазон значений: –128 … 127

Byte – целочисленный тип данных, диапазон значений: 0 … 255

Float – вещественный тип данных, диапазон значений: 3.4\*10–38… 3.4\*1038

Bool – логический тип данных, принимает значения true и false

Вы так же научились получать случайные числа, а значит вы можете добавить непредсказуемости в поведение робота. В конце главы мы увидели как может быть использован оператор цикла while для создания бесконечного цикла.

# Управляющие структуры

В предыдущих главах мы видели операторы "repeat" и "while". Эти операторы управляют тем, каким образом выполняются или не выполняются другие операторы программы. Такие операторы называются “управляющие структуры”. В этой главе мы познакомимся с другими управляющими структурами.

## Оператор "if"

Иногда возникает необходимость выполнять часть программы только в какой-то ситуации. В этом случае используется оператор "if". Давайте я вам покажу как это делается на примере. Для этого мы добавим новый трюк в программу, с которой мы работали в предыдущей главе. Мы хотим, чтобы робот проезжал прямо, а потом поворачивал случайным образом либо вправо, либо влево. Чтоб добиться такого поведения от робота мы опять будем использовать случайные числа. Мы берём случайное число, которое либо положительное, либо отрицательное. Если это число меньше 0 мы делаем поворот направо, иначе поворот налево. Вот получившаяся программа:

#define MOVE\_TIME 500

#define TURN\_TIME 360

task main()

{

while(true)

{

OnFwd(OUT\_AC, 75);

Wait(MOVE\_TIME);

if (Random() >= 0)

{

OnRev(OUT\_C, 75);

}

else

{

OnRev(OUT\_A, 75);

}

Wait(TURN\_TIME);

}

}

Оператор "if" выглядит похожим на оператор "while". Если условие внутри круглых скобок истинно, тогда выполняется часть программы заключенная в фигурные скобки. В противном случае, выполняется часть программы заключенная в фигурные скобки после ключевого слова "else". Давайте подробнее взглянем на условие которое мы использовали. Оно выглядит как "Random() >= 0". Это означает что значение Random() должно быть больше или равно 0, чтобы условие считалось истинным. Вы можете сравнивать значения различными способами, вот наиболее важные из них:

* "==" равно;
* "<" меньше;
* "<=" меньше или равно;
* ">" больше;
* ">=" больше или равно;
* "!=" не равно.

Вы можете комбинировать условия используя ключевое связку "&&", которая означает логическое "и", или связку "||", которая означает логическое “или”. Вот некоторые примеры условий:

"true" всегда истинно

"false" никогда не истинно

"ttt != 3" истинно, когда "ttt" не равно 3

"(ttt >= 5) && (ttt <= 10)" истинно, когда "ttt" лежит между 5 и 10 включительно

"(aaa == 10) || (bbb == 10)" истинно, если "aaa" или "bbb" (или и то и другое) равно 10

Обратите внимание, что оператор "if" может имеет два блока кода внутри себя. Часть начинающаяся сразу после условия, которая будет выполнена если условие истинно, и часть после ключевого слова "else", которая будет выполнена в противном случае. Ключевое слово "else" и блок кода после него являются необязательными. Так что вы можете пропустить их, если не нужно ничего делать когда условие ложно.

## Оператор "do"

Еще одна управляющая структура - это оператор "do". Его используют следующим образом:

do

{

statements;

}

while (condition);

Операторы внутри фигурных скобок после "do" выполняются до тех пор, пока условие в круглых скобках истинно. Условие может быть задано тем же способом, что и в операторе "if" рассмотренном выше. Вот пример программы с использованием этого оператора. Под её управлением робот перемещается случайным образом в течение 20 секунд, после чего останавливается.

int move\_time, turn\_time, total\_time;

task main()

{

total\_time = 0;

do

{

move\_time = Random(1000);

turn\_time = Random(1000);

OnFwd(OUT\_AC, 75);

Wait(move\_time);

OnRev(OUT\_C, 75);

Wait(turn\_time);

total\_time += move\_time;

total\_time += turn\_time;

}

while (total\_time < 20000);

Off(OUT\_AC);

}

Заметим, что оператор "do" практически один в один повторяет оператор "while". Но в операторе "while" условие проверяется перед выполнением блока операторов внутри него, а в операторе "do" это делается после. Таким образом в операторе "while" команды могут вообще не выполниться ни разу, а в операторе "do" они выполнятся 1 раз в любом случае.

## Подводим итоги

В этой главе вы познакомились с двумя новыми управляющими структурами - оперторами "if" и "do". Вместе с рассмотренными ранее операторами "repeat" и "while" они управляют тем какие фрагменты программы будут выполняться и в каком порядке. Очень важно, чтобы вы полностью понимали как они это делают. Поэтому будет полезным перед переходом к следующей главе поэкспериментировать с новыми операторами, пока вы не будете уверены, что освоили их использование.

# Сенсоры

[](http://roboforum.ru/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:NXT-Tribot-sens1.jpg)Разумеется вы можете подключить сенсоры к модулю NXT чтобы робот реагировал на внешние события. Перед тем как я покажу вам, как это сделать, мы должны немного модифицировать робота, добавив ему сенсор касания. Как и ранее, используйте инструкцию по сборке Tribot'а для сборки переднего бампера.

Соедините полученный сенсор касания со входом 1 на модулей NXT.

## 

## Ждём информацию с сенсора

Давайте начнем с простой программы по которой робот будет ехать вперед, пока не коснётся чего-нибудь. Вот её текст:

task main()

{

SetSensor(IN\_1,SENSOR\_TOUCH);

OnFwd(OUT\_AC, 75);

until (SENSOR\_1 == 1);

Off(OUT\_AC);

}

В этой программе две строки для нас особо интересны. Первая строка программы указывает роботу какой тип сенсора мы используем. IN\_1 это номер входа, к которому подключен сенсор. Другие входы для сенсоров имеют названия IN\_2, IN\_3 и IN\_4. Идентификатор SENSOR\_TOUCH показывает что это сенсор касания. Для датчика света мы будем использовать SENSOR\_LIGHT. После того как мы указали тип сенсора и куда он подключен, программа включает оба мотора и робот начинает ехать вперед. Следующий оператор очень полезен. Он ожидает пока условие внутри его круглых скобок не станет истинным. Указанное там условие говорит что значение сенсора SENSOR\_1 должно быть равно 1, что означает, что сенсор нажат. Пока сенсор не будет нажат, значение сенсора будет 0. Таким образом этот оператор ждёт нажатия сенсора. После чего оба мотора выключаются и задача считается завершенной.

## Действия при срабатывании сенсора касания

Давайте теперь попытаемся сделать так, чтобы робот избегал препятствия. Когда робот будет сталкиваться с препятствием, мы сделаем чтобы он отъезжал немного назад, поворачивался и продолжал движение. Вот программа, которая это реализует:

task main()

{

SetSensorTouch(IN\_1);

OnFwd(OUT\_AC, 75);

while (true)

{

if (SENSOR\_1 == 1)

{

OnRev(OUT\_AC, 75); Wait(300);

OnFwd(OUT\_A, 75); Wait(300);

OnFwd(OUT\_AC, 75);

}

}

}

Как и в предыдущем примере, мы сначала определяем тип сенсора, затем робот начинает ехать вперед, а дальше в бесконечном цикле мы постоянно проверяем не оказался ли нажатым контактный сенсор, и если это так - движемся назад 0.3 секунды, поворачиваем направо в течение 0.3 секунд и затем продолжаем движение вперед.

## 

## Сенсор освещенности

Кроме датчика касания, у вас в Mindstorms NXT также имеется датчик освещенности, датчик звука и цифровой ультразвуковой дальномер. Датчик освещенности может быть настроен на излучение света или только на приём света. Таким образом мы можем измерять отраженный свет или просто уровень освещенности с какого-то направления. Измерение отраженного света полезно когда мы строим робота, который будет ездить по линии нанесённой на пол. Именно это мы и будем делать в следующем примере. А теперь, чтобы продолжить наши эксперименты, нужно достроить Tribot'a. Соедините датчик освещенности с входом 3, датчик звука ко входу 2 и ультразвуковой дальномер ко входу 4, как указано в инструкции.

[](http://roboforum.ru/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:NXT-Tribot-sens2.jpg)

Кроме того нам потребуется тестовая поверхность с черной линией, которая идёт в комплекте с NXT. Простейший принцип отслеживания линии заключается в том, что робот пытается держаться датчиком освещенности по правой границе черной линии, поворот правее сделает уровень отраженного света слишком высоким, а поворот левее слишком низким. Вот простая программа, под управлением которой робот может следовать по линии с одним датчиком освещенности.

#define THRESHOLD 40

task main()

{

SetSensorLight(IN\_3);

OnFwd(OUT\_AC, 75);

while (true)

{

if (Sensor(IN\_3) > THRESHOLD)

{

OnRev(OUT\_C, 75);

Wait(100);

until(Sensor(IN\_3) <= THRESHOLD);

OnFwd(OUT\_AC, 75);

}

}

}

Программа сначала объявляет порт 3 датчиком освещенности. Затем робот включается на движение вперед и запускается бесконечный цикл. В нём, если уровень освещенности больше чем 40 (мы использовали константу, чтобы можно было удобно настраивать робота, так как наш метод отслеживания линии сильно зависит от уровня освещенности помещения, где проводится эксперимент) мы включаем один из моторов в режим обратного хода пока снова не вернемся на линию.

Как можно увидеть из выполнения программы - движение робота не очень плавные. Попробуйте добавить Wait(100); перед оператором until, чтобы немного улучшить движение робота. Обратите внимание, что программа не работает для движения против часовой стрелки. Чтобы можно было двигаться по любому направлению линии потребуется другая, более сложная, программа.

Чтобы получить уровень отраженного освещения с выключенным светодиодом подсветки, настройте сенсор следующим образом:

SetSensorType(IN\_3,IN\_TYPE\_LIGHT\_INACTIVE);

SetSensorMode(IN\_3,IN\_MODE\_PCTFULLSCALE);

ResetSensor(IN\_3);

## Звуковой сенсор

Используя звуковой сенсор вы можете превратить свой дорогостоящий набор NXT в "выключатель по хлопку"! Мы напишем программу, которая ждёт громкого звука, после чего включает робота на движение вперед, пока не услышит еще один громкий звук. Не забудьте проверить подключение звукового сенсора к порту 2, как указано в инструкции по сборке Tribot'a.

#define THRESHOLD 40

#define MIC SENSOR\_2

task main()

{

SetSensorSound(IN\_2);

while(true){

until(MIC > THRESHOLD);

OnFwd(OUT\_AC, 75);

Wait(300);

until(MIC > THRESHOLD);

Off(OUT\_AC);

Wait(300);

}

}

Сначала мы определяем константу уровня звука с которого мы будем считать его громким - THRESHOLD и псевдоним для SENSOR\_2; в самой задаче мы настраиваем порт 2 на чтение данных со звукового сенсора и запускаем бесконечный цикл.

Используя оператор until программа ждёт уровня звука который будет больше чем объявленный в THRESHOLD: обратите внимание, что SENSOR\_2 это не просто имя, а функция, которая возвращает уровень звука замеренный сенсором.

Если будет обнаружен громкий звук робот начнёт движение вперед, пока еще один громкий звук не остановит его.

Операторы wait были добавлены в программу потому что иначе робот будет запускаться и тут же останавливаться: на самом деле NXT настолько быстр, что у него практически не занимает времени выполнение программы между двумя операторами until. Пожалуй, если вы попробуете закомментировать оба оператора wait, вы лучше поймёте о чем идёт речь. Альтернативой использованию оператора until для ожидания событий может быть оператор while. Достаточно поместить в его круглые скобки соответствующее условие, например

while(MIC <= THRESHOLD).

На самом деле не так много нужно помнить про аналоговые сенсоры из набора NXT; просто не забывайте, что оба датчика - освещенности и звуковой возвращают вам число от 0 до 100.

## Ультразвуковой дальномер

Ультразвуковой дальномер работает как сонар. Излучая громкий звук он посылает пачку ультразвуковых волн и замеряет время, которое потребовалось им, чтобы отразиться от объектов в поле зрения и вернуться к датчику. Это цифровой сенсор, что означает, что в него встроено устройство для анализа и обработки данных. С этим сенсором вы можете построить робота, который будет избегать препятствий, а не обнаруживать их путём столкновения (как это приходилось делать с датчиком касания).

#define NEAR 15 //cm

task main(){

SetSensorLowspeed(IN\_4);

while(true){

OnFwd(OUT\_AC,50);

while(SensorUS(IN\_4)>NEAR);

Off(OUT\_AC);

OnRev(OUT\_C,100);

Wait(800);

}

}

Программа объявляет порт 4 для чтения данных с ультразвукового дальномера, запускает бесконечный цикл в котором робот едет прямо, пока не обнаружит препятствие ближе чем NEAR сантиметров (15см в нашем случае), после этого он отъезжает и отворачивает немного и снова пытается ехать прямо.

## Подводим итоги

В этой главе вы ознакомились с тем, как работать со всеми видами датчиков включенных в набор NXT. Мы также увидели, как полезны операторы while и until при работе с сенсорами.

Я рекомендую вам попробовать самостоятельно написать несколько программ до перехода к следующей главе. Теперь у вас есть все компоненты, чтобы ваш робот показывал достаточно сложное поведение: попробуйте адаптировать под NXC простейшие программы из "Robo Center programming guide", поставляемого с набором.

# 

# Задачи и функции

До этого момента все наши программы состояли из просто одной задачи. Но программы на языке NXC могут иметь множественные задачи. Также возможно размещать куски программы в так называемых функциях, которые вы можете использовать в различных местах своей программы. Использование задач и функций делает вашу программу более лёгкой для понимания и более компактной. В этой главе мы рассмотрим эти возможности.

## Задачи

Каждая программа NXC может состоять максимум из 255 задач; каждая из которых имеет уникальное имя. Задача с именем "main" всегда должна присутствовать в любой программе, так как это первая задача которая отправляется на выполнение. Другие задачи будут выполняться только если уже запущенная на выполнение задача скажет системе чтобы она их выполнила, или они могут быть неявно запланированы к выполнению; главная задача должна завершиться в этом случае перед тем, как будут выполнены такие задачи. После её завершения запланированные задачи будут выполняться параллельно.

Давайте я покажу вам использование задач на следующем примере. Мы хотим сделать программу по которой робот ездит по квадрату, как и раньше, но если он встречает препятствие - он должен на него среагировать. Это тяжело реализовать в одной задаче, так как робот должен делать две вещи в один и тот же момент - ездить по нужной траектории (т.е. включать\выключать моторы в нужные моменты времени) и следить за сенсорами. Так что лучше мы будем использовать для этого две задачи, первая из которых будет ездить по квадрату, а вторая реагировать на показания датчиков. Вот текст этой программы:

mutex moveMutex;

task move\_square()

{

while (true)

{

Acquire(moveMutex);

OnFwd(OUT\_AC, 75); Wait(1000);

OnRev(OUT\_C, 75); Wait(500);

Release(moveMutex);

}

}

task check\_sensors()

{

while (true)

{

if (SENSOR\_1 == 1)

{

Acquire(moveMutex);

OnRev(OUT\_AC, 75); Wait(500);

OnFwd(OUT\_A, 75); Wait(500);

Release(moveMutex);

}

}

}

task main()

{

Precedes(move\_square, check\_sensors);

SetSensorTouch(IN\_1);

}

Главная задача просто объявляет какой сенсор куда подключен и после этого запускает две другие задачи, планируя их к выполнению, после чего завершает свою работу. Задача "move\_square" бесконечно перемещает робота по квадрату. Задача "check\_sensors" проверяет, сработал ли датчик касания и, если это так, отводит робота от препятствия.

Очень важно помнить, что будучи запущенными задачи работают одновременно, что может привести к непредсказуемым результатам, если задачи одновременно попытаются получить управление моторами робота.

Чтобы избежать таких проблем, мы объявили странный тип переменной, "mutex" (в русском языке этому слову соответствует "мьютекс", это сокращение от "**mut**ual **ex**clusion", т.е. взаимное исключение): мы можем взаимодействовать с таким типом переменных только с помощью функций Acquire и Release, записывая критические куски кода между этими функциями, при этом оставаясь уверенными, что только одна задача в один момент времени имеет полное управление над моторами робота.

Такие "мьютекс"-переменные еще называются семафорами, а такая техника программирования называется конкурентной. Более подробно это обсуждается в 10-й главе.

## Функции

Иногда вам нужно один и тот же кусок программы использовать в разных местах. В этом случае вы можете поместить этот кусок программы в функцию и дать ей имя. Тогда вы сможете вызывать этот кусок программы на выполнение просто вызывая это имя из задачи. Давайте взглянем на следующий пример.

sub turn\_around(int pwr)

{

OnRev(OUT\_C, pwr); Wait(900);

OnFwd(OUT\_AC, pwr);

}

task main()

{

OnFwd(OUT\_AC, 75);

Wait(1000);

turn\_around(75);

Wait(2000);

turn\_around(75);

Wait(1000);

turn\_around(75);

Off(OUT\_AC);

}

В этой программе мы объявили функцию, которая разворачивает робота на месте. Главная задача вызывает эту функцию три раза. Обратите внимание, что мы вызываем функцию, указывая в круглых скобках числовой аргумент. Если у функции нет параметров - нужно будет просто указывать пустые скобки.

Так что использование функции выглядит также, как и использование многих других команд, которые вы уже видели.

Основное преимущество функций в том, что они хранятся в NXT только раз и таким образом экономится память. Но когда функции короткие, может быть лучше использовать встроенные ("inline") функции. Они не хранятся отдельно, а копия их размещается в каждом месте, откуда вызывается функция. Это использует больше памяти, но на количество таких функций ограничения нет. Такие функции могут быть объявлены следующим образом:

inline int Name( Args ) {

//body;

return x\*y;

}

Определение и вызов встроенных функций делается так же как и для обычных. Смотрите пример использования этих функций ниже:

inline void turn\_around()

{

OnRev(OUT\_C, 75); Wait(900);

OnFwd(OUT\_AC, 75);

}

task main()

{

OnFwd(OUT\_AC, 75);

Wait(1000);

turn\_around();

Wait(2000);

turn\_around();

Wait(1000);

turn\_around();

Off(OUT\_AC);

}

В указанном выше примере, мы можем сделать время для разворота аргументом функции, как показано тут:

inline void turn\_around(int pwr, int turntime)

{

OnRev(OUT\_C, pwr);

Wait(turntime);

OnFwd(OUT\_AC, pwr);

}

task main()

{

OnFwd(OUT\_AC, 75);

Wait(1000);

turn\_around(75, 2000);

Wait(2000);

turn\_around(75, 500);

Wait(1000);

turn\_around(75, 3000);

Off(OUT\_AC);

}

Заметим, что при определении в скобках после имени функции мы указываем аргументы этой функции. В этом случае мы показываем, что аргумент функции - целое число (можно выбрать другие типы), а имя этой переменной - "turntime". Когда у функции больше аргументов, вы должны разделять их запятыми. Обратите внимание, что в NXC, ключевое "sub" является синонимом "void"; также, функции могут иметь другой тип, отличный от "void", они могут возвращать целые числа, строковые значения вызывающему, для подробной информации на эту тему смотрите полную инструкцию по языку NXC.

## Определяем макрос

Существует еще один способ дать небольшому фрагменту кода собственное имя для обращения. В языке NXC вы можете определить макрос ( не путайте с макросами BricxCC). Мы видели ранее, что можем определять константы используя ключевое слово "#define", назначая им имена. Но на самом деле мы можем таким же образом определить любой кусок кода. Вот та же самая программа, но с использованием макроса для отделения фукнции разворота.

#define turn\_around \

OnRev(OUT\_B, 75); Wait(3400);OnFwd(OUT\_AB, 75);

task main()

{

OnFwd(OUT\_AB, 75);

Wait(1000);

turn\_around;

Wait(2000);

turn\_around;

Wait(1000);

turn\_around;

Off(OUT\_AB);

}

После ключевого слова #define идёт имя turn\_around которое будет означать текст указанный следом. Теперь, где бы вы не указали turn\_around, это имя будет замещено определенным вами ранее текстом. Обратите внимание, что текст должен быть в одну строку. (Вообще то есть способ использования множественных строк внутри ключевого слова #define, но такой подход не рекомендован к использованию.) На самом деле возможности #define еще более широкие. Они так же могут иметь аргументы. Например, мы можем сделать время поворота аргументом. Вот пример в котором мы определяем 4 макро: двигаться вперед, двигаться назад, поворачивать налево и поворачивать направо. Каждый из них имеет 2 аргумента - скорость и время выполнения.

#define turn\_right(s,t) \

OnFwd(OUT\_A, s);OnRev(OUT\_B, s);Wait(t);

#define turn\_left(s,t) \

OnRev(OUT\_A, s);OnFwd(OUT\_B, s);Wait(t);

#define forwards(s,t) OnFwd(OUT\_AB, s);Wait(t);

#define backwards(s,t) OnRev(OUT\_AB, s);Wait(t);

task main()

{

backwards(50,10000);

forwards(50,10000);

turn\_left(75,750);

forwards(75,1000);

backwards(75,2000);

forwards(75,1000);

turn\_right(75,750);

forwards(30,2000);

Off(OUT\_AB);

}

Очень полезно использовать такие макро, они делают вашу программу более компактной и читаемой. Кроме того вы можете более просто модифицировать код, например, переподключая двигатели к другим портам.

## Подводим итоги

В этой главе вы увидели, как используются задачи, функции, встраиваемые функции и макро. Все они используются в разных случая. Задачи обычно работают одновременно и занимаются разными делами в один и тот же момент. Функции удобны, когда большие куски кода используются в различных местах одной и той же задачи.

Встроенные функции полезны, когда куски когда используются в различных местах программы в различных задачах, но они занимают память. И наконец макро очень полезны для небольших кусочков кода, которые могут использоваться в различных местах. Кроме того они имеют аргументы, что делает их еще более мощным инструментом.

Теперь, когда вы прошли все главы включая последнюю у вас достаточно знаний, чтобы делать программы под управлением которых робот может делать действительно сложные и интересные вещи. Оставшиеся главы в этом курсе покажут вам приёмы важные только в отдельных случаях.

# Создаём музыку

Модуль NXT имеет встроенный динамик и может воспроизводить музыку и даже звуковые файлы. Это может быть полезно в том случае, когда вы хотите чтобы NXT сказал вам, что что-то случилось. Но это также может быть просто более интересно, когда робот играет музыку или разговаривает в процессе выполнения заданий.

## Проигрываем звуковые файлы

BricxCC содержит в себе встроенную утилиту для преобразования .wav-файлов в .rso-файлы, доступную через меню "Tools" / "Sound conversion".

Завершив преобразование, вы можете сохранить .rso-файлы со звуками во флеш-память NXT при помощи другой утилиты (меню "Tools" / "NXT explorer"), после чего их можно будет воспроизводить командой

PlayFileEx(filename, volume, loop?)

Аргументы команды - имя файла, громкость (число от 0 до 4), и необязательный флаг зацикливания "loop", выставьте его в 1 (или TRUE), если хотите, чтобы файл вопроизвёлся не один раз, выставление его в 0 (или FALSE) воспроизведёт файл один раз.

#define TIME 200

#define MAXVOL 7

#define MINVOL 1

#define MIDVOL 3

#define pause\_4th Wait(TIME)

#define pause\_8th Wait(TIME/2)

#define note\_4th \

PlayFileEx("! Click.rso",MIDVOL,FALSE); pause\_4th

#define note\_8th \

PlayFileEx("! Click.rso",MAXVOL,FALSE); pause\_8th

task main()

{

PlayFileEx("! Startup.rso",MINVOL,FALSE);

Wait(2000);

note\_4th;

note\_8th;

note\_8th;

note\_4th;

note\_4th;

pause\_4th;

note\_4th;

note\_4th;

Wait(100);

}

Эта забавная программа сначала проигрывает звук начала работы, который вы возможно уже слышали, а затем использует стандартный звук "click" чтобы сыграть джингл “Shave and a haircut” который приводит в восторг кролика Роджера! Макрос действительно очень удобен в данном случае для упрощения записи команд в главное задаче: попробуйте модифицировать настройки громкости чтобы сделать мелодию наиболее приятной.

## Играем музыку

Чтобы воспроизвести тоновый сигнал вы можете использовать команду PlayToneEx(frequency, duration, volume, loop?) Она имеет 4 параметра. Первый это частота тона в Герцах, второй - продолжительность в 1/1000 секунды (как в операторе "wait"), и последний - такой же флаг зацикливания, как и в предыдущей команде. Кроме того может быть использована команда PlayTone(frequency, duration), в этом случае громкость используется настроенная в меню вашего модуля NXT, а функция зацикливания недоступна.

Вот таблица с полезными частотами музыкальных нот по октавам:

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

Как и в случае с PlayFileEx, модуль NXT не дожидается когда нота доиграет. Так что если вы хотите воспроизвести подряд несколько нот - вам придётся добавить между ними достаточно долгие команды "wait". Например вот так:

#define VOL 3

task main()

{

PlayToneEx(262,400,VOL,FALSE); Wait(500);

PlayToneEx(294,400,VOL,FALSE); Wait(500);

PlayToneEx(330,400,VOL,FALSE); Wait(500);

PlayToneEx(294,400,VOL,FALSE); Wait(500);

PlayToneEx(262,1600,VOL,FALSE); Wait(2000);

}

Вы можете с легкость создавать музыкальные фрагменты с использованием инструмента "Brick Piano", являющегося частью BricxCC. Если вы хотите, чтобы NXT играл музыку, когда выполнял различные задачи - лучше используйте для этого отдельную задачу. Вот пример достаточно достаточно глупой программы, где NXT ездит взад-вперед и параллельно играет музыку.

task music()

{

while (true)

{

PlayTone(262,400); Wait(500);

PlayTone(294,400); Wait(500);

PlayTone(330,400); Wait(500);

PlayTone(294,400); Wait(500);

}

}

task movement()

{

while(true)

{

OnFwd(OUT\_AC, 75); Wait(3000);

OnRev(OUT\_AC, 75); Wait(3000);

}

}

task main()

{

Precedes(music, movement);

}

## Подводим итоги

В этой главе вы научились как с помощью NXT проигрывать звуки и музыку. Кроме того вы увидели, как использовать отдельную задачу для воспроизведения музыки.

# Еще раз о моторах

Существует несколько дополнительных команд для моторов, которые вы можете использовать для более точного управления моторами. В этой главе мы обсудим эти команды: ResetTachoCount, Coast (Float), OnFwdReg, OnRevReg, OnFwdSync, OnRevSync, RotateMotor, RotateMotorEx, а также основы ПИД-управления.

## Плавная остановка

Когда вы используете команду Off(), сервомотор мгновенно останавливается, блокируя вал и удерживая позицию. Существует способ более мягкой остановки мотора не используя блокировку вала. Для этого используйте команды Float() или что то же самое - Coast(), они просто отключают питание моторов. Вот пример, сначала робот останавливается используя блокировку вала, а потом останавливается без использования блокировки. Обратите внимание на разницу. На самом деле для конкретно этого робота разница достаточно незаметна, но для других роботов эта разница будет гораздо больше.

task main()

{

OnFwd(OUT\_AC, 75);

Wait(500);

Off(OUT\_AC);

Wait(1000);

OnFwd(OUT\_AC, 75);

Wait(500);

Float(OUT\_AC);

}

## Дополнительные команды

Команды OnFwd() и OnRev() являются простейшими командами для управления моторами.

Сервомоторы из набора NXT имеют встроенные энкодеры, которые позволяют вам достаточно точно контролировать положение вала и скорость его вращения.

Встроенная в NXT прошивка реализует алгоритм ПИД-управления (Пропорциональный Интегральный Дифференциальный - от названия основных трёх его компонент) положением вала и скоростью его вращения с обратной связью через энкодеры.

Если вы хотите, чтобы ваш робот ездил строго по прямой, вы можете использовать возможность синхронизации двигателей, которая объединяет их в пару, в которой каждый двигатель замедляется, если второй не успевает крутиться с нужной скоростью. Подобным образом вы можете настроить пару моторов на работу в синхронизированном режиме с процентом скорости левого двигателя по отношению к правом, для поворота направо, налево или вращения на месте, при этом траектория будет выполняться достаточно точно. Существует множество команд, которые позволяют полностью использовать возможности сервомоторов из набора NXT!

Команда OnFwdReg(‘ports',‘speed',‘regmode') включает двигатели указанные в ‘ports' на движение вперед со скоростью ‘speed' применяя режим регуляции, который может быть OUT\_REGMODE\_IDLE, OUT\_REGMODE\_SPEED или OUT\_REGMODE\_SYNC. Если выбран режим IDLE, ПИД-регулирование будет отключено; если режим SPEED, Модуль NXT будет регулировать скорость отдельных моторов чтобы получить постоянную скорость, даже если нагрузка на мотор меняется; и наконец, если выбран режим SYNC, пара моторов указанная в ‘ports' работают в синхронизированном режиме, как было описано выше.

Команда OnRevReg() работает так же как и выше, только в другую сторону.

task main()

{

OnFwdReg(OUT\_AC,50,OUT\_REGMODE\_IDLE);

Wait(2000);

Off(OUT\_AC);

PlayTone(4000,50);

Wait(1000);

ResetTachoCount(OUT\_AC);

OnFwdReg(OUT\_AC,50,OUT\_REGMODE\_SPEED);

Wait(2000);

Off(OUT\_AC);

PlayTone(4000,50);

Wait(1000);

OnFwdReg(OUT\_AC,50,OUT\_REGMODE\_SYNC);

Wait(2000);

Off(OUT\_AC);

}

Эта программа показывает различные способы регулирования. Если вы попробуете остановить колёса робота, удерживая их в руках, при первом режиме (IDLE) ничего не будет происходить, во втором режиме (SPEED) попытка замедлить колесо вызовет увеличение мощности подаваемой на этот мотор, так как NXT пытается добиться заданной ему скорости; и наконец, в режиме синхронизации (SYNC), если попытаться остановить одно колесо - второе так же остановится, дожидаясь заблокированного колеса.

OnFwdSync(‘ports',‘speed',‘turnpct') делает то же самое что OnFwdReg() в режиме SYNC, но можно указать 'turnpct' - процент поворота влево\вправо (от -100 до 100).

OnRevSync() делает то же самое что и предыдущая команда, только направления обратные. Следующая программа показывает эти команды в работе: попробуйте менять процент поворота, чтобы увидеть как это влияет на работу двигателей.

task main()

{

PlayTone(5000,30);

OnFwdSync(OUT\_AC,50,0);

Wait(1000);

PlayTone(5000,30);

OnFwdSync(OUT\_AC,50,20);

Wait(1000);

PlayTone(5000,30);

OnFwdSync(OUT\_AC,50,-40);

Wait(1000);

PlayTone(5000,30);

OnRevSync(OUT\_AC,50,90);

Wait(1000);

Off(OUT\_AC);

}

И наконец, моторы можно поворачивать на конкретное число градусов (помните, что полный оборот это 360°).

Для обоих следующих команд, вы можете указывать направление вращения моторов или знаком скорости или знаком угла: так что если скорость или знак имеют один и тот же знак - мотор будет крутиться вперед, иначе назад.

RotateMotor(‘ports',‘speed',‘degrees') вращает вал мотора указанного в ‘ports' на ‘degrees' градусов со скоростью ‘speed' (в диапазоне 0-100).

task main()

{

RotateMotor(OUT\_AC, 50,360);

RotateMotor(OUT\_C, 50,-360);

}

RotateMotorEx(‘ports',‘speed',‘degrees',‘turnpct',‘sync', 'stop') это расширение предыдущей команды, которое позволяет синхронизировать два мотора (например, OUT\_AC) указывая процент поворота ‘turnpct' (от -100 до 100) и флаг ‘sync' (который может быть или истиной или ложью). Также команда позволяет указать - нужно ли по её завершении блокировать вал двигателя, для этого используется флаг 'stop'.

task main()

{

RotateMotorEx(OUT\_AC, 50, 360, 0, true, true);

RotateMotorEx(OUT\_AC, 50, 360, 40, true, true);

RotateMotorEx(OUT\_AC, 50, 360, -40, true, true);

RotateMotorEx(OUT\_AC, 50, 360, 100, true, true);

}

## ПИД-управление

Встроенное программное обеспечение NXT реализует цифровой ПИД-регулятор для точного управления положением и скорости сервомоторов. Этот тип управления один из самых простых и при этом эффективных способов управления с обратной связью, широко используемый во всяких задача автоматизации. Если кратко, он работает так (Я буду говорить об управлении положением вала цифровым контроллером):

Ваша программа задаёт контроллеру требуемое положение R(t); он выдаёт мощность U(t) на двигатель, замеряя положение вала Y(t) встроенными энкодерами и рассчитывает отклонение E(t) = R(t) – Y(t): вот почему это называется контроллер с обратной связью, потому что выходное положение вала мотора возвращается на вход контроллера, чтобы посчитать насколько он сейчас промахивается. Контроллер при этом превращает размер отклонения E(t) в новую команду для двигателя U(t) таким образом:

U(t) = P(t) + I(t) + D(t), где

P(t) = KP·E(t),

I(t) = KI·( I(t–1) + E(t) )

и D(t) = KD·(E(t) – E(t –1)).

Это может выглядеть достаточно сложно для новичка, но я предприму все усилия, чтобы помочь вам объяснить этот механизм. Команда является суммой трёх частей, Пропорциональной части P(t), интегральной I(t) и дифференциальной D(t).

P(t) даёт контроллеру быстроту реакции, но может мешать стабильности в долгосрочной перспективе;

I(t) даёт контроллеру “память”, в том смысле, что накапливает ошибку и компенсирует её, гарантируя долгосрочную стабилизацию в нужном положении;

D(t) даёт контроллеру “возможность предугадывать” (на основе производной из математики), ускоряя время реакции системы.

Я понимаю, что это может всё еще казаться запутанным, на самом деле на эту тему написаны целые научные книги! Но мы всё еще можем попробовать это в реальности с нашим NXT! Простая программа, которая поможет зафиксировать в памяти эту тему:

#define P 50

#define I 50

#define D 50

task main(){

RotateMotorPID(OUT\_A, 100, 180, P, I, D);

Wait(3000);

}

Функция RotateMotorPID(port,speed, angle, Pgain,Igain,Dgain) позволяет управлять мотором устанавливая различные параметры ПИД-регулятора, отличные от настроек по умолчанию. Попробуйте выставлять следующие значения:

(50,0,0): мотор не поворачивается точно на 180°, так как остаётся нескомпенсированная ошибка

(0,x,x): без пропорциональной части ошибка очень большая

(40,40,0): здесь у нас перелёт, т.е. мотор поворачивается дальше чем надо и потом возвращается

(40,40,90): хорошая точность и время выполнения команд

(40,40,200): вал колеблется, так как дифференциальная часть слишком большая

Попробуйте другие значения, чтобы посмотреть как разные настройки ПИД-регулятора влияют на выполнение команд.

## Подводим итоги

В этой главе вы освоили дополнительные команды для управления моторами: Float(),Coast() которые плавно останавливают моторы; OnXxxReg() и OnXxxSync() которые реализуют управление с обратной связью как по скорости, так и по синхронизации со вторым мотором; RotateMotor() и RotateMotorEx() используемые для поворота на нужный угол выходного вала моторов. Вы немного изучили что такое ПИД-регулирование; хотя это было не исчерпывающее объяснение, может быть я вас даже немного запутал, попробуйте поискать материал на эту тему в Интернете!

# Дополнительная информация о сенсорах

В главе 5 мы обсудили простейшие приёмы использования сенсоров, но на самом деле с ними можно делать намного больше. В этой главе мы обсудим отличие между режимом и типом сенсора, увидим как использовать старые RCX-совместимые сенсоры и подключать их к блоку NXT используя кабели "Lego converter cable".

## Режимы и типы сенсоров

Команда SetSensor(), которую мы видели ранее, реально делает две вещи: она устанавливает тип сенсора и режим, в котором этот сенсор будет работать. Путём установки режима и типа сенсора отдельными полями вы можете более точно контролировать поведение сенсора, что полезно в различных задачах.

Тип сенсора устанавливается командой SetSensorType(). Существует множество различных типов, но я опишу только основные:

* SENSOR\_TYPE\_TOUCH, сенсор касания,
* SENSOR\_TYPE\_LIGHT\_ACTIVE, датчик освещенности (с включенным светодиодом, т.е. работающий по отраженному свету),
* SENSOR\_TYPE\_SOUND\_DB, датчик звука и
* SENSOR\_TYPE\_LOWSPEED\_9V - ультразвуковой дальномер.

Установка типа сенсора важна для определения нужно ли подавать питание (включает светодиод в датчике освещенности), или для указания модулю NXT, что сенсор цифровой и надо работать с ним по протоколу I2C. Кроме всего прочего допускается использование с NXT сенсоров из старого набора, поставляемого с модулем RCX:

* SENSOR\_TYPE\_TEMPERATURE, датчик температуры,
* SENSOR\_TYPE\_LIGHT - старый датчик освещенности,
* SENSOR\_TYPE\_ROTATION - датчик положения вала (датчик вращения) - этот тип мы обсудим позже.

Режим датчика задаётся командой SetSensorMode(). Существует 8 различных режимов. Наиболее важный - это SENSOR\_MODE\_RAW. В этом режиме, значение, которое вы получаете от сенсора является числом и лежит в диапазоне 0..1023. Это исходное значение, которое выдаёт датчик. Что оно означает - зависит от того, какой датчик подключен. Например, для датчика касания, когда он не нажат, значение близко к 1023, а когда полностью нажат - близко к 50. При частичном нажатии значение варьируется между 50 и 1000. Так что если вы установите датчик касания в этот режим, вы сможете определять, когда датчик нажат частично. С датчиком освещенности значение варьируется между 300 (очень светло) и 800 (очень темно). Это даёт гораздо значение большей точности, чем при использовании команды SetSensor(). Для более подробной информации смотрите инструкцию по программированию к языку NXC.

Второй режим работы сенсоров - SENSOR\_MODE\_BOOL. В этом режиме возвращаемое значение 0 или 1. Оно получается из исходного значения 0..1023 сравнением, если меньше 565, возвращается 0, иначе 1. SENSOR\_MODE\_BOOL является режимом по умолчанию для датчика касания, но может быть быть применен к любому другому сенсору, чтобы убрать аналоговую информацию и оставить двоичный ответ. Режимы SENSOR\_MODE\_CELSIUS и SENSOR\_MODE\_FAHRENHEIT полезны только для датчиков температуры и дают ответ в соответствующих единицах измерения. SENSOR\_MODE\_PERCENT превращает исходное значение 0..1023 в значение между 0 и 100, просто деля его на 1023. SENSOR\_MODE\_PERCENT является режимом по умолчанию для датчика освещенности. SENSOR\_MODE\_ROTATION используется только для сенсора вращения (см. ниже).

Есть еще два интересных режима: SENSOR\_MODE\_EDGE и SENSOR\_MODE\_PULSE. Они считают переходы между высоким и низким уровнем сигнала с датчика или наоборот. Например, когда мы нажимаете на датчик касания это вызывает переход с высокого к низкому уровню сигнала. Когда вы отпускаете его это вызывает обратный переход. Когда вы устанавливаете сенсор в режим SENSOR\_MODE\_PULSE, только переходы с низкого на высокий уровень считаются. Так что каждое нажатие и отпускание сенсора касания считается за 1. А когда вы устанавливаете режим датчика в SENSOR\_MODE\_EDGE, оба перехода засчитываются, т.е. когда вы нажимаете и отпускаете датчик - счетчик увеличивается на 2. Вы можете использовать этот режим для подсчета того, сколько раз нажимали на датчик касания. Или с датчиком освещенности вы можете посчитать как часто включали/выключали освещение. Разумеется, когда вы считаете количество срабатываний или переходов, у вас должна быть возможность сбросить счетчик в ноль. Для этого используйте команду ClearSensor(), которая сбрасывает счетчик для указанного датчика.

Давайте разберем пример. Следующая программа использует датчик касания для управления роботом. Соедините датчик касания длинным проводом со входом 1. Если быстро нажать датчик касания 2 раза - робот поедет вперед, если нажать 1 раз - остановится.

task main()

{

SetSensorType(IN\_1, SENSOR\_TYPE\_TOUCH);

SetSensorMode(IN\_1, SENSOR\_MODE\_PULSE);

while(true)

{

ClearSensor(IN\_1);

until (SENSOR\_1 > 0);

Wait(500);

if (SENSOR\_1 == 1) {Off(OUT\_AC);}

if (SENSOR\_1 == 2) {OnFwd(OUT\_AC, 75);}

}

}

Обратите внимание, что мы сначала установили тип, а только потом - режим работы датчика. Это важно, так как установка типа датчика автоматически сбрасывает режим в значение по умолчанию для этого типа датчика.

## Сенсор вращения

Сенсор вращения это очень полезный вид датчика, представляющий из себя оптический энкодер, практический такой же, как встроенный в сервомоторы набора NXT. Датчик вращения имеет отверстие, в которое вы можете вставить вал, относительное угловое положение которого он может измеряет. Один полный оборот вала соответствует 16 шагам (или -16, если вращать в другую сторону), т.е. угловое разрешение датчика - 22.5 градуса, достаточно грубо, если сравнивать с разрешением энкодеров сервомоторов в 1 градус. Этот старый датчик вращения можно использовать для отслеживания положения вала без необходимости задействовать для этого сервомотор; кроме того не забывайте, что старый датчик ввиду отсутствия двигателя внутри гораздо легче вращается.

Если вам нужно большее разрешение, чем 16 шагов на оборот - вы всегда можете использовать шестерни, чтобы механически увеличить его.

Следующий пример унаследован из старого курса для RCX.

Одной из типичных ситуаций является размещение двух датчиков вращения на двух колёсах, которыми вы вращаете с помощью установленных на них же двигателей. Для прямолинейного движения вам нужно, чтобы колёса вращались с одинаково быстро. К сожалению, обычно моторы не вращаются одинаково. Используя датчики вращения вы можете отследить ситуацию, когда один мотор обгоняет другой. Вы можете временно приостановить этот мотор (лучше всего - используя команду Float()), пока оба датчика вращения не выдадут одинаковое значение. Следующая программа реализует этот подход к управлению двигателями. Она просто позволяет роботу ехать строго по прямой. Чтобы проверить её на роботе - подключите датчики вращения к моторам. Подцепите сенсоры на входы 1 и 3.

task main()

{

SetSensor(IN\_1, SENSOR\_ROTATION); ClearSensor(IN\_1);

SetSensor(IN\_3, SENSOR\_ROTATION); ClearSensor(IN\_3);

while (true)

{

if (SENSOR\_1 < SENSOR\_3)

{OnFwd(OUT\_A, 75); Float(OUT\_C);}

else if (SENSOR\_1 > SENSOR\_3)

{OnFwd(OUT\_C, 75); Float(OUT\_A);}

else

{OnFwd(OUT\_AC, 75);}

}

}

Программа сначала определяет, что оба датчика являются датчиками вращения, после чего сбрасывает их значения в 0. Потом запускается бесконечный цикл. В цикле мы проверяем, если значения датчиков совпадают - просто едем вперед. Если же один больше, чем другой - мы приостанавливаем соответствующий двигатель, пока другой не догонит его и датчики снова не покажут равные значения.

На самом деле это очень простая программа. Вы можете расширить её, чтобы робот перемещался на четко заданное расстояние или точно выполнял указанные повороты.

## Подключение нескольких сенсоров к одному входу

Есть смысл сделать небольшое предупреждение перед тем, как вы приступите к чтению этого раздела! Из-за новой структуры улучшенных датчиков платформы NXT и кабелей на 6 линий, эта возможность стала сложнее, чем с RCX-набором подключать больше чем 1 датчик на порт модуля. По моему мнению единственное имеющее смысл и более менее простое применение указанной ниже информации - сборка разветвителя датчика касания с помощью кабеля конвертера. Альтернатива которому - полноценный цифровой мультиплексор, который может работать по протоколу I2C с модулем NXT, но это однозначно не лучшее решение для начинающих.

Модуль NXT имеет 4 входя для подключения датчиков. Когда вы хотите собрать более сложного робота (и купили несколько дополнительных датчиков) этого может оказаться недостаточно для вас. К счастью есть несколько трюков, с помощью которых вы можете подключить два (или даже больше) датчиков к одному входу. Проще всего подключить два датчика касания к одному входу. Когда один (любой) из них нажат (или оба) - значение датчика будет 1, иначе 0. Вы не можете отличить какой датчик сработал, но иногда это и не нужно. Например, когда вы размещаете один датчик спереди, а второй сзади, вы можете понять каким стукнулись о препятствие по тому, куда вы сейчас ехали - вперед или назад. Еще вы можете установить режим сенсора RAW (см. выше). Теперь вы сможете получать больше информации. Если вам повезёт - значения нажатого и не нажатого сенсора будет разным для обоих сенсоров. Тогда вы сможете отличать какой из них нажат. А если нажаты сразу оба - вы получите намного меньшее значение (около 30) и тоже сможете это обнаружить.

Вы можете так же соединить датчик касания и датчик освещенности с одним входом (только для датчиков из набора RCX). Установите при этом тип сенсора "датчик света" (иначе датчик света не будет работать). Установите режим RAW. В этом случае, если нажат датчик касания - вы получите значение ниже 100. Если же он не нажат - вы получите значение датчика освещенности, которое никогда не бывает ниже 100.

Следующая программа использует этот подход. Робот должен быть снабжён датчиком освещенности, направленным вниз и датчиком касания выставленным в качестве бампера вперед. Соедините оба датчика ко входу 1. Робот будет ездить случайным образом по освещенной территории, отъезжая от препятствий. Когда датчик освещенности (значение с датчика > 750) попадает на тёмное место робот немного отъезжает назад. Когда что-то нажимает датчик касания (значение < 100) он делает то же самое. Вот текст это программы:

mutex moveMutex;

int ttt,tt2;

task moverandom()

{

while (true)

{

ttt = Random(500) + 40;

tt2 = Random();

Acquire(moveMutex);

if (tt2 > 0)

{ OnRev(OUT\_A, 75); OnFwd(OUT\_C, 75); Wait(ttt); }

else

{ OnRev(OUT\_C, 75); OnFwd(OUT\_A, 75); Wait(ttt); }

ttt = Random(1500) + 50;

OnFwd(OUT\_AC, 75); Wait(ttt);

Release(moveMutex);

}

}

task submain()

{

SetSensorType(IN\_1, SENSOR\_TYPE\_LIGHT);

SetSensorMode(IN\_1, SENSOR\_MODE\_RAW);

while (true)

{

if ((SENSOR\_1 < 100) || (SENSOR\_1 > 750))

{

Acquire(moveMutex);

OnRev(OUT\_AC, 75); Wait(300);

Release(moveMutex);

}

}

}

task main()

{

Precedes(moverandom, submain);

}

Я надеюсь программа получилась понятная. В ней есть две задачи. Задача "moverandom" заставляет робота случайным образом блуждать. Главная задача запускает "moverandom", потом переходит в своё продолжение "submain" чтобы запустилась "moverandom", настраивает сенсоры и ждет пока что-нибудь случится. Если значения сенсоров будут слишком малы (касание) или слишком велики (выехали из освещенного района) она остановит случайные блуждания, отъедет немного назад и затем снова продолжит случайные движения.

## Подводим итоги

В этой главе вы познакомились с некоторым количеством дополнительных возможностей сенсоров. Мы увидели, как отдельно устанавливать тип и режим работы датчиков, как эти режимы могут быть использованы для получения дополнительной информации. Мы научились использовать датчик вращения и увидели, как несколько датчиков можно подключить к одному входу NXT. Все эти трюки очень полезны при сборке достаточно сложных роботов, в которых сенсоры имеют огромное значение.

# Параллельные задачи

Как показывалось ранее, задачи в NXC выполняются одновременно, или как обычно говорят - параллельно. Это очень удобно. Такая возможность позволяет вам следить за датчиками в одной задаче, пока вторая задача управляет движением робота, а, скажем, третья задача играет какую-нибудь музыку. Но параллельные задачи могут создавать проблем. Одна задача может начать мешать другой.

## Неправильная программа

Разберем следующую программу. В ней одна задача управляет движением робота по квадратам (как мы часто уже делали раньше), а другая задача проверяет датчик касания, когда датчик нажат, она отводит робота немного назад и делает поворот на 90 градусов.

task check\_sensors()

{

while (true)

{

if (SENSOR\_1 == 1)

{

OnRev(OUT\_AC, 75);

Wait(500);

OnFwd(OUT\_A, 75);

Wait(850);

OnFwd(OUT\_C, 75);

}

}

}

task submain()

{

while (true)

{

OnFwd(OUT\_AC, 75); Wait(1000);

OnRev(OUT\_C, 75); Wait(500);

}

}

task main()

{

SetSensor(IN\_1,SENSOR\_TOUCH);

Precedes(check\_sensors, submain);

}

Вероятно программа выглядит как совершенно правильная, но если вы попробуете загрузить её в робота вы скорее всего обнаружите некоторое необычное его поведение. Попробуйте следующее, чтобы убедиться в этом: сделайте так, чтобы робот задел что-то бампером при повороте. Он начнет отъезжать назад, но немедленно снова начнет двигаться вперед, снова ударяясь о препятствие. Причина этого в том, что задачи начинают пересекаться. А происходит вот что. Робот едет в повороте и ударяется об препятствие. Он начинает отъезжать назад, но в этот момент главная задача (submain) завершает ожидание конца разворота и даёт команду ехать прямо на препятствие, игнорируя тот факт, что сейчас выполняется маневр отъезда от него. Вторая задача в это время находится в состоянии ожидания и даже не сможет обнаружить столкновение. Это точно не то поведение робота, которое мы хотели бы видеть. Проблема заключается в том, что пока вторая задача ждёт мы забыли, что первая задача продолжает работу и её действия начинают мешать работе задаче избегания препятствий.

## Критические секции и "мьютекс"-переменные

Один из способов решения этой проблемы это введение механизма, гарантирующего, что только одна задача в данный момент работает с двигателями робота. Такой подход мы уже применяли в главе 6. Давайте посмотрим на эту программу еще раз.

mutex moveMutex;

task move\_square()

{

while (true)

{

Acquire(moveMutex);

OnFwd(OUT\_AC, 75); Wait(1000);

OnRev(OUT\_C, 75); Wait(850);

Release(moveMutex);

}

}

task check\_sensors()

{

while (true)

{

if (SENSOR\_1 == 1)

{

Acquire(moveMutex);

OnRev(OUT\_AC, 75); Wait(500);

OnFwd(OUT\_A, 75); Wait(850);

Release(moveMutex);

}

}

}

task main()

{

SetSensor(IN\_1,SENSOR\_TOUCH);

Precedes(check\_sensors, move\_square);

}

Загвоздка здесь в том, что обе задачи (check\_sensors и move\_square) контролируют моторы только если другая задача не использует их в данный момент. Это делается с помощью команд Acquire, которые ждут, чтобы "мьютекс"-переменная moveMutex стала свободной, перед тем как начать работать с моторами, после чего команда блокирует эту переменую и передаёт управление следующей команде. Команда Acquire всегда используется на пару с командой Release, которая сообщает системе, что "мьютекс"-переменная освободилась, и теперь другие задачи могут использовать соответствующий общий ресурс, в нашем случае - двигатели робота. Кусок программы внутри пары Acquire - Release называется критической областью программы, в том смысле, что используются общие ресурсы. Это гарантирует что в таких областях задачи не будут конфликтовать друг с другом по объявленным общим ресурсам.

## Использование семафоров

Существует возможность собрать самому альтернативу "мьютекс"-переменым, являющуюся явной реализацией команд Acquire и Release.

Стандартный способ решить эту программу - объявить переменную, которая будет указывать, какая задача контроллирует двигатели. Другие задачи не должны использовать двигатели, пока первая задача не объявит их свободными. Такая переменная часто называется семафором. Пусть sem это такой семафор (это то же самое по сути, что и "мьютекс"). Мы будем считать, что значение 0 показывает, что двигатели свободны. Теперь, если задача хочет использовать моторы - она должна выполнить следующие команды:

until (sem == 0);

sem = 1; //Acquire(sem);

// Чего-то делаем с моторами

// (Это критическая область программы)

sem = 0; //Release(sem);

Сначала мы ждем, пока моторы будут свободны, затем забираем этот ресурс себе, выставляя 1. Теперь мы управляем ими. Когда мы всё, что хотели, сделали - пишем в sem снова 0. Ниже можно увидеть программу, реализующую семафор. Когда обнаруживается нажатие датчика касания, семафор выставляется в 1 и выполняется процедура отката назад. Во время неё задача move\_square обязана ждать. Когда откат выполнен - семафор выставляется снова в 0 и задача move\_square может продолжить.

int sem;

task move\_square()

{

while (true)

{

until (sem == 0); sem = 1;

OnFwd(OUT\_AC, 75);

sem = 0;

Wait(1000);

until (sem == 0); sem = 1;

OnRev(OUT\_C, 75);

sem = 0;

Wait(850);

}

}

task submain()

{

SetSensor(IN\_1, SENSOR\_TOUCH);

while (true)

{

if (SENSOR\_1 == 1)

{

until (sem == 0); sem = 1;

OnRev(OUT\_AC, 75); Wait(500);

OnFwd(OUT\_A, 75); Wait(850);

sem = 0;

}

}

}

task main()

{

sem = 0;

Precedes(move\_square, submain);

}

Вы можете сказать, что не ясно зачем в задаче move\_square мы выставляем семафор в 1 и обратно в 0. Но на самом деле это полезно. Причина в том, что команда OnFwd() на самом деле состоит из двух команд (см. главу 8). Если вы не хотите, чтобы эту пару команд прервала другая задача - нужно так же блокировать здесь доступ к устройству.

Семафоры очень полезны и, когда вы пишете достаточно сложные программы с параллельными задачами, они почти всегда будут вам нужны. (Кстати, существует некоторый небольшой шанс, что такой семафор может не сработать. Попробуйте разобраться, почему.)

## Подводим итоги

В этой главе мы научились решать некоторые проблемы, которые могут возникнуть при параллельной работе нескольких задач. Всегда будьте внимательны к возможным побочным эффектам при такой работе. Очень много случаев незапланированной работы программ возникает при работе с параллельными задачами. Мы увидели два возможных пути решения таких проблем. Первый заключается в том, что задачи выполняются в критической своей области по очереди. Второй подход использует семафоры для управления порядком выполнения команд. Это гарантирует что только одна задача в каждый момент времени будет находиться в своей критической области связанной с соответствующим общим ресурсом.

# Коммуникации между роботами

Если у вас есть больше чем один модуль NXT, тогда эта глава для вас (хотя вообще можно устанавливать связь не только между роботами, но и робота с ПК). Роботы могут связываться друг с другом через радиоканал Bluetooth: так что вы можете иметь нескольких роботов одновременно выполняющих общую задачу (или сражающихся друг с другом), кроме того вы можете построить большого сложного робота из двух NXT, так что можно будет использовать 6 сервомоторов и 8 сенсоров. Для старого доброго RCX, это очень просто: он отправляет ИК-сообщение и все роботы вокруг принимают его.

Но в NXT реализован совершенно другой подход! Во-первых вам нужно соединить два или более NXT (или NXT и ПК) с помощью меню "Bluetooth" на модуле NXT; только тогда вы сможете отправлять сообщения к подключенным устройствам. NXT который инициирует подключение называется Мастером, и может иметь до 3 подчиненных устройств, подключенных к линиям 1,2 и 3. Подчиненный модуль всегда видит мастера на линии 0. Вы можете отправлять сообщения в 10 доступных ячеек.

## Отправка сообщений мастер-подчиненный

Ниже приведены 2 программы - одна для мастера, а вторая для подчиненного. Эти простые программы покажут вам, как быстрый непрерывный поток строковых сообщений может быть передан с помощью радиоканала между двумя модулями NXT. Программа для мастера сначала убеждается, что подчиненный правильно подключен на 1 линию (константа BT\_CONN) используя функцию BluetoothStatus(conn); после чего создает и отправляет сообщения состоящие из префикса "M" и возрастающих чисел с помощью функции SendRemoteString(conn,queue,string), одновременно получая сообщения от подчиненного функцией ReceiveRemoteString(queue,clear,string) и отображая эту информацию.

//MASTER

#define BT\_CONN 1

#define INBOX 1

#define OUTBOX 5

sub BTCheck(int conn){

if (!BluetoothStatus(conn)==NO\_ERR){

TextOut(5,LCD\_LINE2,"Error");

Wait(1000);

Stop(true);

}

}

task main(){

string in, out, iStr;

int i = 0;

BTCheck(BT\_CONN); //check slave connection

while(true){

iStr = NumToStr(i);

out = StrCat("M",iStr);

TextOut(10,LCD\_LINE1,"Master Test");

TextOut(0,LCD\_LINE2,"IN:");

TextOut(0,LCD\_LINE4,"OUT:");

ReceiveRemoteString(INBOX, true, in);

SendRemoteString(BT\_CONN,OUTBOX,out);

TextOut(10,LCD\_LINE3,in);

TextOut(10,LCD\_LINE5,out);

Wait(100);

i++;

}

}

Программа подчиненного модуля очень похожа, но использует функцию SendResponseString(queue,string) вместо SendRemoteString, потому что подчиненный может отправлять сообщения только мастеру, находящемуся на линии 0.

//SLAVE

#define BT\_CONN 1

#define INBOX 5

#define OUTBOX 1

sub BTCheck(int conn){

if (!BluetoothStatus(conn)==NO\_ERR){

TextOut(5,LCD\_LINE2,"Error");

Wait(1000);

Stop(true);

}

}

task main(){

string in, out, iStr;

int i = 0;

BTCheck(0); //check master connection

while(true){

iStr = NumToStr(i);

out = StrCat("S",iStr);

TextOut(10,LCD\_LINE1,"Slave Test");

TextOut(0,LCD\_LINE2,"IN:");

TextOut(0,LCD\_LINE4,"OUT:");

ReceiveRemoteString(INBOX, true, in);

SendResponseString(OUTBOX,out);

TextOut(10,LCD\_LINE3,in);

TextOut(10,LCD\_LINE5,out);

Wait(100);

i++;

}

}

Как можно заметить, при отключении программы на одно из сторон, вторая сторона продолжит отправлять сообщения с возрастающими номерами, не зная, что все сообщения теряются, потому что на той стороне никто их не принимает. Чтобы избежать этой проблемы мы планируем улучшить протокол с помощью подтверждения доставки.

## Отправка чисел с подтверждением

Как мы видим в приведенной ниже паре программ: в этот раз мастер отправлять числа командой SendRemoteNumber(conn,queue,number) и останавливается, ожидая подтверждения получения от подчиненного (цикл until, внутри которого мы видим ReceiveRemoteString); только если на той стороне подчиненный работает и отправил нам подтверждение - мастер переходит к отправке следующего сообщения. Подчиненный просто получает числа фукнцией ReceiveRemoteNumber(queue,clear,number) и отправляет подтверждение командой SendResponseNumber. Обе программы должны иметь общий код для подтверждения. В нашем случае я принял 0xFF в качестве такого кода.

Мастер отправляет случайные числа и ждёт подтверждение подчиненного; каждый раз после этого переменная ack должна быть очищена, иначе мастер продолжит отправлять сообщения не ожидая больше никаких подтверждений, потому что мы забыли очистить эту переменну.

Подчиненный в бесконечном цикле проверяет ячейку и если она не пуста, показывает значение на экран и отправляет мастеру подтверждение о получении. В начале программы я отправил подтверждение без информации, которую подтверждаю, чтобы разблокировать мастера. Без этого, если бы мастер запустился раньше - первое его сообщение было бы отправлено в никуда и он бы начал ждать ответа, а подчиненный запустился бы и начал ждать первое сообщение, так что всё бы "зависло", даже если подчиненный нормально запущен. То как мы сделали сейчас - может привести к потере части сообщений, но хотя-бы нет риска зависания пары модулей.

//MASTER

#define BT\_CONN 1

#define OUTBOX 5

#define INBOX 1

#define CLEARLINE(L) \

TextOut(0,L," ");

sub BTCheck(int conn){

if (!BluetoothStatus(conn)==NO\_ERR){

TextOut(5,LCD\_LINE2,"Error");

Wait(1000);

Stop(true);

}

}

task main(){

int ack;

int i;

BTCheck(BT\_CONN);

TextOut(10,LCD\_LINE1,"Master sending");

while(true){

i = Random(512);

CLEARLINE(LCD\_LINE3);

NumOut(5,LCD\_LINE3,i);

ack = 0;

SendRemoteNumber(BT\_CONN,OUTBOX,i);

until(ack==0xFF) {

until(ReceiveRemoteNumber(INBOX,true,ack) == NO\_ERR);

}

Wait(250);

}

}

//SLAVE

#define BT\_CONN 1

#define OUT\_MBOX 1

#define IN\_MBOX 5

sub BTCheck(int conn){

if (!BluetoothStatus(conn)==NO\_ERR){

TextOut(5,LCD\_LINE2,"Error");

Wait(1000);

Stop(true);

}

}

task main(){

int in;

BTCheck(0);

TextOut(5,LCD\_LINE1,"Slave receiving");

SendResponseNumber(OUT\_MBOX,0xFF); //unblock master

while(true){

if (ReceiveRemoteNumber(IN\_MBOX,true,in) != STAT\_MSG\_EMPTY\_MAILBOX) {

TextOut(0,LCD\_LINE3," ");

NumOut(5,LCD\_LINE3,in);

SendResponseNumber(OUT\_MBOX,0xFF);

}

Wait(10); //take breath (optional)

}

}

## 

## Прямые команды

Вот еще одна отличная возможность доступная при использовании Bluetooth: мастер может напрямую контролировать подчиненных. В этом примере мастер отправляет подчиненным прямые команды для проигрывания звуков, включения двигателей и для подчиненного модуля NXT даже не нужно писать никакой программы. Это всё реализуется встроенным в NXT программным обеспечением!

//MASTER

#define BT\_CONN 1

#define MOTOR(p,s) RemoteSetOutputState(BT\_CONN, p, s, \

OUT\_MODE\_MOTORON+OUT\_MODE\_BRAKE+OUT\_MODE\_REGULATED, \

OUT\_REGMODE\_SPEED, 0, OUT\_RUNSTATE\_RUNNING, 0)

sub BTCheck(int conn){

if (!BluetoothStatus(conn)==NO\_ERR){

TextOut(5,LCD\_LINE2,"Error");

Wait(1000);

Stop(true);

}

}

task main(){

BTCheck(BT\_CONN);

RemotePlayTone(BT\_CONN, 4000, 100);

until(BluetoothStatus(BT\_CONN)==NO\_ERR);

Wait(110);

RemotePlaySoundFile(BT\_CONN, "! Click.rso", false);

until(BluetoothStatus(BT\_CONN)==NO\_ERR);

//Wait(500);

RemoteResetMotorPosition(BT\_CONN,OUT\_A,true);

until(BluetoothStatus(BT\_CONN)==NO\_ERR);

MOTOR(OUT\_A,100);

Wait(1000);

MOTOR(OUT\_A,0);

}

## Подводим итоги

В этой главе вы изучили некоторые простые аспекты использования Bluetooth для связи между роботами. Мы соединили по радиоканалу 2 модуля NXT, отправляли и принимали строки и числа, ожидали подтверждение получения. Последнее очень важно в случае необходимости в надежном канале передачи информации.

В качестве дополнительной возможности вы также научились отдавать мастером прямые команды для подчиненного модуля NXT.

# Дополнительные возможности

NXC имеет несколько сервисных команд. В этой главе мы обсудим три типа таких команд: для работы с таймером, дисплеем и для использования файловой системы NXT.

## Таймеры

В модуле NXT есть таймер, который работает в постоянном режиме. Этот таймер увеличивается на единицу каждые 1/1000 секунды. Вы можете получить текущее значение таймера функцией CurrentTick(). Вот пример использования таймера, в котором робот будет двигаться случайным образом 10 секунд.

task main()

{

long t0, time;

t0 = CurrentTick();

do

{

time = CurrentTick()-t0;

OnFwd(OUT\_AC, 75);

Wait(Random(1000));

OnRev(OUT\_C, 75);

Wait(Random(1000));

}

while (time<10000);

Off(OUT\_AC);

}

Можно попробовать сравнить эту программу с приведенной в главе 4, делающей то же самое. На таймере это делается определенно проще.

Таймеры очень полезны в качестве заменителя команд Wait(). Вы можете ожидать нужного момента времени сбросив таймер и ожидая нужного его значения. Но вы так же можете в это время реагировать на происходящие события (например на поступление информации от сенсоров), пока ждете. Следующая простая программа является примером такого использования таймеров. Она двигает робота вперед либо пока не пройдёт 10 секунд, либо пока у него не сработает бампер, упершись во что-нибудь.

task main()

{

long t3;

SetSensor(IN\_1,SENSOR\_TOUCH);

t3 = CurrentTick();

OnFwd(OUT\_AC, 75);

until ((SENSOR\_1 == 1) || ((CurrentTick()-t3) > 10000));

Off(OUT\_AC);

}

Не забывайте, что таймер работает с шагом 1/1000 секунды, как и команда wait.

## Дисплей

Модуль NXT имеет черно-белый графический дисплей с разрешением 100х64 пиксела. К нему есть множество функций API для рисования на нём текста, чисел, точек, линий, прямоугольников, кругов и даже картинок (из .ric-файлов). Следующий пример пытается покрыть все эти случаи. Пиксель (0,0) находится внизу слева экранчика.

#define X\_MAX 99

#define Y\_MAX 63

#define X\_MID (X\_MAX+1)/2

#define Y\_MID (Y\_MAX+1)/2

task main(){

int i = 1234;

TextOut(15,LCD\_LINE1,"Display", true);

NumOut(60,LCD\_LINE1, i);

PointOut(1,Y\_MAX-1);

PointOut(X\_MAX-1,Y\_MAX-1);

PointOut(1,1);

PointOut(X\_MAX-1,1);

Wait(200);

RectOut(5,5,90,50);

Wait(200);

LineOut(5,5,95,55);

Wait(200);

LineOut(5,55,95,5);

Wait(200);

CircleOut(X\_MID,Y\_MID-2,20);

Wait(800);

ClearScreen();

GraphicOut(30,10,"faceclosed.ric"); Wait(500);

ClearScreen();

GraphicOut(30,10,"faceopen.ric");

Wait(1000);

}

Все эти функции достаточно очевидны, но я всё-таки опишу их параметры детально:

* ClearScreen() очищает экран;
* NumOut(x, y, number) выводит число в указанные координаты;
* TextOut(x, y, string) тоже самое, только выводит строку
* GraphicOut(x, y, filename) выводит картинку из .ric-файла
* CircleOut(x, y, radius) выводит окружность с заданными координатами центра и радиусом;
* LineOut(x1, y1, x2, y2) рисует линию из (x1,x2) в (x2,y2)
* PointOut(x, y) ставит точку
* RectOut(x, y, width, height) рисует прямоугольник с нижним левым углом в (x,y) и указанных размеров;
* ResetScreen() сбрасывает экран.

## Файловая система

Модуль NXT может читать и писать файлы, имеющиеся у него в флеш-памяти. Так что вы можете создать лог-файл с датчиков или читать числа во время выполнения программы. Единственное ограничение в количестве и размере файлов - это размер флеш-памяти. Для работы с файлами используются функции API, которые позволяют создавать, переименовывать, удалять и искать файлы, а также с конкретным файлом вы можете читать и писать в него текстовые строки, числа и отдельные байты. В следующем примере мы посмотрим, как создать файл, записать в него несколько строк и переименовать этот файл.

Сначала, программа удаляет файлы с именами, которые мы хотим использовать: вообще это плохая практика (приличнее будет сначала проверить существование файла, вручную удалить его, или выбрать другое имя для нашего рабочего файла), но в нашей простой ситуации мы можем не обращать внимание на это. Дальше программа создает наш файл функцией CreateFile("Danny.txt", 512, fileHandle), указывая имя, размер и переменную, куда NXT поместит свой внутренний идентификатор открытого файла.

После чего программа готовит строки и пишет их в файл с возвратом каретки в конце функцией WriteLnString(fileHandle,string, bytesWritten), в которой все параметры должны быть переменными. И в конце, файл закрывается и переименовывается.

**Помните:** файл должен быть явно закрыт перед тем как начать следующую операцию, так что если вы создали файл, вы можете писать в него, если вы хотите прочитать из него - надо его закрыть и открыть на чтение командой OpenFileRead(). И перед удалением/переименованием файла он тоже должен быть закрыт.

#define OK LDR\_SUCCESS

task main(){

byte fileHandle;

short fileSize;

short bytesWritten;

string read;

string write;

DeleteFile("Danny.txt");

DeleteFile("DannySays.txt");

CreateFile("Danny.txt", 512, fileHandle);

for(int i=2; i<=10; i++ ){

write = "NXT is cool ";

string tmp = NumToStr(i);

write = StrCat(write,tmp," times!");

WriteLnString(fileHandle,write, bytesWritten);

}

CloseFile(fileHandle);

RenameFile("Danny.txt","DannySays.txt");

}

Чтобы увидеть, что получилось, зайдите через меню "BricxCC => Tools => NXT Explorer", в просмотрщик NXT-флеш-памяти,загрузите файл DannySays.txt на ПК и посмотрите, что внутри.

Посмотрите на следующий пример! Мы создадим таблицу ASCII-символов.

task main(){

byte handle;

if (CreateFile("ASCII.txt", 2048, handle) == NO\_ERR) {

for (int i=0; i < 256; i++) {

string s = NumToStr(i);

int slen = StrLen(s);

WriteBytes(handle, s, slen);

WriteLn(handle, i);

}

CloseFile(handle);

}

}

Правда просто? Эта программа создает файл и если при этом не произошла ошибка - пишет в него числа от 0 до 255 (конвертируя их предварительно в соответствующие символы функцией WriteBytes(handle, s, slen), которая предоставляет путь писать в файл строки без возврата каретки); после чего пишет это же число функцией WriteLn(handle, value), которая уже добавляет в конец записанной строки возврат каретки.

Результат, который вы видите при открытии файла ASCII.txt текстовым редактором (например Блокнотом из Windows), содержит в каждой строке символ и его код записанный в десятичном виде.

Остались еще две важные функции, которые вы должны знать: ReadLnString для чтения строк из файлов и ReadLn для чтения чисел.

Теперь для примера использования первой функции мы создадим программу, в которой из главной задачи будет вызвана функция CreateRandomFile, создающая файл со случайными числами внутри (записанными в строковом представлении); вы можете закомментировать эту строку и использовать любой другой текстовый файл для этого примера.

После этого главная задача откроет этот файл на чтение, и до конца файла будет читать строку из файла, вызывая ReadLnString и показывать прочитанный текст.

В функции CreateRandomFile мы генерируем заданное количество случайных чисел, преобразуем их в строки и пишем их в файл.

Функция ReadLnString в качестве аргументов имеет внутренний идентификатор открытого файла и строковую переменную, в которую после вызова функции будет прочитанная строка, а функция вернет код ошибки, который мы используем для того, чтобы понять, что достигли конца файла.

#define FILE\_LINES 10

sub CreateRandomFile(string fname, int lines){

byte handle;

string s;

int bytesWritten;

DeleteFile(fname);

int fsize = lines\*5;

//create file with random data

if(CreateFile(fname, fsize, handle) == NO\_ERR) {

int n;

repeat(FILE\_LINES) {

int n = Random(0xFF);

s = NumToStr(n);

WriteLnString(handle,s,bytesWritten);

}

CloseFile(handle);

}

}

task main(){

byte handle;

int fsize;

string buf;

bool eof = false;

CreateRandomFile("rand.txt",FILE\_LINES);

if(OpenFileRead("rand.txt", fsize, handle) == NO\_ERR) {

TextOut(10,LCD\_LINE2,"Filesize:");

NumOut(65,LCD\_LINE2,fsize);

Wait(600);

until (eof == true){ // read the text file till the end

if(ReadLnString(handle,buf) != NO\_ERR) eof = true;

ClearScreen();

TextOut(20,LCD\_LINE3,buf);

Wait(500);

}

}

CloseFile(handle);

}

В последней программе я покажу вам, как читать числа из файла. Кроме того я покажу вам пример условной компиляции. В начале кода мы определим значение, которое не будет использоваться ни как макрос, ни как псевдоним, мы просто определим INT.

Вот директива препроцессора

#ifdef INT

…Code…

#endif

которая сообщает компилятору, что компилировать код внутри "#ifdef" нужно только если предварительно было определено значение INT. Так что если мы в начале программы объявим INT, скомпилируется задача "main" из первой половины кода, а если же мы определим LONG вместо INT, тогда будет скомпилирована вторая версия задачи "main".

Этот метод позволил мне в одной программе показать вам, как работать одной и той же функцией ReadLn(handle,val) с обоими типами чисел - int (16 бит) и long (32 бита).

Как и предыдущая, эта команда принимает в качестве аргументов идентификатор открытого файла и числовую переменную, возвращая код ошибки.

Функция прочитает 2 байта из файла, если переданная переменная типа int, и 4 байта, если переменная типа long. Таким же образом вообще говоря можно читать и писать переменные типа bool.

#define INT // INT or LONG

#ifdef INT

task main () {

byte handle, time = 0;

int n, fsize,len, i;

int in;

DeleteFile("int.txt");

CreateFile("int.txt",4096,handle);

for (int i = 1000; i<=10000; i+=1000){

WriteLn(handle,i);

}

CloseFile(handle);

OpenFileRead("int.txt",fsize,handle);

until (ReadLn(handle,in)!=NO\_ERR){

ClearScreen();

NumOut(30,LCD\_LINE5,in);

Wait(500);

}

CloseFile(handle);

}

#endif

#ifdef LONG

task main () {

byte handle, time = 0;

int n, fsize,len, i;

long in;

DeleteFile("long.txt");

CreateFile("long.txt",4096,handle);

for (long i = 100000; i<=1000000; i+=50000){

WriteLn(handle,i);

}

CloseFile(handle);

OpenFileRead("long.txt",fsize,handle);

until (ReadLn(handle,in)!=NO\_ERR){

ClearScreen();

NumOut(30,LCD\_LINE5,in);

Wait(500);

}

CloseFile(handle);

}

#endif

## Подводим итоги

В этой последней главе мы познакомились с дополнительными функциями модуля NXT: таймером высокого разрешения, графическим дисплеем и файловой системой.

# Как определить цвет, когда датчик "врет"?

|  |  |
| --- | --- |
| При работе с датчиками цвета разных производителей случаются ситуации, когда датчик не определяет цвет корректно. Например, в тех ситуациях, когда на определение влияет недостаток освещенности или наоборот - слишком много света. Еще такое может случаться, когда механизм движется слишком быстро или нет какого-то фиксированного расстояния до предмета, цвет которого датчик пытается определить - например, когда робот движется по направлению к этому предмету. |  |

Один из способов, победить такое поведение датчика - снять показания с каждой из трех цветовых компонент: красной, зеленой и синей, так чтобы по их значения самостоятельно (собственным алгоритмом) попытаться определить какого же цвета предмет. Такой способ уже был описан вот в этом материале. Но дело в том, что подразумевается, что у нас есть достаточно много времени для определения цвета, производя последовательно 3 или даже 4 (замер освещенности без подсветки вообще), и датчик и предмет неподвижны относительно друг друга. А это не всегда возможно достичь, особенно в условиях робототехнических состязаний - когда время работы робота является критичной величиной.

Для языка программирования Not eXactly C, тоже есть подобный инструментарий, позволяющий получить значения цветовых компонент за раз. Делается это с помощью функции ReadSensorColorRaw.

**task** main() {

SetSensorColorFull(S3);

**unsigned int** rawData[];

**int** result = ReadSensorColorRaw(S3, rawData);

NumOut(8, LCD\_LINE1, rawData[INPUT\_RED]);

NumOut(8, LCD\_LINE2, rawData[INPUT\_GREEN]);

NumOut(8, LCD\_LINE3, rawData[INPUT\_BLUE]);

**until**(ButtonPressed(BTNCENTER));

}

Данный функционал тоже производит измерения достаточно быстро:

**task** main() {

SetSensorColorFull(S3);

**unsigned int** rawData[];

**int** result;

**unsigned long** strt = CurrentTick();

**for**(**int** i=0; i<5000; i++) {

result = ReadSensorColorRaw(S3, rawData);

NumOut(8, LCD\_LINE1, rawData[INPUT\_RED]);

NumOut(8, LCD\_LINE2, rawData[INPUT\_GREEN]);

NumOut(8, LCD\_LINE3, rawData[INPUT\_BLUE]);

}

NumOut(8, LCD\_LINE5, CurrentTick() - strt);

**until**(ButtonPressed(BTNCENTER));

}

Результат работы - 5000 итераций за 6,2 секунды.  
Если необходимо сделать за один вызов и определение цвета и получить значения компонент, то можно использовать функцию SysColorSensorRead.

# NXC: кнопочная интерактивность

|  |  |
| --- | --- |
| Довольно часто те, кто начинает знакомится с языком NXC, сперва обращаются к широкоизвестному NXC Tutorial (на русском), написанному Daniele Benedettelli. Это довольно хорошее для первого знакомства с этим языком программирования пособие и дает представление о многих базовых возможностях языка. |  |

Но тем не менее, некоторые широкоиспользуемые базовые области, к сожалению, пропущены. Например, цветовой сенсор и работа с кнопками NXT блока. И если работа в NXC с цветовым сенсором была уже освещена ранее, то пора закрыть пробел и во взаимодействии с кнопками.

**task** main ()

{

//Вторая переменная, чтобы отслеживать изменения первой

**int** count = 0, oldcount = -1;

//Флаг, отслеживающий нажатие центроальной кнопки

**bool** flExit = **false**;

TextOut(0, LCD\_LINE2, *"Press"*);

TextOut(0, LCD\_LINE3, *"<, > or Enter"*);

**while** (!flExit)

{

//Если значение счетчика поменялось - вывести на экран

**if** (count != oldcount)

{

oldcount = count;

NumOut(10, LCD\_LINE4, count);

}

//Нажата ли правая кнопка - увеличить счетчик

**if** (ButtonPressed(BTNRIGHT, **true**))

{

**if** (count < 7)

count++;

}

//Нажата ли левая кнопка - уменьшить счетчик

**else if** (ButtonPressed(BTNLEFT, **true**))

{

**if** (count > 0)

count--;

}

//Нажата ли центральная кнопка - выйти из цикла

**else if** (ButtonPressed(BTNCENTER, **true**))

flExit = **true**;

//Небольшая пауза - короткое нажатие различается от долгого

Wait(200);

}

}

# NXC: работаем с датчиками вращения мотора

|  |  |
| --- | --- |
| При программировании Lego-роботов иногда необходимо проверить на сколько повернулся мотор. Например, чтобы определить не застрял ли он. |  |

NXC учебник, идущий в комплекте с данным языком программирования, не описывает работу с датчиками вращения мотора. А в NXC справочнике, также идущем в комплекте, не так уж легко ориентироваться, чтобы найти необходимые функции. А их на самом деле не много.  
  
На самом деле, чаще всего, все ограничится использованием двух из них:

* MotorRotationCount - позволяет получить текущее значение с датчика вращения соответствующего мотора.
* ResetRotationCount - сбрасывает текущее значение датчика вращения соответствующего мотора в 0.

Имеет смысл пояснить использование этих двух функций примером.

**sub** MoveHeadBack()

{

OnRev(HEAD, HEAD\_SPEED\_REV);

**int** Start;

**do** {

Start = MotorRotationCount(HEAD);

Wait(100);

} **while** (abs(MotorRotationCount(HEAD) - Start) > 5)

Coast(HEAD);

Wait(200);

ResetRotationCount(HEAD);

Wait(1);

}

Эта функция предназначена для того, чтобы сдвинуть выбрасывающий механизм робота для игры в крестики-нолики назад.  
Поскольку до запуска этой подпрограммы неизвестно, где находися механизм, он должен двигаться наза до тех пор пока не упрется в соответствующие ограничители движения.  
Именно это происходит в цикле - через каждые 100 милисекунд проверяется на сколько повернулась ось двигателя со времени последнего измерения, и если поворот произошел меньше чем на 5 градусов, то значит выбрасывающий механизм уперся в ограничитель, и его нужно останавливать.  
  
В конце функции происходит сброс счетчика поворотов мотора (просто для примера, на самом деле он здесь не обязателен).  
Заметьте, что после сброса стоит пауза в 1 милисекунду, наличие которой может вызвать обоснованные вопросы.  
  
Дело в том, что подобно функциям OnFwd() и OnRev(), сброс счетчика имеет асинхронную природу. Функция не дожидается реального сброса показаний сенсора, а просто модифицирует нужную область памяти, подразумевая, что модуль Output, запущенный виртуальной машиной блока NXT, обработает изменение памяти соответствующим образом, что приведет к реальному сбросу счетчиков.  
Так вот, по своей природе, виртуальная машина NXT блока позволяет запускаться всем модулям не чаще, чем один раз в 1 мс. И если ResetRotationCount() вызывается в самом конце милисекундного интервала, то Output модуль вызовется практически сразу же после этой функции, что приведет почти к мгновенному сбросу счетчиков. А если ResetRotationCount() вызывается в начале этого интервала, то последующий вызов, например, MotorRotationCount() будет показывать старые значения и будет казаться, что ResetRotationCount() как будто бы не работает (англоязычное объяснение, описанного выше, можно найти на багртрекере BricxCC).  
Те, кто привык все понимать через анализ рабочего кода, могут рассмотреть два примера:

* Внешне выглядит, что показания счетчика не сбрасываются
* OnRev(OUT\_A, 50);
* Wait(1000);
* TextOut(0, LCD\_LINE1, MotorRotationCount(OUT\_A));
* ResetRotationCount(OUT\_A);

TextOut(0, LCD\_LINE2, MotorRotationCount(OUT\_A));

* Добавление паузы решает проблему
* OnRev(OUT\_A, 50);
* Wait(1000);
* TextOut(0, LCD\_LINE1, MotorRotationCount(OUT\_A));
* ResetRotationCount(OUT\_A);
* Wait(1);

TextOut(0, LCD\_LINE2, MotorRotationCount(OUT\_A));

# 

# NXC: работаем с цветовым сенсором.

|  |  |
| --- | --- |
| Среда NXT-G привлекательна тем, что она позволяет сосредоточиться программисту на самом алгоритме, и практически не задумываться о таких вещах, свойственных текстовым языкам, как инициализация, синтаксис и т.п. Она предоставляет простой и удобный доступ практически ко всем возможностям моторов и сенсоров, без необходимости помнить вызовом какой процедуры или функции это происходит. |  |

Поэтому, если вы программируете на текстовых средах, довольно часто для поиска решения проблемы по работе какого-то сенсора, которая бы на NXT-G не заняла бы у вас и одной минуты, необходимо потратить значительное количество времени.  
Так, например, работая с языком NXC, вы еще достаточно легко можете найти готовый код для работы со стандартным цветовым сенсором:

**task** main()

{

SetSensorColorFull(S1);

ColorSensorReadType csr;

csr.Port = S1;

**while**(1)

{

SysColorSensorRead(csr);

**if** (csr.Result == NO\_ERR) {

NumOut(0, LCD\_LINE1, csr.ColorValue);

}

Wait(1000);

}

}

В то время как, поиск примера по работе с ним в режиме светового сенсора займет определенное количество времени.  
Чтобы облегчить эту задачу, можете изучить приведенный ниже пример:

**task** main()

{

SetSensorColorRed(S1);

Wait(500);

**while**(1)

{

NumOut(0, LCD\_LINE1, SENSOR\_1);

Wait(300);

}

}

И на будущее, если у вас возникли какаие-то вопросы связанные с использованием того или иного языка программирования - идите на форум The MindBOARDS Community. И если вы там не найдете готовый ответ - не стесняйтесь, спрашивайте. Там сидит довольно много широко известных в Lego мире разработчиков, которые смогут дать вам квалифицированный ответ.

# NXC: датчики, енкодеры, кнопки - взаимодействуем с внешним миром

|  |  |
| --- | --- |
|  | Программирование робота, в большинстве случаев, - изменение поведения робота как реакция на информацию, пришедшую к управляющему устройству с датчиков и сенсоров. Эта заметка рассматривает примеры опроса датчиков в языке Not eXactly C, а также некоторые особенности, которые необходимо учитывать при работе с ними. |

Пример 1. Датчик расстояния  
Для датчика расстояния (ультразвуковой) используются специальные функции для его инициализации и опроса.

**task** main() {

*/\**

*Робот ждет пока перед ним не окажется предмет (ближе чем 20 см.), после*

*чего отодвигается от предмета на 50 см.*

*\*/*

*//Говорим, что медленный (ультразвуковой) сенсор установлен в первый*

*//порт.*

SetSensorLowspeed(S1);

Wait(100);

**while** (**true**) {

*//Опрашиваем порт первый, как ультразвуковой датчик*

*//Ждем, пока расстояние до предмета не станет меньше 20 см.*

**until** (SensorUS(S1) < 20);

*//Отодвигаемся назад до тех пор, пока расстояние до предмета не*

*//станет больше 50 см.*

OnRev(OUT\_AB, 50);

**until** (SensorUS(S1) > 50);

*//Явно останавливаем двигатели*

Off(OUT\_AB);

}

}

Пример 2. Измерение отраженного света.  
Работа датчика в режиме измерения отраженного света (пространство перед датчиком дополнительно освещается светодиодом) позволяет повысить его чувствительность. Без использования этого режима, решение такой задачи как распознавание цвета или градаций серого была бы довольно трудно выполнима – обычно окружающий свет не имеет достаточной интенсивности, поэтому соседние градации цветов сливались бы один цвет при измерении. К тому же, измерения при неравномерном освещении давали бы разный результат для одного и того же предмета. Поэтому использование дополнительной подсветки – наиболее частый вариант при управлении роботом с помощью датчика освещенности.

**task** main() {

*/\**

*Робот двигается пока поверхность под световым сенсором не потемнеет.*

*\*/*

*//Говорим, что cветовой сенсор установлен в третий порт.*

SetSensorLight(S3);

Wait(500);

*//Двигаемся вперед, пока значение сенсора в третьем порту не станет*

*//меньше 55 процентов.*

OnFwd(OUT\_AB, 50);

**until**(Sensor(S3) < 55);

Off(OUT\_AB);

}

Пример 3. Измерение окружающего света  
В некоторых задачах, дополнительная подсветка только мешает. Например, определить какое место в комнате самое темное. Для этого нужно измерять именно окружающее освещение.

**task** main() {

*/\**

*Робот отображает на экране текущую освещенность.*

*\*/*

*//Говорим, что cветовой сенсор установлен в третий порт*

*//лампа подсветки - неактивна*

SetSensorType(S3, SENSOR\_TYPE\_LIGHT\_INACTIVE);

*//Сенсор, устанавливаемый через SetSensorType по-умолчанию показывает*

*//ненормализованные данные (не путать с RAW). Переводим, его в режим для*

*//считывания процентов.*

SetSensorMode(S3, SENSOR\_MODE\_PERCENT);

Wait(500);

**while**(**true**) {

*//Выводим на эркан текущие показатели сенсора, перед выводом -*

*//очищаем экран (4-ый параметр - true)*

NumOut(0, LCD\_LINE1, Sensor(S3), **true**);

Wait(50);

}

}

Пример 4. Использование необработанных данных  
По умолчанию, датчики выдают измерения в процентах. В то время как NXC предоставляет возможность использовать необработанные (RAW) данные. Преимущество – очевидно – поскольку необработанные данные могут быть в диапазоне от 0 до 1023, измерения будут в 10 раз чувствительнее.

**task** main() {

*/\**

*Робот отображает на экране показания отраженного света.*

*Темная поверхность:*

*1-ая строка (проценты): значения маленькие (<50%)*

*2-ая строка (raw): значения большие (>600)*

*Светлая поверхность:*

*1-ая строка (проценты): значения маленькие (>50%)*

*2-ая строка (raw): значения большие (<400)*

*\*/*

*//Говорим, что cветовой сенсор установлен в третий порт*

SetSensorLight(S3);

Wait(500);

**while**(**true**) {

*//Выводим на эркан в первой строке текущие показатели сенсора в*

*//процентах, перед выводом - очищаем эркан (4-ый параметр - true)*

NumOut(0, LCD\_LINE1, Sensor(S3), **true**);

*//Выводим на эркан во второй строке текущие необработанные показатели*

*//сенсора, экран не очищается (4-ый параметр не указывается - берется*

*//значение по-умолчанию false)*

NumOut(0, LCD\_LINE2, SensorRaw(S3));

*//Чтобы избежать мерцания цифр на экране, делаем небольшую паузу.*

Wait(50);

}

}

Пример 5. Датчик вращения двигателя (енкодер)  
Поскольку датчик вращения двигателя (енкодер) скрыт от глаз конструктора, часто о его существовании забывают. Тем не менее, с помощью этого датчика можно решать множество задач: для определения столкновения с препятствием, для синхронизированного движения моторов, для контроля положения двигателя в одном положении с использованием регуляторов и т.п.

**task** main() {

*/\**

*Робот начинает двигаться, затем, в цикле дважды опрашивается датчик*

*поворота двигателя с небольшой паузой.*

*Поскольку пауза происходит параллельно движению двигателя (использование*

*функции OnFwd), оценивается поворот двигателя, произошедший во время*

*этой паузы.*

*Если поворот меньше ожидаемого, то скорее всего робот наткнулся на*

*препятствие - препятствие мешает провернуться двигателю. После*

*обнаружения такой ситуации - движение прекращается.*

*\*/*

**int** Start;

**int** Diff;

**int** Desired;

*//Переменная используется, чтобы при первой итерации цикла определить*

*//поворот двигателя во время паузы и взять этот поворот за эталон*

**int** iter = 0;

OnFwd(OUT\_AB, 40);

**do** {

*//Считываем показание датчика поворота перед паузой*

Start = MotorRotationCount(OUT\_A);

Wait(100);

*//Считываем показание датчика после паузы и определяем угол поворота*

Diff = MotorRotationCount(OUT\_A) - Start;

*//Если это первая итерация цикла - принять угол поворота за эталон*

**if** (iter == 0) {

Desired = Diff \* 8 / 10;

*//Указать, что больше в этот 'if' заходить не надо*

iter = 1;

}

} **while** (Diff > Desired) *//Если угол поворота двигателя меньше эталона*

Off(OUT\_AB);

}

Пример 6. Скорость опроса датчика расстояния  
При работе с датчиком расстояния необходимо помнить, что он является низкоскоростным устройством, т.е. опрос состояния этого датчика занимает определенное время. Это в первую очередь связано с физической природой процесса измерения расстояния: для того чтобы звуковая волна прошла путь до препятствия и вернулась обратно в датчик, определенно, необходимо какое-то время.

#define OneSecond 1000

**task** main() {

*/\**

*В каждой итерации цикла опрашивается датчик расстояния. Количество*

*итераций определяется выяснением, сколько миллисекунд прошло с начала*

*работы программы. Как только количество миллисекунд становится больше*

*1000 - цикл прерывается.*

*Поскольку ультразвуковой сенсор является низкоскоростным устройством,*

*его опрос вызывает задержку.*

*Результат показывает, что ультразвуковой сенсор может быть опрошен 34*

*раза за одну секунду.*

*Т.е. задержка опроса сенсора составляет - 1000/34 = 30 миллисекунд.*

*\*/*

*//Указываем, что низкоскоростной сенсор (ультразвуковой) установлен в*

*//первый порт*

SetSensorLowspeed(S1);

Wait(100);

*//Считываем начальное показание внутреннего таймера*

**unsigned long** Start = CurrentTick();

**unsigned long** Diff;

**unsigned long** i = 0, value;

**do** {

*//Опрашиваем сенсор, его показания не важны для данной программы,*

*//важен факт его вызова*

value = SensorUS(S1);

*//Вычисляем разницу между начальным показанием таймера и текущим*

Diff = CurrentTick() - Start;

i++;

} **while** (Diff < OneSecond) *//Повторять, пока разница - меньше секунды*

NumOut(0, LCD\_LINE1, i);

Wait(3000);

}

Пример 7. Определение цвета  
Работа с датчиком цвета практически не отличается от работы с датчиком освещенности, следует только помнить, что в датчике цвета светодиодов три.

**task** main()

{

*/Инициализация датчика в первом порту - включить все три светодиода*

SetSensorColorFull(S1);

**while**(1)

{

*//Опрашиваем датчик и выводим на экран определившийся цвет*

NumOut(0, LCD\_LINE1, Sensor(S1));

Wait(1000);

}

}

Пример 8. Опрос датчика цвета в режиме датчика освещенности  
В определенных задачах датчик цвета может заменить датчик освещенности.

**task** main()

{

*//Инициализация датчика в первом порту - включить красный светодиод*

SetSensorColorRed(S1);

**while**(1)

{

*//Опрашиваем датчик и выводим на экран значения отраженного*

*//света в процентах и "сырых" показаниях*

NumOut(0, LCD\_LINE1, Sensor(S1));

NumOut(0, LCD\_LINE2, SensorRaw(S1));

Wait(1000);

}

}

# NXC: использование циклов

|  |  |
| --- | --- |
| http://for-schoolboy.ru/media/images/Operatoryi-organizatsii-tsikla/15.jpg | Как и в обычном C, в NXC поддерживается несколько видов циклов: for, while, do ... while, repeat. |

Есть даже не совсем обычная команда until, которая является макросом к использованию while().  
  
В этой заметке основной акцент сделан не на стандартное поведение программы при использовании циклов, а на особенности, существующие в NXC.  
  
Пример 1. while() как функция  
В NXC while может быть использован и как стандартный цикл "выполнять пока", внутри которого выполняются различные действия, так и как функция "выполнять пока", которая возвращает управление программе, только тогда когда не выполняется условие, переданное в данную функцию.

**task** main() {

*/\**

*Программа ниже может быть описана следующим образом:*

*- начать движение прямо*

*- двигаться до тех пор пока до препятствия больше 10 см.*

*- остановится как только до препятствия осталось меньше (либо равно)*

*10 см.*

*Следует отметить, что команда "движение вперед" выполняется не цикле, но*

*в цикле уже опрашивается сенсор расcтояния.*

*\*/*

SetSensorLowspeed(S1);

Wait(100);

OnFwd(OUT\_AB, 50);

**while** (SensorUS(S1) > 10);

Off(OUT\_AB);

}

Пример 2. while() в одну строчку  
Реализация while() в NXC также поддерживает вызов с блоком повторения, записанным одной командой.

**task** main() {

*/\**

*Движение будет происходить до тех пор пока расстояние до препятствия*

*Больше 10 см. Остановка двигателей (Off) уже на другой строчке - он не*

*Выполняется в цикле.*

*Поскольку для движения используется RotateMotor(), который отдает*

*Управление следующей итерации цикла только после того как повернет*

*двигатели на 180 градусов, проверка на расстояние может произойти уже*

*когда до препятствия останется уже гораздо меньше расстояния. Т.е. за*

*эти 180 градусов, робот может пройти через рубеж в 10 см.*

*\*/*

SetSensorLowspeed(S1);

Wait(100);

**while** (SensorUS(S1) > 10)

RotateMotor(OUT\_AB, 50, 180);

Off(OUT\_AB);

}

Пример 3. until()  
Как уже было сказано ранее, until() – макрос к функции while(). По сути, он позволяет выполнять повторять предыдущее действие до тех пор, пока условие неверно.

**task** main() {

*/\**

*Движение назад будет происходить до тех пор, пока расстояние до*

*препятствия не станет больше 30 см.*

*\*/*

SetSensorLowspeed(S1);

Wait(100);

OnRev(OUT\_AB, 50);

**until** (SensorUS(S1) > 30);

Off(OUT\_AB);

}

Пример 4. Скорость исполнения цикла.  
В данном примере не рассматривается никакая особенность циклов. Зато он предназначен для получения ответа на вопрос "сколько итераций цикла исполняется в течение одной секунды".

#define OneSecond 1000

**task** main() {

*/\**

*Перед началом цикла засекаем текущее состояние таймера (Start).*

*Затем, проверяем разницу между этим значением и новым состоянием таймера*

*Как только разница - 1000 милисикунд - останавливаем цикл.*

*Полученные данные выводим на экран.*

*\*/*

Wait(100);

**unsigned long** Start = CurrentTick();

**unsigned long** Diff;

**unsigned long** i = 0;

**do** {

Diff = CurrentTick() - Start;

i++;

} **while** (Diff < OneSecond)

*// Координата X*

*// | Координата Y (задается константой, обозначающей номер строки)*

*// | | Значение для вывода на экран*

*// | | |*

NumOut(0, LCD\_LINE1, Start);

NumOut(0, LCD\_LINE2, i);

Wait(3000);

}

# NXC: повороты - практика программирования моторов

|  |  |
| --- | --- |
| При задании сложных траекторий движения робота развороты и повороты - неотъемлемая часть программы. В этой статье будут рассматриваться способы, как программируются эти элементы движения с помощью функций и команд языка Not eXactly C рассмотренных ранее. | http://upload.wikimedia.org/wikipedia/commons/thumb/e/e5/Korean_Traffic_sign_%28Left_Turn%29.svg/395px-Korean_Traffic_sign_%28Left_Turn%29.svg.png |

Пример 1. Поворот одним двигателем  
На самом деле в этом примере рассматриваются два способа поворота одним двигателем.  
В качестве входных данных для обоих способов, задается необходимость выполнения разворота вправо.  
  
Итак, первый способ.  
*Поворот одним двигателем вправо. Ось вращения - правое колесо.*

**task** main() {

*/\**

*Робот крутится вправо до тех пор, пока расстояние до препятствия не*

*станет меньше 20 см. Вращение происходит за счет того, что правый мотор*

*(OUT\_A) не двигается, а левый мотор (OUT\_B) двигается вперед. Поэтому*

*робот, как бы крутится вокруг правого колеса.*

*\*/*

SetSensorLowspeed(S1);

Wait(100);

*//Вместо двух моторов, указываем только один.*

OnFwd(OUT\_B, 50);

**until** (SensorUS(S1) < 20);

Off(OUT\_AB);

}

Второй способ –   
*Поворот вправо "задним ходом". Вращение вокруг левого колеса.*

**task** main() {

*/\**

*Робот крутится вправо до тех пор, пока расстояние до препятствия не*

*станет меньше 20 см. Вращение происходит за счет движения правого мотора*

*(OUT\_А) назад, в то время как левый мотор (OUT\_B) остается неподвижным.*

*Поэтому робот, как бы пятится назад вокруг левого колеса.*

*\*/*

SetSensorLowspeed(S1);

Wait(100);

*//Вместо двух моторов, указываем только один.*

*//Мотор двигается назад.*

OnRev(OUT\_A, 50);

**until** (SensorUS(S1) < 20);

Off(OUT\_AB);

}

Пример 2. Поворот двумя двигателями  
Основной недостаток поворота одним двигателем - небольшая скорость: усилия одного двигателя направлены и на перемещение тележки и на преодоление силы сопротивления оказываемой неподвижным колесом. Поэтому, популярностью пользуется быстрый поворот вокруг своей оси (точнее ось вращения проходит между колесами).  
В данном примере рассматривается поворот робота двумя двигателями влево.

**task** main() {

*/\**

*Робот крутится влево до тех пор, пока расстояние до преграды не станет*

*меньше 20 см. Вращение происходит за счет движения обеими моторами –*

*правый мотор двигается вперед, левый мотор двигается назад. За такое*

*распределение движения между моторами отвечает третий параметр в*

*OnFwdSync().*

*\*/*

SetSensorLowspeed(S1);

Wait(100);

*// Двигатели*

*// | мощность*

*// | | распределение нагрузки между моторами*

*// | | |*

OnFwdSync(OUT\_AB, 50, -100);

**until** (SensorUS(S1) < 20);

Off(OUT\_AB);

}

Пример 3. Распределение мощности при управлении двумя двигателями  
В прошлом примере вместе с функцией OnFwdSync() использовался параметр "распределение нагрузки между моторами". Этот параметр имеет смысл только тогда, когда происходит управление двумя двигателями. Он позволяет в одном числе выразить не только, насколько быстро будет крутиться один мотор относительно другого, но также и направление вращения.

**task** main() {

SetSensorLowspeed(S1);

Wait(100);

*/\**

*Первым действием робот вращается влево, правое колесо крутится медленней*

*левого.*

*\*/*

*// Двигатели*

*// | мощность*

*// | | угол поворота оси мотора в градусах*

*// | | | распределение движения между моторами*

*// | | | | синхронизация моторов*

*// | | | | | включать торможение после*

*// | | | | | | окончания движения или*

*// | | | | | | останавливаться своим ходом*

RotateMotorEx(OUT\_AB, 50, 360, -60, **true**, **true**);

Wait(1000);

*/\**

*Сейчас, робот вращается влево, правый мотор стоит, левый мотор двигается*

*назад.*

*\*/*

RotateMotorEx(OUT\_AB, 50, -360, -50, **true**, **true**);

Wait(1000);

*/\**

*Робот вращается влево двигаясь, правый мотор двигается быстрее левого.*

*\*/*

RotateMotorEx(OUT\_AB, 50, 360, -25, **true**, **true**);

Wait(1000);

*/\**

*Робот двигается без поворотов, оба колеса двигаются одинаково назад.*

*\*/*

RotateMotorEx(OUT\_AB, 50, -360, 0, **true**, **true**);

}

Пример 4. Альтернативный поворот двумя двигателями  
Предыдущий пример показал, что движение при использовании параметра "распределение нагрузки между моторами" по сути – работа двух двигателей с разной скоростью и разным направлением. Следовательно, его можно запрограммировать и без использования этого параметра.

**task** main() {

*/\**

*Робот крутится влево до тех пор, пока расстояние до преграды не станет*

*меньше 20 см.*

*Вращение вокруг своей оси происходит без использования команд, которые*

*принимают парамерт распределения движения между моторами.*

*Заместо этого, используется знание того, что команды OnFwd/OnRev отдают*

*управление следующим командам сразу после запуска моторов.*

*Таким образом, сначала запускается один двигатель, потом сразу же*

*запускается второй двигатель. Направление движения каждого двигателя и*

*мощность, подаваемая на каждый из них, задают направление и форму*

*поворота.*

*\*/*

SetSensorLowspeed(S1);

Wait(100);

*//Правый мотор двигается вперед*

OnFwd(OUT\_A, 50);

*//Левый мотор двигается назад с такой же мощностью как и правый.*

OnRev(OUT\_B, 50);

**until** (SensorUS(S1) < 20);

Off(OUT\_AB);

}

# Сортировка данных на NXT

|  |  |
| --- | --- |
|  |  |

В начале этой неделе я опубликовал вопрос: как в произвольном тексте найти 10 наиболее часто используемых слов? В программах для NXT обработка кучи данных с сенсоров является обыденным делом, но что если тебе надо узнать только 10 самых больших или маленьких значений? Как ты будешь решать этот вопрос? В языке RobotC нет специальных функций для сортировки; в NXC есть единственная ArraySort функция, упорядочивающая численные массивы по возрастанию. Поэтому в данной статье я решил провести небольшое исследование различных алгоритмов сортировки. Я помню эту классную демонстрацию работы алгоритмов, которая наглядно показывает различия в принципах сортировки. Я подумал, что будет реально здорово, если я сделаю тоже самое на NXC. И я сделал это.

### Пузырьковая сортировка

Это первый алгоритм, который я реализовал, потому что я подумал, что он выглядит круто. Так как же он работает? Представьте себе кучу неупорядоченных данных и самый большой элемент "всплывает" наверх, подобно пузырьку воздуха в кипящей воде. Ты можешь сделать это, сравнивая попарно каждый элемент с последующим и меняя их местами, если они расположены в неправильной последовательности. Таким образом, за один "проход" массива наибольший элемент окажется в  начале списка - первый "пузырек" всплыл. Эта процедура повторяется до тех пор, пока не "всплывут" все оставшиеся пузырьки - все элементы не будут отсортированы.  
Код этого алгоритма довольно простой:

**for**(**int** i = 0; i < MAX\_NUMBERS; i++)

{

**for**(**int** j = 0; j < (MAX\_NUMBERS - 1); j++)

{

**if** (arr[j] > arr[j+1])

{

temp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

}

}

}

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № итерации |  |  |  |  |  |  |
| Исходный массив | 10 | 5 | 2 | 7 | 1 | 9 |
| 1 | **5** | **10** | 2 | 7 | 1 | 9 |
| 2 | 5 | **2** | **10** | 7 | 1 | 9 |
| 3 | 5 | 2 | **7** | **10** | 1 | 9 |
| 4 | 5 | 2 | 7 | **1** | **10** | 9 |
| 5 | 5 | 2 | 7 | 1 | **9** | **10** |
| 6 | **2** | **5** | 7 | 1 | 9 | 10 |
| 7 | 2 | 5 | **1** | **7** | 9 | 10 |
| 8 | 2 | **1** | **5** | 7 | 9 | 10 |
| 9 | **1** | **2** | 5 | 7 | 9 | 10 |

Единственным его недостатком является то, что он достаточно медленный. Его вычислительная сложность равна O(n2). Это означает, что при увеличении массива время работы алгоритма будет расти экспоненциально. Например, массив из 11 показаний сенсоров будет отсортирован на 20% медленнее, чем массив из 10 элементов, а обработка 50 замеров потребуют в 25 раз больше времени.

### Сортировка вставками

Этот алгоритм сначала он сортирует два первых элемента массива. Затем алгоритм вставляет третий элемент в соответствующую порядку позицию по отношению к первым двум элементам. После этого он вставляет четвертый элемент в список из трех элементов. Таким образом, на каждом шаге новый элемент вставляется в нужную позицию в уже отсортированной части массива. Этот процесс повторяется до тех пор, пока не будут вставлены все элементы.

**for** (i = 1 ; i < MAX\_NUMBERS ; i++ )

{

**for** (j = 0 ; j < i ; j++ )

{

**if** ( arr[j] > arr[i] )

{

temp = arr[j] ;

arr[j] = arr[i] ;

**for** (k = i ; k > j ; k-- )

{

arr[k] = arr[k - 1] ;

}

arr[k + 1] = temp ;

}

}

}

В таблице ниже кроме самих итераций алгоритма я покажу, чему равнялась переменная temp.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| № итерации |  |  |  |  |  |  | temp |
| Исходный массив | 10 | 5 | 2 | 7 | 1 | 9 | - |
| 1 | **5** | **5** | 2 | 7 | 1 | 9 | 10 |
| 2 | 5 | **10** | 2 | 7 | 1 | 9 | - |
| 3 | **2** | 10 | **10** | 7 | 1 | 9 | 5 |
| 4 | **2** | **2** | 10 | 7 | 1 | 9 | 5 |
| 5 | 2 | **5** | 10 | 7 | 1 | 9 | - |
| 6 | 2 | 5 | **7** | **7** | 1 | 9 | 10 |
| 7 | 2 | 5 | 7 | **10** | 1 | 9 | - |
| 8 | **1** | 5 | 7 | 10 | **10** | 9 | 2 |
| 9 | 1 | 5 | 7 | **7** | 10 | 9 | 2 |
| 10 | 1 | 5 | **5** | 7 | 10 | 9 | 2 |
| 11 | 1 | **1** | 5 | 7 | 10 | 9 | 2 |
| 12 | 1 | **2** | 5 | 7 | 10 | 9 | - |
| 13 | 1 | 2 | 5 | 7 | **9** | **9** | 10 |
| 14 | 1 | 2 | 5 | 7 | 9 | **10** |  |

### Сортировка Шелла

Сортировка Шелла является модификацией алгоритма сортировки вставками. Разница вот в чем: вместо того, чтобы сравнивать соседние элементы, как в сортировке пузырьком, сортировка Шелла несколько раз сравнивает элементы, находящиеся на определенном расстоянии друг от друга (зададим расстояние буквой m). Это расстояние равняется половине размера входного массива и на каждой последующей итерации m уменьшается вдвое.

В нашем примере сначала будут сортироваться между собой элементы, отстоящие друг от друга на расстоянии 3 элемента (6 / 2 = 3), а затем на расстоянии 1-го элемента (3 / 2 = 1 - целочисленное деление).

**for**(m = MAX\_NUMBERS/2 ; m > 0; m /= 2)

{

**for**(j = m; j < MAX\_NUMBERS; j++)

{

**for**(i = j - m; i >= 0; i -= m)

{

**if**(arr[i + m] >= arr[i])

**break**;

**else**

{

mid = arr[i];

arr[i] = arr[i + m];

arr[i + m] = mid;

}

}

}

}

В таблице ниже вы можете увидеть, как это работает:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № итерации |  |  |  |  |  |  |
| Исходный массив | 10 | 5 | 2 | 7 | 1 | 9 |
| 1 | **7** | 5 | 2 | **10** | 1 | 9 |
| 2 | 7 | **1** | 2 | 10 | **5** | 9 |
| 3 | **1** | **7** | 2 | 10 | 5 | 9 |
| 4 | 1 | **2** | **7** | 10 | 5 | 9 |
| 5 | 1 | 2 | 7 | **5** | **10** | 9 |
| 6 | 1 | 2 | **5** | **7** | 10 | 9 |
| 7 | 1 | 2 | 5 | 7 | **9** | **10** |